



CSCI 5408

PROJECT REPORT

DPG7 - WINTER'22



Submitted By:

Qurram Zaheer Syed: B00902314
Benny Tharigopala: B00899629
Trushita Maurya: B00913134
Arshdeep Singh: B00890183
Jay Kirankumar Patel: B00906433

CONTENTS

System Architecture – A Bird’s Eye View	5
Design Decisions	5
Persistence Storage	7
Handling persistence storage in memory	8
Local Metadata	9
Global Meta Data	10
Query Parsing	12
Additional Achievements	13
Query Execution	14
Validation	14
1. Schema validation	14
2. Table validation.....	14
1. Column validation	14
2. Data Type validation	14
3. Condition validation.....	14
4. Primary key and unique validation	14
5. Foreign key validation	15
1. USE	15
2. SELECT	15
3. INSERT	15
4. UPDATE	15
5. DELETE.....	16
6. CREATE (table).....	16
Execution.....	16
1. USE	16
2. SELECT	16
3. INSERT	16
4. UPDATE	16
5. DELETE.....	17
6. CREATE (schema)	17

7. CREATE (table).....	17
Log Generation	18
Query Log:.....	18
Event Log:.....	21
General Log:	24
Analytics	26
User registration and login	29
Export structure and values	33
Module 1: DB Design	36
Module 2: Query implementation	37
Query parsing.....	37
Query validation.....	42
Query execution.....	43
Module 4: Log Management Implementation	46
Module 5: Data Modelling – Reverse Engineering Implementation	46
Module 6: Export structure and values Implementation	48
Module 7: Analytics Implementation	49
Module 8: User Interface	50
References	57

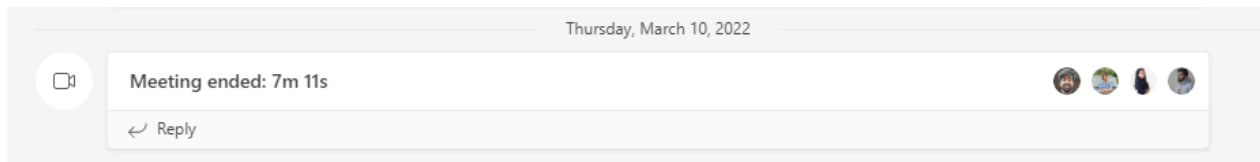
SECTION I: MEETING REPORTS

10.03.2022

Online meeting

Attendees: Qurram Zaheer Syed, Jay Kirankumar Patel, Benny Tharigopala, Trushita Maurya, Arshdeep Singh

1. Breakdown of the workflow of each module
2. Discussion on the possible technical challenges and brainstorming solutions.



15.03.2022

Offline meeting, Killam Memorial Library

Attendees: Qurram Zaheer Syed, Jay Kirankumar Patel, Benny Tharigopala, Trushita Maurya, Arshdeep Singh

1. Discussing the implementation of distributed System and file system.
2. Discussion on the connection between instances through program.
3. Exploring options to test connection to instances in GCP.

19.03.2022

Offline meeting, Kellogg Health Sciences Library

Attendees: Jay Kirankumar Patel, Benny Tharigopala, Trushita Maurya, Arshdeep Singh

1. Basic implementation of the project structure.
2. Implementation of the file system and decision on the location of data and meta files to be stored.

25.03.2022

Offline meeting, Kellogg Health Sciences Library

Attendees: Arshdeep Singh, Benny Tharigopala, Qurram Zaheer Syed, Jay Kirankumar Patel, Trushita Maurya,

1. Brainstorming Implementation of Modules.
2. Discussing design decisions for implementing the metadata and its impact on other modules.

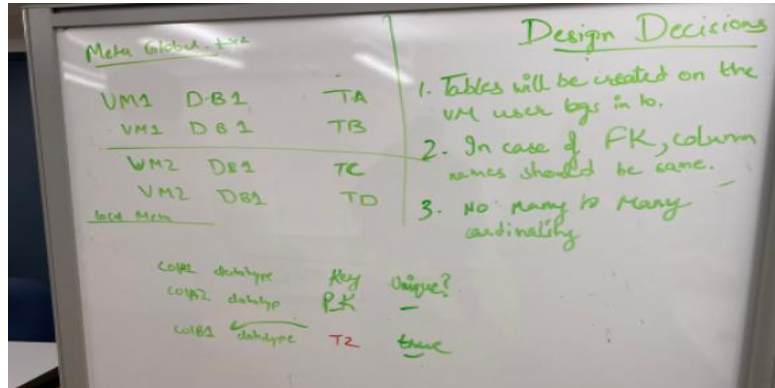


Figure 1: Brainstorming Design decisions on white board

02.04.2022

Offline meeting, Goldberg Computer Science Building

Attendees: Arshdeep Singh, Benny Tharigopala, Qurram Zaheer Syed, Jay Kirankumar Patel, Trushita Maurya

1. Implementing first iteration of Modules
2. Created second iteration for project Structure based on previous meeting.

07.04.2022

Offline meeting, Goldberg Computer Science Building

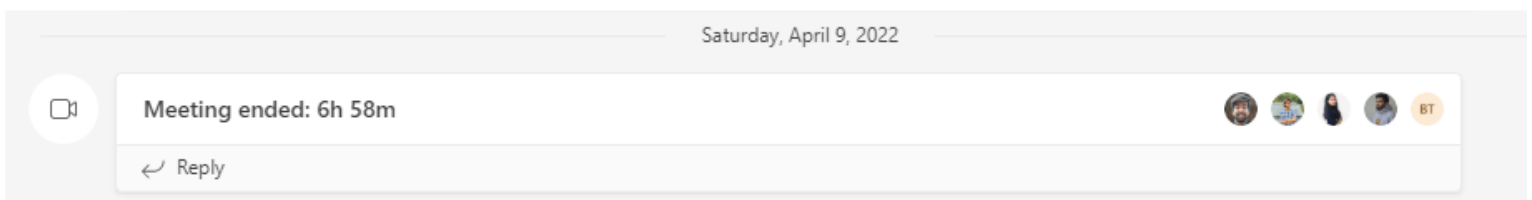
Attendees: Arshdeep Singh, Benny Tharigopala, Qurram Zaheer Syed, Jay Kirankumar Patel, Trushita Maurya

1. Code walkthrough of modules developed by each member.
2. Assigned pending modules and distributed task to each member.

09.04.2022

Online + Offline meeting

1. Integrating all modules
2. Testing each module in GCP instance.



NOTE: ALL MEETINGS PRIOR TO FEASIBILITY REPORT SUBMISSION HAVE BEEN DOCUMENTED IN THE FEASIBILITY REPORT.

SECTION II: ARCHITECTURE AND DESIGN DETAILS

GitLab Repository URL: <https://git.cs.dal.ca/qsyed/csci-5408-w2022-dpg7>

SYSTEM ARCHITECTURE – A BIRD’S EYE VIEW

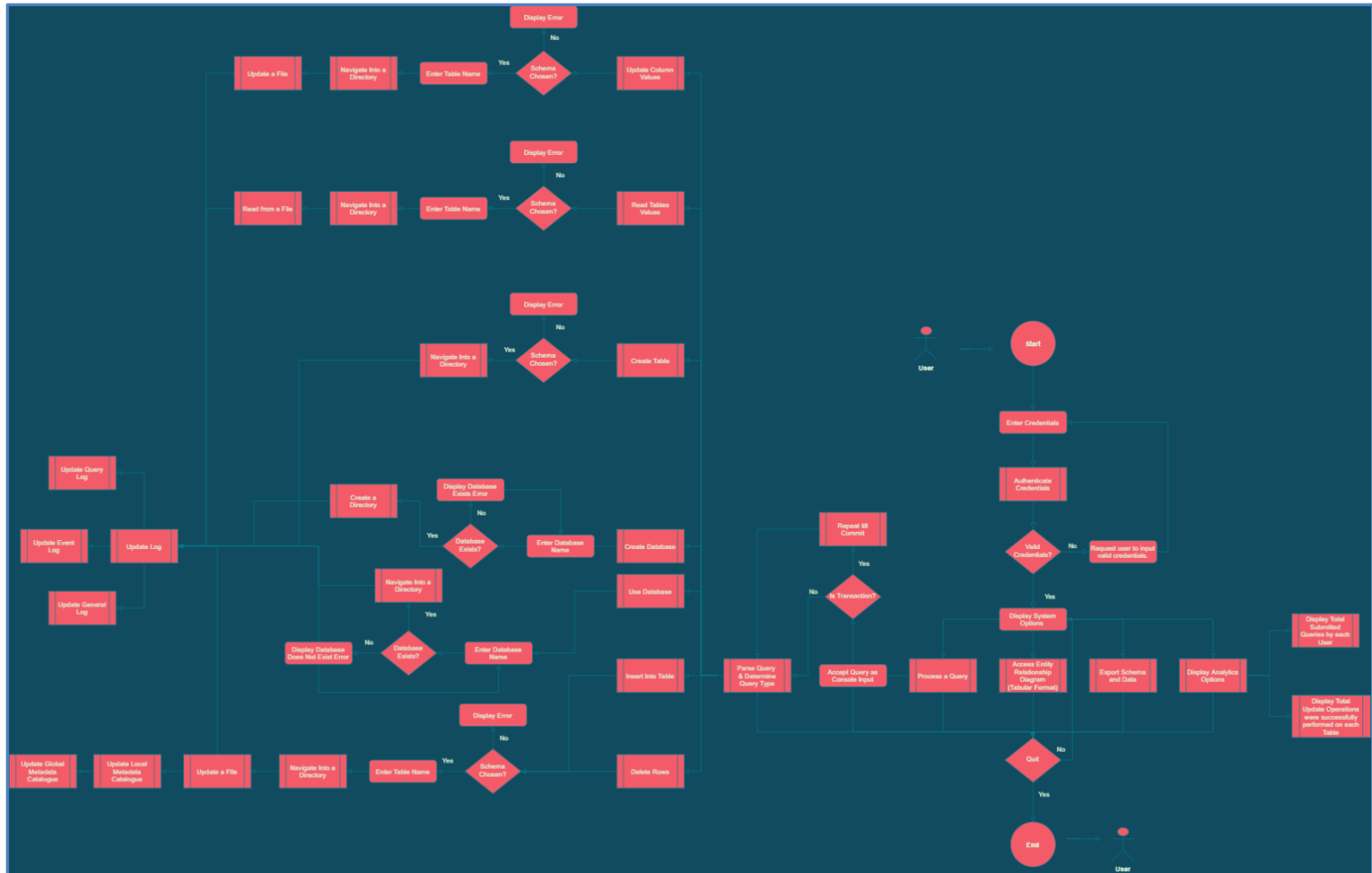


Figure 2: Overall System Architecture

DESIGN DECISIONS

This section contains some critical design decisions that were made while the development stage and the reasons behind them.

1. The parser accepts text values with and without single quotes. For instance, the following two queries are equivalent and valid:



- a. `SELECT ID FROM EMPLOYEE WHERE NAME=JACOB`
- b. `SELECT ID FROM EMPLOYEE WHERE NAME='JACOB'`

The decision was taken to simplify the parsing of textual tokens.

2. For a foreign key constraint, the name of the two participating columns must be the same. This decision was done to simplify the code. Since the name of the columns is the same, to add a foreign key constraint, the user just needs to specify the table name where the foreign key references.

Consider the following tables:

```
EMPLOYEE(ID PRIMARY_KEY, NAME, DEPARTMENT_ID)
DEPARTMENT(DEPARTMENT_ID PRIMARY_KEY, DEPARTMENT_NAME)
```

To add a foreign key from `EMPLOYEE(DEPARTMENT_ID)` to `DEPARTMENT(DEPARTMENT_ID)`, the user will input the following CREATE queries:

```
CREATE TABLE DEPARTMENT(DEPARTMENT_ID FLOAT PRIMARY_KEY, DEPARTMENT_NAME)

CREATE TABLE EMPLOYEE(ID FLOAT PRIMARY_KEY, NAME TEXT, DEPARTMENT_ID FLOAT DEPARTMENT)
```

3. When inserting values user must specify, the columns in order, in which the values are to be inserted. The below format of `INSERT INTO` is not valid:


```
INSERT INTO EMPLOYEE VALUES(1, JACOB)
```
4. In the Query Log, if the syntax of a query is incorrect, the log entry labels the query as “Invalid”.
5. In the Query Log, if the query is not associated with any database or table, the corresponding fields are “null”.
6. In the Event Log, for User Registration and Login Events, a “No User Logged In” label is logged, instead of a `UserId`.
7. We have decided that the metadata files for our distributed database system should contain information about the datatype of each column, Key classification details and if the unique constraint applies to the column.
8. If the column is a foreign key, the name of the table which this foreign key references, is recorded, whereas if the column is neither of the keys, “null” is rendered

PERSISTENCE STORAGE

The persistence storage for our distributed database was designed with memory efficiency and query efficiency in mind. The folder structure used in our persistent storage is shown in the figure below:

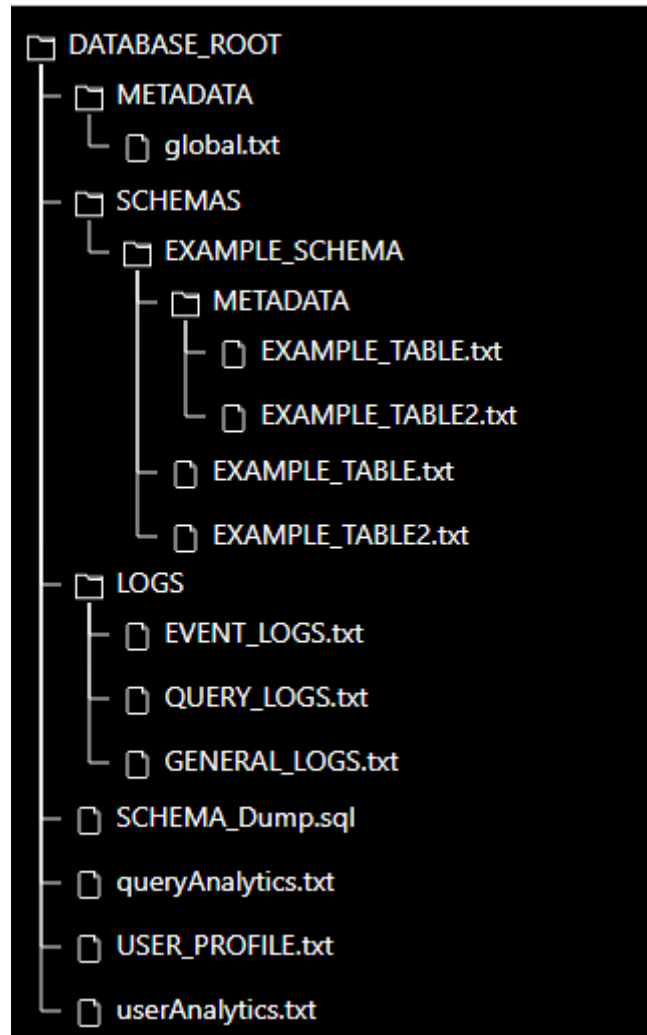


Figure 3 File structure of the DDBMS

The `global.txt` file in the `DATABASE_ROOT/METADATA` folder contains the global metadata, and the table level metadata is stored in the `{SCHEMA_NAME}/METADATA` folder with the same name as the table it corresponds to. This structure makes it easier to load schemas and tables into memory whenever needed. Although currently reads from storage are done per file, the possibility of using “pages” (i.e., set of files determined by another metadata file) to improve reading efficiency was considered. This can be implemented given some extra time.

Each “data” file in the persistence storage (any file that stores information, e.g., Global metadata, local metadata, table rows) is a `txt` file which contains tab-separated values. Every file is required to have a header row. All the database logs are recorded in the `LOGS` folder, classified into `EVENT_LOGS`, `QUERY_LOGS`, and

GENERAL_LOGS. The root folder (DATABASE_ROOT) also contains any generated dump files, analytics files, and the authentication file (USER_PROFILE.txt).

HANDLING PERSISTENCE STORAGE IN MEMORY

The in-memory processing is slightly different for different kinds of “information” files:

- The global metadata is processed into a `List` of `LinkedHashMap<String, String>`, with the key to the Map being the name of the column. Initially, a normal `HashMap` was used in place of a `LinkedHashMap`, but this was changed because using a regular `HashMap` caused the order of the columns to change repeatedly. This was not a software-breaking issue, but it just added extra volatility to our persistence storage, which we didn’t want. This data structure was used for the global metadata since it allowed for easy iteration and $O(1)$ access for values based on the column name (key).
- The table level metadata is processed into a `LinkedHashMap<String, HashMap<String,String>>`, where the key to the `LinkedHashMap` was the name of the column, and the value `HashMap` contained the various attributes for that column. An example of this structure is provided in **Fig. 2**. This structure was used because it provided increased code-efficiency and readability, which was necessary due to the repeated reading of the local metadata.

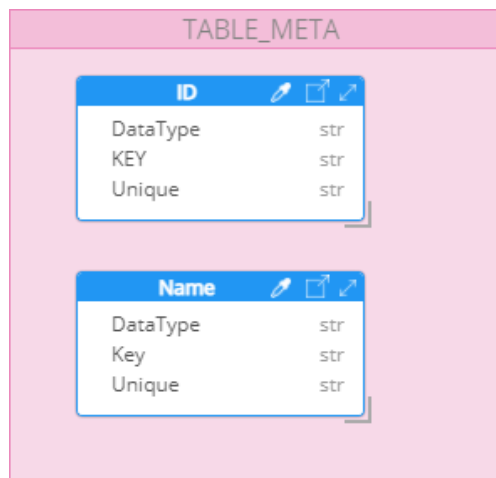


Figure 4 Example in memory structure for table metadata

- The table data is read as a `List` of `LinkedHashMap<String,String>`, similar as the global metadata. This also used to be a `List<HashMap<String,String>`, but was changed because the column order kept changing randomly.
- The logs file is also read as a `List<LinkedHashMap<String,String>`.
- The analytics files are read as `List<String>`.
- The USER_PROFILE file that contains the authentication information for registered users is read as a `List<LinkedHashMap<String,String>>`.

LOCAL METADATA

The local metadata file for each table provides a description of the table in the form of a structured reference. We have decided that the metadata files for our distributed database system should contain information about the datatype of each column, Key classification details and if the unique constraint applies to the column. The following table illustrates the structure of a local metadata file in our application:

Table 1: Local Metadata File Structure

<u>Column Name</u>	<u>Datatype</u>	<u>Key</u>	<u>Unique</u>
--------------------	-----------------	------------	---------------

Column Definitions:

Column_Name: The titles of each column in the table.

Datatype: The datatype of elements in a column. Currently, our system supports float and text datatypes.

Key: This field specifies whether a column is any one of the following values: **Primary Key, Foreign Key or Null**. If the column is a foreign key, the name of the table which this foreign key references, is recorded, whereas if the column is neither of the keys, “null” is rendered

Unique: This field specifies whether a column contains unique values. A value of null corresponds to acceptance of duplicate values in this column.

The Local Metadata file for a table is generated during the creation of a table in the application. Therefore, the method to create the metadata file is invoked, whenever the query parser and processor encounter a “Create Table...” statement that is syntactically and semantically valid.

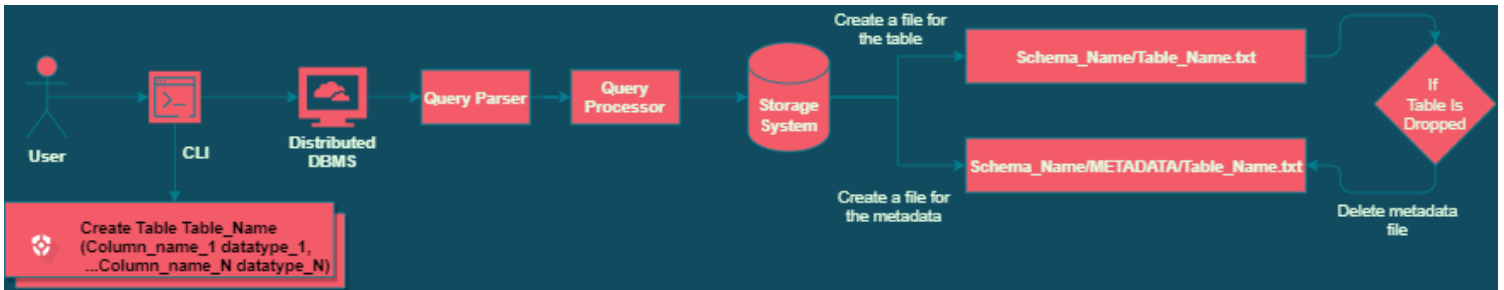


Figure 5: Local Metadata Generation - Process Overview

Sample Metadata File for a Table:

Table 2: Sample Local Metadata file

Column_Name	Datatype	Key	Unique
id	TEXT	PRIMARY KEY	null
name	TEXT	Walmart	null
age	FLOAT	null	null

In this table the column id is a Primary Key. The column – “name” is a Foreign Key, and it references a **column with the same name** from the Table – “Walmart”. All of the columns can contain non-unique values.

```

imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS$ ls
Students db2
imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS$ cd db2
imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS/db2$ ls
METADATA t10.txt t12.txt t14.txt t3.txt t4.txt t5.txt t8.txt
imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS/db2$ cd METADATA/
imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS/db2/METADATA$ ls
t10.txt t12.txt t14.txt t3.txt t4.txt t5.txt t8.txt
imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS/db2/METADATA$ cat t10.txt
Column_Name  Datatype    Key         Unique
id           FLOAT      null       nullimaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/SCHEMAS/db2/METADATA$

```

Figure 6: Sample Local Metadata from a VM Instance

GLOBAL META DATA

Global Metadata is information on data that is used to identify the file location of our file system through global attributes such as the instance name, table and the database names. This file will be present at both instances to achieve distributed database system. It is one of the core concepts for distribution as it holds the overall information about the instances and its details.

Global Meta Data file contains information such as the instance name, database name and the names of the tables. This helps in identifying which table belongs to which instance. Considering distributed database, vertical fragmentation is done by distributing the tables. Therefore, the same database would contain different set of tables in different instances.

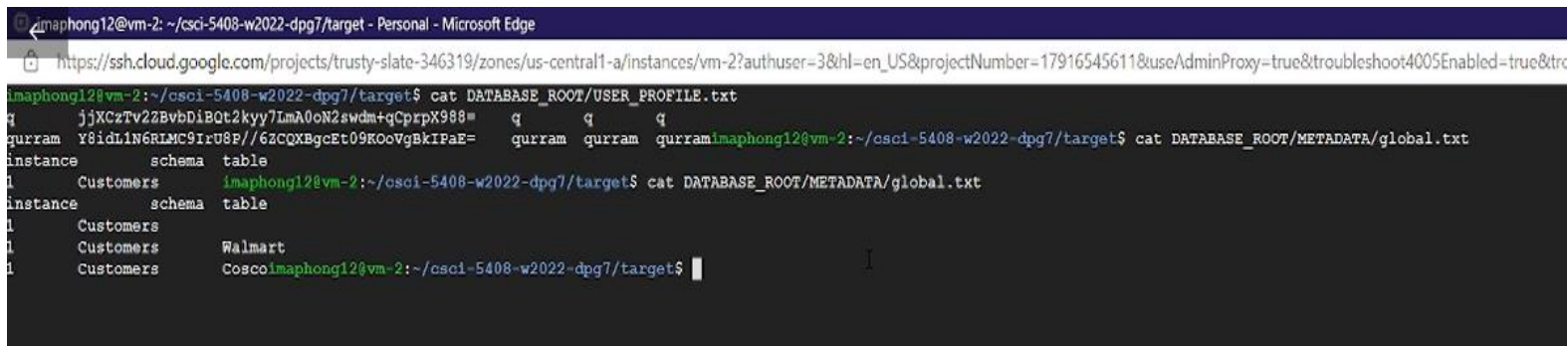
Table 3: Sample Global Metadata file

Instance Name	Database	Table
1	Customers	Walmart
1	Customers	Cosco
2	Customers	Atlantic Store

Table 3 gives a sample of the Global Meta file. It would contain three columns which would be the first row in the text file followed by rows with values corresponding to each column. Here, this file captures two instances “1” and “2” that contains the same database “customers” but has different tables. Instance 1 has two tables “Walmart” and “Cosco” in the “Customers” database. On the other hand, instance 2 has one table “Atlantic store” in the “Customers” database.

How is the entry made in Global Metadata?

- The user will fire a query from any of the instances.
- Semantic analysis of the query would be done.
- Following probabilities are there to process the data further:
 - Creating new database: If the query is on creating a new database, then a entry would be made to both the instances which would denote that the databases will be created in each of these instances.
 - Creating new table: if the query is on creating a new table, then an entry would be made on both global metadata file, but the entry of the table would be against the instance from where the query had been fired.



```

imaphong12@vm-2: ~/csci-5408-w2022-dpg7/target$ cat DATABASE_ROOT/METADATA/global.txt
instance schema table
1 Customers Walmart
1 Customers Cosco
2 Customers Atlantic store
  
```

Figure 7: Global Metadata Generated in VM2 instance (same would be created on VM 1 instance)

Figure 2 shows Global Meta Data in GCP VM2 instance. The Global Meta is created in both the instances and are synchronized with each other to process the query based on the table being used. For example, if a user fires a query on a table that does not exist in one instance, our DBMS system would look up the global meta data to search for the instance where the table exist and provide the result accordingly.

QUERY PARSING

The SQL parser has two parts:

1. Tokenization

Breaking down the SQL query into tokens. Consider the following example:

SQL Query: SELECT ID, NAME FROM EMPLOYEE;

The output of the tokenizer would look like this:

Tokens:{"SELECT", "ID", ",", "NAME", "FROM", "EMPLOYEE"}

2. Syntactic Analysis

Going over the tokens one by one, and validating if syntax is correct. In this phase, we iterate over the tokens, making sure that they make sense. The parsing is halted if the parser reaches the end of query or a syntactic mistake is encountered. After the syntactic analysis, the output looks like:

```
Query{type='SELECT',
      tableName='EMPLOYEE',
      fields=[ID, NAME],
      valuesToUpdate={},
      conditions=[],
      inserts=[],
      datatypes=[],
      constraints=[]}
```

We will discuss the meaning of each of these attributes in the following sections.

INTERNAL WORKING

1. Tokenize the input query
2. Observing some arbitrary queries, we can conclude that the first token will always signify the type of the query, and would be equal to one of the values: INSERT INTO, SELECT, UPDATE, DELETE FROM, CREATE etc. Any other input would lead to a syntactic error. This is the first stage in parsing the tokens.
3. After identifying the type of query, there are only certain inputs that are valid. For instance, if the query type is "SELECT", the next thing a parser expects is a valid table name. If the query type is "CREATE", the next token a parser expects is either the keyword "DATABASE" or the keyword "TABLE".
4. If we go with our initial example query, "SELECT ID, NAME FROM EMPLOYEE", the parser expects either "*" or comma-separated column names (ID, NAME in this instance) after the SELECT keyword.
5. These certain input requirements of a parser are analogous to a state in a Finite State Machine.
6. Our parser is also a simple Finite State Machine that has states and based on the input tokens at each state, it transitions to the next state.

7. Possible states and transitions for our parser are defined in Figure 1

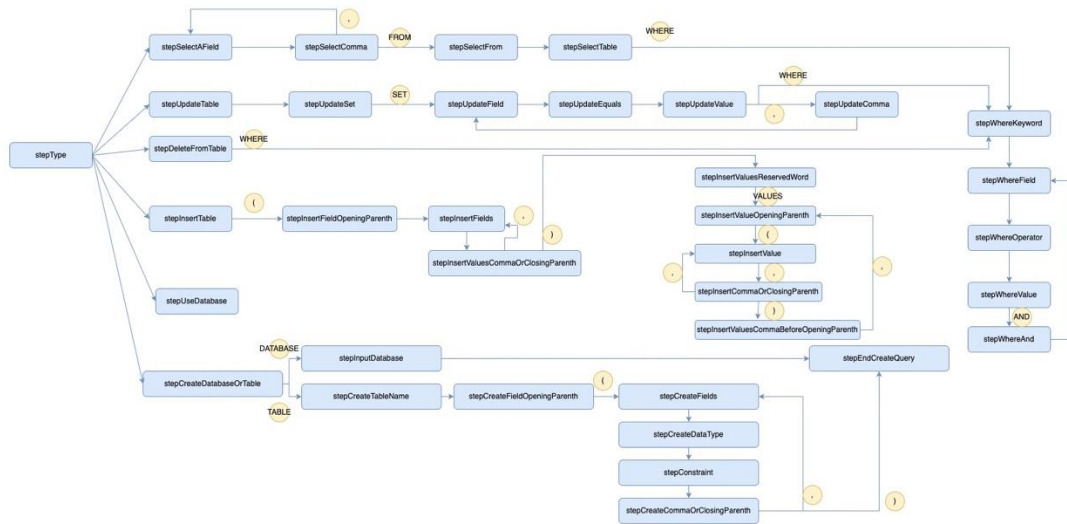


Figure 8: Parser stages and transitions

ADDITIONAL ACHIEVEMENTS

1. INSERT INTO supports the insertion of multiple rows at a time. For instance, the query INSERT INTO EMPLOYEE (ID, NAME) VALUES (1, JACOB), (2, JANE) will insert two rows into the table EMPLOYEE.
2. The WHERE clause supports multiple AND chaining. This was made possible by making 'conditions' attributes in Query class a list, and looping back to the step *stepWhereField* if an AND is encountered. This is an additional functionality over the requirements specified in the project handout.
3. A complete Continuous Integration and Development Pipeline that automates Build, Test and Code Quality processes. The application is also linked to DesigniteJava and QScored. These tools provide insights about an application's code quality and helps reduce code smells.

QUERY EXECUTION

VALIDATION

After a query is parsed and its syntax is validated, the ExecutionDriver proceeds to validate the query in terms of the metadata. This validation happens in various steps, with some steps being common for all queries, and some specific to certain queries. All the different kinds of validations are mentioned below:

1. SCHEMA VALIDATION

The first step in validating a query is to validate the schema (check whether it exists). This validation is used only for the USE query.

2. TABLE VALIDATION

Following schema validation, the existence of the table is then verified using the global metadata. In addition to checking for the existence of the table, this step in the validation also returns the ID of the instance the table is present in if it exists, to facilitate easier retrieval of table metadata.

1. COLUMN VALIDATION

Once the table is validated and the location of the table (which one of the instances its present in) is retrieved, the columns of the query are then validated. Based on the location of the table, the relevant table-level metadata is read and processed, and then used to do the validation. This validation has two different steps:

1. Check if the columns passed in the query exist in the table.
2. For insert and update queries, check if all table columns have been mentioned in the query.

2. DATA TYPE VALIDATION

When the columns are all validated to be existing and correct, the data type of the columns are validated using the table-level metadata.

3. CONDITION VALIDATION

Once the data types are validated, the WHERE condition is validated too. For TEXT type fields, only “=” is allowed, whereas for FLOAT type fields, “<”, “>” or “=” are allowed.

4. PRIMARY KEY AND UNIQUE VALIDATION

For insert and update queries, uniqueness constraint is also validated once the columns, the data types, and the conditions are validated.

5. FOREIGN KEY VALIDATION

Following the primary key and unique constraint validation, the foreign key validation is done. This is a slightly more complex process, in which the following steps are involved.

1. Find the table where the foreign key exists and get its location
2. Based on its location, read the foreign key table's table-level metadata and all its rows.
3. Validate the data type of the foreign key column
4. Validate the uniqueness constraint of the foreign key column by reading all the records
5. Validate the existence of foreign key in foreign key table

The above validations are used individually for different queries based on the requirement. Each query has a different set of validations done. The individual validations needed for each query are provided below.

1. USE

- Schema validation

2. SELECT

- Schema validation
- Table validation
- Column validation

3. INSERT

- Schema validation
- Table validation
- Column validation
- Data type validation
- Primary key and unique validation
- Foreign key validation

4. UPDATE

- Schema validation
- Table validation
- Column validation
- Data type validation
- Condition validation
- Primary key and unique validation
- Foreign key validation

5. DELETE

- Schema validation
- Table validation
- Condition validation

6. CREATE (TABLE)

- Schema validation

EXECUTION

After all the required validations are done, the ExecutionDriver proceeds to execute each query. The steps followed to execute different kinds of queries are shown below:

1. USE

- Validate use
- Set the current schema in the application state, which is a Singleton class
- Invoke logger

2. SELECT

- Validate select
- Load table based on table location (gathered from the validation)
- Identify columns (all columns if *, or specific columns)
- Parse and implement WHERE condition, filter records
- Print out filtered records to console
- Invoke logger

3. INSERT

- Validate insert
- Get table location
- Append record to correct table location
- Invoke logger

4. UPDATE

- Validate update
- Get table location
- Load all table records based on location
- Find records that satisfy WHERE condition
- Update relevant columns of satisfying records

- Overwrite table file in correct location with updated table records
 - Invoke logger
-

5. DELETE

- Validate delete
 - Get location of table
 - Read table records and table metadata from correct location
 - Find indices to delete based on the WHERE condition
 - Remove those indices from the `List<LinkedHashMap<String,String>>`
 - Overwrite table file in correct location with updated table records
 - Invoke logger
-

6. CREATE (SCHEMA)

- Make entry in global metadata
 - Create a folder SCHEMA_NAME in the SCHEMAS directory in DATABASE_ROOT for both instances
 - Invoke logger
-

7. CREATE (TABLE)

- Validate create
- Create entry in global metadata of schema name, table name, and instance
- Create a txt file called TABLE_NAME.txt in the SCHEMA_NAME directory.
- Create a txt file called TABLE_NAME.txt in the SCHEMA_NAME/METADATA directory (this is the metadata file)
- Populate metadata file with the information used in the CREATE query.



Figure 9 Overall flow of ExecutionDriver

LOG GENERATION

This section details the various logs that are generated during the occurrence of various incidents in the system. Our application maintains 3 different types of logs, namely: Query Log, Event Log and a General Log.

```

imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$ ls
EVENT_LOG.txt  GENERAL_LOG.txt  QUERY_LOG.txt
  
```

Figure 10: Logs - Directory Structure

QUERY LOG:

The Query Log, logs information pertaining to any query that a user submits to the system, irrespective of its validity. It plays a vital role in the analytics module since the module utilizes the records present in the query log to furnish insights relevant to query submission. The query log primarily lists, the user who submitted the query, the query type, its execution time and corresponding table & database. The following table represents the structure of the query log in our application.

Query Log – Structure:

Table 4: Query Log Structure

<u>logTimestamp</u>	<u>instanceName</u>	<u>databaseName</u>	<u>tableName</u>	<u>userId</u>	<u>queryValidity</u>	<u>queryType</u>	<u>queryExecutionTime</u>	<u>query</u>
---------------------	---------------------	---------------------	------------------	---------------	----------------------	------------------	---------------------------	--------------

Column Definitions:

logTimestamp – The timestamp corresponding to when a query is logged into the system.

Timestamp format: **YYYY-MM-DDTHH:MM:SS.Milliseconds**

instanceName – The instance in which a query is submitted.

databaseName – The current schema against which a query is being executed.

tableName – The name of the table which a query is relevant to. If the user submits queries relevant to the creation and “Use” of a database, this field is “null”.

userId – The id of the user who submitted a query.

queryValidity – This field specifies whether a query is valid or invalid subsequent to the query being parsed and processed by the system.

queryType – This field specifies the nature of the query. Possible values include:
[“Create”, “Select”, “Update”, “Include”, “Use”, “Delete”, “Drop”]

queryExecutionTime – The time taken by the system to parse and execute the query.

query – The actual query submitted by an user.

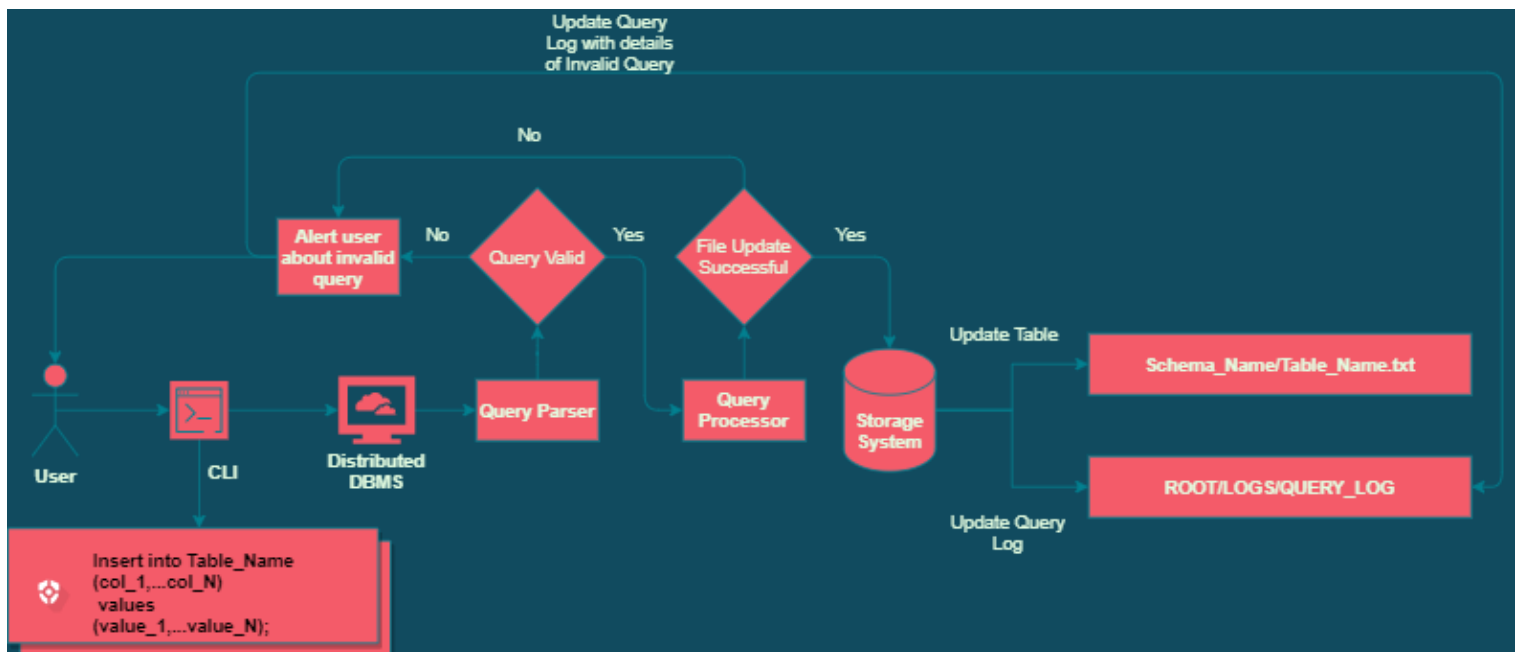


Figure 71: Query Log Generation - Process Overview

```

imaphong12@vm-1:~/csci-5408-w2022-dpg7$ cd DATABASE_ROOT/
imaphong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT$ cd LOGS/
imaphong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$ cat QUERY_LOG.txt
2022-04-10T22:38:36.350803900Z null null NULL Danny INVALID NULL 0 Create Table tb1(sds)
2022-04-10T22:41:53.654028Z null null null Danny INVALID CREATE 0 create t1
2022-04-10T22:47:29.109992600Z null null null Danny INVALID USE 4 Use db1
2022-04-10T22:50:25.766508500Z null null null Danny INVALID USE 6 use db1
2022-04-10T22:51:41.105996800Z null db2 null Danny VALID USE 13 use db2
2022-04-10T22:53:06.387076400Z null db2 NULL Danny INVALID NULL 0 5
2022-04-10T23:14:54.455745700Z null db2 null Danny VALID USE 3 use db2
2022-04-10T23:15:16.565462600Z null db2 t3 Danny VALID CREATE 25 create table t3(id float, name text)
2022-04-10T23:20:01.379393500Z null db2 null Danny VALID USE 2 use db2
2022-04-10T23:20:28.694727700Z null db2 t4 Danny VALID CREATE 29 create table t4(id float, name text, last_name text)
2022-04-10T23:24:40.816932800Z null db2 null Danny VALID USE 3 use db2
2022-04-10T23:24:53.247845600Z null db2 t5 Danny VALID CREATE 16 create table t5(id float)
2022-04-10T23:45:21.693668600Z null null null Danny INVALID USE 2 use d2
2022-04-10T23:45:29.000536400Z null db2 null Danny VALID USE 9 use db2
2022-04-10T23:45:36.772753700Z null db2 null Danny INVALID CREATE 0 create tale t7(id float)
2022-04-10T23:54:15.356161900Z null db2 null Danny VALID USE 5 use db2
2022-04-10T23:54:29.094133200Z null db2 t8 Danny VALID CREATE 19 create table t8(id float, name text)
2022-04-10T23:58:34.419897200Z null db2 null Danny VALID USE 2 use db2
2022-04-10T23:58:49.592421400Z null db2 t10 Danny VALID CREATE 17 create table t10(id float)
2022-04-11T00:01:58.724519600Z null db2 null Danny VALID USE 3 use db2
2022-04-11T00:02:07.344274600Z null db2 t12 Danny VALID CREATE 12 create table t12(id float)
2022-04-11T00:03:09.704567700Z null db2 null Danny VALID USE 2 use db2
2022-04-11T00:03:20.661763300Z null db2 t14 Danny VALID CREATE 13 create table t14(name text)
imaphong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$

```

Figure 82: Sample Query Log from a VM Instance

Notice in the snippet above:

- If the syntax of a query is incorrect, the log entry labels the query as “Invalid”.
- If the query is not associated with any database or table, the corresponding fields are “null”.

Class Hierarchy & Members:

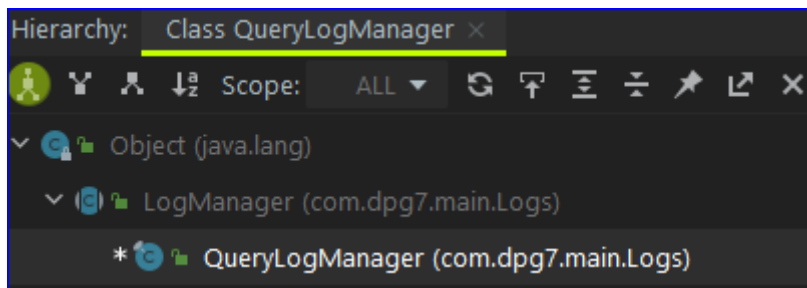


Figure 93: Query Log Class Hierarchy

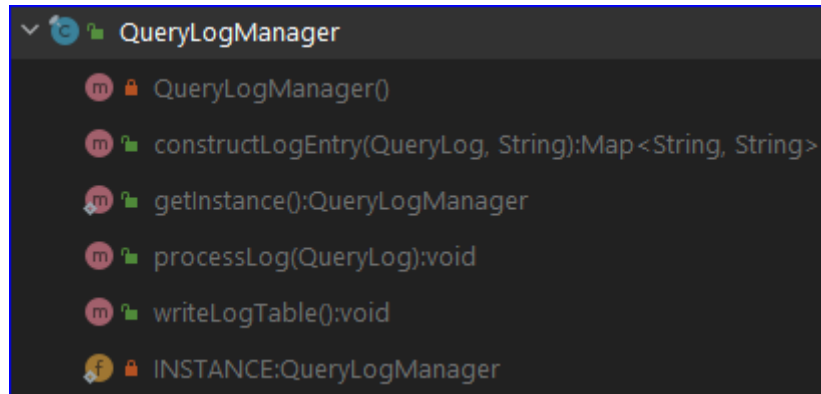


Figure 104: Query Log Class Members

EVENT LOG:

The event log records any event, be it successful or a failure, that occurs in our system. It primarily describes the event through a message, multiple timestamps, once which corresponds to the event occurrence and another which's when the log entry was made and details of the user who triggered the event. The primary purpose of the event log can be interpreted as follows. The event log can be utilized for determining if an event was successful. It is analogous to the success and failure messages displayed to a user through a console in a traditional Relational DBMS.

Event Log – Structure:

Table 5: Event Log Structure

<u>logTimestamp</u>	<u>instanceName</u>	<u>databaseName</u>	<u>tableName</u>	<u>userId</u>	<u>eventTimestamp</u>	<u>eventMessage</u>
---------------------	---------------------	---------------------	------------------	---------------	-----------------------	---------------------

Column Definitions:

logTimestamp – The timestamp corresponding to when a query is logged into the system.

Timestamp format: **YYYY-MM-DDTHH:MM:SS.Milliseconds**

instanceName – The instance in which a query is submitted.

databaseName – The current schema against which a query is being executed.

tableName – The name of the table which a query is relevant to. If the user submits queries relevant to the creation and “Use” of a database, this field is “null”.

userId - The id of the user who submitted a query.

eventTimestamp – The timestamp corresponding to when an event is triggered in the system.

eventMessage – This field specifies the nature of the event by describing the event in addition to the status of the event (Success / Failure).

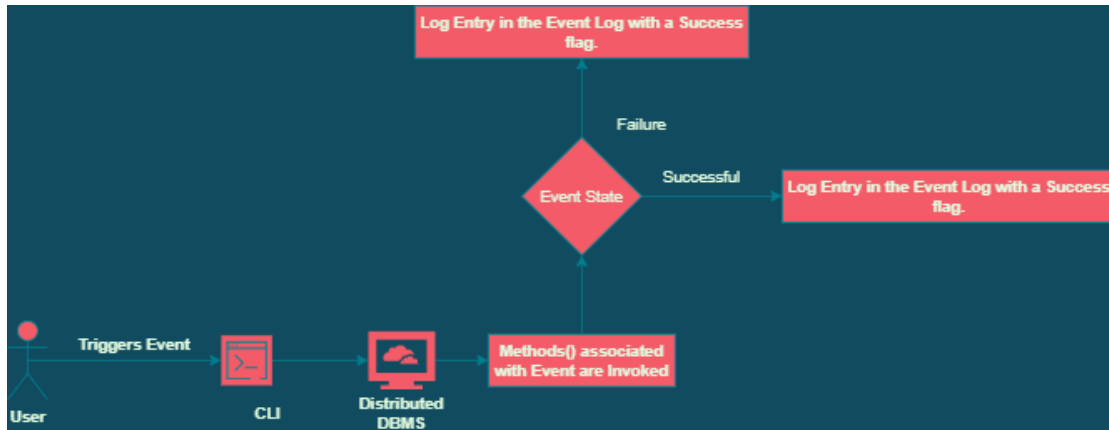


Figure 115: Event Log Generation - Process Overview

```

laophong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT$ cd LOGS/
laophong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$ ls
EVENT_LOG.txt  GENERAL_LOG.txt  QUERY_LOG.txt
laophong12@vm-1:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$ cat EVENT_LOG.txt
2022-04-10T22:21:06.006859900Z null null NULL No USER LOGGED IN! 2022-04-10T22:21:05.950018100Z Use
rid already exists!
2022-04-10T22:26:17.402395900Z null null NULL No USER LOGGED IN! 2022-04-10T22:26:17.334564900Z Inv
alid input for password !
2022-04-10T22:27:10.095592400Z null null NULL No USER LOGGED IN! 2022-04-10T22:27:10.094598200Z New
Profile Created Successfully !
2022-04-10T22:32:16.129337500Z null null NULL No USER LOGGED IN! 2022-04-10T22:32:16.071492200Z New
Profile Created Successfully !
2022-04-10T22:32:22.405079Z null null NULL No USER LOGGED IN! 2022-04-10T22:32:22.405079Z Inc
orrect password.
2022-04-10T22:32:53.033567200Z null null NULL No USER LOGGED IN! 2022-04-10T22:32:53.031581800Z Log
in successful. Welcome Danny
2022-04-10T22:38:15.899067200Z null null NULL No USER LOGGED IN! 2022-04-10T22:38:15.845203600Z Log
in successful. Welcome Danny
2022-04-10T22:41:45.436183Z null null NULL No USER LOGGED IN! 2022-04-10T22:41:45.372357400Z Log
in successful. Welcome Danny
2022-04-10T22:41:53.668988600Z null null NULL No USER LOGGED IN! 2022-04-10T22:41:53.668988600Z Inv
alid Query !
2022-04-10T22:47:23.327380800Z null null NULL No USER LOGGED IN! 2022-04-10T22:47:23.270536600Z Log
in successful. Welcome Danny
2022-04-10T22:47:29.123947200Z null null NULL No USER LOGGED IN! 2022-04-10T22:47:29.123947200Z Inv
alid Query !
2022-04-10T22:50:17.097192300Z null null NULL No USER LOGGED IN! 2022-04-10T22:50:17.021386600Z Log
in successful. Welcome Danny
2022-04-10T22:50:25.788437800Z null null NULL No USER LOGGED IN! 2022-04-10T22:50:25.787446200Z Sch
ema Access Failed
2022-04-10T22:50:25.804401100Z null null NULL No USER LOGGED IN! 2022-04-10T22:50:25.804401100Z Inv
alid Query !
2022-04-10T22:51:15.959636400Z null null NULL No USER LOGGED IN! 2022-04-10T22:51:15.901788200Z Log
in successful. Welcome Danny
2022-04-10T22:51:36.197972300Z null null NULL No USER LOGGED IN! 2022-04-10T22:51:36.197972300Z Que
ry Executed Successfully !
2022-04-10T22:51:41.114970600Z null db2 NULL No USER LOGGED IN! 2022-04-10T22:51:41.114970600Z Dat
abase in Use
2022-04-10T22:51:41.120956900Z null db2 NULL No USER LOGGED IN! 2022-04-10T22:51:41.120956900Z Que
ry Executed Successfully !
2022-04-10T22:53:06.393075500Z null db2 NULL No USER LOGGED IN! 2022-04-10T22:53:06.393075500Z S:i
s not a valid query
2022-04-10T22:53:32.620921100Z null null NULL No USER LOGGED IN! 2022-04-10T22:53:32.553059100Z Log
in successful. Welcome Danny
2022-04-10T23:14:46.290870500Z null null NULL No USER LOGGED IN! 2022-04-10T23:14:46.224044400Z Log
in successful. Welcome Danny
  
```

Figure 126: Sample Event Log

For User Registration and Login Events, a “No User Logged In” label is logged.

The event log contains entries for any operations associated with the following options:

- Store metadata
- View or read metadata
- Database creation
- Removal of a database
- Use a database
- Create a table
- Select records from a table
- Drop a table
- Start a transaction
- Commit a query
- Generate a SQL dump
- Import a SQL dump
- Generate an ERD for a database

Class Hierarchy & Members:

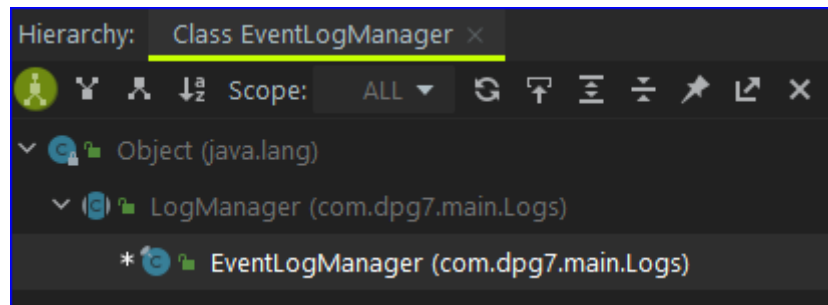


Figure 17: Event Log Class Hierarchy

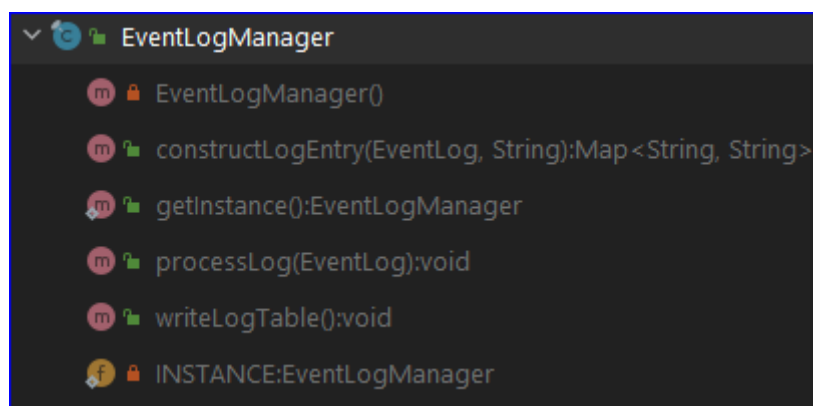


Figure 18: Event Log Class Members

GENERAL LOG:

The General log maintains the state of the database. It renders information about the number of tables and the total number of rows in each database across the instances.

General Log – Structure:

Table 6: General Log Structure

<u>logTimestamp</u>	<u>instanceName</u>	<u>databaseName</u>	<u>userId</u>	<u>numberOfTables</u>	<u>numberOfRecords</u>
---------------------	---------------------	---------------------	---------------	-----------------------	------------------------

Column Definitions:

logTimestamp – The timestamp corresponding to when a query is logged into the system.

Timestamp format: **YYYY-MM-DDTHH:MM:SS.Milliseconds**

instanceName – The instance in which a query is submitted.

databaseName – The current schema against which a query is being executed.

userId – The id of the user who submitted a query.

numberOfTables – This field the total number of tables in the database.

numberOfRecords – This field the cumulative number of records in the database.

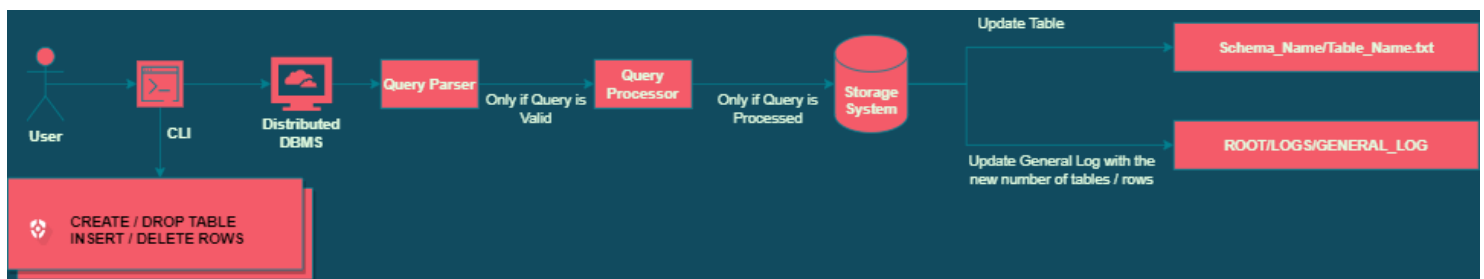


Figure 19: General Log Generation - Process Overview

...

```

imaphong12@vm-2:~/csci-5408-w2022-dpg7/DATABASE_ROOT/LOGS$ cat GENERAL_LOG.txt
2022-04-10T23:14:54.510600Z      null      db2      Danny    0         0
2022-04-10T23:15:16.637264300Z null      db2      Danny    1         1
2022-04-10T23:20:28.791231100Z null      db2      Danny    2         2
2022-04-10T23:24:53.350573100Z null      db2      Danny    3         3
2022-04-10T23:54:29.214813100Z null      db2      Danny    4         6
2022-04-10T23:58:49.655249600Z null      db2      Danny    5         3
2022-04-11T00:02:07.421072400Z null      db2      Danny    6         0
2022-04-11T00:03:20.736563300Z null      db2      Danny    7         0

```

Figure 20: Sample General Log from a VM Instance

Class Hierarchy & Members:

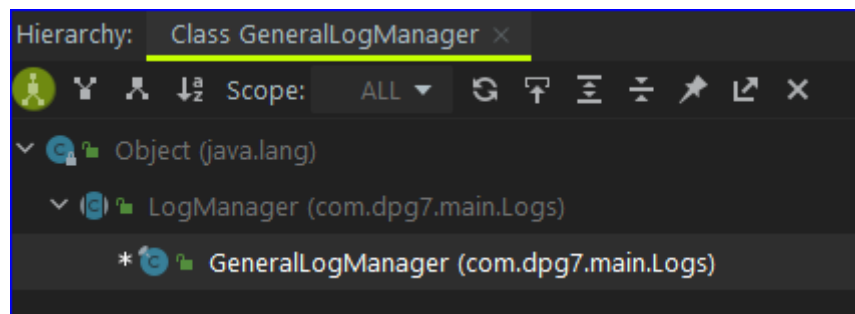


Figure 21: Class Hierarchy

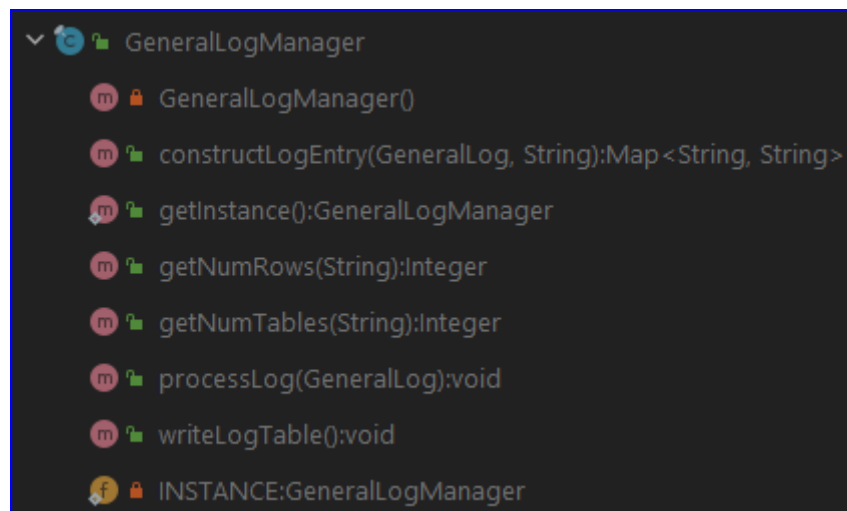


Figure 22: General Log Manager - Class Members

ANALYTICS

Menu Driven Analytics: The user can use this facility by selecting the option 4 from the main menu. After selecting the analytics option the user would be further given a sub menu to select an option.

```
1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 4

1. Queries Submitted By Database.
2. Update/Select/Create Operations By Tables.
```

Figure 23: Menu Driven Analytics Option Provided to User.

Analytics consists of two categories based on which a report is generated. These are:

- **Queries (Invalid and Valid) that are submitted by the database.**

The source for analytics report is based on the query logs. Query logs captures the details such as the table name, instance name, query that was fired by the user (invalid or valid), user name and the database name. All these information is a necessary input to process and generate a report.

Overview when user selects the first option:

Modularized methods that could be reused by both types of analytics. To give an overview of the methods, the following screenshot provides a list of self-explanatory methods that follows SOLID principles.

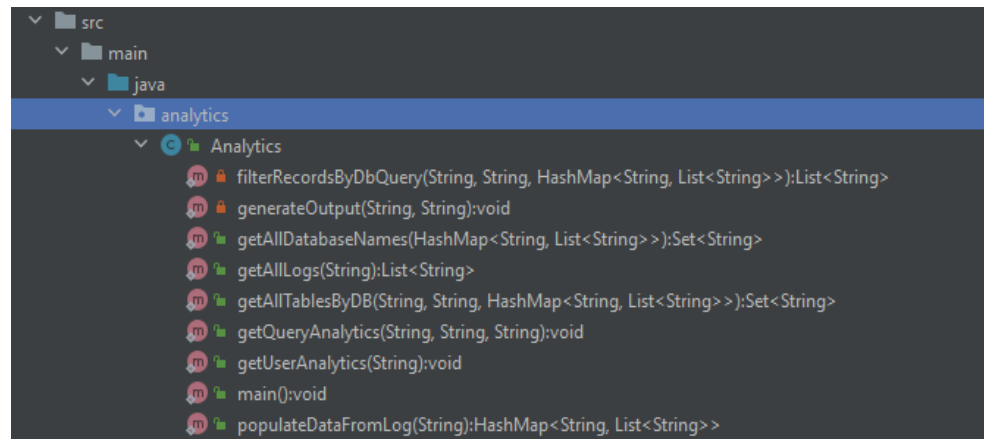


Figure 24: Methods in the Analytics Package.

The entry point to execute any of the reports is done through main method that is called in the **Driver Class**. When the user selects the option 4 from the user menu, the main method of Analytics class would be executed.

User would be provided by two sub menus to choose from, based on the user input provide the report would be generated. The user needs to input either 1 or 2. 1 would generate the report for all queries (valid or invalid) that are done in the database from both instances. 2 would generate the operations on each table for a given database.

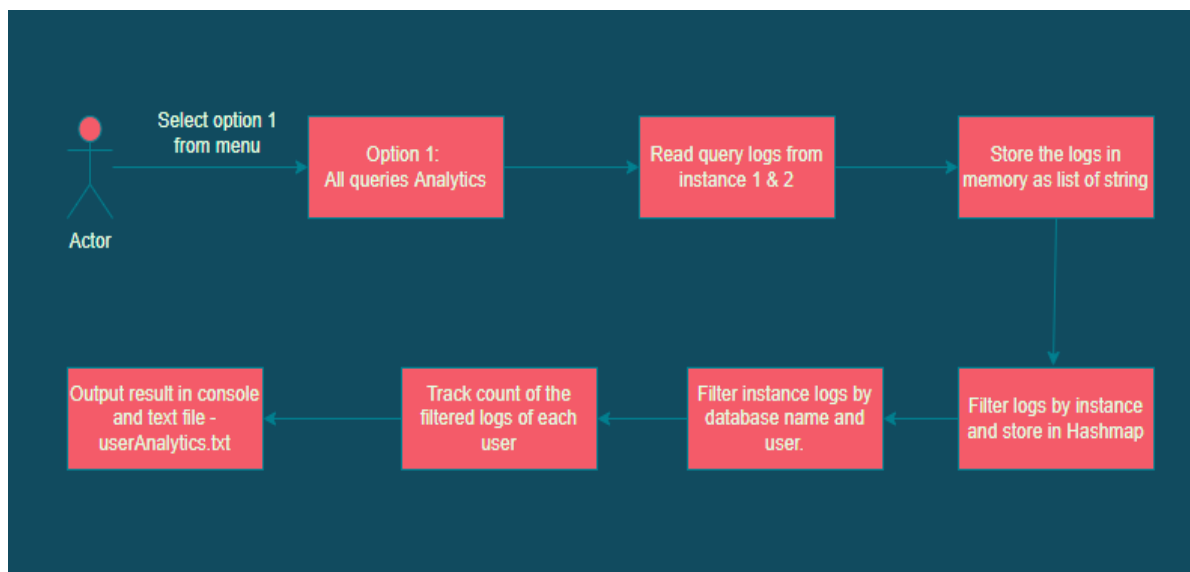


Figure 25: User Analytics Processing when user selects first option

```

1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 4

1. Queries Submitted By Database.
2. Update/Select/Create Operations By Tables.

Select Option:1
Generating Reports...
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target

Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
REMOTE STATUS: false
LOCAL
REMOTE STATUS: true
/home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/LOGS/QUERY_LOG.txt
recieved all logs22
user qurram submitted 2 queries for null running on 1
user qurram submitted 22 queries for Customers running on 1
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target

```

Figure 26 : Output of program for analysis of a user selecting option 1.

- **Operations [UPDATE/SELECT/CREATE] that are submitted by the database for all tables**

Second option gives the analytics of the total operations done on each table for a database entered by user. Once the user selects option 2, data would be processed to output a file as well as display the result in the console.

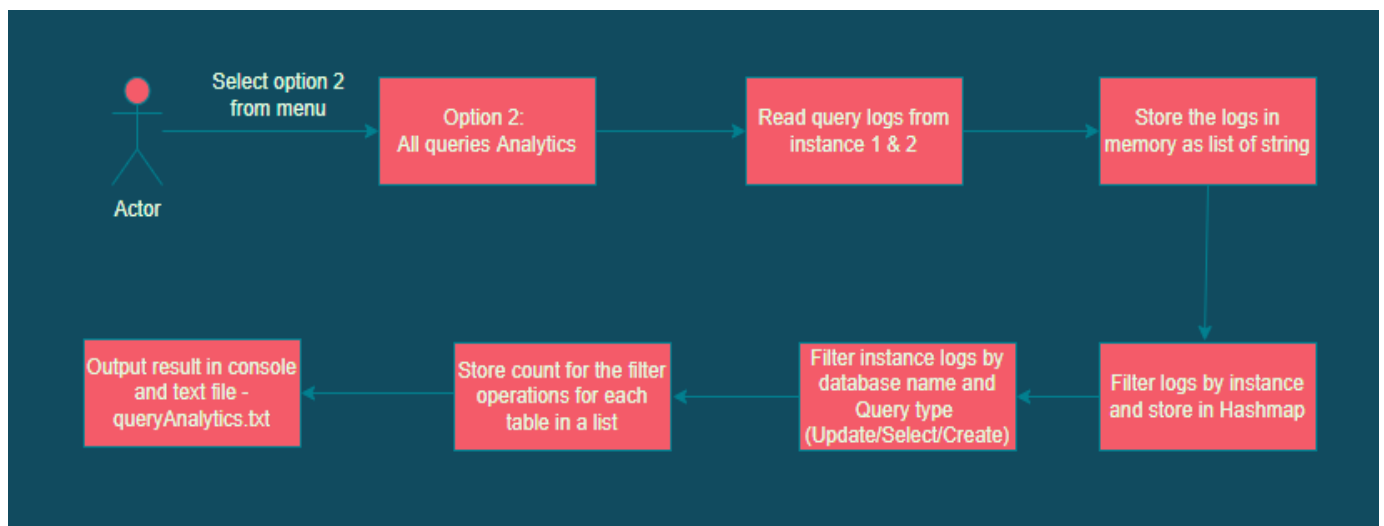


Figure 27: User Analytics Processing when user selects second option

```

imaphong12@vm-1: ~/csci-5408-w2022-dpg7/target - Personal - Microsoft Edge
https://ssh.cloud.google.com/projects/trusty-slate-346319/zones/us-central1-a/instances/vm-1?authuser=3&hl=en_US&projectNumber=17916545611&useAdminProxy=true&troubleshoot4005Enabled=tr
Schema: Customers-2
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-2
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target

1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 4

1. Queries Submitted By Database.
2. Update/Select/Create Operations By Tables.

Select Option:2
Enter Operation [UPDATE,SELECT,INSERT,CREATE] :
CREATE
Enter databaseName :
Customers

Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
REMOTE STATUS: false
LOCAL
REMOTE STATUS: true
/home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/LOGS/QUERY_LOG.txt
received all logs22
Total 2 CREATE operations are performed on Walmart
Total 2 CREATE operations are performed on Cosco
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-2
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-2
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target

```

Figure 28 : Output of program for analysis of a user selecting option 2

USER REGISTRATION AND LOGIN

User Registration:

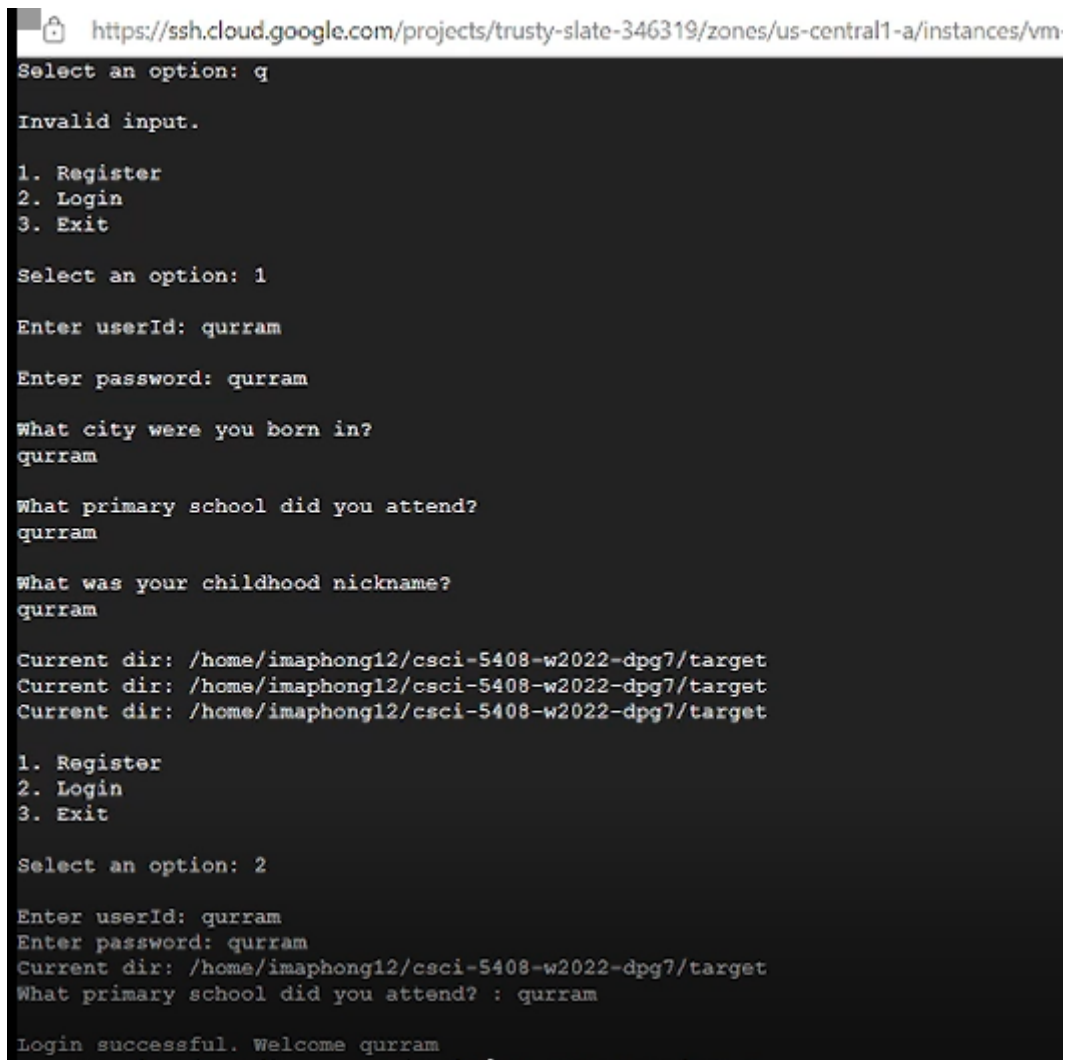
Like any other database, a user should be registered to access the database. To register on this Distributed Relational Database Management System, a user needs to provide a unique userID, password, and answers to 3 security questions. Once the user submits these values, we run some basic validations to check if the submitted values are invalid (duplicate userID, blank values, etc.) If the inputs are invalid, we show appropriate error message and log the information in log. If the inputs are valid, we hash the password and store all the information in the UserProfile.txt file. Since we are building a Distributed DBMS, we copy the UserProfile.txt file in other instance/server, so that a user will be able to access the DBMS through both the instances.

User Login:

To login in the system, a user needs to provide 3 inputs:

1. User ID
2. Password
3. Answer for a randomly selected security question.

Once the user provides the inputs, we run basic validations to check if the inputs are not blank.



```
https://ssh.cloud.google.com/projects/trusty-slate-346319/zones/us-central1-a/instances/vm-
Select an option: q
Invalid input.

1. Register
2. Login
3. Exit

Select an option: 1
Enter userID: gurram
Enter password: gurram
What city were you born in?
gurram
What primary school did you attend?
gurram
What was your childhood nickname?
gurram

Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target

1. Register
2. Login
3. Exit

Select an option: 2
Enter userID: gurram
Enter password: gurram
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
What primary school did you attend? : gurram

Login successful. Welcome gurram
```

Figure 29: Output for registration of user

If the inputs are valid, we check if the User ID, password, and the answer to the security question matches with the information stored in the UserProfile.txt file. If the data matches, we store the user profile information in

session and show the main menu options to the user. If the data doesn't match, we show appropriate error message and log the information in the logs. **Figure 30 and 31** represents the workflow of the user login process.

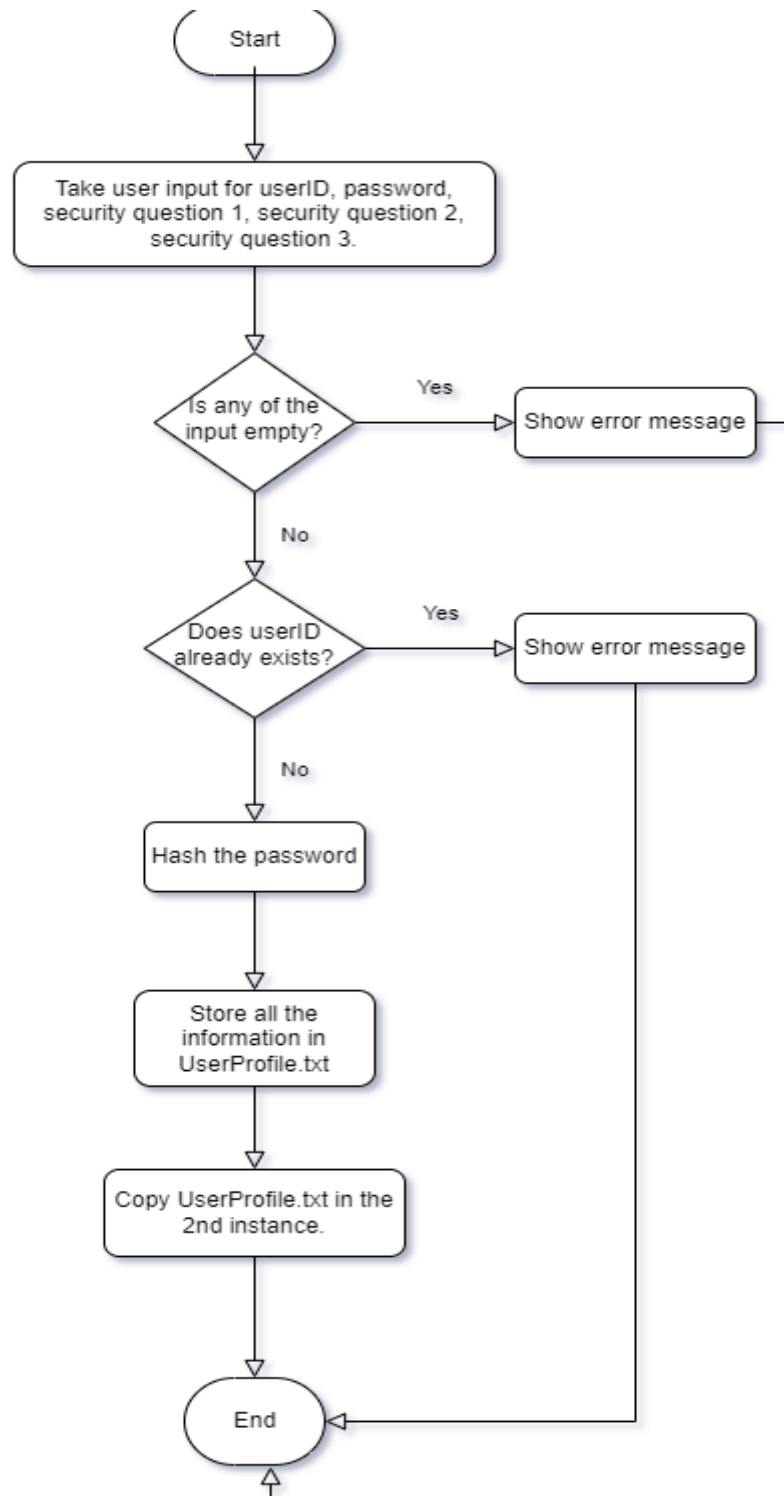
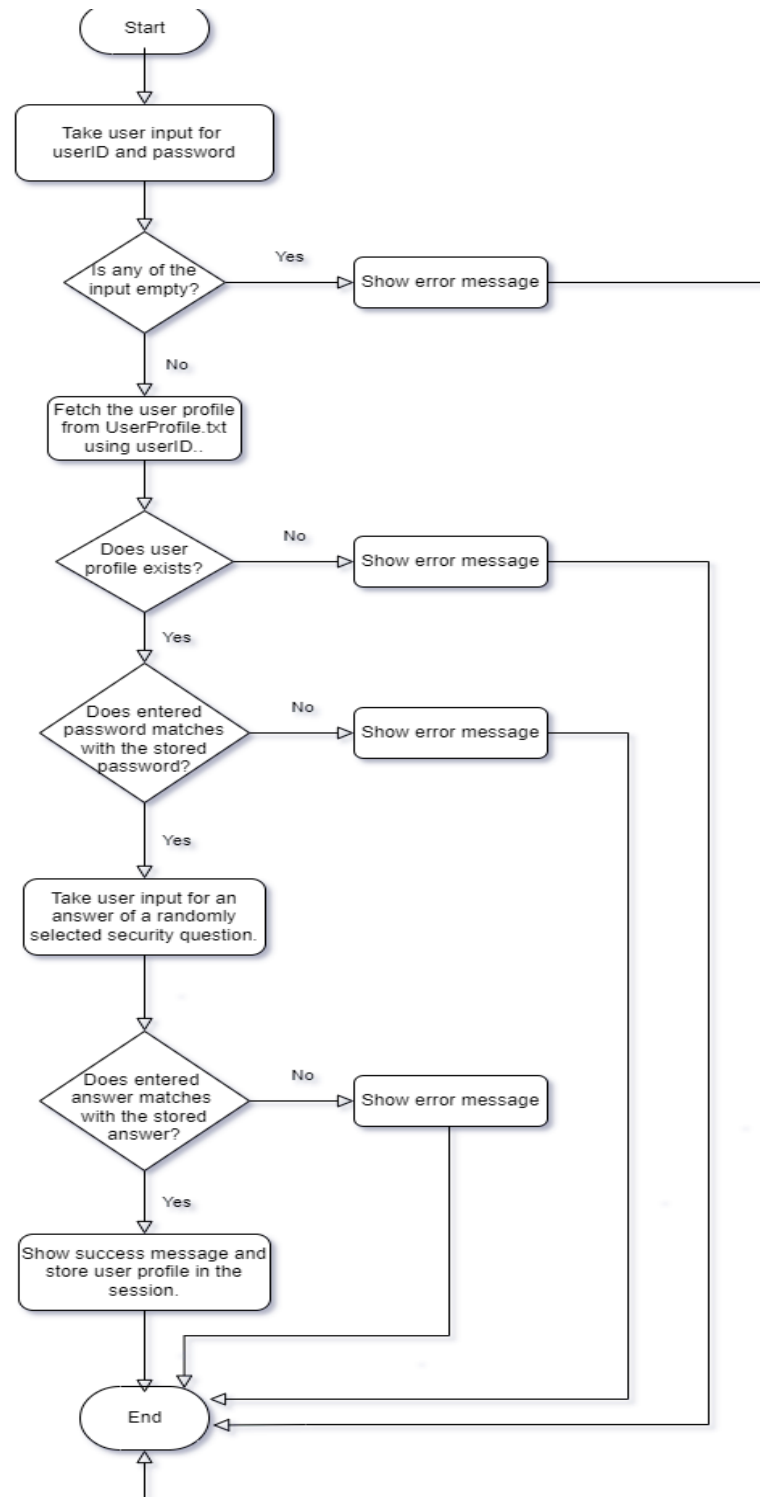


Figure 30: User registration workflow.**Figure 31: User login workflow.**

EXPORT STRUCTURE AND VALUES

Export SQL file:

This is the module where a logged in user can export a specific database into a standard SQL file. The user just needs to provide the name of the database which they want to export.

To generate a SQL file for a specific database, the system needs to read 3 types of files.

1. Global metadata file
2. Table level metadata file
3. Data file of the table

Global metadata file will provide the information about which tables are present under a database and in which server.

Table level metadata will provide the information of columns of the table along with its datatype, primary key, foreign key, and unique key constraints.

Data file of the table will contain the records of a particular table.

Using these 3 types of files, program will be able to generate “Create table” and “Insert into table” SQL queries which are required to generate the final database dump file. **Figure 9** shows the SQL file for Customers database.

```

DATABASE_ROOT > Customers_Dump.sql
1  DROP TABLE IF EXISTS `Walmart`;
2  CREATE TABLE `Walmart` (
3    `id` TEXT,
4    `Name` TEXT,
5    `age` FLOAT,
6    PRIMARY KEY (`Name`)
7  );
8  LOCK TABLES `Walmart` WRITE;
9  INSERT INTO `Walmart` VALUES ('1', 'Jim', 45), ('2', 'Jane', 46), ('3', 'John', 99);
10 UNLOCK TABLES;
11
12
13 DROP TABLE IF EXISTS `Costco`;
14 CREATE TABLE `Costco` (
15   `id` FLOAT,
16   `Name` TEXT,
17   PRIMARY KEY (`id`),
18   UNIQUE KEY `UK_46367eed-127d-4017-bd78-63f3b08886de` (`Name`),
19   KEY `FK2c9c8704-fbe4-4181-b0be-ab04a8c52826` (`Name`),
20   CONSTRAINT `FK2c9c8704-fbe4-4181-b0be-ab04a8c52826` FOREIGN KEY (`Name`) REFERENCES `Walmart` (`Name`)
21 );
22 LOCK TABLES `Costco` WRITE;
23 INSERT INTO `Costco` VALUES (1, 'Jim');
24 UNLOCK TABLES;
25

```

Figure 32: SQL file for Customers database.

Data Modelling – Reverse Engineering

This option allows the user to generate a textual ERD (Entity Relation Diagram) for a given schema. It is available as option 3 under the main driver program,

```
Current dir: /Users/arshtejay/Desktop/csci-5408-w2022-dpg7

1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 3
```

Figure 33: Main menu for the driver after logging in

Once the user selects option 3, they are prompted to enter a schema name as in **Figure 17**.

```
Current dir: /Users/arshtejay/Desktop/csci-5408-w2022-dpg7

1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 3

----- Handle Data model -----
Enter schema name:
Customer
```

Figure 34:13 User input for schema name for which ERD is to be generated

If the schema name is correct and exists, the tool outputs the result and saves it in a text file termed “ERD.txt” in the current directory. The output is also displayed on the screen. The output includes the following:

1. Table names of all the tables in schema
2. Metadata of each table with details including column names, datatypes, size, if the column is a key, and if the column contains unique values
3. Cardinality relationships, if any. The relationships are based on foreign key constraints. Only one-to-one and one-to-many relationships are calculated for now. **Figure 35** shows the final output.

```

1. Write queries
2. Export
3. Data model
4. Analytics
5. Exit

Select an option: 3

----- Handle Data model -----
Enter schema name:
Customers
SCHEMA: Customers
TABLES: Walmart, Cosco

Walmart
columnName, dataType, size, key, unique
id, TEXT, null, PRIMARY KEY, null
name, TEXT, null, null, null

Cosco
columnName, dataType, size, key, unique
id, TEXT, null, PRIMARY KEY, null
name, TEXT, null, Walmart, True

CARDINALITY
Walmart, Cosco, name, one-to-many
|

```

Figure 35: Generated ERD with tables metadata and cardinality

Table Metadata

The table metadata is picked up from the meta files stored locally and remotely. Names of all the tables are taken from GLOBAL metadata along with the instance number they are stored on, remote or local. Then we connect to local/remote based on the location of a particular table and pull LOCAL metadata files for those tables.

Cardinality

Cardinality is calculated based on the following rules:

1. Metadata for all the tables is pulled
2. If there is a foreign key reference in a column in the metadata, there definitely exists a **one-to-many** relationship.
3. Check if the column is unique, if yes, then the relationship is **one-to-one**

SECTION III: MODULE-WISE IMPLEMENTATION

MODULE 1: DB DESIGN

The data structures identified for use in our application were List, LinkedHashMap, and HashMap. Three data structures were used for all in-memory data storage depending on the use case. HashMap was used for O(1) access to values, LinkedHashMap was used to preserve column order in tables and global metadata, and List was used to maintain sequence in table records.

All the persistence storage in the application was stored in tab-separated files. Each file had a header row at the start which facilitated populating the keys for the LinkedHashMap and the HashMap.

id	Name	age
1	Jim	45
2	Jane	46
3	John	99

Figure 36 Example of table

The global metadata file contained three columns of information: **Schema**, **Table**, and **Instance**. The **Instance** field was used to identify which instance a particular table was present in. All schemas were accessible in both instances, irrespective of which schema they were created in.

1	instance	schema	table
2	1	Customers	
3	1	Customers	Walmart
4	1	Customers	Costco
5	1	Customers	Shoppers
6	1	Customers	Shoppers

Figure 37 Example of global metadata file (formatted for clarity)

The local metadata file contained four columns of information for each column in the table: **Column_Name**, **DataType**, **Key**, and **Unique**. For **Foreign key** relations, the **Key** column simply contained the name of the table in which the foreign key existed. For this iteration of our software, we only supported foreign keys with the same column names. The value of **Unique** could be either **true** or **null**.

...

1	Column_Name	Datatype	Key	Unique
2	id	TEXT	PRIMARY KEY	null
3	Name	TEXT	null	null
4	age	FLOAT	null	null

Figure 38 Example of table-level metadata file (formatted for clarity)

On instantiation of the program in one instance, the program first checks for a **DATABASE_ROOT** folder and the remaining folder structure. If the correct folder structure exists on both instances, nothing is done. If the structure doesn't exist, it is created.

```

imaphong12@vm-1:~/csci-5408-w2022-dpg7/target$ java -jar DPG7.jar
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Created DATABASE_ROOT directory successfully
Created SCHEMAS directory successfully
Created METADATA directory successfully
Created global.txt file successfully
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Created USER_PROFILE.txt file successfully
Created LOGS directory successfully
Created QUERY_LOG.txt file successfully
Created EVENT_LOG.txt file successfully
Created GENERAL_LOG.txt file successfully

```

Figure 39 Creation of folder structure for local and remote instances

MODULE 2: QUERY IMPLEMENTATION

Every query present in the requirements for this module has been implemented. The implementation of this module has two broad steps, with multiple sub-steps in between. The first step is the query parser, which was tested extensively to handle various edge cases. The details for these test cases have been provided in **Section I**. Following the parsing, the execution was done following a three-step approach: 1. Classify query, 2. Validate query with global and table-level metadata and 3. Execute query.

QUERY PARSING

We implemented three classes Query, ParserConstants, Parser and an InvalidQueryException class. When a parser parses a query, it returns an object of type 'Query', ParserConstants contains all pre-defined values and

states of transition for the parser. Parser class contains all methods and attributes needed to parse an input Query.

In order to use the parser, we first create an object of Parser with a constructor that takes in the string input of the query. Then call the method *Parser.parse()* which returns a parsed query object or throws the exception *InvalidQueryException* with an appropriate message, in case of a syntax error.

The parser is inspired by Mariano Gappa's blogpost [1]. Find the description of classes and interfaces used in the following pages.

Table 7 The Data Structures Used For Query Parsing

Class/Interface	Description
Parser	Class to model a parser which parses a given query
Query	Class to represent a particular query after tokenisation and parsing
ParserConstants	Final class containing all the constants and states to be used by the Parser class
InvalidQueryException	An exception is thrown by the Parser in case of a syntax error

Parser

Class to model a parser that parses a given query. Has the data members of a processed SQL query, an index pointing to the start of the current token, the current state of the parser, an unformatted SQL query testing required, a string to indicate the next field name to be updated in case of an update query, and a timestamp.

Table 8 Attributes of Parser class

Variable	Usage
private String SQL	The query to be parsed, pre-processed to remove trailing semi-colons
private String orgSql	Original query input by the user
private int currPos	The current position in query
private Query query	The query object after parsing is returned by the parse() method

private ParserConstants.ParsingStep step	Current parsing step/state, say, selectTable or selectField
private String nextUpdateField	A string to indicate the next field name to be updated in case of an update query
private Instant startTimestamp	Timestamp when the parser started parsing, to be used in logs

Apart from some getters and setters of the above-mentioned variables, the following additional methods are defined:

Table 9 Methods of Parser class

Method	Usage
public Query parse() throws InvalidQueryException	Takes no input parameters, does the actual parsing of query and returns an object of type Query. If there's a syntactical error, it throws an InvalidQueryException
public String peek()	This method uses the currPos index and returns the next token to parse. If none is available, it returns an empty string. It doesn't update the value of the index currPos. It looks for the token in RESERVED_PHRASES, VALID_CONSTRAINTS, and DATA_TYPES defined in ParserConstants. If the token is not found in any of the above, it looks for a quoted string and parses the token using a regex that accepts only alphanumeric words.
public void invokeQueryLog(String tableName, String queryValidity, String queryType)	This method takes in the table name, query validity and query type and logs these values in the query log, along with the timestamp, and whether the execution was successful or not.
public String pop()	This method internally calls the peek() method and gets a valid token. It then increments the currPos index by a value equal to the length of the extracted token
public void popWhiteSpace()	This method skips any spaces inside a query and moves the index currPos to the next available non-whitespace character
private boolean isReserved(String token)	Tells if the current token is a reserved word and returns a Boolean value
private boolean isDataType(String token)	Tells if the current token is a valid datatype
private boolean isConstraint(String token)	Tells if the current token is a valid constraint
private boolean isAnIdentifier(String token)	Tells if the current token is an identifier. An identifier is an alpha-numeric string
private boolean isOperatorValid(String operator)	Tells if the operator in a where the condition is valid

private void validateQuery()	Validate the final generated query object. It contains any additional validations after a query has been parsed
-------------------------------------	---

Query

Class to model a query as returned by the parser. Has the data members type, tableName, databaseName, fields, valuesToUpdate, conditions, inserts, datatypes, constraints and useQuery.

Table 10 Attributes of Query Class

Variable	Usage
private String type	Represents type of query like select, update, insert, delete
private String tableName	Represents the corresponding table
private String databaseName	Represent the database
private List<String> fields	The field names to perform the insert operation on
private Map<String, String> valuesToUpdate	Field names with corresponding update values
private List<List<String>> conditions	A list of all the conditions in where. Each entry is of type {Operand1, Operator, Operand2}
private List<List<String>> inserts	A list of values to be inserted into fields

Query class contains only getters and setters for the respective data members and an overridden toString() method.

Table 31 Methods of Query Class

Method	Usage
public String getType()	Getter for type of query
public void setType(String type)	Setter for type of query
Public String getTableName()	Getter for table name
public void setTableName(String type)	Setter for table name

public List<String> getFields()	Getter for fields
public void setFields(List<String> fields)	Setter for fields
public Map<String, String> getValuesToUpdate()	Getter for update values map
public void setValuesToUpdate(Map<String, String> valuesToUpdate)	Setter for values to update
public List<List<String>> getConditions()	Getter for conditions specified
public void setConditions(List<List<String>> conditions)	Setter for conditions specified
public List<List<String>> getInserts()	Getter for insert values
public void setInserts(List<List<String>> inserts)	Setter for insert values
public String getDatabaseName()	Getter for database name
public void setDatabaseName(String databaseName)	Setter for database name
public List<String> getDatatypes()	Getter for datatypes
public void setDatatypes(List<String> datatypes)	Setter for datatypes
public List<String> getConstraints()	Getter for constraints
public void setConstraints(List<String> constraints)	Setter for constraints
public String getUserQuery()	Get the original string query of the user
public void setUserQuery(String userQuery)	Set the original string query of the user
public String toString()	Overridden toString method to output all attributes

ParserConstants

It is a final class that defines all transition states while parsing, reserved keywords, valid operators, datatypes and valid constraints.

Table 42 Attributes of ParserConstants class

Variable	Usage
static final String[] RESERVED_PHRASES	All the reserved words used in SQL supported by the parser
static final String[] VALID_OPERATORS	All the operators supported. This list includes "=", "<", ">", ">=", "<=", "!="
static final String[] DATA_TYPES	Represent the currently supported data types. The parser currently allows only TEXT and FLOAT datatypes
static final String[] VALID_CONSTRAINTS	Represents all the constraints currently supported. PRIMARY_KEY and UNIQUE keywords. The parser also supports foreign key but it requires only the table name of the referred column.
public enum ParsingStep	Enum to describe the current state of the parser

The class doesn't have any methods defined. The RESERVED_PHRASES includes the following list:

```
"CREATE", "USE", "SELECT", "UPDATE", "DELETE FROM", "FROM", "INSERT INTO", "SET", "VALUES", "WHERE",
"BEGIN TRANSACTION", "END TRANSACTION", "COMMIT", "ROLLBACK", "=", "<", ">", ">=", "<=", "(", ")",
"TABLE"
```

Figure 40: Reserved words

InvalidQueryException

InvalidQueryException extends the RuntimeException and takes in a string message. Whenever the parser encounters an invalid syntax or an unexpected token, it throws the InvalidQueryException.

QUERY VALIDATION

For query validation, one class named **MetadataValidation** has been used. A brief overview of all the important functions in this class is provided below:

Table 53 Methods of Query Validation

Method	Usage
validateSchema(String schemaName)	Used to validate existence of schema using global metadata
getInstanceNumber(String tableName, String schemaName)	Get instance identifier for a particular table

<code>validateTable(String tableName, String schemaName)</code>	Validate existence of table in a schema
<code>validateSingleColumn(String columnName, String columnInsertValue, LinkedHashMap<String, HashMap<String, String>> meta, Table table)</code>	Validate data type of a single column for an insertion value
<code>validateColumns(List<String> columns, LinkedHashMap<String, HashMap<String, String>> localMeta)</code>	Used to validate existence of columns and their ordering
<code>validateInsertDataTypes(List<List<String>> inserts, LinkedHashMap<String, HashMap<String, String>> localMeta, List<String> columns)</code>	Validate all insert data types
<code>validateUniqueValueInColumn(String columnName, String value, String tableName, String schemaName)</code>	Validate unique constraint
<code>validateUniqueInsert(Query query, LinkedHashMap<String, HashMap<String, String>> localMeta, String schemaName)</code>	Validate insert query for unique constraints
<code>validateUniqueUpdate(Query query, LinkedHashMap<String, HashMap<String, String>> localMeta, String schemaName)</code>	Validate update query for unique constraints
<code>validateForeignKeys(Query query, LinkedHashMap<String, HashMap<String, String>> localMeta, String schemaName)</code>	Validate foreign key constraint for a query
<code>validateInsertQuery(Query insertQuery, String schemaName)</code>	All validations for an insert query
<code>validateSelectQuery(Query selectQuery, String schemaName)</code>	All validations for select query
<code>validateUpdateDataTypes(Map<String, String> valuesToUpdate, LinkedHashMap<String, HashMap<String, String>> localMeta)</code>	Validate data types for update query
<code>validateConditionDataType(Query query, LinkedHashMap<String, HashMap<String, String>> localMeta)</code>	Validate data types for where condition
<code>validateUpdateQuery(Query updateQuery, String schemaName)</code>	All validations for update query
<code>validateDeleteQuery(Query deleteQuery, String schemaName)</code>	All validations for delete query

QUERY EXECUTION

After parsing and validation, queries are executed according to the project requirements. Individual functions for execution driver are not provided in a table since they are very straightforward. Instead, screenshots of all required queries are shown below.

```
CREATE DATABASE Customers
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
[INFO] Inserting to Global Meta
```

Figure 41: Successful create query

...

```
USE Customers
REMOTE STATUS: false
LOCAL
FILENAMEEEEE: METADATA
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Using database Customers
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
```

Figure 42 Successful use query

```
CREATE TABLE Walmart (id TEXT PRIMARY KEY,name TEXT, age FLOAT)
REMOTE STATUS: false
LOCAL
Created Walmart.txt file successfully
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
[INFO] Inserting to Global Meta
TABLE META FILE PATH WOWIE/home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/METADATA/Walmart.txt
Created Walmart.txt file successfully
File path: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/METADATA/Walmart.txt
[INFO] INSERTING TO METAFILE
WRITE META ROWS RECORD: (Column_Name=id, Datatype=TEXT, Key=PRIMARY KEY, Unique=null)
FINAL ADDING STRING: id TEXT PRIMARY KEY null
WRITE META ROWS RECORD: (Column_Name=name, Datatype=TEXT, Key=null, Unique=null)
FINAL ADDING STRING: name TEXT null null
WRITE META ROWS RECORD: (Column_Name=age, Datatype=FLOAT, Key=null, Unique=null)
FINAL ADDING STRING: age FLOAT null null
[INFO] SUCCESSFULLY INSERTED IN THE META FILE
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-0
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-0
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
```

Figure 43 Successful create table query

```
INSERT INTO Walmart (id,name,age) values (1,John,45)
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Columns validated
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-0
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-0
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Inserted successfully
```

Figure 44: Successful insert query

```
SELECT * FROM Walmart WHERE age<50
{id=1, Name=Jim, age=45}
{id=2, Name=Jane, age=46}
```

Figure 14: Successful select query

...

```

UPDATE Cosco set name=Jane WHERE name=John
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-3
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-3
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Updated successfully

```

Figure 15: Successful update query

```

DELETE FROM Walmart WHERE id=2
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-2
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Cosco.txt
File: /home/imaphong12/csci-5408-w2022-dpg7/target/DATABASE_ROOT/SCHEMAS/Customers/Walmart.txt
Schema: Customers-2
Current dir: /home/imaphong12/csci-5408-w2022-dpg7/target
Deleted successfully

```

Figure 16: Successful delete query

MODULE 4: LOG MANAGEMENT IMPLEMENTATION

Table 64: Log Management Implementation Details

Method	Usage
private QueryLogManager() / private EventLogManager() / private GeneralLogManager()	This is the private constructor. It serves as a point-of-entry for the Log Manager since this is a Singleton class.
public static QueryLogManager getInstance() / public static EventLogManager getInstance() / public static GeneralLogManager getInstance()	This function is used to retrieve the single instance of the Log Manager.
public Map<String,String> constructLogEntry(QueryLog querylog, String instanceName)	This method is used to input the fields required in the corresponding Log.
public void writeLogTable()	This method is used to write the Log Entries to their corresponding files – QUERY_LOG.txt / EVENT_LOG.txt / GENERAL_LOG.txt
public void processLog(QueryLog log)	This method is a wrapper function that contains contains all 3 functions necessary to generate a log entry: constructLogEntry(),formatLogEntry() and writeLogTable().
public static String formatLogEntry()	This method is used to transform log entries into rows of log entries by converting the variables to DELIMITER-separated strings, where DELIMITER refers to the string chosen as the Delimiter in our system – “\t”.
public integer getNumTables()	This method is used to retrieve the number of tables in each schema.
public integer getNumRows()	This method is used to retrieve the cumulative number of rows in each schema.

MODULE 5: DATA MODELLING – REVERSE ENGINEERING IMPLEMENTATION

We implemented one class ERDiagram which contains a public static method generateERD() and other private helper functions. There are no data members. Since the method generateERD() is static, the driver program doesn't create any instance of the class, but directly calls it whenever user selects to generate an ERD.

Find the description of classes and interfaces used, and data members and methods of the class in the following pages.

Table 75: Methods for Data Modelling

Method	Usage
public static String generateERD(String schemaName) throws IllegalAccessException	The main function called by the Driver to generate an ERD. It takes in the schemaName as the input. Validates the name of schema. If invalid, it returns an empty string and no data is written to disk.

	<p>If there's an access issue with schema the method raises an <code>IllegalAccessError</code> with appropriate message.</p> <p>If valid, it reads the GLOBAL metadata to get information about the schema and all the tables, where those tables are stored, locally or remotely. Then fetches metadata of all the tables.</p> <p>It then calls the following two functions:</p> <p><code>stringifyTableMetadata()</code> to format the metadata of a table and append to output</p> <p><code>getTableCardinalities()</code> to calculate relationships and cardinalities.</p> <p><code>writeErdToFile()</code> is called in the end to write the output to a file named <code>ERD.txt</code> in the project root directory.</p>
<pre>private static String stringifyTableMetadata(LinkedHashMap<String, HashMap<String, String>> meta)</pre>	<p>It's a private function used by the class to format table metadata and return a single string containing the output. It takes in metadata of a particular table as an input. The metadata is a linked hash map, containing mapping from column names to corresponding attributes of metadata like datatype, size, etc</p>
<pre>private static String getTableCardinalities(LinkedHashMap<String, HashMap<String, String>> meta, String tableName)</pre>	<p>It's a private function used by the class to generate cardinality table. It takes in input as metadata and current table name. Cardinalities are generated based on the following rules:</p> <ol style="list-style-type: none"> 1. If there is a foreign key reference in a column in the metadata, there definitely exists a one-to-many relationship. 2. Check if the column is unique, if yes, then the relationship is one-to-one

MODULE 6: EXPORT STRUCTURE AND VALUES IMPLEMENTATION

Table 86: Attributes for Exporting

Variable	Usage
String schemaName	Used to store the name of database that is to be exported.
List<LinkedHashMap<String, String>> globalMeta;	Used to store the content of global metadata file.
HashMap<String, LinkedHashMap<String, HashMap<String, String>>> tableMetaHashMap;	Used to store the metadata of each table present in the database
HashMap<String, List<String>> tableDataHashMap;	Used to store the data/records of each table present in the database
List<String> primaryKeyColumnList;	Used to store primary key related columns
List<String> uniqueKeyColumnList;	Used to store unique key related columns
List<String> foreignKeyColumnList;	Used to store foreign key related columns

Table 97: Methods for Exporting with explanation

Method	Usage
public ExportDatabase()	This is the constructor. It initializes several class level variables like globalMeta, primaryKeyColumnList, uniqueKeyColumnList, foreignKeyColumnList, tableMetaHashMap, and tableDataHashMap.
public void handleExportDB()	This is the entry point of the module. This function will be called from the main Driver program. This function internally calls various private methods which are required to generate the SQL file.
private Boolean handleUserInputs()	This method is responsible to get the database name from user and validate if the database exists. It returns true if database exists and false if database doesn't exist.
private void loadMetaAndDataForAllTables(List<LinkedHashMap<String, String>> filteredGlobalMeta)	This method is used to load the table level metadata and the actual data files of the tables which are in the database. This method receives globalMeta as the parameter.
private String handleCreateQuery(String tableName)	This method is used to generate, and return CREATE table SQL query. It receives tableName as a parameter.
private String handleColumns(String tableName)	This method is used to generate and return the column names along with its datatypes. The output of this method is used by handleCreateQuery() method. This method is also responsible to store the primary, foreign, and unique key constraints in

	primaryKeyColumnList, foreignKeyColumnList, and uniqueKeyColumnList respectively (If constraints exists).
private String handlePrimaryKey()	This method is used to generate and return the SQL syntax of PRIMARY KEY
private String handleUniqueKey()	This method is used to generate and return the SQL syntax of UNIQUE KEY
private String handleForeignKey()	This method is used to generate and return the SQL syntax of FOREIGN KEY
private String handleInsertQuery(String tableName)	This method is used to generate and return the INSERT query for the table. It receives tableName as a parameter
private void generateSQLFile(String sqlContent)	This method generates the actual SQL file.

MODULE 7: ANALYTICS IMPLEMENTATION

Analytics is a separate package in the project which focuses on generating two types of report. Each methods used in the file is described in the table.

Table 108: Methods for Analytics with explanation

Method	Usage
Public void main()	Takes no input parameters and is the entry point for execution of analytics program
public static void getUserAnalytics(String fileName)	This method has filename as input. The file name would be of the query log for instance 1 and 2. Since analysis would be done for both the instances, therefore we read both the query logs
public static HashMap<String, List<String>> populateDataFromLog(String fileName)	This method takes in the query log file name. As the filename and path for both the instances is similar, it would avoid any errors in the program. The purpose of this method is to read all the logs and store it in a hashmap. The first key would be the instance name followed by all the logs for that particular instance as the value.
public static List<String> getAllLogs(String filePath)	This method will read the logs from both the instances and store in memory to avoid multiple I/O operations by storing it in a List of string datatype.
public static Set<String> getAllTablesByDB(String databaseName, String queryType, HashMap<String, List<String>> instanceRecordsMap)	This method will store all the tables in a Hashset datatype from both the instances. The reason to store the tables in a hashset is to avoid storing duplicate table names. This is a helper method that would retrieve all the tables.
public static Set<String> getAllDatabaseNames (HashMap<String, List<String>> instanceRecordsMap)	This method gets all the database names from any one of the query logs as both the instance will hold the same databases. So we retrieve the records from the hashmap for one of the instance and store all the database name in a Set datatype to avoid storing duplicate database names. The reason to

	avoid duplicity is because the database names are read from the query log file which could contain multiples entries for a given database.
private static List<String> filterRecordsByDbQuery(String databaseName, String queryType, HashMap<String, List<String>> instanceRecordsMap)	This method takes input parameter such as the database name, the query type like (select,update,create) from the user and a hashmap of logs from both the instances. It is used for filtering records by first fetching all the logs for a given database entered by the user and then the filtered records go through the next level of filter by type of query. Based on these two filters, a count is done that would be used for generating report.
private static void generateOutput(String result, String analyticsFile)	This method is a common method used for both type of analytics. It is used for creating output files and dump all the results to it. The parameter is result, analytics file
public static void getQueryAnalytics(String fileName, String operation, String databaseName)	This method is the entry point when the user selects the second option. It is responsible for generating the report based on the operation entered by the user.

MODULE 8: USER INTERFACE

The user interface of application is handled by 2 functions in the Driver class of main package.

1. public static void showRegisterLoginMenu()
2. public static void showMainMenu()

The Main menu will be only displayed to a user if he logs in the application using his UserID and password.

SECTION IV: TESTING

Some parts of the program were determined to be important for testing, and these were tested extensively to make sure as many edge cases as possible have been satisfied. The project has a **total of 88 test cases**, with most of them in the query parsing and execution section.

QUERY PARSER TESTS

Each type of query has a set of tests associated with it. All these tests are provided in the screenshots in this section. Each test has a self-explanatory test name.

```
20
21  @Test
22  > void emptyQuery() { ...
33
34  @Test
35  > void nothingAfterCreateKeyword() { ...
46
47  @Test
48  > void nothingAfterDatabaseKeyword() { ...
59
60  @Test
61  > void nothingAfterTableKeyword() { ...
72
73  @Test
74  > void validCreateDatabaseQuery() { ...
87
88
89  @Test
90  > void nothingAfterTableName() { ...
101
102  @Test
103  > void nothingAfterOpeningParenthesis() { ...
114
115  @Test
116  > void noDataTypeSpecified() { ...
127
128  @Test
129  > void noClosingParenth() { ...
140
141  @Test
142  > void closingParenthAfterComma() { ...
153
154  @Test
155  > void validCreateTableQueryWithOneColumn() { ...
168
169  @Test
170  > void validCreateTableQueryWithTwoColumn() { ...
183
184  @Test
185  > void invalidQueryWithExtraCharAtEnd() { ...
196
197  @Test
198  > void fieldNameIsAReservedWord() { ...
209
```

Figure 17: Create query tests (i)

```

210  @Test
211  > void tableNameIsReservedWord() { ...
222
223  // Tests for foreign key constrain
224  @Test
225  > void validCreateTableQueryWithOneColumn_FK() { ...
238
239  @Test
240  > void validCreateTableQueryWithMultipleColumns_FK() { ...
253
254  // Tests for foreign key constrain
255  @Test
256  > void validCreateTableQueryWithOneColumn_FK_Unique() { ...
269
270  @Test
271  > void validCreateTableQueryWithMultipleColumns_FK_Unique() { ...
284 }

```

Figure 49: Create query tests (ii)

```

13  @Test
14  > void emptyQuery() { ...
25
26  @Test
27  > void nothingAfterDeleteKeyword() { ...
38
39  @Test
40  > void validQueryWithNowhere() { ...
53
54  @Test
55  > void noConditionsAfterWhereClause() { ...
66
67  @Test
68  > void conditionPartiallySpecified() { ...
79
80  @Test
81  > void invalidOperatorInCondition() { ...
92
93  @Test
94  > void tableNameIsReservedKeyword() { ...
105
106
107  @Test
108  > void validQueryWithOneCondition() { ...
121
122  @Test
123  > void validQueryWithMultipleConditions() { ...
136
137
138
139  @Test
140  > void invalidQueryWithExtraCharAtEnd() { ...
151
152  @Test
153  > void fieldNameIsReservedWord() { ...
164
165  @Test
166  > void tableNameIsReservedWord() { ...
177 }

```

Figure 50: Delete query tests

```

21  @Test
22  > void emptyQuery() { ...
33
34  @Test
35  > void nothingAfterInsertIntoKeyword() { ...
46
47  @Test
48  > void noRowsSpecified() { ...
59
60  @Test
61  > void noColumnsSpecified() { ...
72
73  @Test
74  > void noRowsAfterColumns() { ...
85
86  @Test
87  > void InsertQueryUnitTests.emptyParenthAfterValuesKeyword()
98  emptyParenthAfterValuesKeyword() (Not yet run).
109 > void emptyParenthAfterValuesKeyword() { ...
111
112  @Test
113 > void noCommaBetweenFieldNames() { ...
124
125  @Test
126 > void columnParenthMissing() { ...
137
138  @Test
139  > void validQueryWithOneRow() { ...
153
154  @Test
155 > void validQueryWithTwoRows() { ...
168
169  @Test
170 > void invalidQueryWithExtraComma() { ...
181
182  @Test
183 > void invalidQueryWithExtraCharAtEnd() { ...
194
195  @Test
196 > void fieldNameIsAReservedWord() { ...
207
208  @Test
209 > void tableNameIsAReservedWord() { ...
220
}

```

Figure 18: Insert query tests

```

21  @Test
22  > void emptyQuery() { ...
33
34  @Test
35  > void nothingAfterSelectKeyword() { ...
46
47  @Test
48  > void nothingAfterFieldNames() { ...
59
60  @Test
61  > void noCommaBetweenFieldNames() { ...
72
73  @Test
74  > void noTableNameAfterFromKeyword() { ...
85
86  @Test
87  > void validQueryWithAsteriskAndSemicolon() { ...
100
101  @Test
102 > void validQueryWithAsteriskAndNoSemicolon() { ...
115
116  @Test
117 > void validQueryWithOneFieldAndSemicolon() { ...
131
132  @Test
133 > void validQueryWithOneFieldAndNoSemicolon() { ...
147
148  @Test
149 > void validQueryWithMultipleFieldsAndSemicolon() { ...
163
164  @Test
165 > void validQueryWithMultipleFieldsAndNoSemicolon() { ...
179
180  @Test
181 > void validQueryWithExtraComma() { ...
192
193  @Test
194 > void FieldNameIsAReservedWord() { ...
205
206  @Test
207 > void tableNameIsAReservedWord() { ...
218
}

```

Figure 19: Select query tests

```

14  @Test
15  > void emptyQuery() { ...
26
27  @Test
28  > void nothingAfterBegin() { ...
39
40  @Test
41  > void nothingAfterEnd() { ...
52
53  @Test
54  > void validBeginTransaction() { ...
67
68  > void validEndTransaction() { ...
81
82  > void validCommitTransaction() { ...
95
96  > void validRollbackTransaction() { ...
109
110 @Test
111 > void invalidCharAfterBegin() { ...
122
123 @Test
124 > void invalidCharAfterEnd() { ...
135
136 @Test
137 > void invalidTokenAfterCommit() { ...
148
149 @Test
150 > void invalidTokenAfterRollback() { ...
161
162 }

```

Figure 53: Transaction query tests

```

13  @Test
14  > void emptyQuery() { ...
25
26  @Test
27  > void nothingAfterUpdateKeyword() { ...
38
39  @Test
40  > void noSetKeyword() { ...
51
52  @Test
53  > void noColumnsToSet() { ...
64
65  @Test
66  > void noEqualToSign() { ...
77
78  @Test
79  > void noValueToSet() { ...
90
91  @Test
92  > void noFieldAfterComma() { ...
103
104  @Test
105 > void noConditionAfterWhere() { ...
116
117  @Test
118 > void noConditionsAfterWhereClause() { ...
129
130  @Test
131 > void conditionPartiallySpecified() { ...
142
143  @Test
144 > void invalidOperatorInCondition() { ...
155
156  @Test
157 > void tableNameIsAReservedKeyword() { ...
168
169  @Test
170 > void fieldNameIsAReservedKeyword() { ...
181
182  @Test
183  > void validQueryWithOneCondition() { ...
197
198  @Test
199 > void validQueryWithMultipleConditions() { ...
212

```

Figure 20: Update query tests (i)

...

```

214   @Test
215   > void validQueryWithMultipleUpdateFields() { ...
228
229   @Test
230   > void invalidQueryWithExtraCharAtEnd() { ...
241 }
242

```

Figure 21: Update query tests (ii)

```

14   @Test
15   > void emptyQuery() { ...
26
27   @Test
28   > void nothingAfterUseKeyword() { ...
39
40   @Test
41   > void validUseQuery() { ...
54

```

Figure 22: Use query tests

In addition to query tests, constraint testing was also performed in the remote instances.

```

INSERT INTO Walmart (id,name,age) values (1,Jane,34)
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Columns validated
REMOTE STATUS: false
LOCAL
Unique key constraint failed

```

Figure 57: Failure of unique key constraint

...

```
INSERT INTO Cosco (id,name,age) values (1,Jim,24)
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
Columns validated
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
REMOTE STATUS: false
LOCAL
FK Constraint failed!
```

Figure 58: Failure of foreign key constraint

REFERENCES

- [1] "Database Design - Subclasses," *web.csulb.edu*. [Online]. Available: <https://web.csulb.edu/colleges/coe/cecs/dbdesign/dbdesign.php?page=subclass.php>. [Accessed: Mar. 08, 2022]
- [2] Draw.io, "Flowchart Maker & Online Diagram Software," *app.diagrams.net*, 2021. [Online]. Available: <https://app.diagrams.net/>. [Accessed: Feb. 17, 2022]
- [3] Archiveddocs, "Table and Index Organization," *docs.microsoft.com*. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189051\(v=sql.105\)](https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189051(v=sql.105)). [Accessed: Mar. 16, 2022]
- [4] "MySQL :: MySQL 5.6 Reference Manual :: 21.2 INFORMATION_SCHEMA Table Reference," *dev.mysql.com*. [Online]. Available: <https://dev.mysql.com/doc/refman/5.6/en/information-schema-table-reference.html>. [Accessed: Mar. 21, 2022]
- [5] "sql - How to determine programmatically MySQL relationship type (1:1, 1:n, n:m) between tables?," *Stack Overflow*. [Online]. Available: <https://stackoverflow.com/questions/26103542/how-to-determine-programmatically-mysql-relationship-type-11-1n-nm-betwee>. [Accessed: Mar. 08, 2022]
- [6] V. Singh, "Two Phase Locking (2PL)," *Medium*, Mar. 02, 2019. [Online]. Available: https://medium.com/@vikas.singh_67409/two-phase-locking-2pl-696c49a66a79. [Accessed: Mar. 14, 2022]
- [7] "Instant isBefore() method in Java," *www.tutorialspoint.com*. [Online]. Available: <https://www.tutorialspoint.com/instant-isbefore-method-in-java>. [Accessed: Mar. 24, 2022]
- [8] M. Gappa, "Let's build a SQL parser in go!," *Mariano Gappa's Blog*. [Online]. Available: <https://marianogappa.github.io/software/2019/06/05/lets-build-a-sql-parser-in-go/>. [Accessed: 14-Apr-2022].