

# CS6545 Project Report

## Complex Query Evaluation and Optimization over relational and NoSQL Data Stores in Container based Environments

Sarthak Gupta  
Student# 3603399  
Faculty of Computer Science  
University of New Brunswick, Fredericton  
([Sarthak.gupta@unb.ca](mailto:Sarthak.gupta@unb.ca))

Gaurang Mohanbhai Barot  
Student# 3631546  
Faculty of Computer Science  
University of New Brunswick, Fredericton  
([gbarot@unb.ca](mailto:gbarot@unb.ca))

### Abstract

Modern Applications are developed with the back-end support for both relational and non-relational databases. This is because applications are deployed over a number of devices, over a wide range of technologies. Traditionally, SQL databases (MySQL, SQL Server) adhere to a strict schema, while NoSQL databases (Key-Value stores, Graph databases) are schema-less and more flexible. In this Project implementation, we focus on applications that use both SQL and NoSQL Databases on the back-end and compare which of the two is better to deal with variable amounts of data.

For a set of SQL Queries ranging from simple to complex, we compare the execution times of these queries on both SQL and NoSQL databases, hosted in a simulated cloud environment. In order to run these SQL Queries on NoSQL databases, we design our own parser to process the query tokens and translate into NoSQL GET/SET requests. Through the implementation of these queries, we provide evidence that with a growing data set size, a NoSQL Database is much faster than a relational database, even though it includes Query translation and data processing. This is important for real time applications that access millions of records of data in real time, from SQL and NoSQL databases, and speed of data retrieval is an important factor in the application's overall performance.

**Keywords:** SQL; NoSQL; DB Schema; Query translation; Query Optimization; Containers; Virtual Environments.

### 1. Introduction

Traditionally, SQL databases like MySQL have a strict schema, and emphasize on ACID properties (Atomicity, Consistency, Isolation and Durability). However, they are not flexible and scalable which is an important requirement from a database in case of a growing application. In order to overcome the shortcomings of SQL databases, NoSQL Databases were designed in a way to handle both structured and unstructured data. NoSQL databases also scale well and have a superior performance to its counterpart.

In this project report, we design a parser program that executes read and write queries on both SQL and NoSQL databases. For SQL, we chose MySQL standard edition database running on localhost, and for the NoSQL database, we run a REDIS key value store in a Master-Slave configuration on Virtual Machines. This approach and justification for these choices is described in detail in section 5. The business logic of our parser, which is described in more detail in section 6, is to execute queries on MySQL, and then translate these read and write queries into REDIS GET/SET requests and execute on REDIS.

Our results in section 7 demonstrate that applications that use a variety of heterogeneous databases on the backend suffer from SQL database performance and it is suggested to use a Query parser and migrate to NoSQL databases entirely to increase applications' performance. It also overcomes the drawback of having multiple heterogeneous API's to talk to multiple databases and replacing it with one homogeneous API to

talk to a NoSQL database which is integrated with parser support. By default, SQL databases offer provisions for Joins and complex queries, which is difficult to replicate in a NoSQL database since they have no structure. We discuss a way to overcome this by importing a schema on to the REDIS data store and translating the complex queries to retrieve the same result as if received from an SQL Database. Our results show that even with query translation and data processing, REDIS outperforms the SQL databases in READ Queries and matches the performance of SQL in case of WRITE Queries. These results are discussed in more detail in section 7.

In this report, we discuss the previous work that has been done in this area of research in section 2 and discuss the problem statement in more detail in section 3. We then move on to discuss our approach to tackle this problem into section 4, where we describe the architecture and methodology. In section 5, we discuss the environment we set up to execute this project and how we chose the technologies over its alternatives. In section 6, we go on to describe the code of the parser and its implementation in more detail. We then share the results of our experiments in section 7 and the challenges and limitations of this project in section 8. Since this is a group project, we describe our individual contributions to the project in section 9, concluding with references in section 10 and 11 respectively.

## **2. Related Works**

The related works on similar ideology mainly focuses on variety of data available on multiple data store-based applications. It also focuses on the challenges that developers, analysts and data users have to face while working with multiple data store applications. The challenges faced by the data users are ways to express, execute, and optimize complex queries, code adaption and deployment of an application that is based on multiple data stores. The solutions proposed by related works are based on approaches to either integrate multiple data stores using unified data model and associated query language or a specific query engine to combine the results of multiple data stores to perform orders of magnitude better than traditional relational databases. Our approach is based on the first approach mentioned above that is integrating multiple data stores using unified data model and associated query language. This approach ensures us a unique view over relational and NoSQL data stores.

## **3. Problem Statement**

The focus of our project was Query Evaluation and Complex Query Optimization over SQL and NoSQL Databases, and evaluating the result of the performance on both in a cloud environment. This is because most modern applications that are data intensive access data remotely from a database hosted by a cloud service. To achieve this goal, we created batches of data sets ranging from 10 to 1000 rows, in order to evaluate the performance of the two databases over increasing size of the data sets.

Another problem is to imprint the schema from a SQL database to a NoSQL database. A SQL database has a strict schema and only allows values according to the data type of the column. However, NoSQL does not have a strict schema, and neither does it offer functionalities to adapt/import a schema. After schema import and data population in the NoSQL data store, we dealt with the problem of translating READ/WRITE SQL Queries into NoSQL requests using a self-designed parser, developed from scratch. The requirement of this parser was that it would process a set of Queries efficiently and replicate the results for NoSQL database.

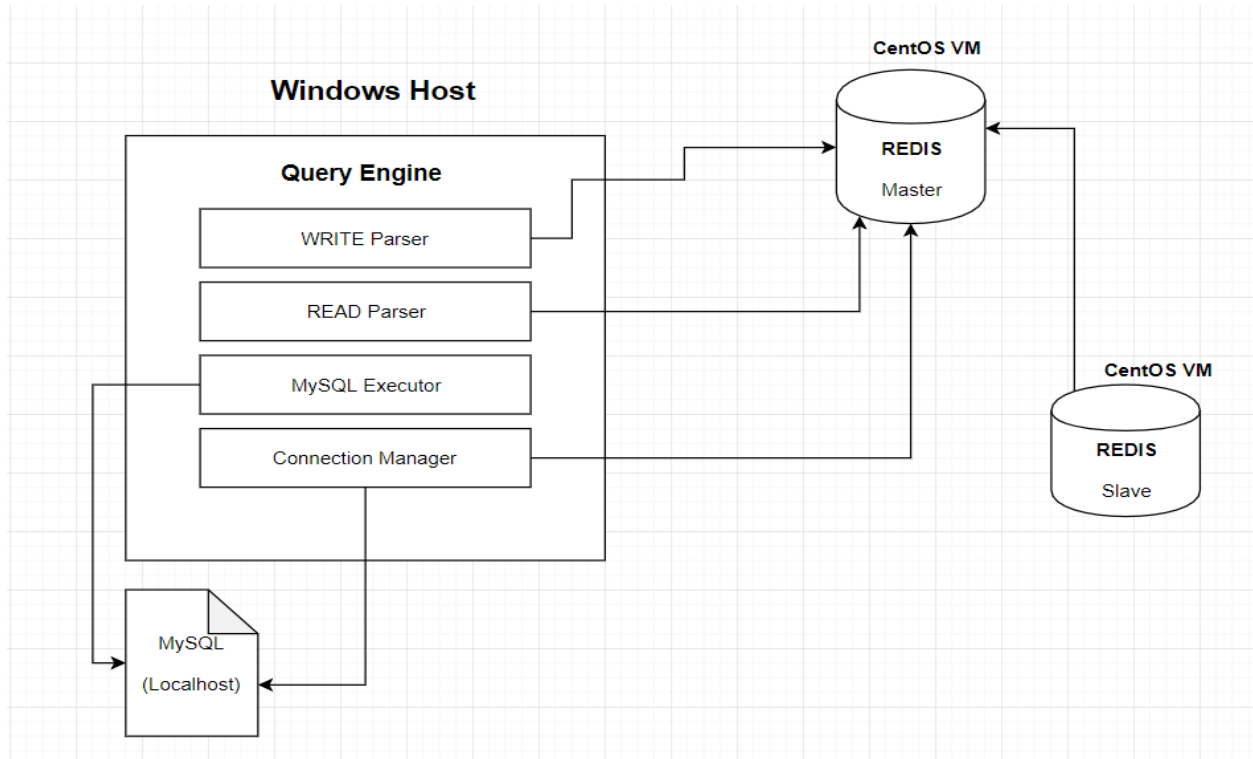
In order to test this parser, it was important to test its performance using a set of Queries defined over SQL data and imported/replicated NoSQL data (same data in different formats over two databases). To design this, our system required a set of tables with a schema, primary and foreign keys so as to test complex queries like Join operations and retrieve and process data similarly when a request for Join operation was

mapped to a NoSQL store. And finally, the problem statement includes the requirement of comparison of Query processing times over SQL and NoSQL data stores.

## 4. Our Approach

### 4.1. Architecture

Fig1. Architecture



As shown in the figure, the architecture of our system consists of three main elements: Query Execution and Parsing Engine, REDIS Data Stores and a MySQL database. The Query Engine was designed in Java programming language using the Eclipse IDE. The program was executed on a Windows Host machine, with Java, JDK, and Eclipse Neon pre-installed. The REDIS data stores were installed using the opensource repository on CentOS virtual machine, in a Master Slave configuration. MySQL was installed on the localhost of the host machine. The Host machine and the two subnets were connected on the network, which means that the IP's assigned to the three machines were from the same subnet.

### 4.2. Setting up the Environment

In order to set up the environment, requirement analysis was done. A machine with a windows host was used with enough memory to install the supported programming languages, Integrated development environments and Virtual machines in order to simulate a Cloud Environment. For the Virtual machine manager, Virtual box was installed and utilized. The operating system used on the virtual machines was CentOS 7, and MySQL was installed locally on the host due to memory constraints. Alternatives and the decision-making process behind these choices are discussed in Section 5.

### **4.3. Populating data in the databases**

Once the environment is setup and the databases are connected to the Query Engine application programming layer, the next step is to populate the MySQL database. We used two tables (Employees and Salaries) to design a schema. Employees had a primary key EMP\_NO defined, which was added to the Salaries table as a foreign key constraint. We populated this schema with data in batches of 10, 100, 500 and 1000. We then used this data to generate CSV Files which were used to enter data into the REDIS data store.

In order to import data into REDIS in a schema like fashion, a Java program was written which takes data from CSV files as input and enters that Data into the REDIS database. The important point to note here is that we used Hashes in REDIS as the preferred Data Structure to store data. The Key for these hashes was created in the Java program using the table name and the unique ID concatenated together. The Value of the Hash was a key value pair where the key was the column name from the table and the value was the actual value of that column in the row corresponding to the unique ID. MySQL Command Line commands were used to generate CSV Files and the external JEDIS (Java-REDIS) library was used to connect to REDIS from Java and perform operations that were equivalent of command line operations.

### **4.4. Designing and Executing Queries**

Given the nature of the two tables in the database, we designed a group of Simple Queries and Complex Queries containing join operations between the two tables. These Queries ranged from reading all rows and columns of a table to reading selected columns of both tables combined using the inner join.

All the Queries were executed on data sets of all sizes, which gave us an evaluation of Query performance over increasing data set size for each query. The Queries were executed manually one-by-one and this process was not automated to make observations for each processing time easily. Another reason for manually executing these queries and not including automation in the business logic of the query engine is that the REDIS query execution time already includes the parsing and query translation time. Including automation would further slowdown the performance of query execution on both databases.

### **4.5. Testing and Evaluation**

After manually executing each query on SQL using Java and Java-Database-Connectivity external libraries, and REDIS using JEDIS (Java-REDIS) external libraries, we had sufficient data to draw an inference out of the results, for each analytical query executed. It was seen that even though Query execution on REDIS includes translation, parsing and presentation of the output on console, READ Query execution in general outperforms on that of MySQL even though MySQL Queries are run on the database hosted locally and does not include network latency time. As far as WRITE queries are concerned, the performance of REDIS either matched that of MySQL or outperformed it. These outcomes are further discussed in Section 7.

## **5. Environment**

This section describes the Hardware and Software components that have been discussed in this report so far and used to implement the ideology of the project. It will also discuss the alternatives, pros and cons, and how the advantages of the technologies used, and the limitations of our project scope led us to make these decisions.

The scope of this project includes evaluation of queries in SQL and NoSQL Data stores in cloud environments. There are plenty of databases available on the cloud, and most of those are available for

commercial/paid use only (For Example: Amazon Web Services). Any databases on the cloud that offer functionality to login in and retrieve data are either paid and protected, or free and public, which is insecure. Therefore, we ruled out using Cloud databases and considered using Container based environments. The options that we considered for containers were Docker and Linux Containers (LxC). On one hand, Docker is free for development, easy to use and has a lot of supporting documentation, LxC containers are not user friendly and the degree of ease of use and learning curve involved with LxC was why we were more inclined towards Docker.

Since the Host operating system is Windows 10, and Docker containers by default are Linux containers, networking and connecting containers was a drawback we were not able to overcome. We created running containers with REDIS images but were not able to connect the Java program on Host with the containers on Docker, even with Bridged Networks and Static IP addresses of the same subnet. Ultimately, we switched our focus to development on virtual machines – a heavy weight version of containers but easier to connect to the host using Virtual box manager. Since it upholds the condition for our databases to not reside on the same operating system and should be reachable remotely over the network instead of locally, we adapted this methodology to set up our environment.

The Host Operating System (Windows 10) runs on top of an Intel(R) Core (TM) i7 – 8550U CPU @ 1.80 GHz, consisting of 8 GB of RAM, 64-bit OS and 2 TB of Hard Disk space. The Virtual OS that were installed on the machines was CentOS 7, running with 2 GB of memory, 20 GB of Hard disk space dynamically allocated and 2 virtual CPU's. Each machine had two working interfaces, a NAT interface to communicate with the External Internet directly and another interface that connects to the host using a Host-only adapter. Static IP addresses are manually assigned to these interfaces after installation, with Gateway IP same as the IP of the host and IP addresses of the same subnet, to create a two-way internet connection.

The MySQL database was installed locally and run using the MYSQLD background service. The user that was used to interact with the Query engine was root and was password protected. Both Virtual Machines had CentOS installed and REDIS and its dependencies were installed on both machines. REDIS was configured to accept traffic from any IP address for ease of testing and was not run in protected mode, without the requirement of a plaintext password. The two REDIS instances were configured in Master-Slave mode and both had backup append-only files enabled for fault tolerance. For the Query Engine to submit requests/ reach the command line interface of this REDIS instance, firewall rules had to be enabled on both machines to update for accepting traffic on port 6379 from any network other than local host.

Since the databases were in a cloud simulated virtual environment, it was the focus of the project to set up the right environment and test its end-to-end working.

## **6. Implementations**

The implementation is composed of 5 steps, which are Step 1-Designing a data structure and dataset. Step 2- Establishing a connection between java and Redis, and, java and MySQL. Step 3-Loading data in both relational database (MySQL) and NoSQL database (REDIS). Step 4-Designing and generating read/get and write/set query parser. Step 5-Designing an application that reads and writes data from and to relational database(MySQL).

### **6.1. Design**

The design consists of the above mentioned 5 Steps.

Step 1- Designing a data structure and dataset. In this step we initially designed a data structure that suits the project concept (Proof of Concept). Based on this data structure we then decided the two different

databases, MySQL and Redis followed by analysing and creating chunks of data that supports these two different databases.

Step 2- Establishing a connection between java and Redis, and, java and MySQL. In this step we established a connection between java and Redis client interface using Jedis and a public IP address of the virtual machine where the Redis Client interface runs. At the same time, we also established a connection between java and MySQL client interface that runs on a localhost.

Step 3- Loading data in both relational database (MySQL) and NoSQL database (REDIS). In this step we loaded the data of different sized chunks (data of 10, 100, 500 and 1000 records) in MySQL using MySQL command line interface. To maintain the exact replica of relational data on Redis we generated a csv of relational data using MySQL CLI. In addition, we also designed a java code that can read the csv file and can load the csv data in Redis.

Step 4- Designing and generating read/get and write/set query parser. In this step we designed a java code that analyses a MySQL query, splits it in sub query, converts it into a corresponding get and set Redis commands and executes these get and set commands by passing it to Redis over java to Redis connection and fetches a result from Redis over a same connection and displays the result.

Step 5- Designing an application that reads and writes data from and to MySQL. In this step we designed a java program that takes an input of MySQL query, passes this query as it is to MySQL client interface over java to MySQL connection and fetches the result of query over the same connection and displays it to user. In step 5 and 6 we also note the time from the instance where the query is sent over connection till, we get back the result of query, in order to compare the performance of MySQL and Redis.

## 6.2. Code Description

In all we have developed five different code for five different purposes. The codes are, 1. MySQL DDL and DML queries, 2. Java2Redis.java, 3. QueryParser.java, 4. RedisInsert.java, 5. MySQL.java. Beginning MySQL DDL and DML queries. These are the queries which were used for creating a data structure and loading a data on MySQL database. These queries were used in step 1 and 3 of design section [6.1] above. For DML queries please refer to code submission with a text file named “DML Statements”.

Fig.2 Create Table Employees

```
CREATE TABLE employees (  
  emp_no      INT          NOT NULL,  
  birth_date  DATE         NOT NULL,  
  first_name  VARCHAR(14)  NOT NULL,  
  last_name   VARCHAR(16)  NOT NULL,  
  gender      ENUM ('M','F') NOT NULL,  
  hire_date   DATE         NOT NULL,  
  PRIMARY KEY (emp_no)  
);
```

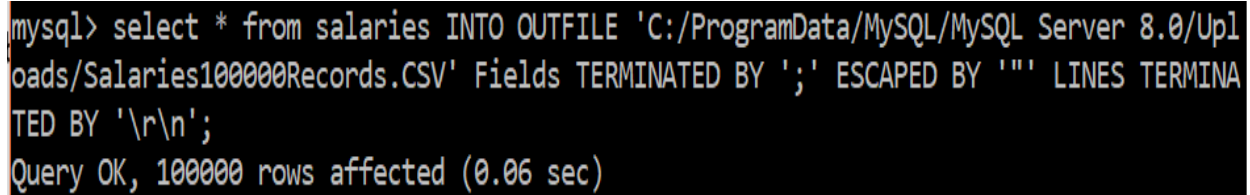
Fig.3 Create Table Salaries

```
CREATE TABLE salaries (  
  emp_no      INT          NOT NULL,  
  salary      INT          NOT NULL,  
  from_date   DATE         NOT NULL,  
  to_date     DATE         NOT NULL,  
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,  
  PRIMARY KEY (emp_no, from_date)  
);
```

Moving on to first code that is ‘Java2Redis.java’, in this code we have developed a logic that reads data from csv (CSV consisting MySQL data that is generated using a MySQL INTO OUTFILE command – Fig below.) and generates ‘HMSET’ command of Redis in order to load the data in Redis database. This is one-

time activity in order to replicate same data in both databases. (for code of Java2Redis.java – please refer to code submission.). This code was used in step 2 and 3 of design section [6.1] above.

Fig.4 MySQL INTO OUTFILE

A screenshot of a MySQL command prompt window. The text is as follows:  
mysql> select \* from salaries INTO OUTFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/Salaries100000Records.CSV' Fields TERMINATED BY ';' ESCAPED BY '"' LINES TERMINATED BY '\r\n';  
Query OK, 100000 rows affected (0.06 sec)

Next is a 'QueryParser.java' and 'RedisInsert.java', In this codes we have developed a logic that checks the syntax of the MySQL 'Select'(read) and 'Insert'(write) query, it identifies the tables used in query, specific columns used instead of '\*' in select query and specific columns used to write the data in database, and other keywords of select query such as 'From', 'Join' and 'Inner Join' of SQL like select query and 'Insert', 'Into', '(', ')', 'Values' of SQL like insert query. After identifying the keywords, it fetches all the keys from Redis database based on the tables mentioned in the select/insert query and sorts these keys and stores it in an array for future use. Now based on the query analysis, whether for simple select query, complex select query such as with join, insert query with all values or insert query with specific values the code generates corresponding 'Set'/'Get' command that reads/writes the same result from Redis database as of MySQL query from MySQL database. In case of select query with join, we have developed a unique solution to join the records from Redis database as Redis being NoSQL database it does not supports join operation, the solution is, at the time of fetching the keys from Redis based on the tables mentioned in query, it stores the key identifier with its occurrence count in a 'Set Collection', this helps to identify whether the record with same identifiers is present in mentioned join tables or not and helps to match the records from join tables. (for code of QueryParser.java and RedisInsert.java – please refer to code submission.). This code was used in step 4 of design section [6.1] above.

Finally, MySQL.java, in this code we developed a logic to take the MySQL query as input and based on whether its read (Select) or write (Insert) query it then fetches the result/writes the data from/to MySQL database. QueryParser.java, RedisInsert.java and MySQL.java also notes the execution time of both read and write operation which is useful to compare the performance of both the databases.

## 7. Evaluation

### 7.1. Experimental Setup

To evaluate our Select and Insert query parsers we performed various test scenarios with different test cases including different select/insert queries with different number of records (10,100,500 and 1000 Records) in both the database. Following are the queries on which we tested are Select query parser. For Insert query parser we considered two different queries, one which inserts data for all columns of table and one which inserts selected columns of table. For insert parser we carried out our evaluation based on 10 and 1000 record insertion query.

Table 1. Test Cases

Query
Select * from Employees
Select * from Employees
Select * from Salaries
Select * from Salaries
Select Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date from Employees
Select Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date from Employees
Select Emp_No, Salary, From_Date, To_Date from Salaries
Select Emp_No, Salary, From_Date, To_Date from Salaries
Select Employees.Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date, Salary, From_Date, To_Date from Employees inner join Salaries on Employees.Emp_No = Salaries.Emp_No
Select Employees.Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date, Salary, From_Date, To_Date from Employees inner join Salaries on Employees.Emp_No = Salaries.Emp_No

## 7.2. Experimental Results

Following table shows the result of all the test scenario of Select (Get) Queries (Commands) with different number of records (10,100,500 and 1000 Records).

Table 2. Execution time of Select (Get) Queries (Commands) in Redis.

Database	Query	Time Taken for Execution(in ms) with 10 Records	Time Taken for Execution(in ms) with 100	Time Taken for Execution(in ms) with 500 Records	Time Taken for Execution(in ms) with 1000 Records
MySQL	Select * from Employees	970	980	1692	990
Redis	Select * from Employees	76	78	248	69
MySQL	Select * from Salaries	556	570	740	659
Redis	Select * from Salaries	72	75	234	73
MySQL	Select Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date from Employees	542	549	620	726
Redis	Select Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date from Employees	73	77	208	71
MySQL	Select Emp_No, Salary, From_Date, To_Date from Salaries	641	653	748	624
Redis	Select Emp_No, Salary, From_Date, To_Date from Salaries	70	76	217	69
MySQL	Select Employees.Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date, Salary, From_Date, To_Date from Employees inner join Salaries on Employees.Emp_No = Salaries.Emp_No	586	624	742	761
Redis	Select Employees.Emp_No, Birth_Date, First_Name, Last_Name, Gender, Hire_Date, Salary, From_Date, To_Date from Employees inner join Salaries on Employees.Emp_No = Salaries.Emp_No	83	89	362	78

Following table shows the result of all the test scenario of Insert (Set) Queries (Commands) with different number of records (10 and 1000 Records).

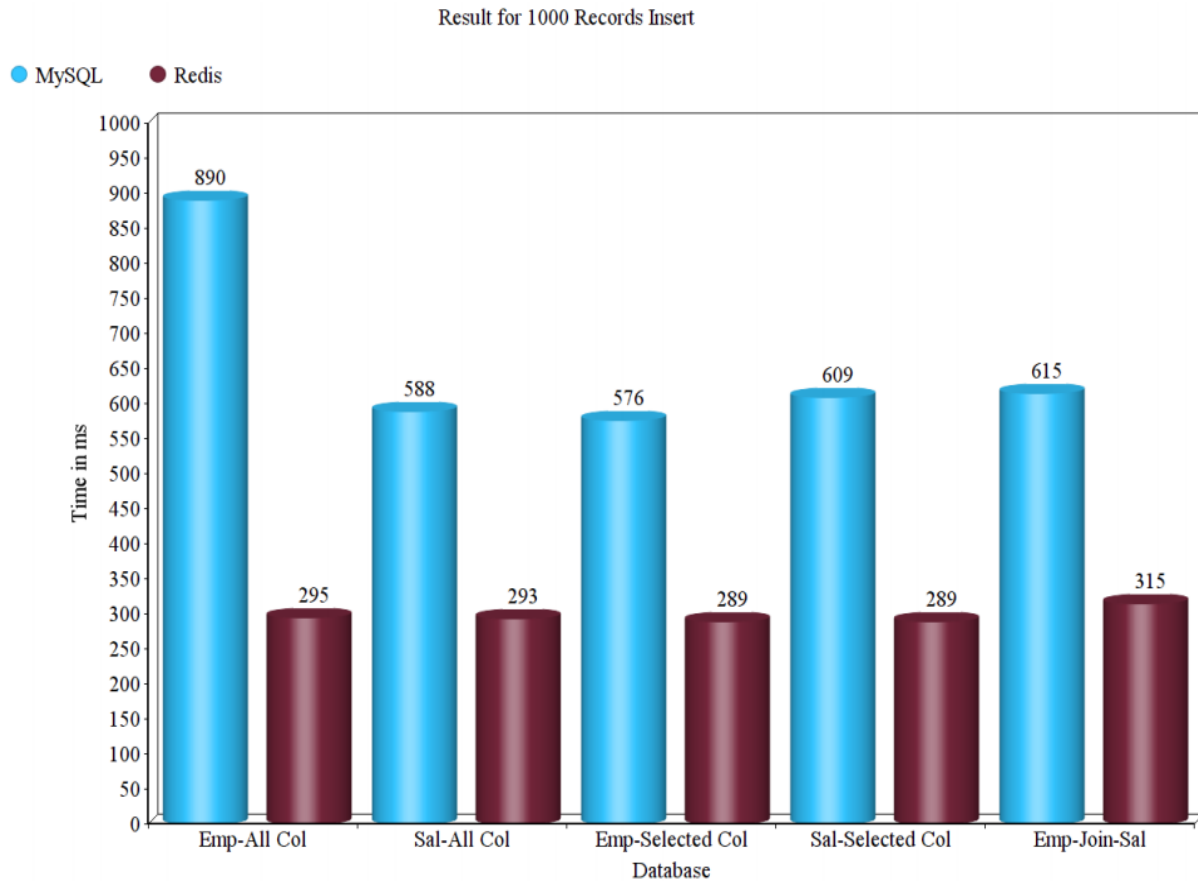


Table 3. Execution time of Insert (Set) Queries (Commands)

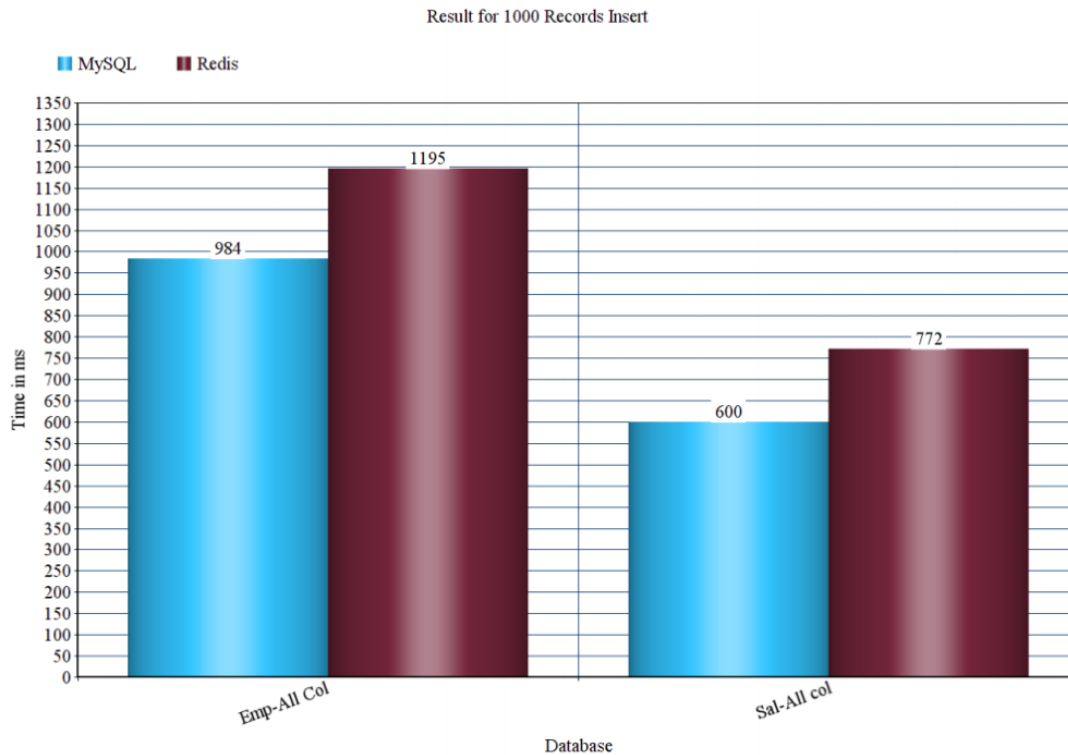
Database	Number of Records	Query	Time Taken for Execution(in ms)
MySQL	10	Insert all Employees	529
Redis	10	Insert all Employees	76
MySQL	10	Insert all Salaries	683
Redis	10	Insert all Salaries	74
MySQL	1000	Insert all Employees	984
Redis	1000	Insert all Employees	1195
MySQL	997	Insert all Salaries	600
Redis	997	Insert all Salaries	772

Based on the execution time noted and as shown in above tables we can say that for read operation (Select query), Redis is much more efficient as compared to MySQL, but in case of write operation (Insert query), Redis and MySQL has similar performance. The graphs below are the representation of above results, to showcase the huge difference in efficiency of Redis vs MySQL.

Graph 1. Average Execution Time of Select Queries with 10, 100, 500 and 1000 Records



Graph 2. Execution time of Insert Queries for 10 and 1000 Records



## 8. Challenges and Limitations

We faced challenges on both software and hardware levels, which led to constant project rescopeing and changes in approaches, led to different methodologies. As of Software, we faced certain programming challenges. First one is to execute set command for every individual key, which makes an application hit the Redis database for the resultant number of records. This involves unnecessary processing time as for each command an application traverse back and forth. In addition, it is also not feasible to store and display the result of Redis command on an application level, as after spending unnecessary processing time on hitting Redis database for individual Redis command, it again requires looping through the collection of result to display for an end user, this again involves extra time for data processing.

As far as Hardware challenges are concerned, we mainly dealt with Trial and Error on the environment front. We tried implementing the business logic of our project on multiple configurations, while dealing with memory constraints, lack of documentations, a mismatch of host and container operating systems, Host to VM and Host to Container Networking issues, along with running virtual machines on a single system with limited memory. It should be noted that at the time of project demonstrations, an average of 80% of CPU was utilised while the environment was completely up and running.

Due to these challenges and limitations, the implementation of a MySQL database was moved from a virtual machine to localhost. There is plenty of documentation to run REDIS on CentOS and there is plenty of documentation available for running Java connected to REDIS and MySQL databases, but there is a special lack of documentation with connecting to REDIS instances running on VM from Java running on Host. Any and all firewall level changes to default policies were made on trial and error basis until a solution (successful connection) was achieved. This is why the environment set up was a time-consuming aspect for this project.

The most consistent limitation we faced was lack of documentation and support regarding running UNIX based containers on a Windows host and networking between them, which is why Docker, a container platform we were most comfortable with had to be discarded as it did not fit well in order for us to accomplish the architecture. Similarly, running LxC on Windows Host was also vetoed. A notable limitation of our evaluation is that since the data from REDIS is retrieved through the network, the network latency is variable, and time of execution in a single scenario is different for each test case.

## 9. Individual Contribution

The scope of this academic project was approximately 12 weeks over the Winter 2019 term.

Week	Phase	Contribution by Sarthak	Contribution by Gaurang
Week 1 - Week 2	Planning	"Project Selection, Reading Research Papers, Discussing Project idea with Dr. Ray"	"Project Selection, Reading Research Papers, Discussing Project idea with Dr. Ray"
Week 3 - Week 6	Planning, Analysis	"Project Proposal, Requirement Analysis, Discussing environment requirements with Dr. Ray"	"Project Proposal, Requirement Analysis, Discussing environment requirements with Dr. Ray"
Week 7 - Week 9	Analysis, Design	"Design, Research, Environment testing, Rescoping according to challenges, Progress Presentation"	"Design, Research Initial Local Coding and Testing, Rescoping according to challenges Progress Presentation"
Week 10 - Week 11	Implementation	"Environment Setup (50%), Insert Query Parser Design (60%), Select Query Parser Design (40%), MySQL automation (50%) Testing Locally and Remotely "	"Environment Setup (50%), Insert Query Parser Design (40%), Select Query Parser Design (60%), MySQL automation (50%) Testing Locally and Remotely "
Week 12	Final Demonstration	"Designing Test Cases Testing, Evaluation, Documentation"	"Designing Test Cases Testing, Evaluation, Documentation"

## 10. Conclusions

Processing on huge amount of data requires a cloud computing environment. At the same time unlike older days, these days the different types of data require different ways of processing it. To work on different types of data it is must to know the relevant database commands and functions. But as an analyst or developer or any data user it is more important to focus more on the processing and insights of data rather than understanding every different type of database. To overcome this problem, we have introduced a parser that converts a relational query in NoSQL database command which makes it easier for an analyst or developer or any data user to work on different types of data by just inputting a command in one of the relational queried language and rest the parser takes care of. At the same time, we have also tried to showcase the benefits of using NoSQL databases in place of Relational database by comparing its performance for similar queries on two different databases with a consideration that relational database is operated locally and NoSQL database is operated on container like environment but still the performance of NoSQL database has an upper hand over relational database.

## 11. References

- [1] Rami Sellami and Bruno Defude, Member IEEE. Complex Queries Optimization and Evaluation over Relational and NoSQL Data Stores in Cloud Environments. IEEE Transactions on Big Data, Vol 5, No. 2, April-June 2018.
- [2] Redis Documentation - <https://redis.io/documentation>
- [3] Redis Commands - <https://redis.io/commands>
- [4] Docker - <https://docs.docker.com/get-started/>
- [5] Dataset - [https://github.com/datacharmer/test\\_db](https://github.com/datacharmer/test_db)