DALHOUSIE UNIVERSITY

CSCI 5408 – Data Management, Warehousing, Analytics

Project Report

D2_DB

Group 12

Guturu Rama Mohan Vishnu (B00871849)

Sharad Kumar (B00889585)

Kalpit Machhi (B00911364)

Kavya Kasaraneni (B0088124)

Kishan Mahendrabhai Savaliya (B00896729)

Table of Contents

Project Objective

The whole purpose of the project is to design a Distributed Database Management system that operates on two Linux virtual machines. It should provide a command-line interface on both instances and should be able to handle two different requests from different users. This database management system should be able to create the database and manage it in a secure way. It also should be able to store the data, retrieve it, generate ERD, Log generation, metadata managing, SQL dump, perform analytics, etc.

Overview

Our project follows a centralized data structure that has a command-line user interface together with backend DBMS. There are 3 parts of the project i.e., User-interface, DBMS, and distributed DBMS of which we have achieved user-interface and DBMS. This project accepts all the queries that are given by the user as input and checks if they are syntactically and semantically correct through the parser. Later, if there are any invalid queries entered it then displays the appropriate message to the user, and if there are valid queries provided by the user it then implements them. Using these queries, the DBMS performs SELECT, INSERT, UPDATE, CREATE and DELETE operations on the database successfully. The data structures that are used are Lists, Maps, and basic data structures like Integer, String, etc.

The command-line interface that is developed is very user-friendly and offers users multiple options. Upon users' selection of the option, it executes the different modules in the backend and provides the user with the expected output.

Distributed implementation is not merged into the code successfully, but we have mentioned a section at the end of this document on how we are planning to implement it.

The brief process of how our database operates and how the system looks can be found in the flowchart below.

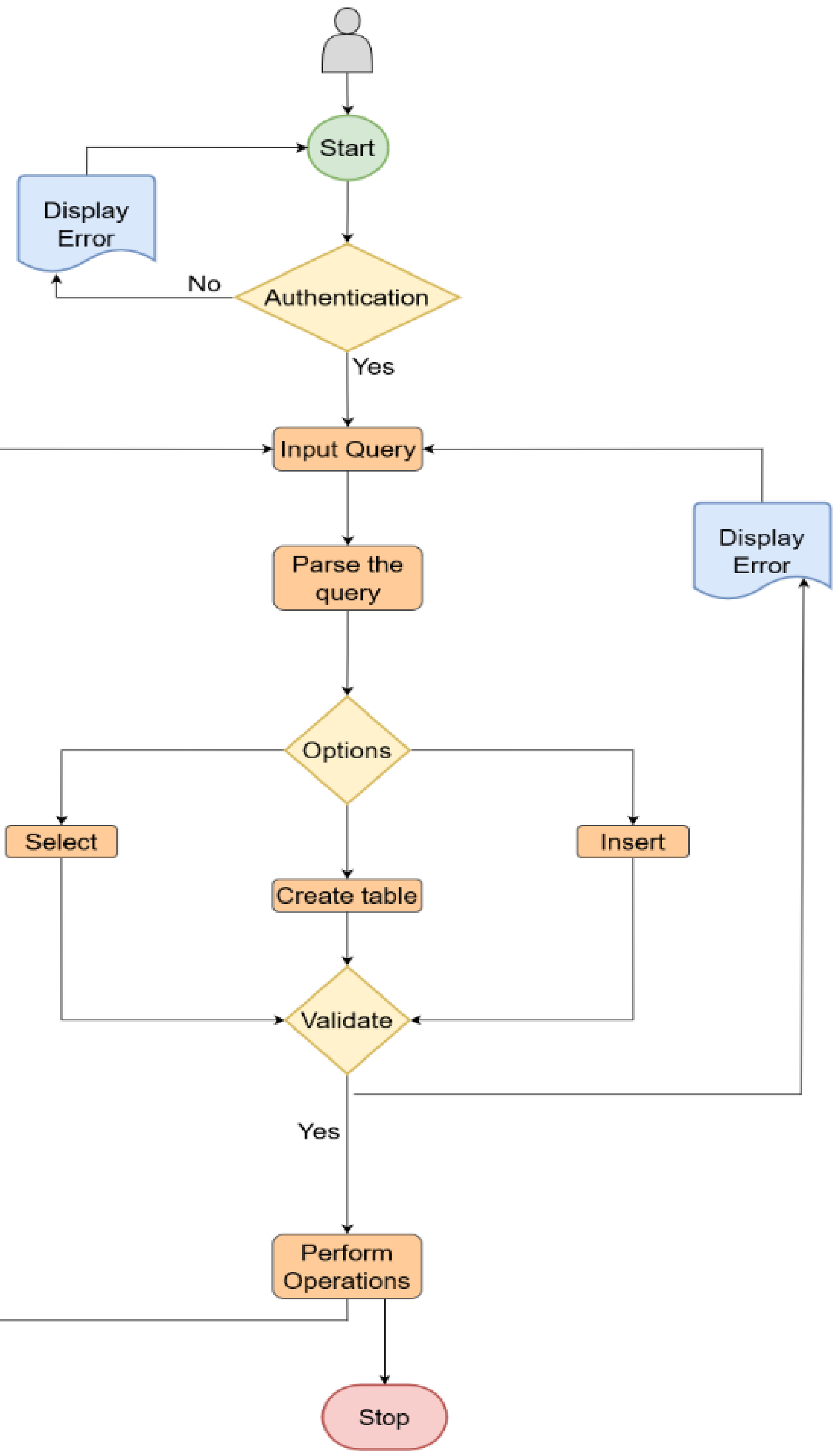


Figure 0: Basic process of the project

Team Strength

- ✓ Responsive members that are ready to pick up any task
- ✓ Ability to solve problems and bugs together
- ✓ Organized work along with contributing a fair share
- ✓ Offer each other support
- ✓ Focused more on the goals and striving for good results
- ✓ Well responsive to all the issues faced during project building
- ✓ Encourage differences in opinions by understanding each other's strengths and weaknesses.

Individual Contribution

Every member of our team has contributed equally and given their best efforts to this project. To know the exact module, which was performed by which team member, here are the details!

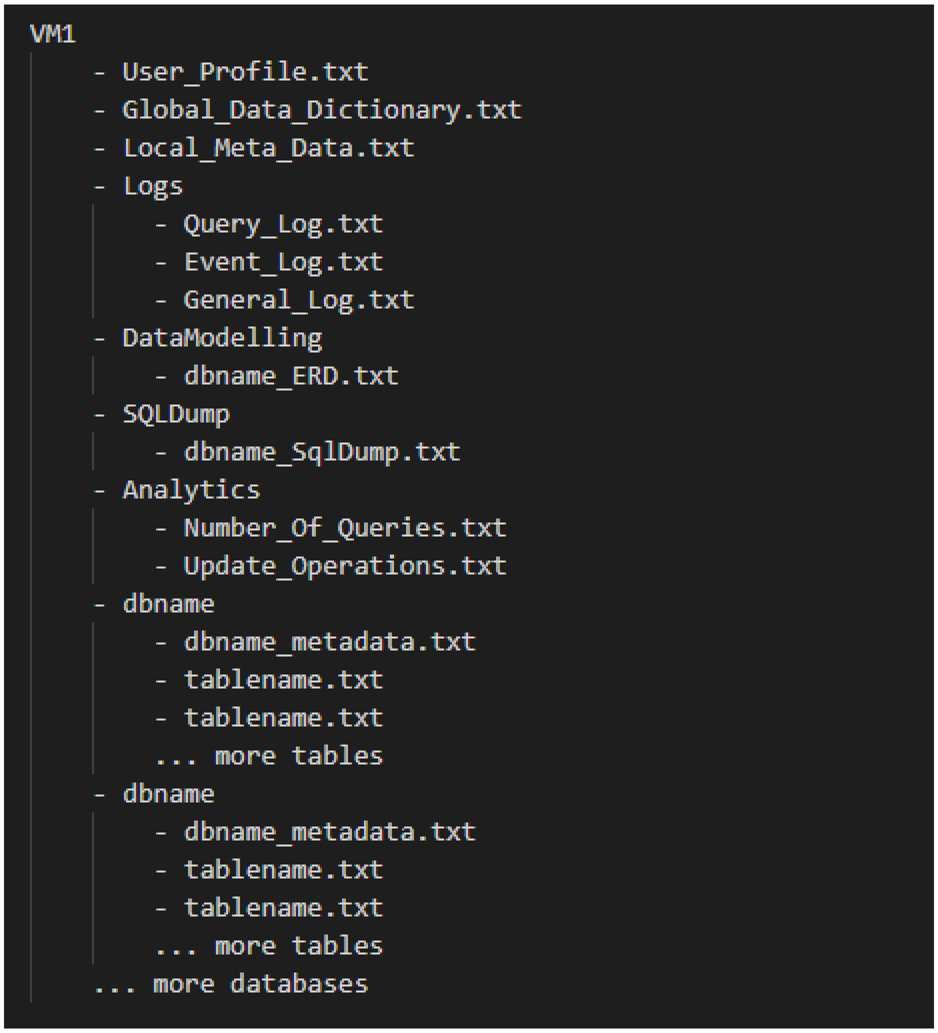
Member	Contribution
Rama Mohan Vishnu Guturu	User Interface & Login, Analytics
Kalpit Machhi	Query Implementation
Kishan Savaliya	Transaction Processing
Kavya Kasaraneni	Log Management, Data Modelling
Sharad Kumar	Export Structure & Value, Connection between two VM's

Table 0: Member Contribution

<u>Design and Implementation Details</u>

Module 1: DB Design

We have designed our project with a particular structure, and we have followed the same structure throughout. Whenever the program starts running for the first time, it will create a folder with the name “VM1” or “VM2” based on which Virtual Machine it is, in the root project directory itself. If that folder already exists in this location, the code will not create it again or will not overwrite it. When the creation of the folder is successful, it creates 3 basic and important files named “Global_Data_Dictionary.txt”, “User_Profile.txt”, and “Local_Meta_Data.txt” inside the VM1 or VM2 folder. After these three files are created, it creates the required folders for their respective modules, with names such as “Logs”, “DataModelling”, “SQLDump”, and “Analytics”. All the files respective to their modules will be created inside their respective folders. Whenever a new database is created, the program creates a folder with the database name inside the VM folder and it creates a metadata file in the format of .txt file to store the information about the database and the tables in it. As for a table, it creates a .txt file inside the database folder considering which database that table belongs to. An overview of the project structure can be seen in the image below.

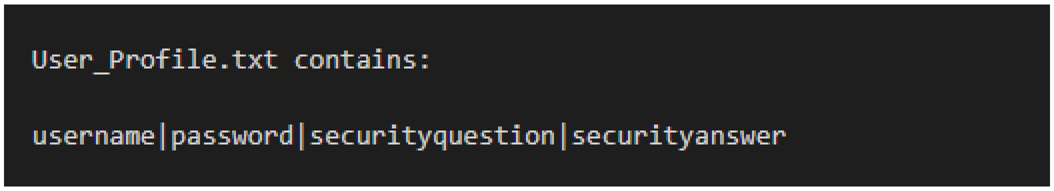


```
VM1
    - User_Profile.txt
    - Global_Data_Dictionary.txt
    - Local_Meta_Data.txt
    - Logs
        - Query_Log.txt
        - Event_Log.txt
        - General_Log.txt
    - DataModelling
        - dbname_ERD.txt
    - SQLDump
        - dbname_SqlDump.txt
    - Analytics
        - Number_Of_Queries.txt
        - Update_Operations.txt
    - dbname
        - dbname_metadata.txt
        - tablename.txt
        - tablename.txt
        ... more tables
    - dbname
        - dbname_metadata.txt
        - tablename.txt
        - tablename.txt
        ... more tables
    ... more databases
```

Figure 1.0: Overview of the project structure

1.1 User Profile File

Starting with User_Profile.txt, this file contains information about the users as their username, password, security question, and security answer. These details in this file are separated by a pipeline character (|). An overview of the file is below. More details about this would be discussed in the User Interface module later.

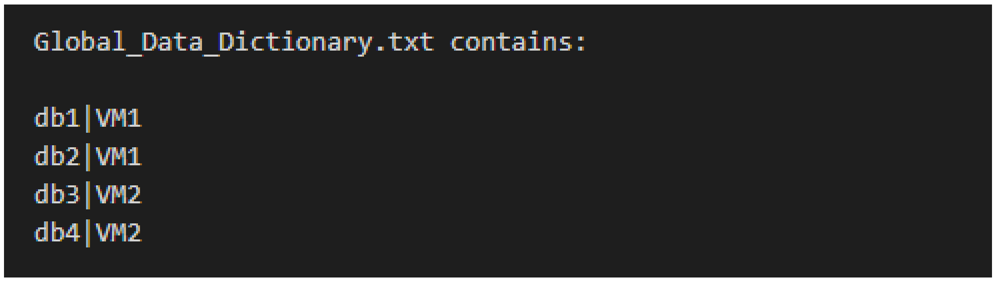


```
User_Profile.txt contains:

username|password|securityquestion|securityanswer
```

Figure 1.1.1: File structure of User Profile file

1.2 Global Data Dictionary File

Global Data Dictionary contains the names of the databases separated by a pipeline character (|) with the name of the Virtual Machine that the database is in. This file will remain a single and the sole file for both virtual machines. So, it has information about both the machines and all the databases in it. Whenever a new database gets created in either of the virtual machines, it gets updated in this file and this file is going to be only on one of the machines and not both. This is to know where our databases are and to help all the other operations smoothly. An overview of the file is below.

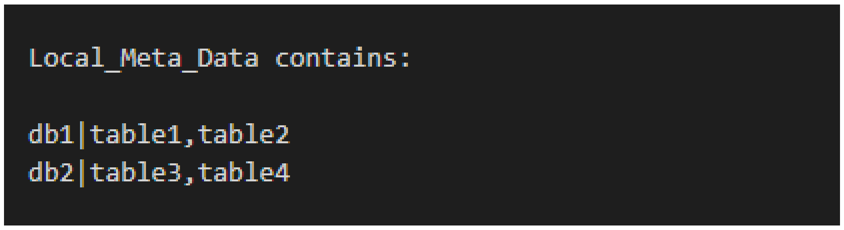


```
Global_Data_Dictionary.txt contains:

db1|VM1
db2|VM1
db3|VM2
db4|VM2
```

Figure 1.2.1: File structure of Global Data Dictionary file

1.3 Local Meta Data File

Coming to the Local Metadata file, this file is specifically designed to be for that particular virtual machine only and since it is that, it contains only the information of the databases in the virtual machine and the tables each database has. Every time a table gets created; this file gets updated with the needful information. An overview of the file is below.

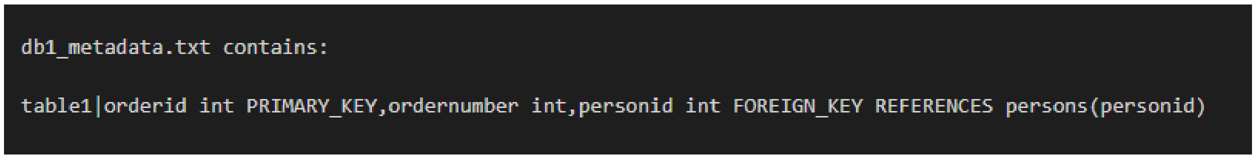


```
Local_Meta_Data contains:

db1|table1,table2
db2|table3,table4
```

Figure 1.3.1: File structure of Local Meta Data file

1.4 Database Meta Data File

And the last file that needs introduction is the metadata file that is respective to one particular database. “dbname_metadata.txt” files contain the information of all the tables in that database along with their column names, their types, their relationships if any, and also their constraints. All these details are needed to validate the entries user gives in before executing any operation. An example of how this file stores the data is as follows.



```
db1_metadata.txt contains:

table1|orderid int PRIMARY_KEY,ordernumber int,personid int FOREIGN_KEY REFERENCES persons(personid)
```

Figure 1.4.1: File structure of Database Meta Data file

Module 2: Query Implementation

This module is divided into two parts:

1. Query Parsing
2. Query Execution

After successful login, users have to choose the ‘write query’ option from all given options. It takes the user directly to the Query Implementation module.

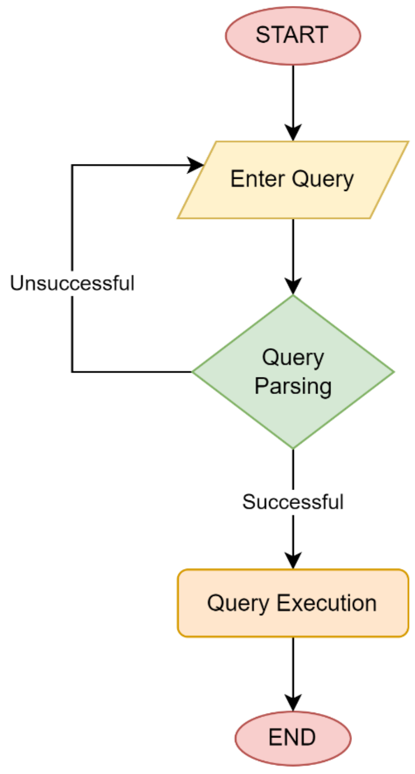


Figure 2.0: Flow chart of Query Implementation

2.1 Query Parsing

Here, the parser will take queries and check that the query is syntactically correct. For that, a different regex is written programmatically to check the spellings, blank spaces between two worlds, and the end query delimiter which is a semicolon (;).
If the entered query is wrong on any aspect, then it will not execute it and displays a specific error message for it.

Regex that is used in our program are the following:

- ‘CREATE DATABASE’ Query

  Sample query:
  - CREATE DATABASE [name_of_database];

  Regex:
  - \s*create\s+database\s+[0-9a-zA-Z_]+\s*;\s*

- ‘CREATE TABLE’ Query

  Sample query:
  - CREATE TABLE [table_name] (id [type_of_field], [further fields] ...);

  Regex:
  - \s*create\s+table\s+[0-9a-zA-Z_]+\s*\(\s*([0-9a-zA-Z_]+\s+(int|float|boolean|varchar\(([1-9]|[1-9][0-9]|[1-4][0-9][0-9]|500)\))\s*,\s*)*[0-9a-zA-Z_]+\s+(int|float|boolean|varchar\(([1-9]|[1-9][0-9]|[1-4][0-9][0-9]|500)\))(\s*,\s*primary\s+key\s*\(\s*[0-9a-zA-Z_]+\s*\))?(\s*,\s*foreign\s+key\s*\(\s*[0-9a-zA-Z_]+\s*\)\s+references\s+[0-9a-zA-Z_]+\s*\(\s*[0-9a-zA-Z_]+\s*\))*\s*\)\s*;\s*

- ‘USE’ Query

  Sample query:

  - USE [name_of_database];

  Regex:

  - \s*use\s+[0-9a-zA-Z_]+\s*;\s*

- ‘INSERT’ Query

  Sample queries:

  - INSERT INTO [table_name] values [field_values];
  - INSERT INTO [column_name] values [field_values];

  Regex:

  - \s*insert\s+into\s+[0-9a-zA-Z_]+(\s*\(\s*([0-9a-zA-Z_]+\s*,\s*)*[0-9a-zA-Z_]+\s*\))?\s+values\s*\(\s*(('[0-9a-zA-Z _?!@&*()-]*'|\d+)\s*,\s*)*('[0-9a-zA-Z _?!@&*()-]*'|\d+)\s*\)\s*;\s*

- ‘SELECT’ Query

  Sample queries:

  - SELECT * FROM [table_name];
  - SELECT * FROM [table_name] WHERE [condition];
  - SELECT [columns] FROM [table_name];

  Regex:

  - \s*select\s+(\*|([0-9a-zA-Z_]+\s*,\s*)*[0-9a-zA-Z_]+)\s+from\s+[0-9a-zA-Z_]+\s*(\swhere\s+[0-9a-zA-Z_]+\s*=\s*('[0-9a-zA-Z _?!@&*()-]*'|\d+))?;\s*

- ‘UPDATE’ Query

  Sample query:

  - UPDATE [table_name] SET [values];

  Regex:

  - \s*update\s+[0-9a-zA-Z_]+\s+set\s+[0-9a-zA-Z_]+\s*=\s*('[0-9a-zA-Z _?!@&*()-]*'|\d+)\s+where\s+[0-9a-zA-Z_]+\s*=\s*('[0-9a-zA-Z _?!@&*()-]*'|\d+)\s*;\s*

- ‘DELETE’ Query

  Sample query:

  - DELETE FROM [table_name] WHERE [condition];

  Regex:

  - \s*delete\s+from\s+[0-9a-zA-Z_]+\s+where\s+[0-9a-zA-Z_]+\s*=\s*('[0-9a-zA-Z _?!@&*()-]*'|\d+)\s*;\s*

- ‘START TRANSACTION’ Query

  Sample query:

  - START TRANSACTION;

  Regex:

  - \s*start\s+transaction\s*;\s*

- ‘COMMIT’ Query

  Sample query:

  - COMMIT;

Regex:

- \s*commit\s*;\s*

- ‘ROLLBACK’ Query

  Sample query:

  - ROLLBACK;

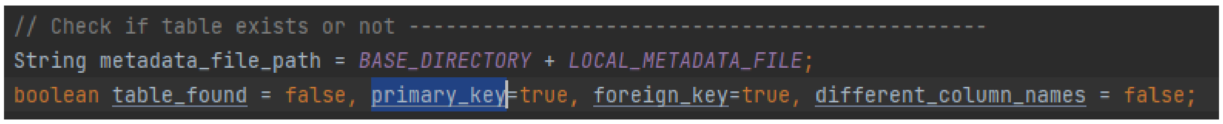
  Regex:

  - \s*rollback\s*;\s*

Program will firstly check the syntax, and syntactical structure of the query and then fetch the table or database names. It will check in local metadata if a table or database exists or not. If it does not exist, then the program will throw the error. After successfully parsing it will pass the query to the execution part of the program.

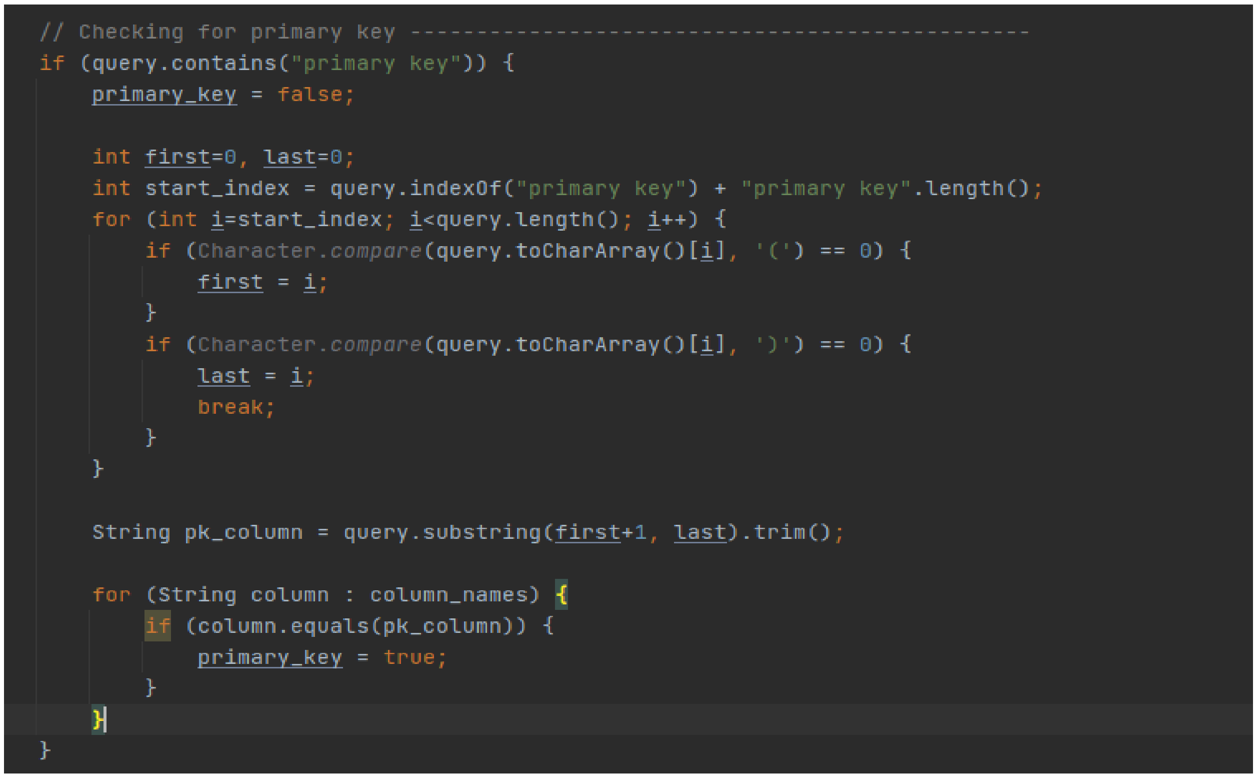
Identifying Primary and Foreign Keys:

As in any database, the concept of the primary and foreign keys is very important and the basis for relationships between tables. Our program is capable of identifying the primary key in the table while parsing it if the user mentions it in the query.



```
// Check if table exists or not ---------------------------------------------
String metadata_file_path = BASE_DIRECTORY + LOCAL_METADATA_FILE;
boolean table_found = false, primary_key=true, foreign_key=true, different_column_names = false;
```

Figure 2.1.1: Primary key and Foreign key flags



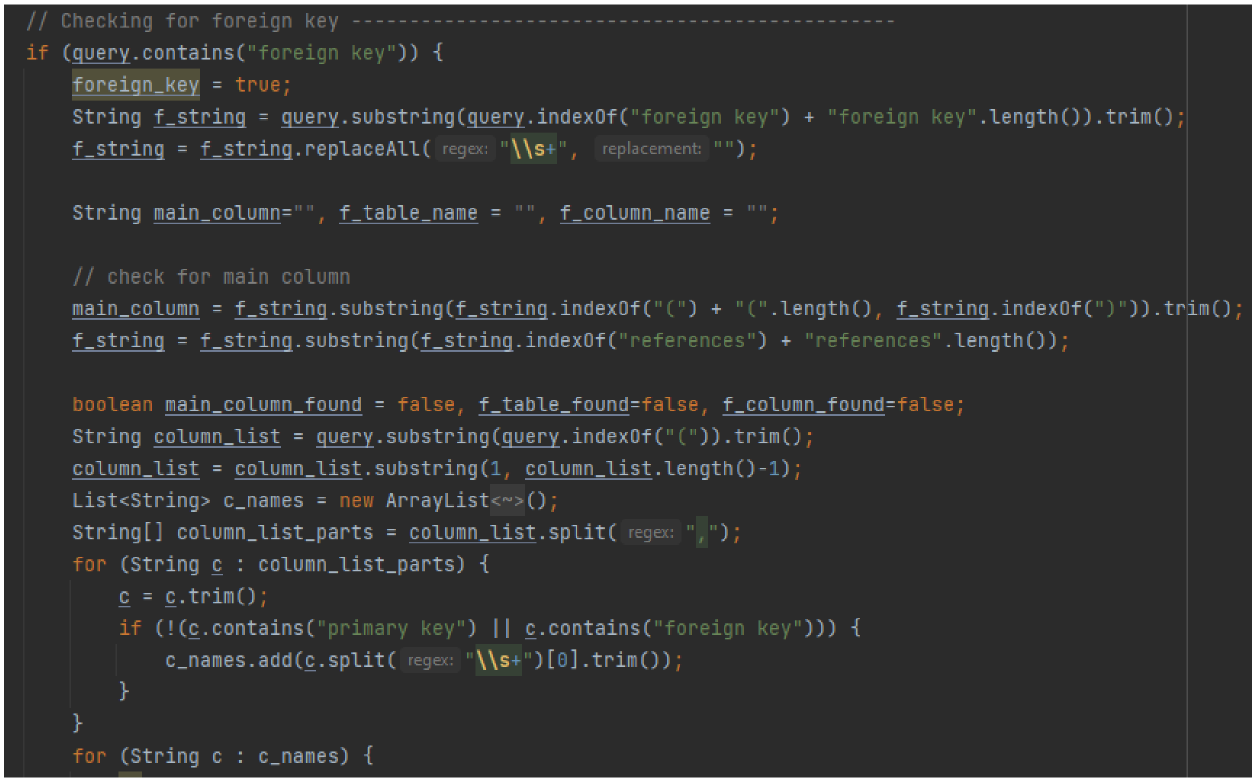
```java
// Checking for primary key -------------------------------------------
if (query.contains("primary key")) {
    primary_key = false;

    int first=0, last=0;
    int start_index = query.indexOf("primary key") + "primary key".length();
    for (int i=start_index; i<query.length(); i++) {
        if (Character.compare(query.toCharArray()[i], '(') == 0) {
            first = i;
        }
        if (Character.compare(query.toCharArray()[i], ')') == 0) {
            last = i;
            break;
        }
    }

    String pk_column = query.substring(first+1, last).trim();

    for (String column : column_names) {
        if (column.equals(pk_column)) {
            primary_key = true;
        }
    }
}
```

Figure 2.1.2: Checking for primary key

In the code, if the user mentions that a particular field of a table is the primary key for the table then here it will identify that. And make this primary_key variable true and add it to the metadata. Sample query for creating a table with the primary key:

CREATE TABLE [table_name] (ID int, NAME varchar(25), PRIMARY KEY (ID));

The field ID will be set as the primary key. This field will be unique and it cannot be null, it should contain a unique value. A table can contain multiple foreign keys. Users have to give the reference of the table from where the table should match the value for this particular field.



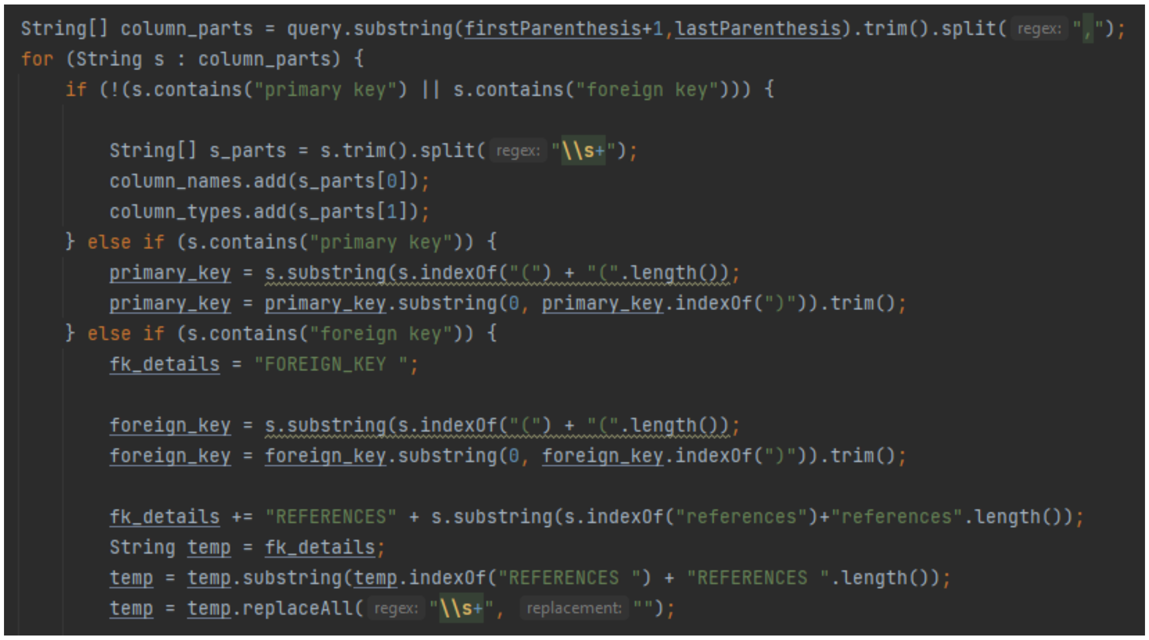
```java
// Checking for foreign key ----------------------------------------------
if (query.contains("foreign key")) {
    foreign_key = true;
    String f_string = query.substring(query.indexOf("foreign key") + "foreign key".length()).trim();
    f_string = f_string.replaceAll("\\s+", "");

    String main_column="", f_table_name = "", f_column_name = "";

    // check for main column
    main_column = f_string.substring(f_string.indexOf("(") + "(".length(), f_string.indexOf(")")).trim();
    f_string = f_string.substring(f_string.indexOf("references") + "references".length());

    boolean main_column_found = false, f_table_found=false, f_column_found=false;
    String column_list = query.substring(query.indexOf("(")).trim();
    column_list = column_list.substring(1, column_list.length()-1);
    List<String> c_names = new ArrayList<~>();
    String[] column_list_parts = column_list.split(",");
    for (String c : column_list_parts) {
        c = c.trim();
        if (!(c.contains("primary key") || c.contains("foreign key"))) {
            c_names.add(c.split("\\s+")[0].trim());
        }
    }
    for (String c : c_names) {
```

Figure 2.1.3: Checking for foreign key

The code snippet is checking if the user has added the foreign key or not and if yes, then it will process further for the taking reference from the other table.

2.2 Query Execution

It will change the table which we store in the form of text files, according to the query. For creating a database, it will create a folder in the folder structure by the name of the database. If a user creates a table, then it will create a text file in that folder. If a user executes any query to perform a select, insert, update or delete a task, the program will fetch the table name and values that should be updated from the query. After that, it will access the file for that particular database and then perform the task according to the query.



```java
String[] column_parts = query.substring(firstParenthesis+1,lastParenthesis).trim().split( regex: ",");
for (String s : column_parts) {
    if (!(s.contains("primary key") || s.contains("foreign key"))) {

        String[] s_parts = s.trim().split( regex: "\\s+");
        column_names.add(s_parts[0]);
        column_types.add(s_parts[1]);
    } else if (s.contains("primary key")) {
        primary_key = s.substring(s.indexOf("(") + "(".length());
        primary_key = primary_key.substring(0, primary_key.indexOf(")")).trim();
    } else if (s.contains("foreign key")) {
        fk_details = "FOREIGN_KEY ";

        foreign_key = s.substring(s.indexOf("(") + "(".length());
        foreign_key = foreign_key.substring(0, foreign_key.indexOf(")")).trim();

        fk_details += "REFERENCES" + s.substring(s.indexOf("references")+"references".length());
        String temp = fk_details;
        temp = temp.substring(temp.indexOf("REFERENCES ") + "REFERENCES ".length());
        temp = temp.replaceAll( regex: "\\s+",  replacement: "");
```

Figure 2.2.1: Query execution

The above image is displaying the code of one of our methods named executeCreateTable().

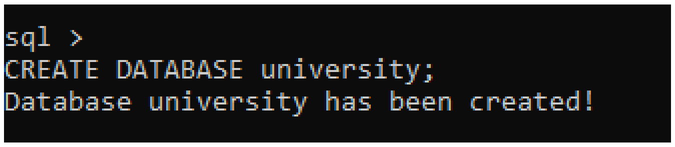
The query should be in the form of Create table [table_name] (id [type_of_field], [further fields]..);.

So, this code will fetch the name of the table, column name, and type from the query and for it, the code will find the index of the opening bracket which is ‘(‘.

After that, it will make a substring to identify all fields of the table.

2.3 Test cases

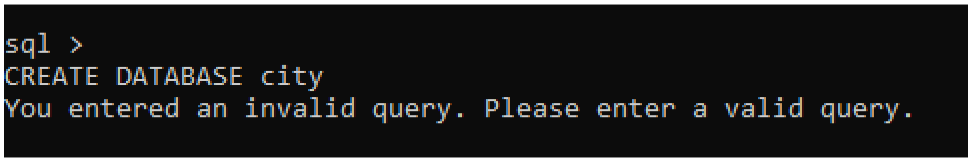
(1) Successful create database



```
sql >
CREATE DATABASE university;
Database university has been created!
```

Figure 2.3.1: Valid CREATE DATABASE query

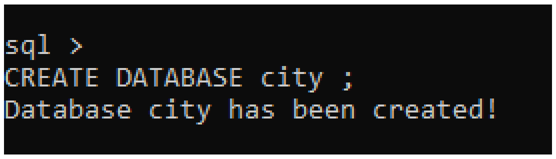
(2) Failed/Wrong create database



```
sql >
CREATE DATABASE city
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.2: Invalid CREATE DATABASE query

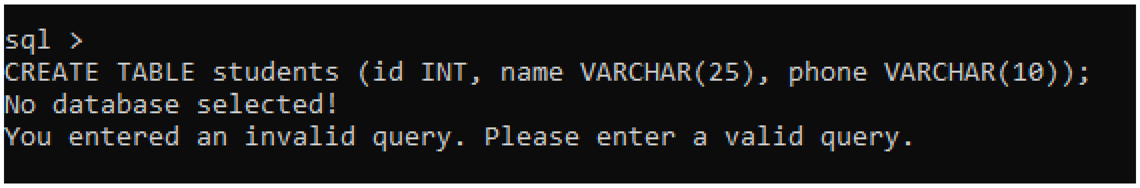
(3) Create database with space before the end delimiter



```
sql >
CREATE DATABASE city ;
Database city has been created!
```

Figure 2.3.3: CREATE DATABASE query with extra spaces

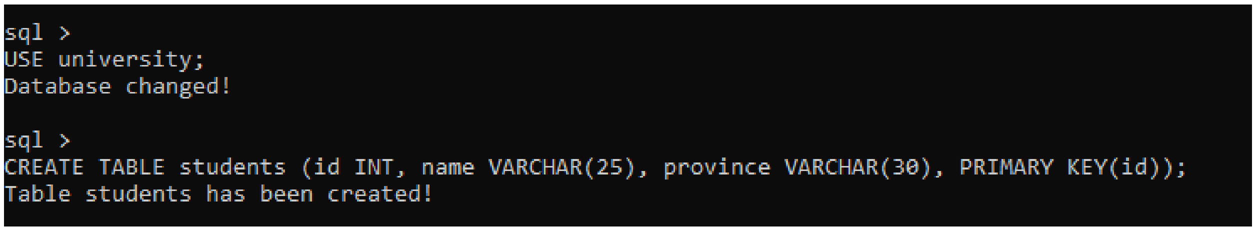
(4) Failed create table without selecting database



```
sql >
CREATE TABLE students (id INT, name VARCHAR(25), phone VARCHAR(10));
No database selected!
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.4: Invalid CREATE TABLE query

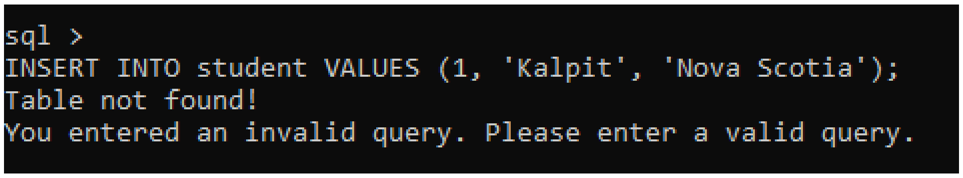
(5) Successful create table with selecting database



```
sql >
USE university;
Database changed!

sql >
CREATE TABLE students (id INT, name VARCHAR(25), province VARCHAR(30), PRIMARY KEY(id));
Table students has been created!
```

Figure 2.3.5: Valid CREATE TABLE query

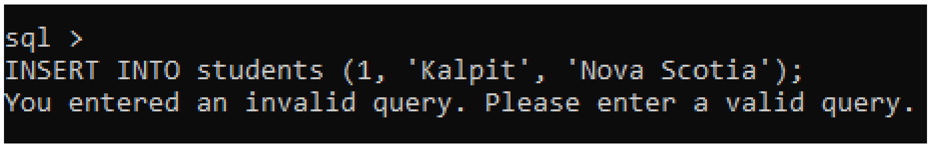
(6) Wrong table insert statement



```
sql >
INSERT INTO student VALUES (1, 'Kalpit', 'Nova Scotia');
Table not found!
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.6: Invalid INSERT query

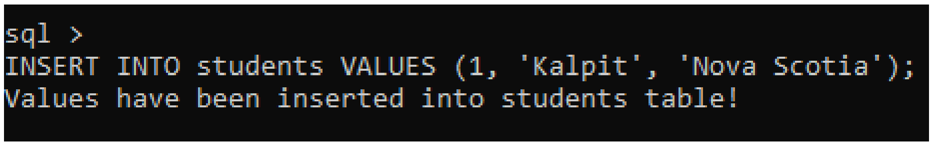
(7) Wrong syntax insert statement



```
sql >
INSERT INTO students (1, 'Kalpit', 'Nova Scotia');
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.7: Wrong INSERT query syntax

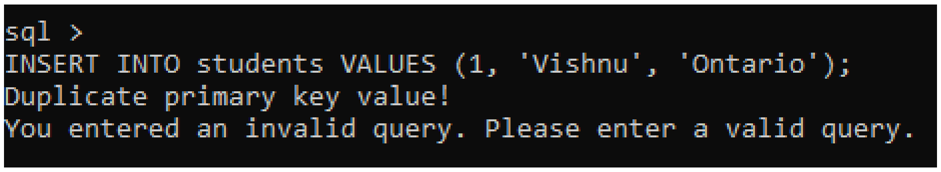
(8) Successful Insert statement



```
sql >
INSERT INTO students VALUES (1, 'Kalpit', 'Nova Scotia');
Values have been inserted into students table!
```

Figure 2.3.8: Valid INSERT query

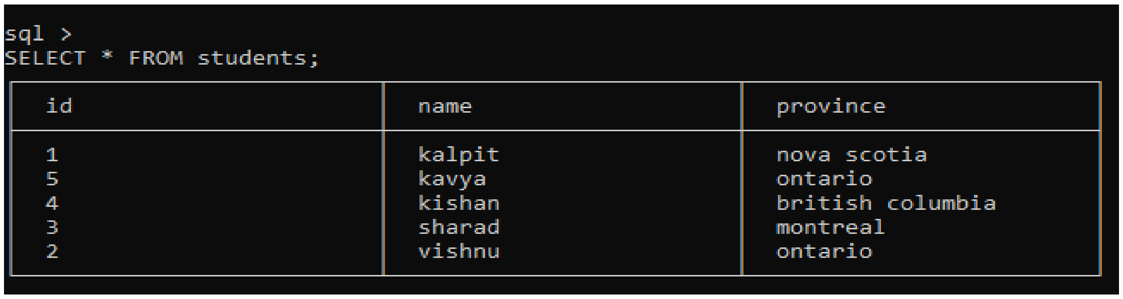
(9) Duplicate Primary key



```
sql >
INSERT INTO students VALUES (1, 'Vishnu', 'Ontario');
Duplicate primary key value!
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.9: Duplicate primary key in INSERT query

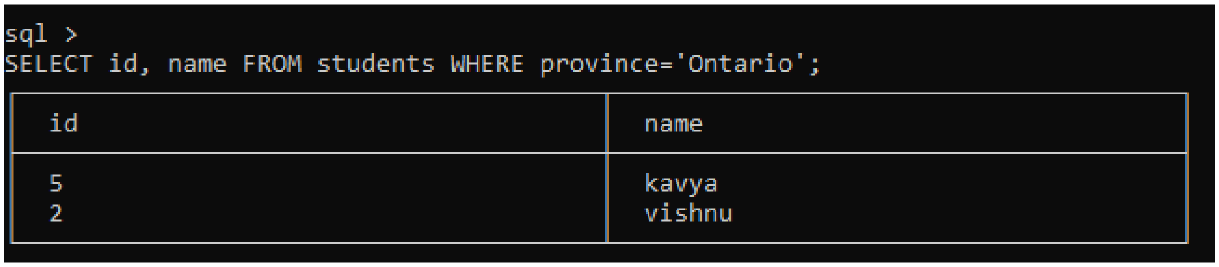
(10) Select statement



```
sql >
SELECT * FROM students;
┌────────────────────┬───────────────────┬───────────────────┐
│ id                 │ name              │ province          │
├────────────────────┼───────────────────┼───────────────────┤
│ 1                  │ kalpit            │ nova scotia       │
│ 5                  │ kavya             │ ontario           │
│ 4                  │ kishan            │ british columbia  │
│ 3                  │ sharad            │ montreal          │
│ 2                  │ vishnu            │ ontario           │
└────────────────────┴───────────────────┴───────────────────┘
```

Figure 2.3.10: SELECT query

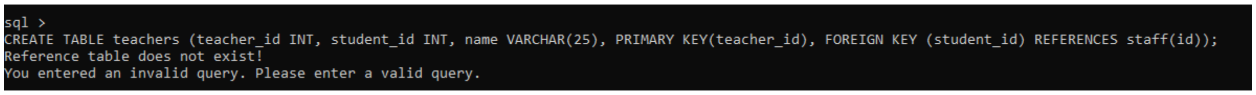
(11) Select specific records



```
sql >
SELECT id, name FROM students WHERE province='Ontario';
┌──────────────────────────┬──────────────────────────┐
│ id                       │ name                     │
├──────────────────────────┼──────────────────────────┤
│ 5                        │ kavya                    │
│ 2                        │ vishnu                   │
└──────────────────────────┴──────────────────────────┘
```

Figure 2.3.11: SELECT query with specific columns

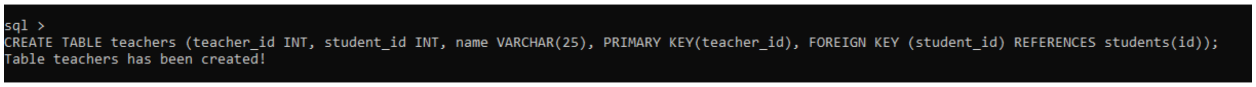
(12) Failed to create table with foreign key



```
sql >
CREATE TABLE teachers (teacher_id INT, student_id INT, name VARCHAR(25), PRIMARY KEY(teacher_id), FOREIGN KEY (student_id) REFERENCES staff(id));
Reference table does not exist!
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.12: Wrong foreign key in CREATE TABLE query

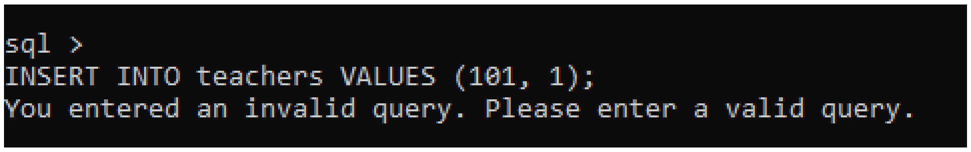
(13) Successful create table with foreign key



```
sql >
CREATE TABLE teachers (teacher_id INT, student_id INT, name VARCHAR(25), PRIMARY KEY(teacher_id), FOREIGN KEY (student_id) REFERENCES students(id));
Table teachers has been created!
```

Figure 2.3.13: Valid CREATE DATABASE query with foreign key

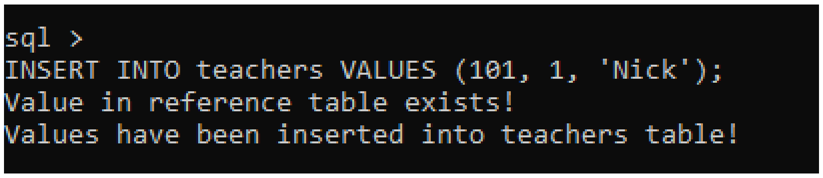
(14) Insert statement without foreign key



```
sql >
INSERT INTO teachers VALUES (101, 1);
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.14: INSERT query without foreign key

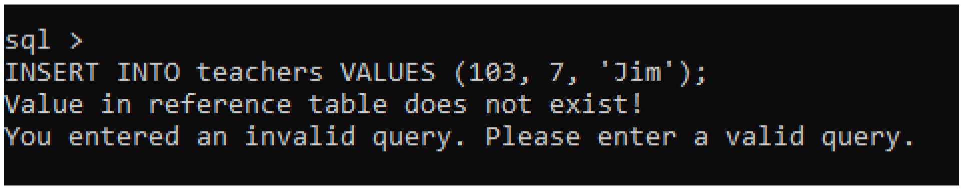
(15) Insert with foreign key and check for reference



```
sql >
INSERT INTO teachers VALUES (101, 1, 'Nick');
Value in reference table exists!
Values have been inserted into teachers table!
```

Figure 2.3.15: Invalid reference key value in INSERT query

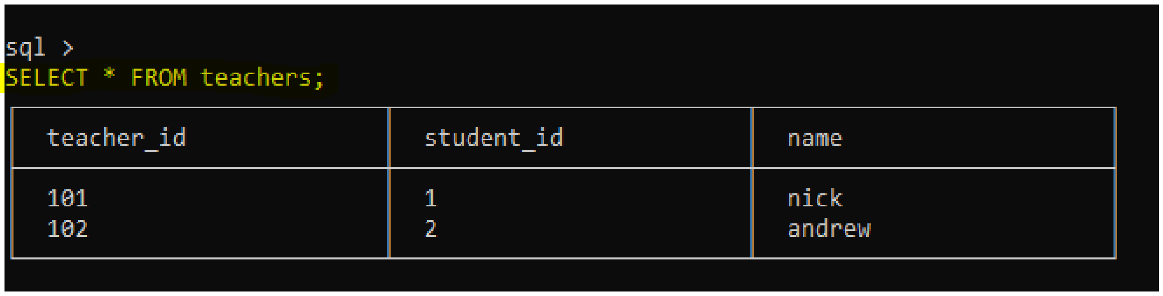
(16) Insert with foreign key and wrong reference



```
sql >
INSERT INTO teachers VALUES (103, 7, 'Jim');
Value in reference table does not exist!
You entered an invalid query. Please enter a valid query.
```

Figure 2.3.16: Invalid reference table in INSERT query

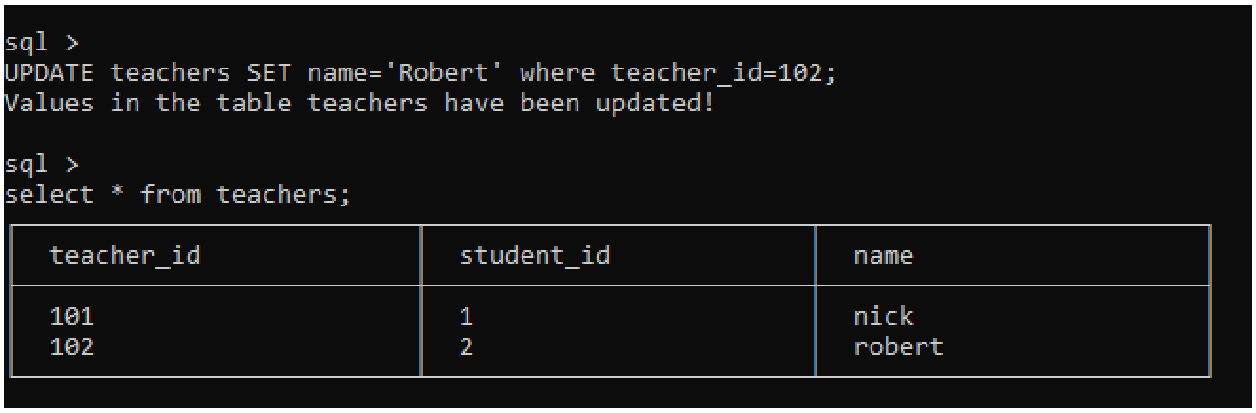
(17) Select all records



```
sql >
SELECT * FROM teachers;
┌─────────────────────┬────────────────────┬────────────────────┐
│ teacher_id          │ student_id         │ name               │
├─────────────────────┼────────────────────┼────────────────────┤
│ 101                 │ 1                  │ nick               │
│ 102                 │ 2                  │ andrew             │
└─────────────────────┴────────────────────┴────────────────────┘
```

Figure 2.3.17: SELECT query for all records

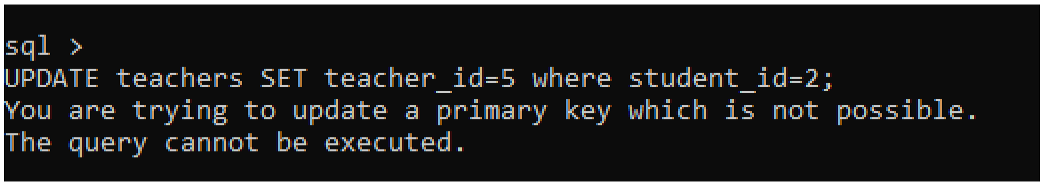
(18) Successful updated records



```
sql >
UPDATE teachers SET name='Robert' where teacher_id=102;
Values in the table teachers have been updated!

sql >
select * from teachers;
┌──────────────────────┬─────────────────────┬─────────────────────┐
│ teacher_id           │ student_id          │ name                │
├──────────────────────┼─────────────────────┼─────────────────────┤
│ 101                  │ 1                   │ nick                │
│ 102                  │ 2                   │ robert              │
└──────────────────────┴─────────────────────┴─────────────────────┘
```

Figure 2.3.18: Valid UPDATE Query

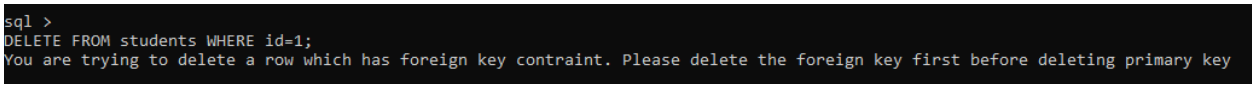
(19) Try to update primary key



```
sql >
UPDATE teachers SET teacher_id=5 where student_id=2;
You are trying to update a primary key which is not possible.
The query cannot be executed.
```

Figure 2.3.19: Invalid UPDATE query

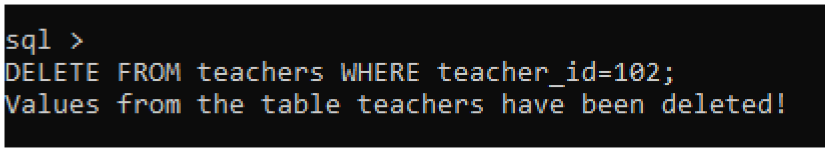
(20) Try to Delete a record with a foreign key



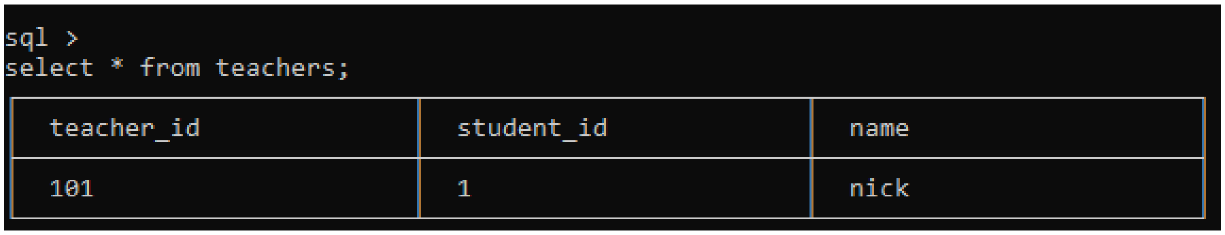
```
sql >
DELETE FROM students WHERE id=1;
You are trying to delete a row which has foreign key contraint. Please delete the foreign key first before deleting primary key
```

Figure 2.3.20: Deleting record with foreign key

(21) Successful delete query



```
sql >
DELETE FROM teachers WHERE teacher_id=102;
Values from the table teachers have been deleted!
```



```
sql >
select * from teachers;
```

teacher_id	student_id	name
101	1	nick

Figure 2.3.21: Valid DELETE query

Module 3: Transaction Processing

The transaction is the set of query statements. All queries in the transaction module are either committed or rollback. The transaction follows the ACID properties. ACID is an acronym for Atomicity, Consistency, Isolation, Durability.

- **Atomicity:** It means that the whole set of statements in the transaction takes place or not at all.
- **Consistency:** It means that the database should be consistent before and after the transaction.
- **Isolation:** The parallel transaction can occur without any inconsistency in the database.
- **Durability:** This property ensures that once the transaction is completed and the modifications are saved on disk and even after failure, it will remain.

The flowchart for this module is as shown below:

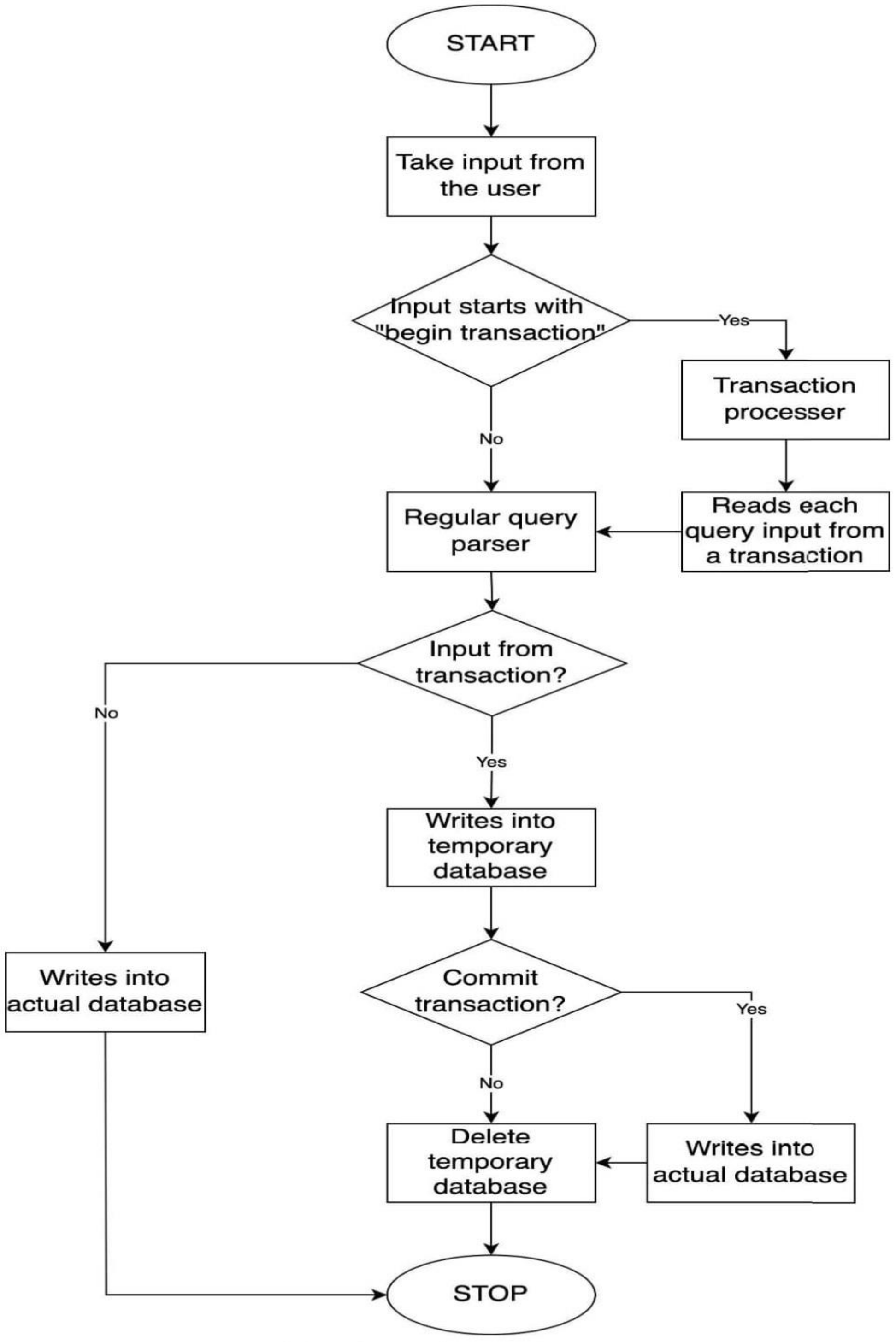

START
Take input from the user
Input starts with "begin transaction"
Yes
Transaction processer
No
Reads each query input from a transaction
Regular query parser
Input from transaction?
No
Yes
Writes into temporary database
Writes into actual database
Commit transaction?
Yes
No
Delete temporary database
Writes into actual database
STOP

Figure 3.0: Flowchart of transaction processing

Here, we have used the concept of DDL (Data Definition Language) and DML (Data Manipulation Language).

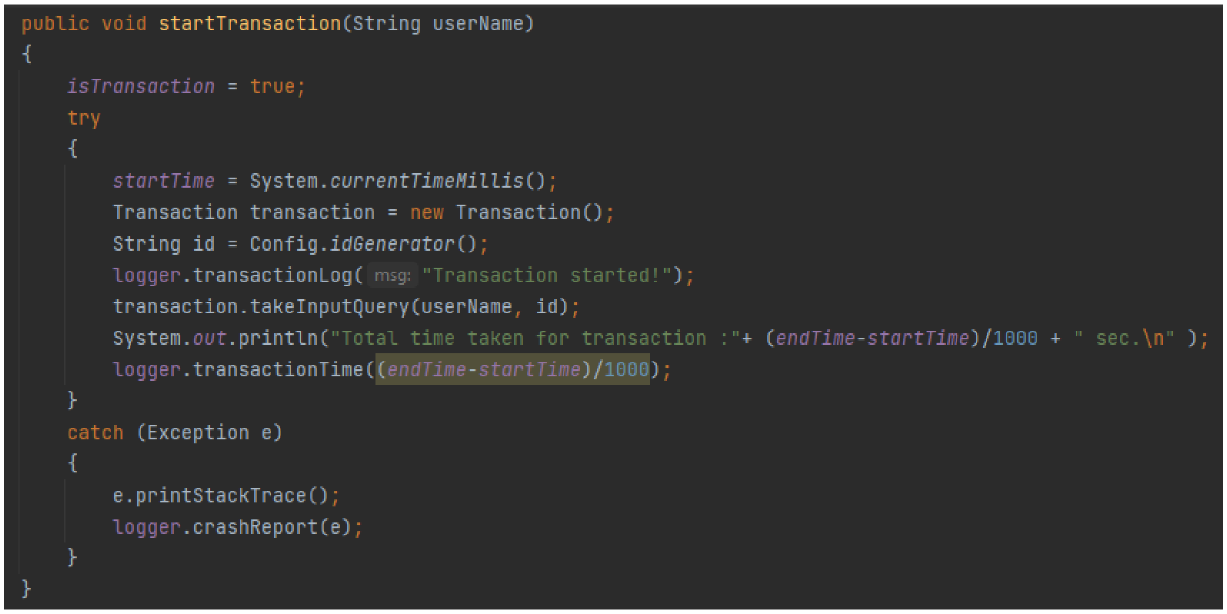
→ DDL means the query statement that creates or defines the database.

→ DML means the query statement that manipulates or modifies the database.

To enter a transaction, the user has to write:

START TRANSACTION;

This will take the user to the transaction module where the user can enter the multiple query statement.

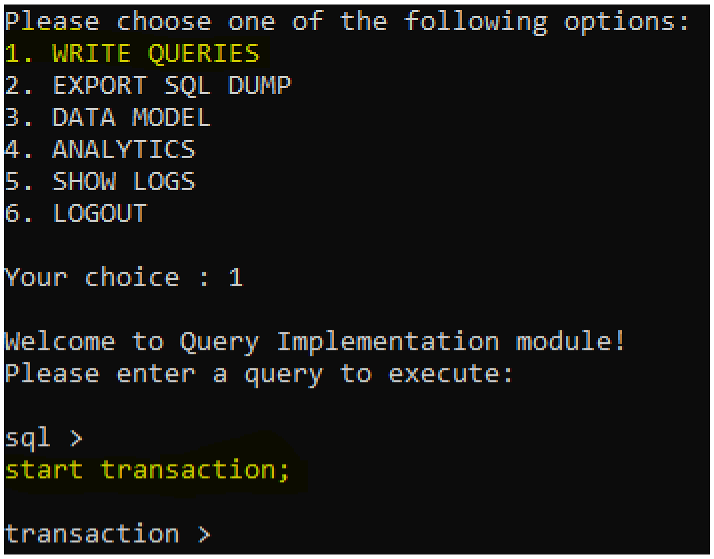


```
public void startTransaction(String userName)
{
    isTransaction = true;
    try
    {
        startTime = System.currentTimeMillis();
        Transaction transaction = new Transaction();
        String id = Config.idGenerator();
        logger.transactionLog( msg: "Transaction started!");
        transaction.takeInputQuery(userName, id);
        System.out.println("Total time taken for transaction :"+ (endTime-startTime)/1000 + " sec.\n" );
        logger.transactionTime((endTime-startTime)/1000);
    }
    catch (Exception e)
    {
        e.printStackTrace();
        logger.crashReport(e);
    }
}
```

Figure 3.1: Code snippet of Transaction module

Whenever a user enters a transaction, the program will generate a unique id and keep it till the end of the transaction. If the entered query uses a database, create a table/database then it will parse it first and then execute it because it is needed to parse the further queries. All other queries will be stored in a data structure. When the user hits commit or rollback then, it will parse every query statement one by one so that user-provided. Here if the query is parsed successfully then it will be executed. If any statement fails, then only the DDL (Data Definition Language) statement before the failed statement will be executed. If all queries are passed, then all statements will be executed. If the transaction is a rollback, then it will only execute the DDL statement from the given set of query statements.

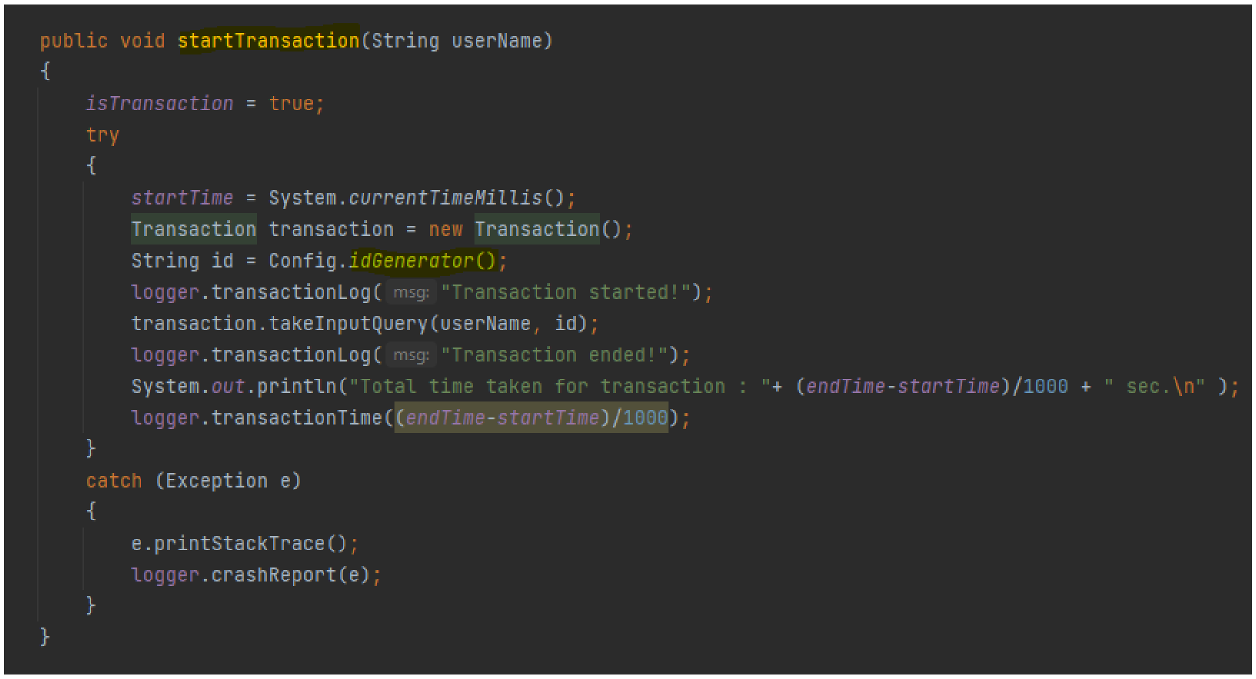
So, here are some Test cases for the implemented Transaction module:



```
Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT

Your choice : 1

Welcome to Query Implementation module!
Please enter a query to execute:

sql >
start transaction;

transaction >
```

Figure 3.2: Options of user menu after login

Firstly, users have to select the “1. Write Queries” option from the given list. After that, you have to just type “Start transaction” to begin a new transaction. As a user entered a transaction, the program will give it a unique ID. The ID is generated by the method named idGenerator() of the Config class in the transaction package. It will pass that id and the name of the user to the takeQueryInput() method. In that method, all the queries are taken as input first. Entered statements are stored in a data structure.



```java
public void startTransaction(String userName)
{
    isTransaction = true;
    try
    {
        startTime = System.currentTimeMillis();
        Transaction transaction = new Transaction();
        String id = Config.idGenerator();
        logger.transactionLog( msg: "Transaction started!");
        transaction.takeInputQuery(userName, id);
        logger.transactionLog( msg: "Transaction ended!");
        System.out.println("Total time taken for transaction : "+ (endTime-startTime)/1000 + " sec.\n" );
        logger.transactionTime((endTime-startTime)/1000);
    }
    catch (Exception e)
    {
        e.printStackTrace();
        logger.crashReport(e);
    }
}
```

Figure 3.3: Code snippet of transaction start point

If the entered query statement is meant for creating a table/database or using any database then, it will parse and execute first. Because if the user does not choose any database, then it can identify the database to use.

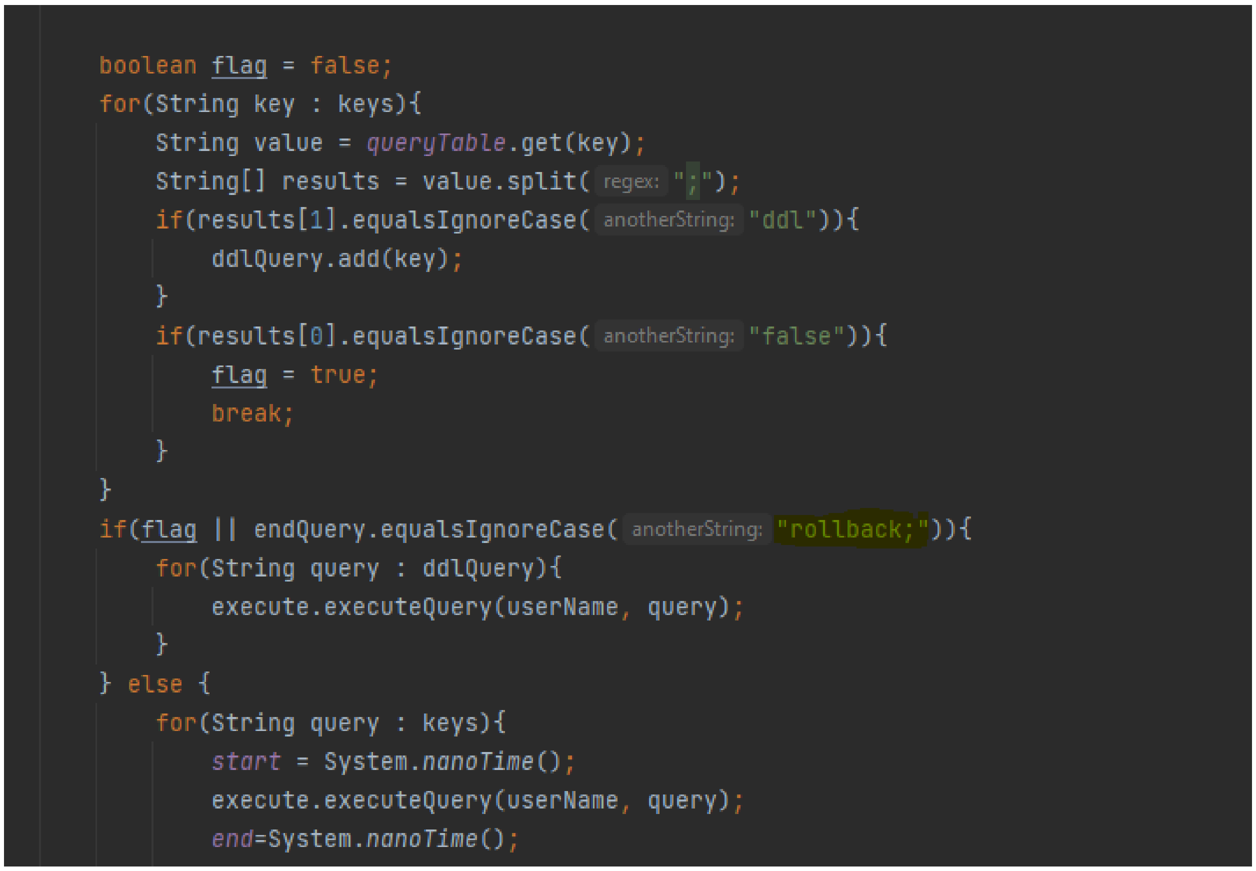
As locking is necessary in the transaction module. So, as the user starts a new transaction, the lock module will activate and lock every table used by the current transaction. The lock module is described laterally in this report.

All other queries are stored in a particular data structure after parsing one after one. The validation and type of query are also stored in the same data structure.

- If the statement is correct and the type is DDL then it will store “true;ddl” as a value.

- If the statement is incorrect and the type is DDL then it will store "false;ddl" as a value.
- If the statement is correct and the type is DML then it will store "true;dml" as a value.
- If the statement is incorrect and the type is DML then it will store "false;dml" as a value.

The program will pass the data structure to process the next steps. As shown in the next figure, if any query is wrong or uses rollback then only DDL statements will execute. If any statement is wrong, then only DDL queries before that statement will execute.



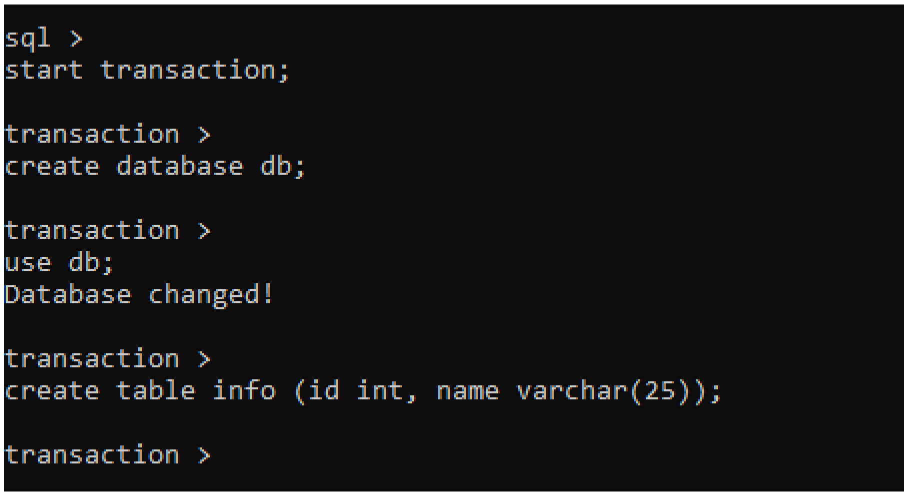
```
boolean flag = false;
for(String key : keys){
    String value = queryTable.get(key);
    String[] results = value.split( regex: ";");
    if(results[1].equalsIgnoreCase( anotherString: "ddl")){
        ddlQuery.add(key);
    }
    if(results[0].equalsIgnoreCase( anotherString: "false")){
        flag = true;
        break;
    }
}
if(flag || endQuery.equalsIgnoreCase( anotherString: "rollback;")){
    for(String query : ddlQuery){
        execute.executeQuery(userName, query);
    }
} else {
    for(String query : keys){
        start = System.nanoTime();
        execute.executeQuery(userName, query);
        end=System.nanoTime();
```

Figure 3.4: Code snippet of transaction

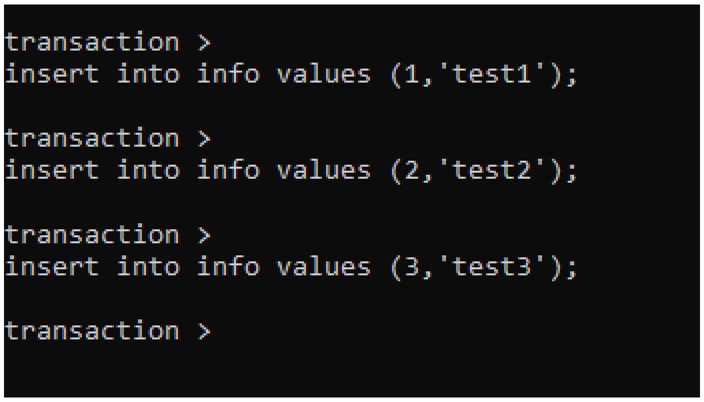
3.1 Test cases

(1) When the transaction is committed or rollback, it will show the total time taken by the transaction to complete the task. It will also display a message that the transaction has ended.

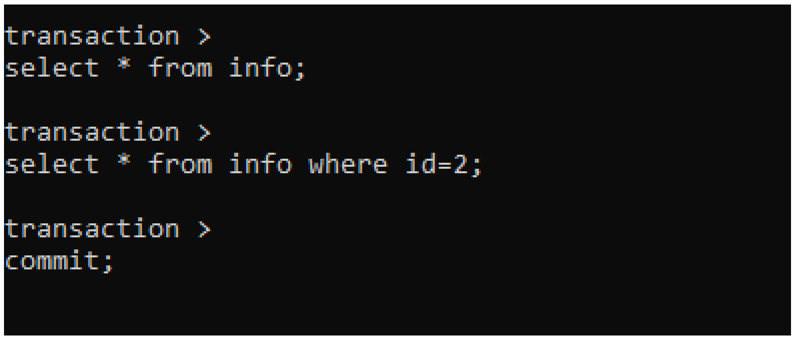
Input:



```
sql >
start transaction;

transaction >
create database db;

transaction >
use db;
Database changed!

transaction >
create table info (id int, name varchar(25));

transaction >
```

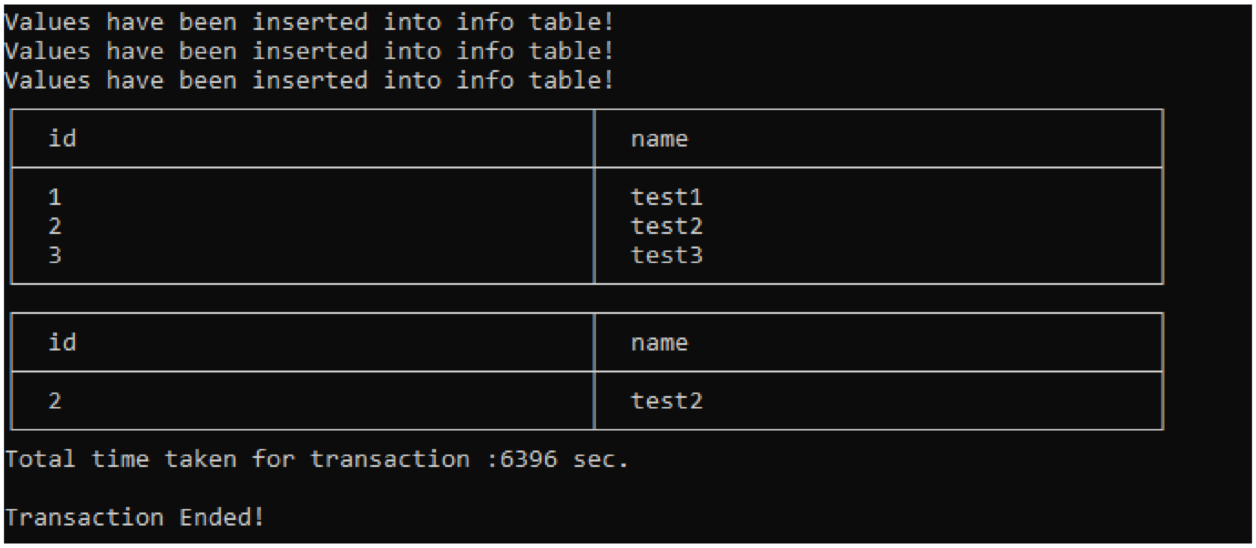


```
transaction >
insert into info values (1,'test1');

transaction >
insert into info values (2,'test2');

transaction >
insert into info values (3,'test3');

transaction >
```



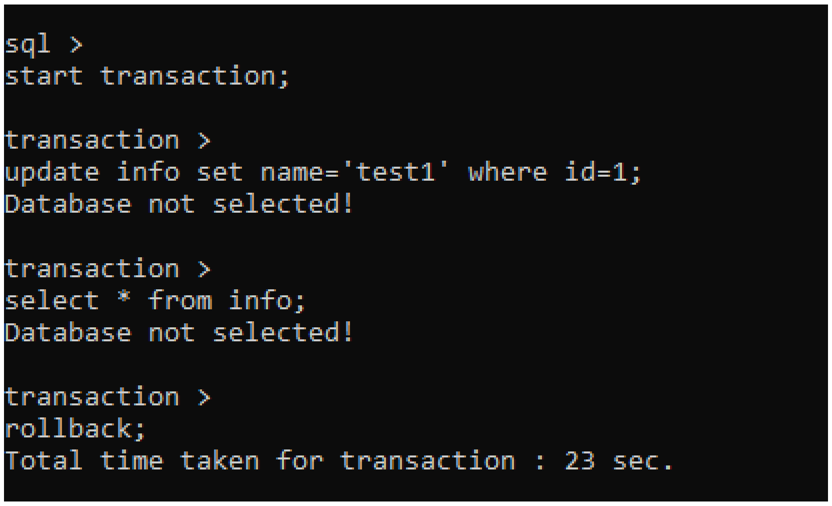
```
transaction >
select * from info;

transaction >
select * from info where id=2;

transaction >
commit;
```

Output:



```
Values have been inserted into info table!
Values have been inserted into info table!
Values have been inserted into info table!
+-------------------------------+------------------------------+
| id                            | name                         |
+-------------------------------+------------------------------+
| 1                             | test1                        |
| 2                             | test2                        |
| 3                             | test3                        |
+-------------------------------+------------------------------+

+-------------------------------+------------------------------+
| id                            | name                         |
+-------------------------------+------------------------------+
| 2                             | test2                        |
+-------------------------------+------------------------------+
Total time taken for transaction :6396 sec.

Transaction Ended!
```

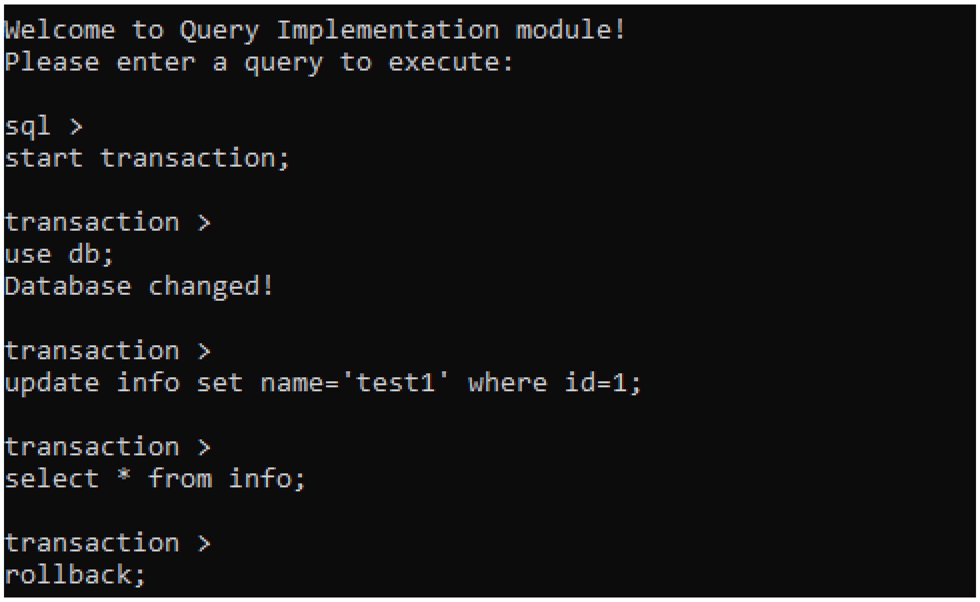
(2) If the database is not selected, it will show this kind of error.



```
sql >
start transaction;

transaction >
update info set name='test1' where id=1;
Database not selected!

transaction >
select * from info;
Database not selected!

transaction >
rollback;
Total time taken for transaction : 23 sec.
```

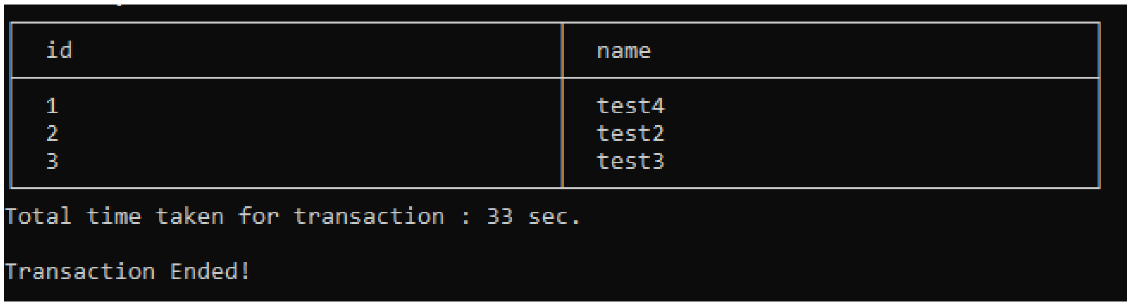
(3) This transaction is rolled back at the end. So according to the process, only the DDL statements which are “select” in this case, will be executed.

Input:



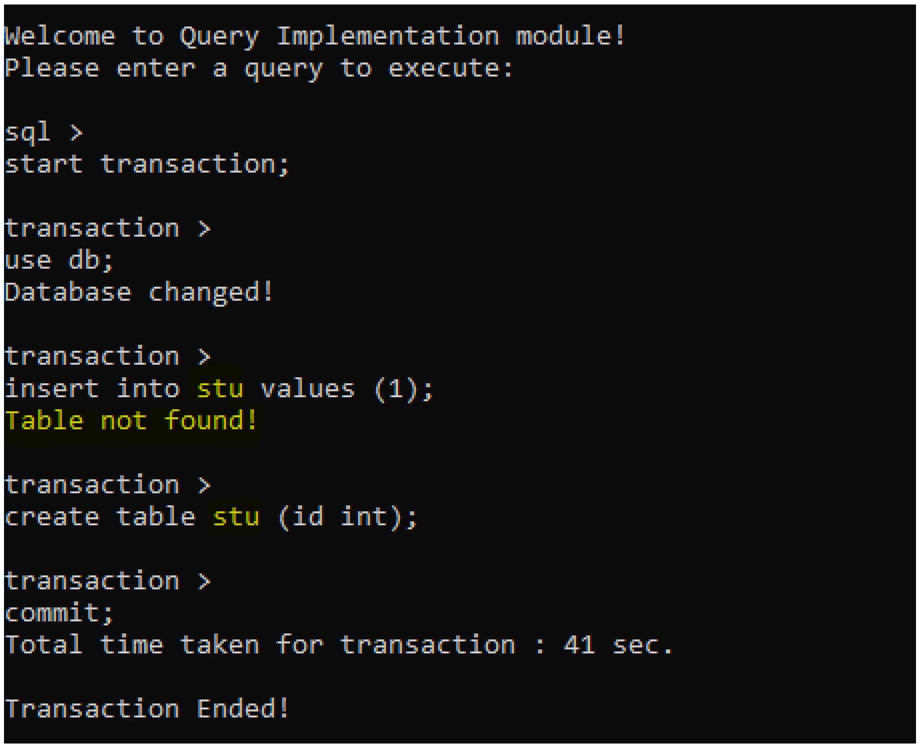
```
Welcome to Query Implementation module!
Please enter a query to execute:

sql >
start transaction;

transaction >
use db;
Database changed!

transaction >
update info set name='test1' where id=1;

transaction >
select * from info;

transaction >
rollback;
```

Output:



```
+----------------------------------+---------------------------------+
| id                               | name                            |
+----------------------------------+---------------------------------+
| 1                                | test4                           |
| 2                                | test2                           |
| 3                                | test3                           |
+----------------------------------+---------------------------------+
Total time taken for transaction : 33 sec.

Transaction Ended!
```

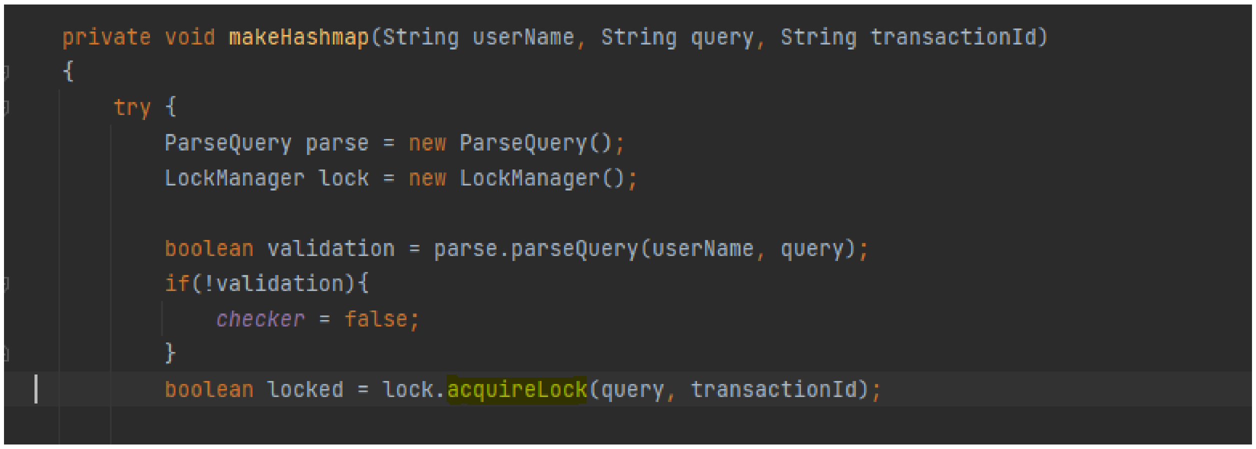
(4) In this case, users try to access a table in a transaction that does not exist. Lately, the same user-created that table. Then this type of transaction should not execute the DML queries.



```
Welcome to Query Implementation module!
Please enter a query to execute:

sql >
start transaction;

transaction >
use db;
Database changed!

transaction >
insert into stu values (1);
Table not found!

transaction >
create table stu (id int);

transaction >
commit;
Total time taken for transaction : 41 sec.

Transaction Ended!
```

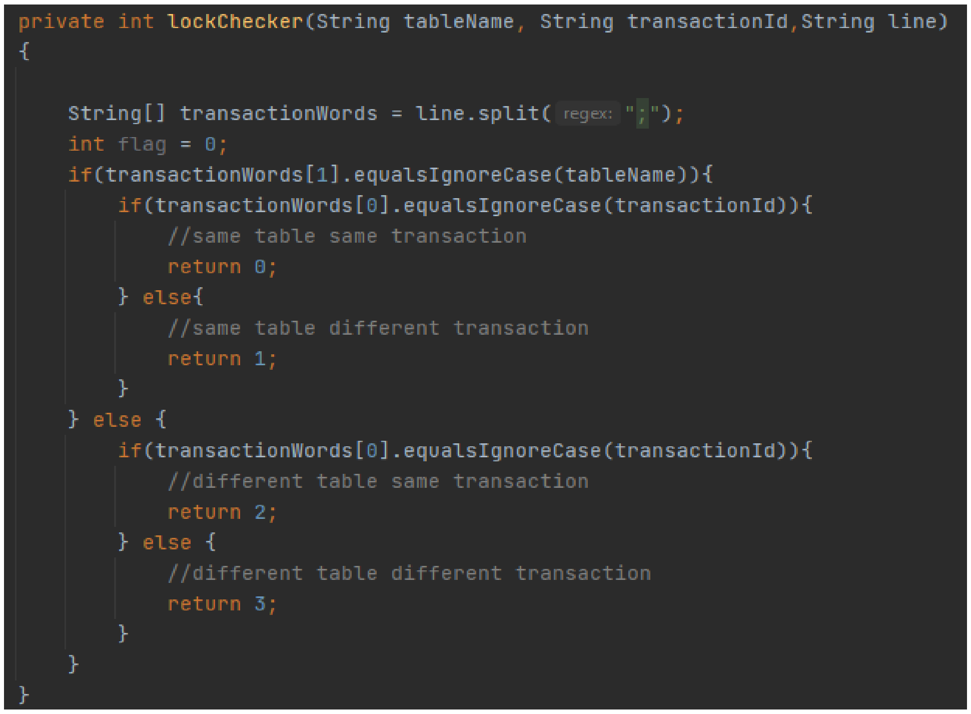
3.2 Lock

When the transaction starts and a query statement is given, the lock on that table will be given to that transaction. After that If any other transaction tries to take a lock on that same table, then it will reject the queries until the first transaction has a lock on that table.



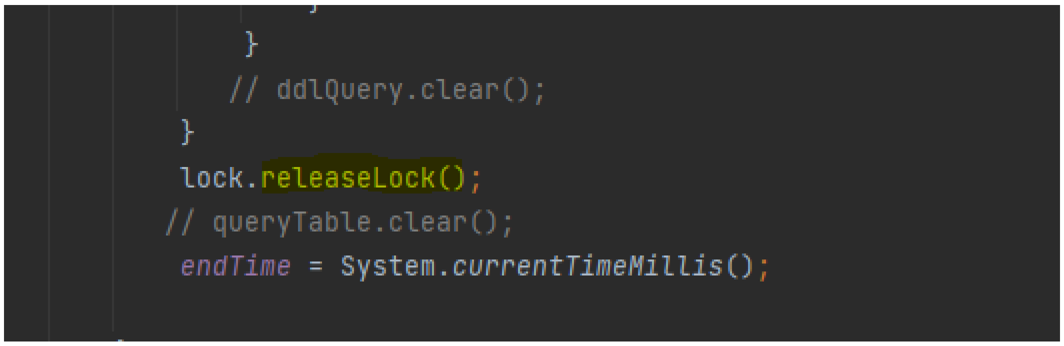
```java
private void makeHashmap(String userName, String query, String transactionId)
{
    try {
        ParseQuery parse = new ParseQuery();
        LockManager lock = new LockManager();

        boolean validation = parse.parseQuery(userName, query);
        if(!validation){
            checker = false;
        }
        boolean locked = lock.acquireLock(query, transactionId);
```

This function (acquireLock()) will pass the query and transaction id to acquire the lock. It will first fetch the table name and then read the "Transaction.txt" file for checking if the same transaction or any other transaction has any lock on the same table. If there is no record found then it will write "transactionid;tablename" in the "Transaction.txt" file.

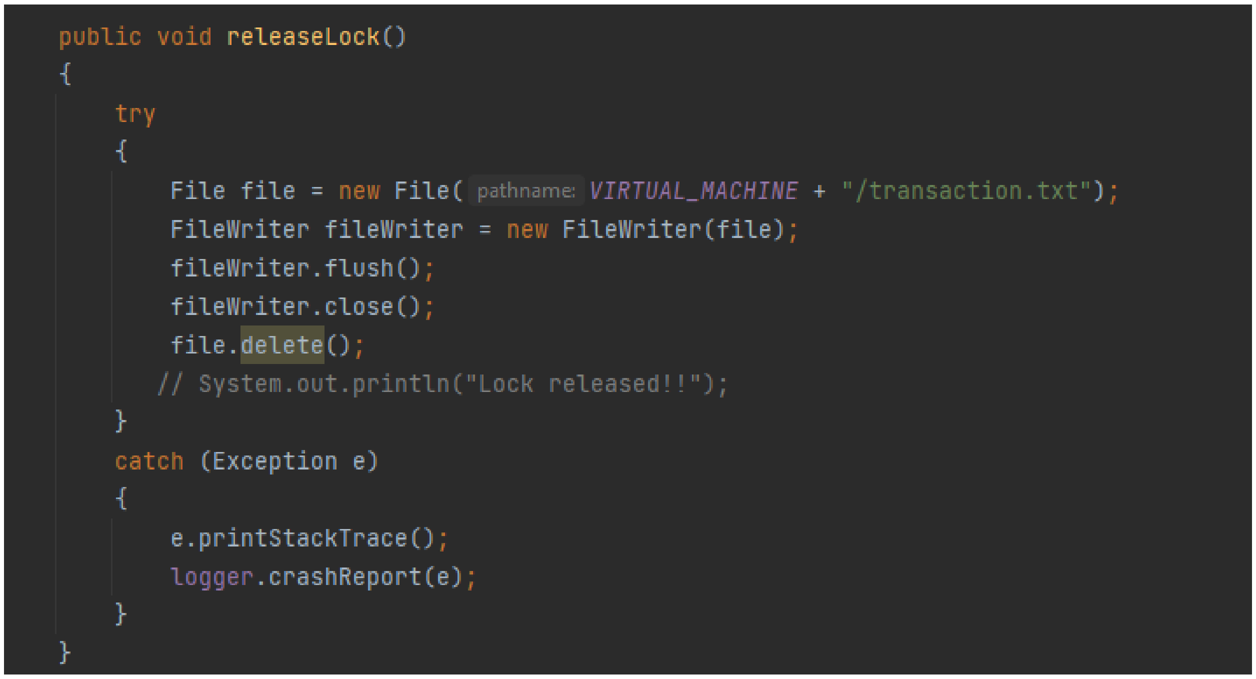


```java
private int lockChecker(String tableName, String transactionId,String line)
{

    String[] transactionWords = line.split( regex: ";");
    int flag = 0;
    if(transactionWords[1].equalsIgnoreCase(tableName)){
        if(transactionWords[0].equalsIgnoreCase(transactionId)){
            //same table same transaction
            return 0;
        } else{
            //same table different transaction
            return 1;
        }
    } else {
        if(transactionWords[0].equalsIgnoreCase(transactionId)){
            //different table same transaction
            return 2;
        } else {
            //different table different transaction
            return 3;
        }
    }
}
```

This lockchecker function will check before giving the lock. For releasing the lock, when a user commits or rollback, it will call the releaseLock() function. releaseLock will clear the "Transaction.txt" file and then delete it. So, the next transaction can create a new file and can acquire a lock on the table to perform another transaction.



```
            }
          // ddlQuery.clear();
        }
        lock.releaseLock();
       // queryTable.clear();
        endTime = System.currentTimeMillis();
```



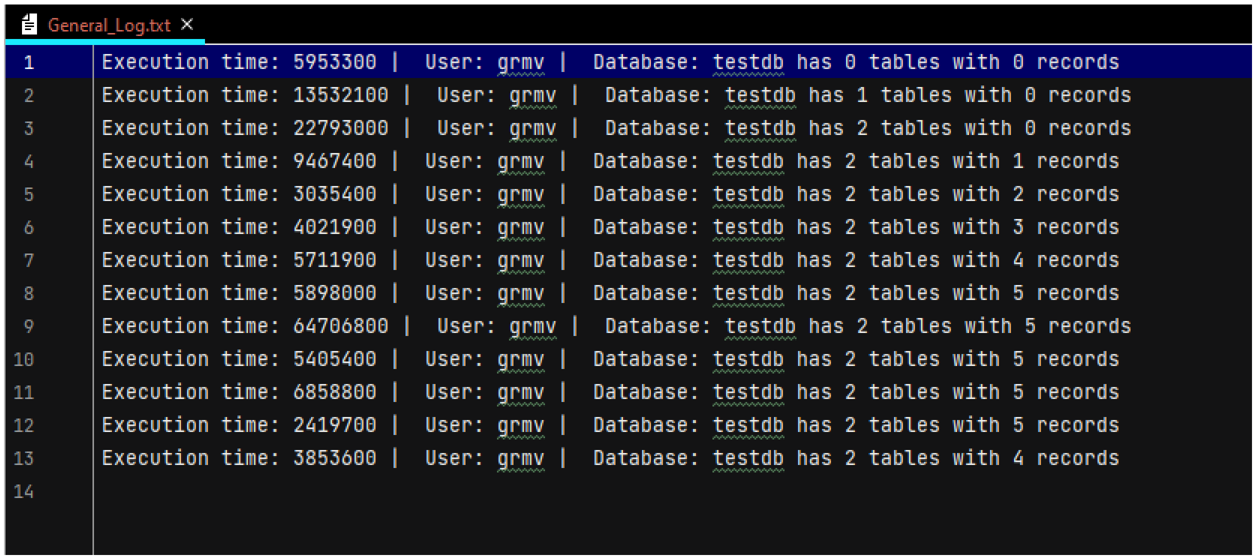
```
public void releaseLock()
{
    try
    {
        File file = new File( pathname: VIRTUAL_MACHINE + "/transaction.txt");
        FileWriter fileWriter = new FileWriter(file);
        fileWriter.flush();
        fileWriter.close();
        file.delete();
       // System.out.println("Lock released!!");
    }
    catch (Exception e)
    {
        e.printStackTrace();
        logger.crashReport(e);
    }
}
```

Module 4: Log Management

This module is used for recording and storing all the logs or events that happen during the query implementation. There are three types of logs:

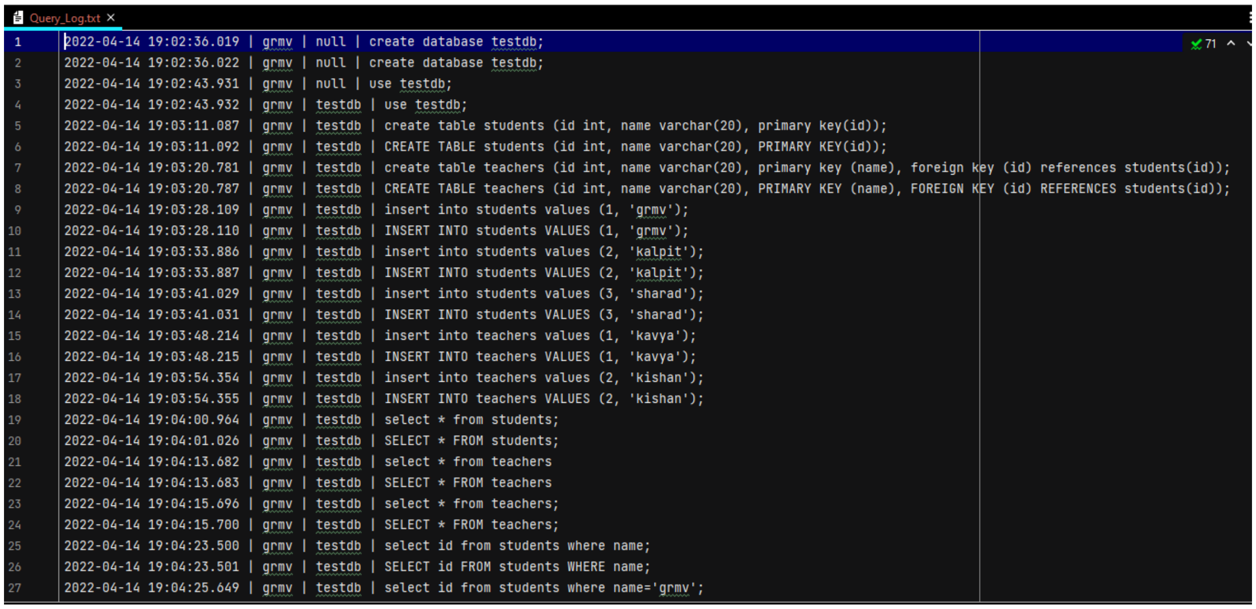
- General logs
- Query logs and
- Event logs

General logs are used to store all the details such as database name, user name, the execution time of each query, and the total number of tables and records that are present at that particular point in time.


General_Log.txt

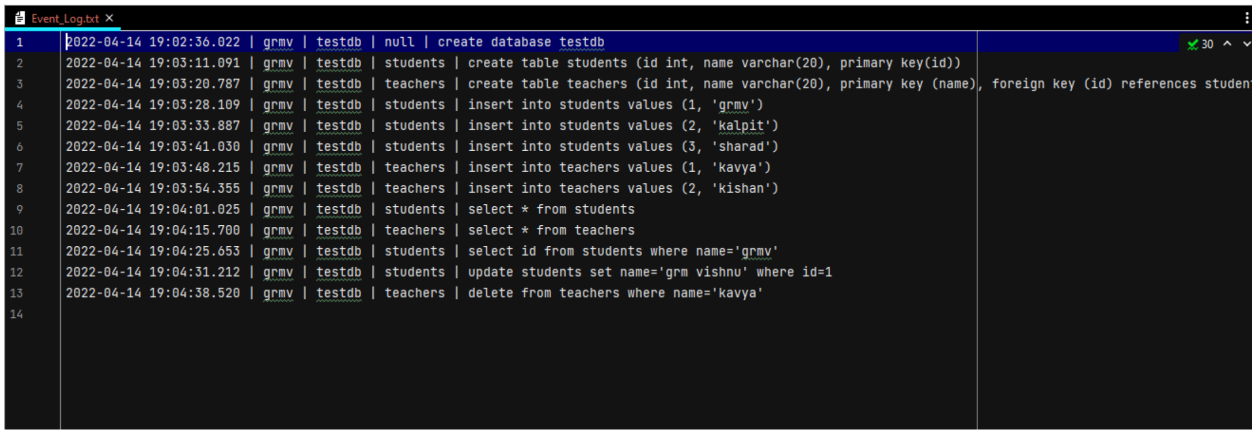
```
Execution time: 5953300 |  User: grmv |  Database: testdb has 0 tables with 0 records
Execution time: 13532100 |  User: grmv |  Database: testdb has 1 tables with 0 records
Execution time: 22793000 |  User: grmv |  Database: testdb has 2 tables with 0 records
Execution time: 9467400 |  User: grmv |  Database: testdb has 2 tables with 1 records
Execution time: 3035400 |  User: grmv |  Database: testdb has 2 tables with 2 records
Execution time: 4021900 |  User: grmv |  Database: testdb has 2 tables with 3 records
Execution time: 5711900 |  User: grmv |  Database: testdb has 2 tables with 4 records
Execution time: 5898000 |  User: grmv |  Database: testdb has 2 tables with 5 records
Execution time: 64706800 |  User: grmv |  Database: testdb has 2 tables with 5 records
Execution time: 5405400 |  User: grmv |  Database: testdb has 2 tables with 5 records
Execution time: 6858800 |  User: grmv |  Database: testdb has 2 tables with 5 records
Execution time: 2419700 |  User: grmv |  Database: testdb has 2 tables with 5 records
Execution time: 3853600 |  User: grmv |  Database: testdb has 2 tables with 4 records
```

Query logs are used to store all the queries that are performed by each user and also the exact time each query is being performed together with the database name, and user name. Query logs capture all the queries even if the query is invalid. So, whatever the user inputs, even if that query is a wrong one, it will be updated in the query log file.



```
2022-04-14 19:02:36.019 | grmv | null | create database testdb;
2022-04-14 19:02:36.022 | grmv | null | create database testdb;
2022-04-14 19:02:43.931 | grmv | null | use testdb;
2022-04-14 19:02:43.932 | grmv | testdb | use testdb;
2022-04-14 19:03:11.087 | grmv | testdb | create table students (id int, name varchar(20), primary key(id));
2022-04-14 19:03:11.092 | grmv | testdb | CREATE TABLE students (id int, name varchar(20), PRIMARY KEY(id));
2022-04-14 19:03:20.781 | grmv | testdb | create table teachers (id int, name varchar(20), primary key (name), foreign key (id) references students(id));
2022-04-14 19:03:20.787 | grmv | testdb | CREATE TABLE teachers (id int, name varchar(20), PRIMARY KEY (name), FOREIGN KEY (id) REFERENCES students(id));
2022-04-14 19:03:28.109 | grmv | testdb | insert into students values (1, 'grmv');
2022-04-14 19:03:28.110 | grmv | testdb | INSERT INTO students VALUES (1, 'grmv');
2022-04-14 19:03:33.886 | grmv | testdb | insert into students values (2, 'kalpit');
2022-04-14 19:03:33.887 | grmv | testdb | INSERT INTO students VALUES (2, 'kalpit');
2022-04-14 19:03:41.029 | grmv | testdb | insert into students values (3, 'sharad');
2022-04-14 19:03:41.031 | grmv | testdb | INSERT INTO students VALUES (3, 'sharad');
2022-04-14 19:03:48.214 | grmv | testdb | insert into teachers values (1, 'kavya');
2022-04-14 19:03:48.215 | grmv | testdb | INSERT INTO teachers VALUES (1, 'kavya');
2022-04-14 19:03:54.354 | grmv | testdb | insert into teachers values (2, 'kishan');
2022-04-14 19:03:54.355 | grmv | testdb | INSERT INTO teachers VALUES (2, 'kishan');
2022-04-14 19:04:00.964 | grmv | testdb | select * from students;
2022-04-14 19:04:01.026 | grmv | testdb | SELECT * FROM students;
2022-04-14 19:04:13.682 | grmv | testdb | select * from teachers
2022-04-14 19:04:13.683 | grmv | testdb | SELECT * FROM teachers
2022-04-14 19:04:15.696 | grmv | testdb | select * from teachers;
2022-04-14 19:04:15.700 | grmv | testdb | SELECT * FROM teachers;
2022-04-14 19:04:23.500 | grmv | testdb | select id from students where name;
2022-04-14 19:04:23.501 | grmv | testdb | SELECT id FROM students WHERE name;
2022-04-14 19:04:25.649 | grmv | testdb | select id from students where name='grmv';
```

Event logs are used to capture all the changes in the database, crash reports, database name, and user name, and also hold all the details of transactions.



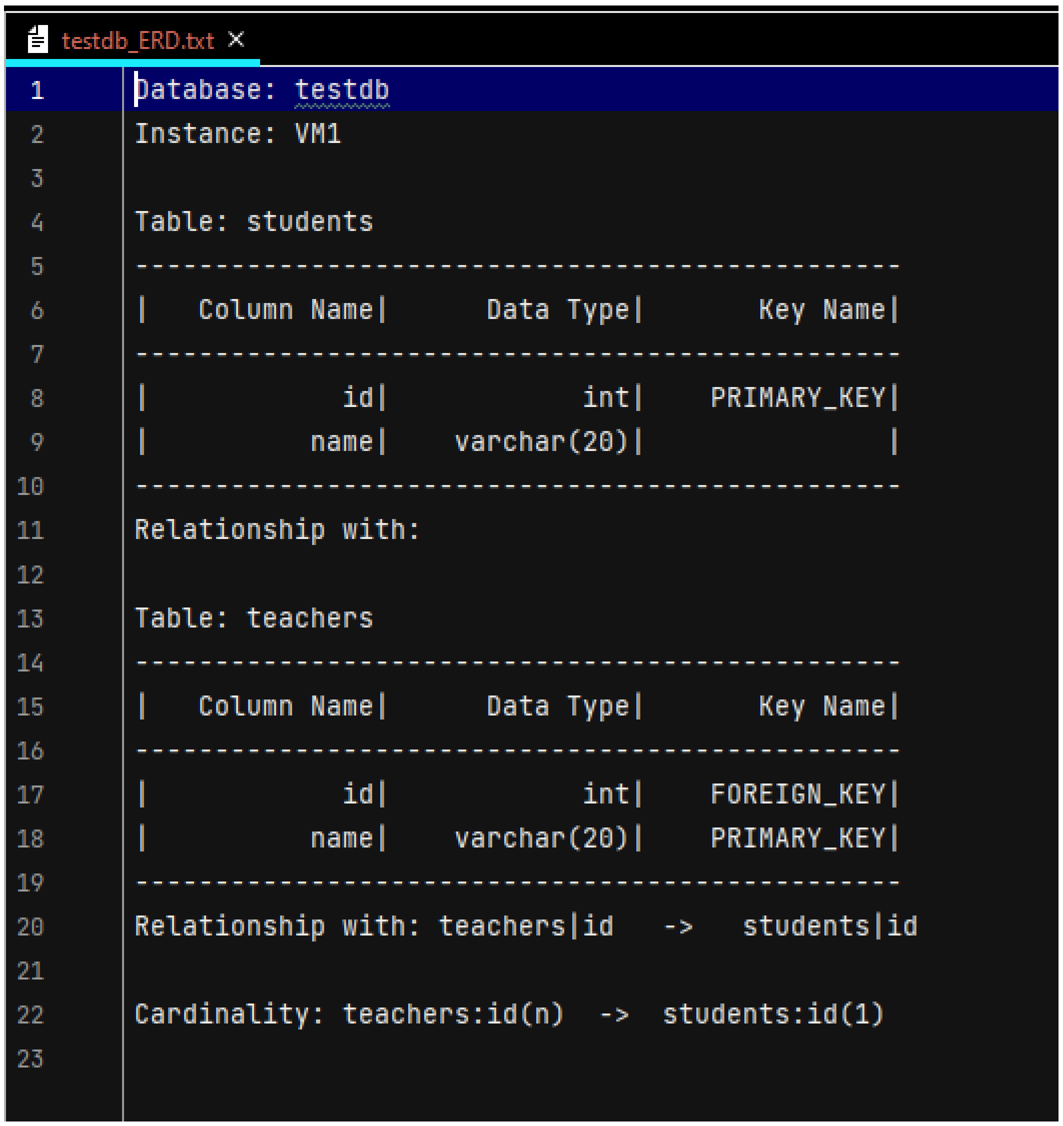
```
2022-04-14 19:02:36.022 | grmv | testdb | null | create database testdb
2022-04-14 19:03:11.091 | grmv | testdb | students | create table students (id int, name varchar(20), primary key(id))
2022-04-14 19:03:20.787 | grmv | testdb | teachers | create table teachers (id int, name varchar(20), primary key (name), foreign key (id) references studen
2022-04-14 19:03:28.109 | grmv | testdb | students | insert into students values (1, 'grmv')
2022-04-14 19:03:33.887 | grmv | testdb | students | insert into students values (2, 'kalpit')
2022-04-14 19:03:41.030 | grmv | testdb | students | insert into students values (3, 'sharad')
2022-04-14 19:03:48.215 | grmv | testdb | teachers | insert into teachers values (1, 'kavya')
2022-04-14 19:03:54.355 | grmv | testdb | teachers | insert into teachers values (2, 'kishan')
2022-04-14 19:04:01.025 | grmv | testdb | students | select * from students
2022-04-14 19:04:15.700 | grmv | testdb | teachers | select * from teachers
2022-04-14 19:04:25.653 | grmv | testdb | students | select id from students where name='grmv'
2022-04-14 19:04:31.212 | grmv | testdb | students | update students set name='grm vishnu' where id=1
2022-04-14 19:04:38.520 | grmv | testdb | teachers | delete from teachers where name='kavya'
```

All the details that are required to be printed in the log files are passed as parameters by each module to the methods implemented in the Log management module. The crash reports method is called whenever an exception occurs in the database.

All the logs are being stored in .txt format and every user can view these logs in the specified file location. These logs help users in understanding where and at what point of time an error is occurred and also the type of error that has been seen. For the implementation of the logging module, we have performed basic file read and write functions without using any built-in libraries or packages.

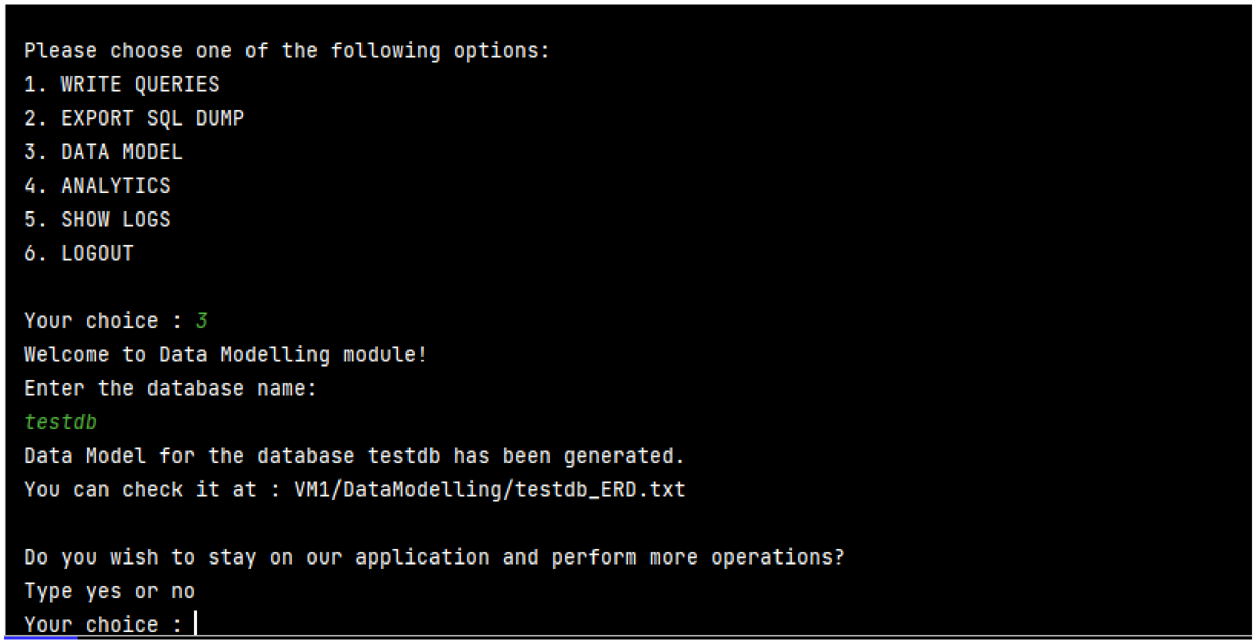
Module 5: Data Modelling – Reverse Engineering

To export an ER diagram, the user will be asked for the database name. If the database with the name user-provided exists, the data dictionary file for that database will be accessed. From this data dictionary, details such as database names, table names, and table attributes are extracted. All this information will then be presented in a tabular structure in the exported .txt file. The below diagram will give you an overview of this module.



```
Database: testdb
Instance: VM1

Table: students
------------------------------------------------
|   Column Name|      Data Type|       Key Name|
------------------------------------------------
|            id|            int|    PRIMARY_KEY|
|          name|    varchar(20)|               |
------------------------------------------------
Relationship with:

Table: teachers
------------------------------------------------
|   Column Name|      Data Type|       Key Name|
------------------------------------------------
|            id|            int|    FOREIGN_KEY|
|          name|    varchar(20)|    PRIMARY_KEY|
------------------------------------------------
Relationship with: teachers|id   ->   students|id

Cardinality: teachers:id(n)  ->  students:id(1)
```

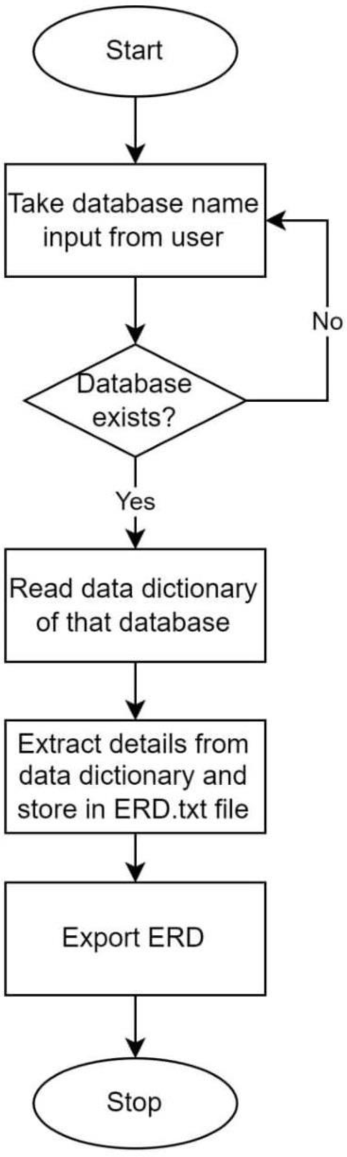
When a user selects the Data model option which is provided as the user-interface option, this module gets executed. Based on the user requirement (Every user is asked to provide a database name when they want to generate an ERD) our application checks if that database is present in the system or not. If the system does not have a database with that particular name, it displays a message to the user saying that the database is not found in the system/database does not exist in the system. If the database exists in the system, then an ERD is generated and the user will be provided with the path which re-directs them to the location where ERD is generated.



```
Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT

Your choice : 3
Welcome to Data Modelling module!
Enter the database name:
testdb
Data Model for the database testdb has been generated.
You can check it at : VM1/DataModelling/testdb_ERD.txt

Do you wish to stay on our application and perform more operations?
Type yes or no
Your choice :
```

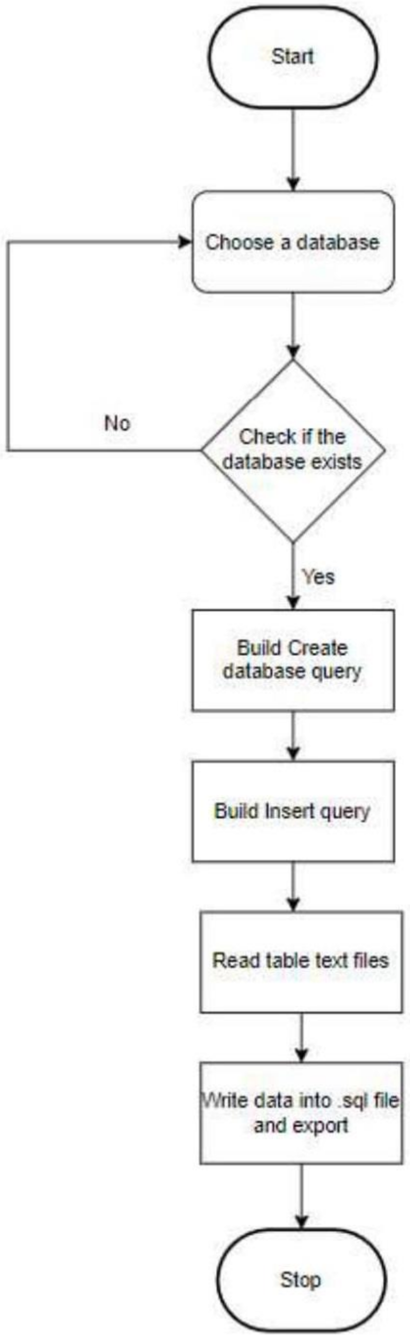
We are storing the generated ERD in a text format, where it prints the database name, table name, columns, datatypes of each column, Key constraints, Relationship with other tables, and also cardinality. Initially, we are checking if the database exists in the VM folder and also if it exists in Global Data Dictionary. If the database exists in both then we are reading the metadata file. Later, we are splitting the data and are storing it in the string that we are accessing later to print the ERD structure. Cardinality is achieved by checking the existence of a foreign key in a table with duplicate values (n) which are pointing to a table with the same key as the primary key (1), here one-to-many is achieved. Similarly, if the foreign key does not have any duplicate values in the table then we are achieving one-to-one.

Following is a flowchart for generating an Entity-Relationship diagram:

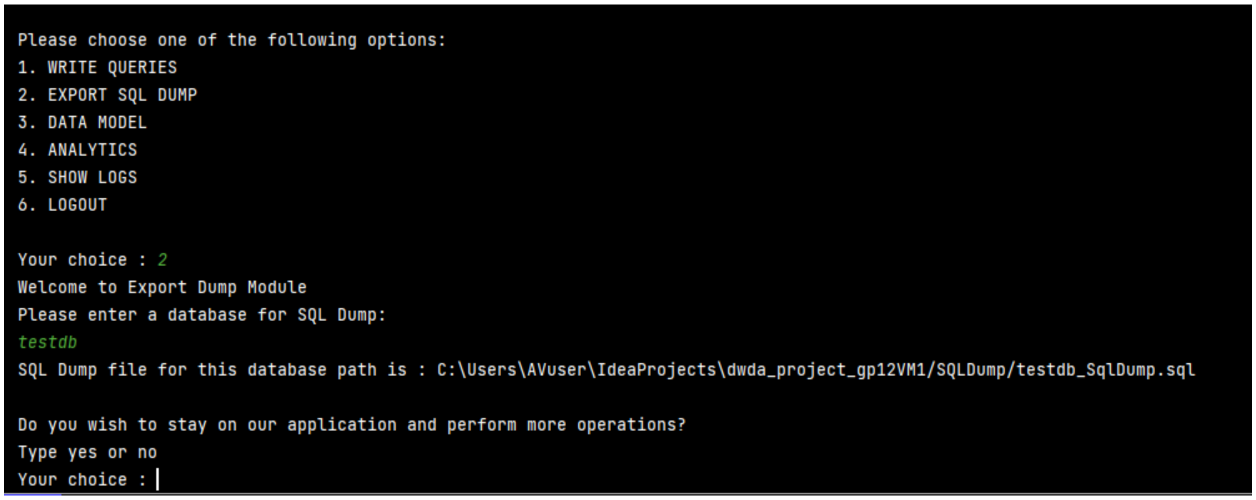

Start
Take database name input from user
No
Database exists?
Yes
Read data dictionary of that database
Extract details from data dictionary and store in ERD.txt file
Export ERD
Stop

Module 6: Export Structure and Value

Flowchart of the process of this module can be understood with the help of this diagram.

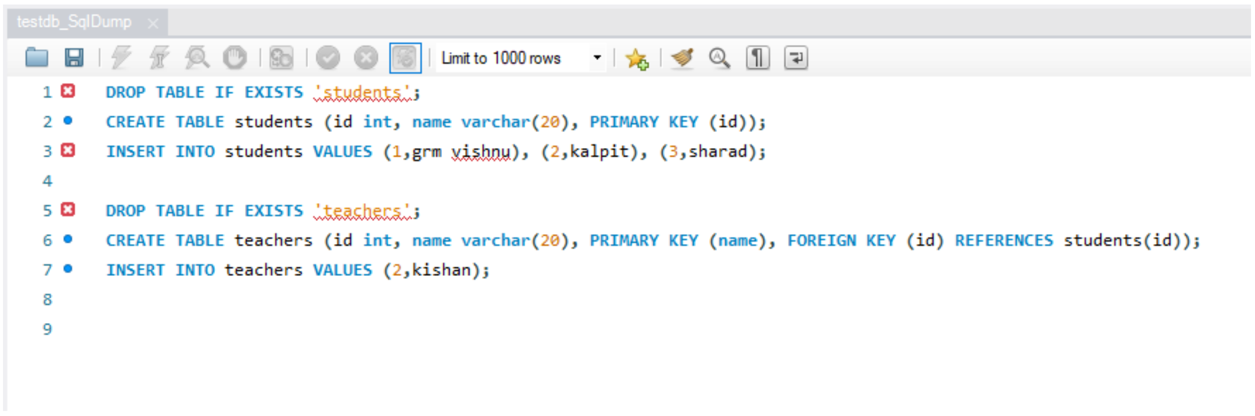

Start
Choose a database
Check if the database exists
No
Yes
Build Create database query
Build Insert query
Read table text files
Write data into .sql file and export
Stop

Every user is provided with an option of exporting an SQL dump file of the database he/she requires. When a user opts for exporting the data dump, the system initially checks if that particular database exists or not. If the database does not exist, it displays that database does not exist. If the database exists in the system if-then exports the dump file of that database and stores it in the specified path. It also displays the specified path file to the user for convenience.



```
Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT

Your choice : 2
Welcome to Export Dump Module
Please enter a database for SQL Dump:
testdb
SQL Dump file for this database path is : C:\Users\AVuser\IdeaProjects\dwda_project_gp12VM1/SQLDump/testdb_SqlDump.sql

Do you wish to stay on our application and perform more operations?
Type yes or no
Your choice :
```

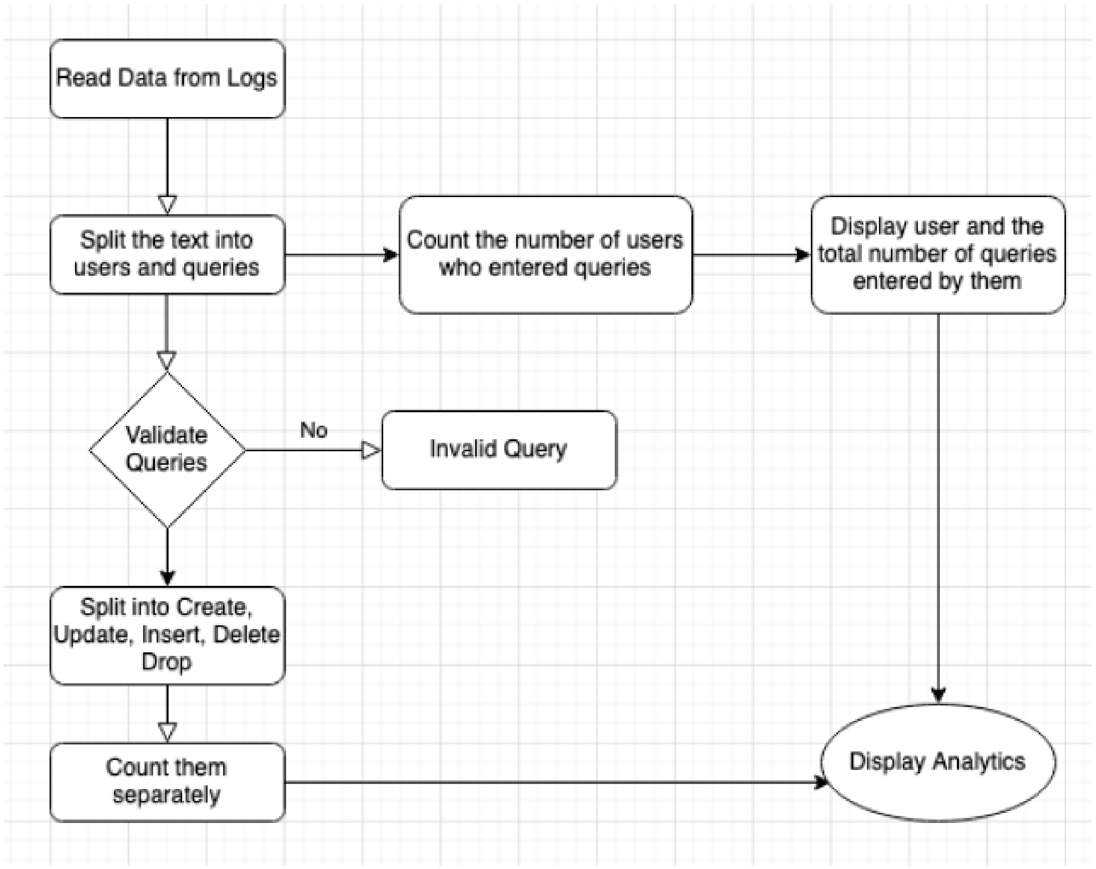
Each dump file that is generated is stored in .sql format and it captures the current state of the database. Here, we haven't used any external packages for the generation of the dump file and have included the generation of the create and insert statements dynamically.



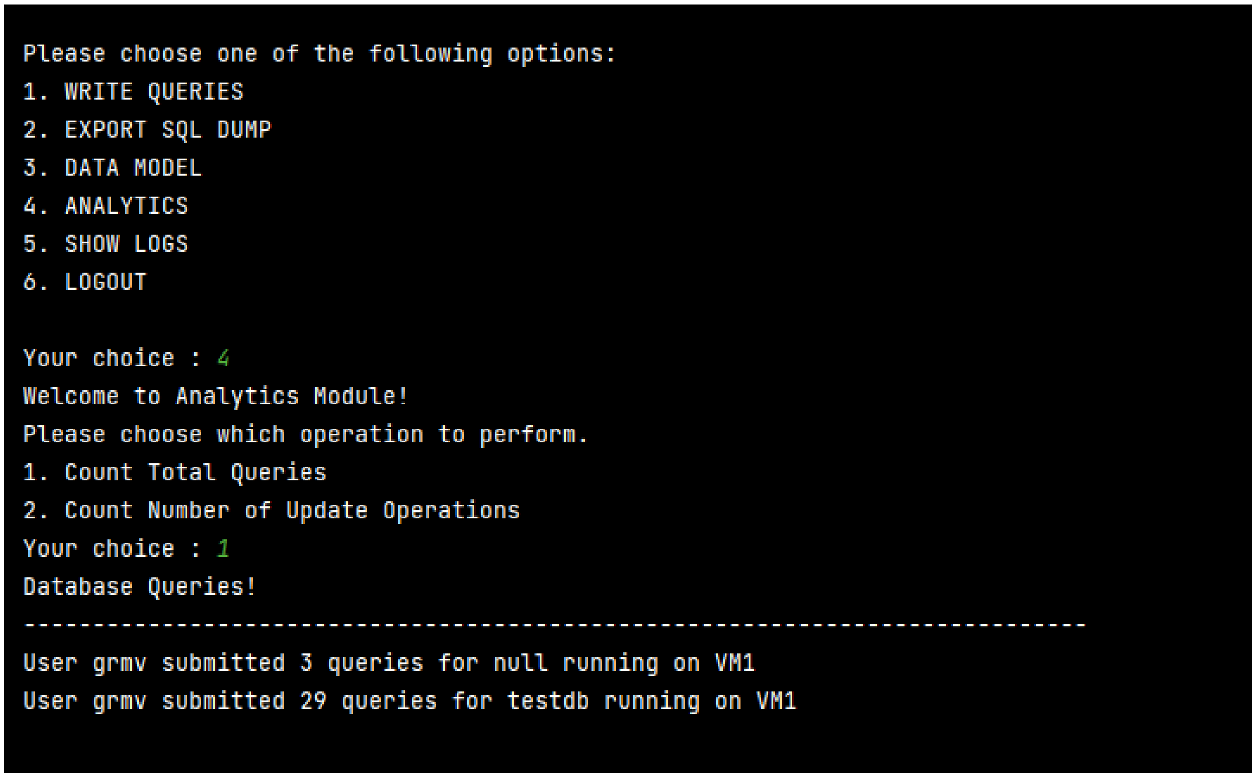
```
DROP TABLE IF EXISTS 'students';
CREATE TABLE students (id int, name varchar(20), PRIMARY KEY (id));
INSERT INTO students VALUES (1,grm vishnu), (2,kalpit), (3,sharad);

DROP TABLE IF EXISTS 'teachers';
CREATE TABLE teachers (id int, name varchar(20), PRIMARY KEY (name), FOREIGN KEY (id) REFERENCES students(id));
INSERT INTO teachers VALUES (2,kishan);
```

Module 7: Analytics

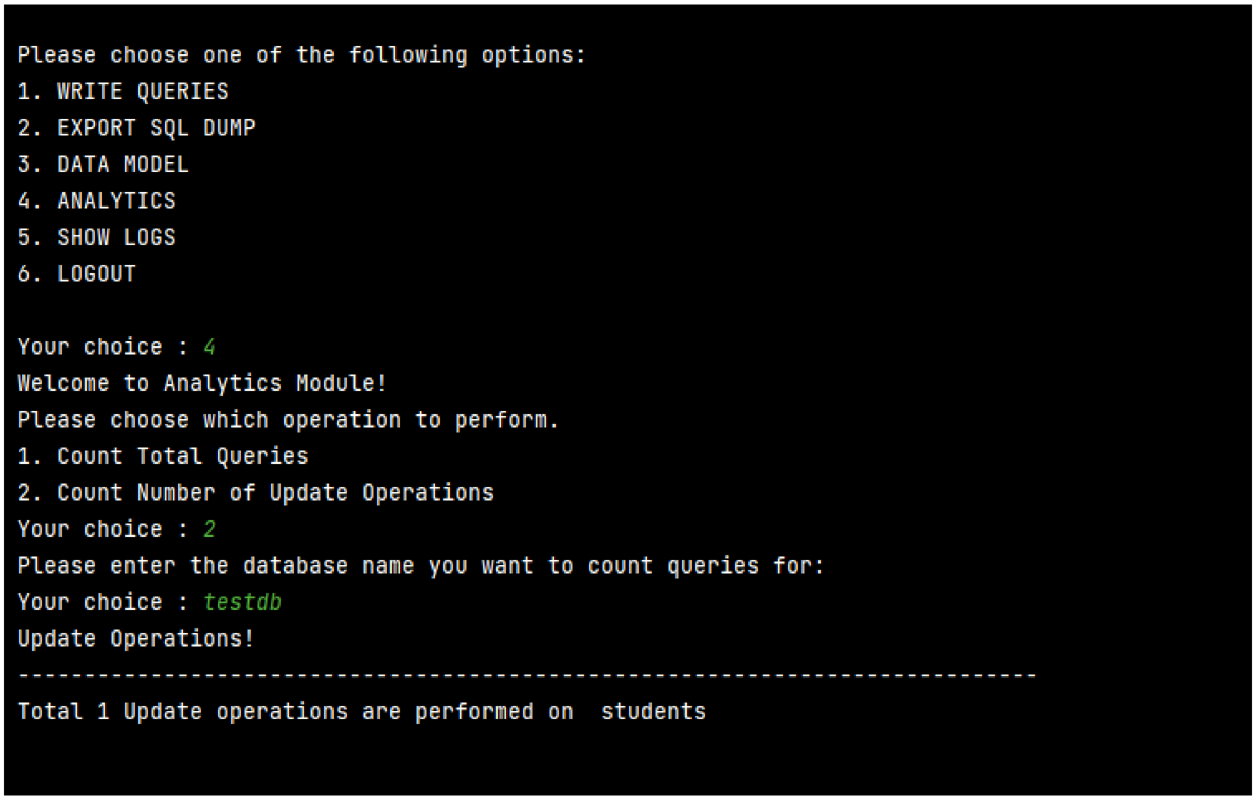
The flowchart of the process of this module can be understood with the help of this diagram.


Read Data from Logs
Split the text into users and queries
Count the number of users who entered queries
Display user and the total number of queries entered by them
Validate Queries
No
Invalid Query
Split into Create, Update, Insert, Delete Drop
Count them separately
Display Analytics

Our application provides analytics when a user selects an option that is provided for analytics. It displays the number of queries that are either valid or invalid that have been executed on a particular database. If the database does not exist in the system, then the user will be displayed that the database does not exist in the system or if it exists, it displays using the specific file path that is being used to store analytics.

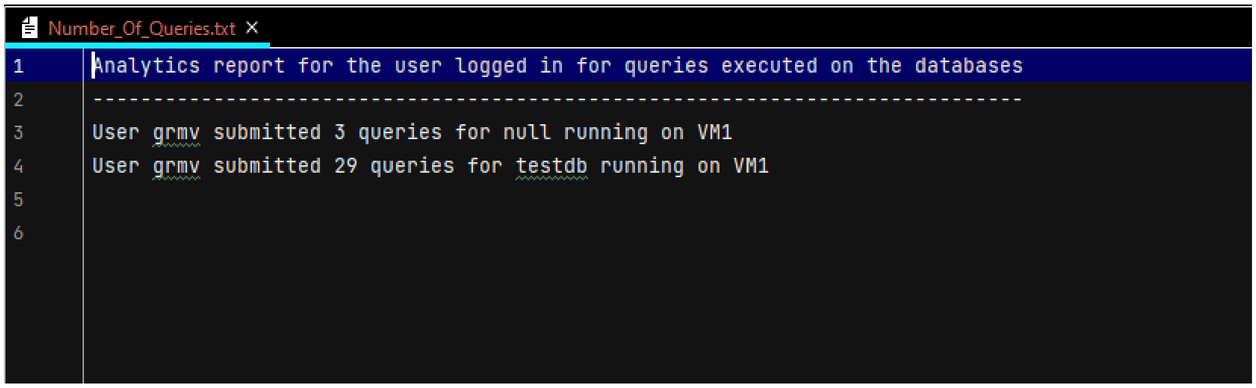


```
Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT

Your choice : 4
Welcome to Analytics Module!
Please choose which operation to perform.
1. Count Total Queries
2. Count Number of Update Operations
Your choice : 1
Database Queries!
-----------------------------------------------------------------------------
User grmv submitted 3 queries for null running on VM1
User grmv submitted 29 queries for testdb running on VM1
```



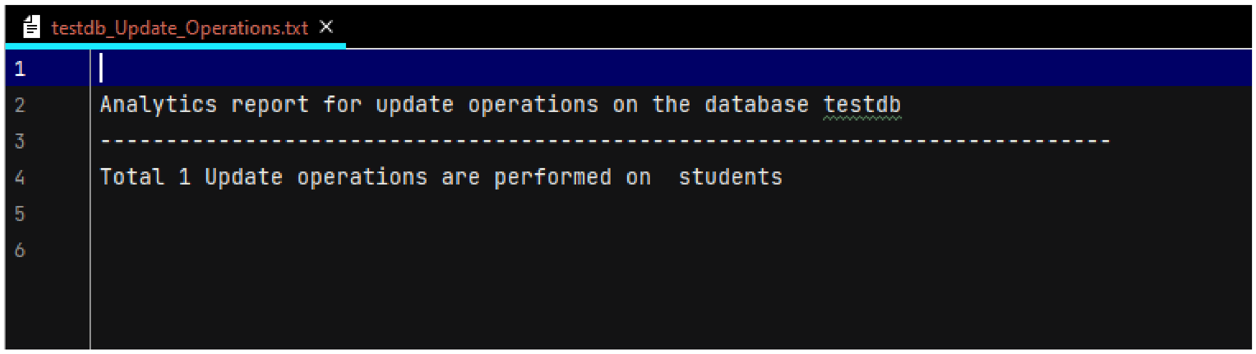
```
Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT

Your choice : 4
Welcome to Analytics Module!
Please choose which operation to perform.
1. Count Total Queries
2. Count Number of Update Operations
Your choice : 2
Please enter the database name you want to count queries for:
Your choice : testdb
Update Operations!
-----------------------------------------------------------------------------
Total 1 Update operations are performed on  students
```

The analytics required are stored in .txt format and it has the total queries that are performed on the database (both valid and invalid) together with the count of update operations that are performed on the database table.



Number_Of_Queries.txt

```
Analytics report for the user logged in for queries executed on the databases
-----------------------------------------------------------------------------
User grmv submitted 3 queries for null running on VM1
User grmv submitted 29 queries for testdb running on VM1
```

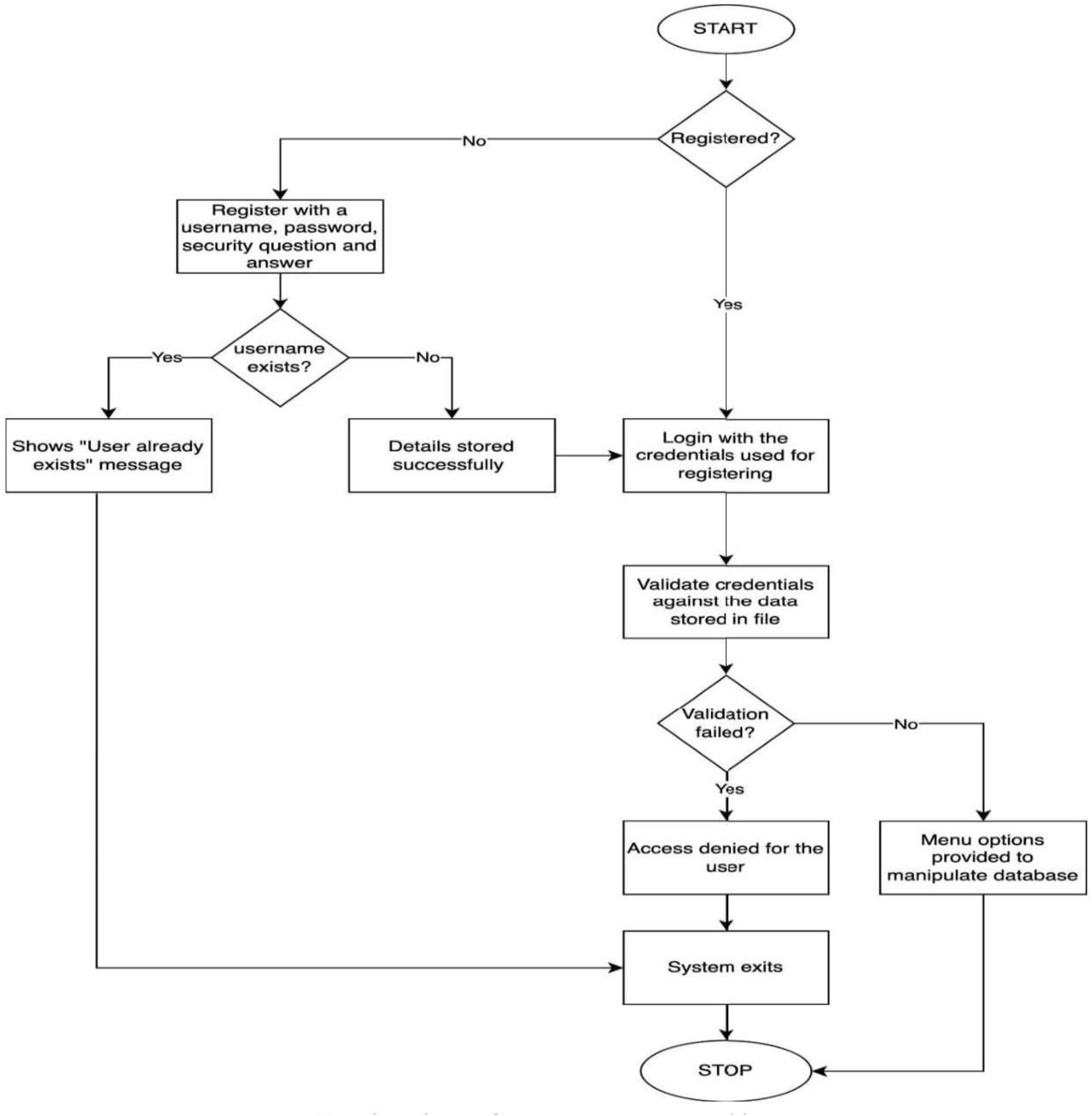


testdb_Update_Operations.txt

```

Analytics report for update operations on the database testdb
-----------------------------------------------------------------------------
Total 1 Update operations are performed on  students
```

Module 8: User Interface and Login Security

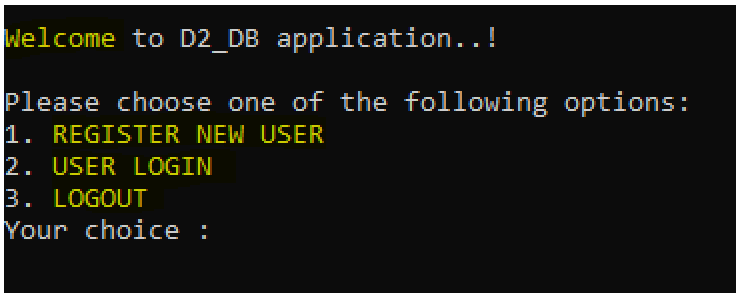
The flowchart of the process of this module can be understood with the help of this diagram.


START
Registered?
No
Yes
Register with a username, password, security question and answer
username exists?
Yes
No
Shows "User already exists" message
Details stored successfully
Login with the credentials used for registering
Validate credentials against the data stored in file
Validation failed?
No
Yes
Access denied for the user
Menu options provided to manipulate database
System exits
STOP

For achieving the project requirements, we have implemented a console-based user interface that is very easy to understand and provides all database options that a user should get.

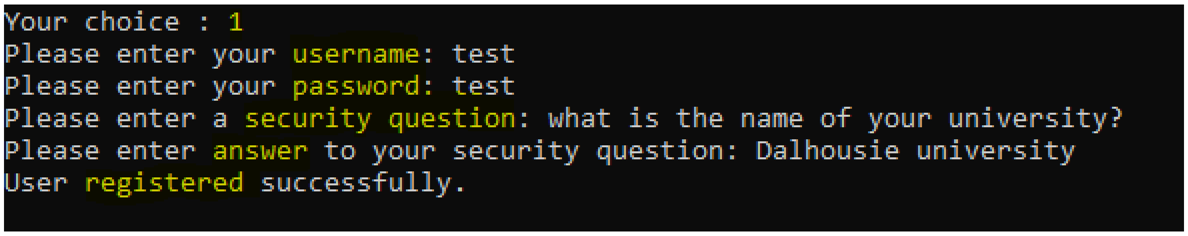
When this project starts it firstly displays three options with a welcome message:

1. Register New User
2. User Login
3. Logout



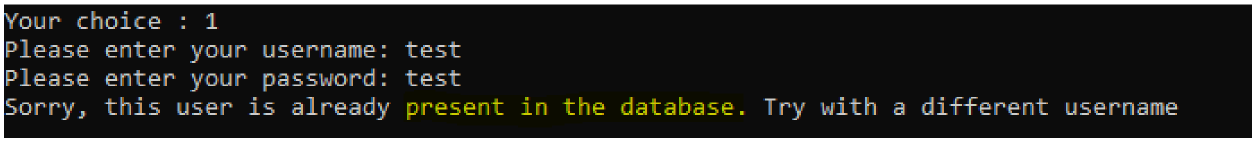
```
Welcome to D2_DB application..!

Please choose one of the following options:
1. REGISTER NEW USER
2. USER LOGIN
3. LOGOUT
Your choice :
```

Here the program will wait for the user's choice (1,2 or 3). If the user is new then they have to select 1 to register himself as a new user. While registering, users have to provide a unique username, password, and a security question and answer.



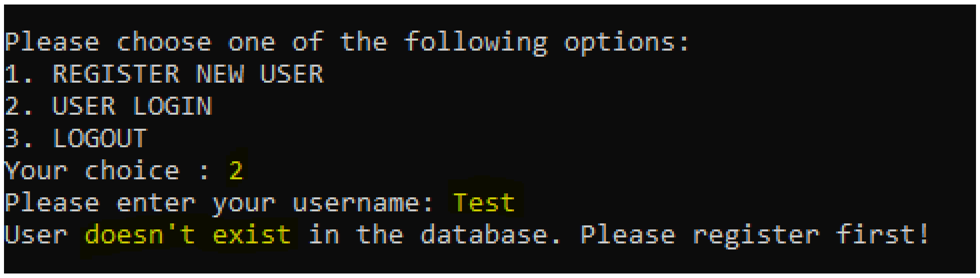
```
Your choice : 1
Please enter your username: test
Please enter your password: test
Please enter a security question: what is the name of your university?
Please enter answer to your security question: Dalhousie university
User registered successfully.
```

If the username is unique and all the answers are valid, then it will register the user successfully. If a user tries to take the existing username, then it will prevent the registration.

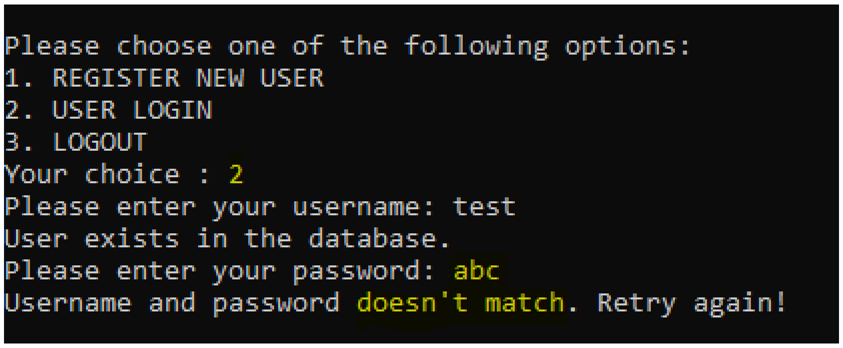


```
Your choice : 1
Please enter your username: test
Please enter your password: test
Sorry, this user is already present in the database. Try with a different username
```

After successful registration, the username and password are hashed and stored in "User_Profile.txt" with the security question and its answer. After this registration user has to login using that username and password.

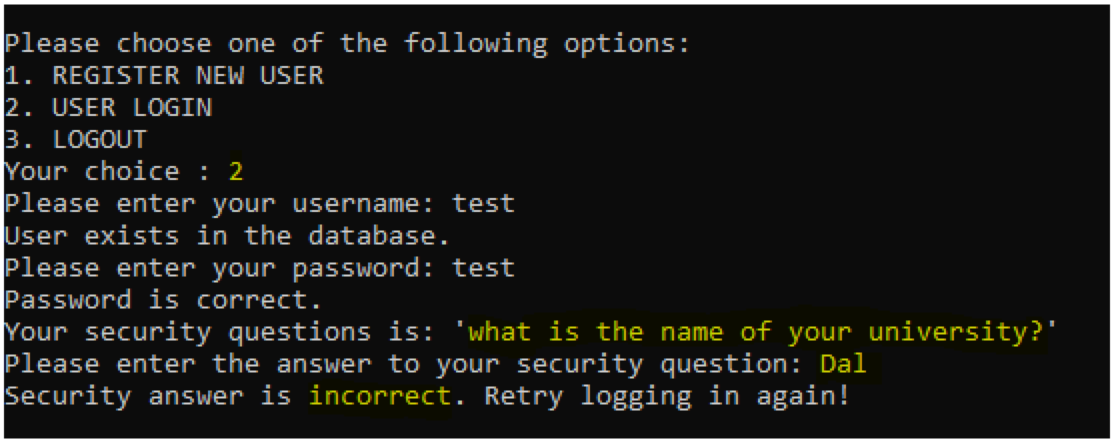


```
Please choose one of the following options:
1. REGISTER NEW USER
2. USER LOGIN
3. LOGOUT
Your choice : 2
Please enter your username: Test
User doesn't exist in the database. Please register first!
```



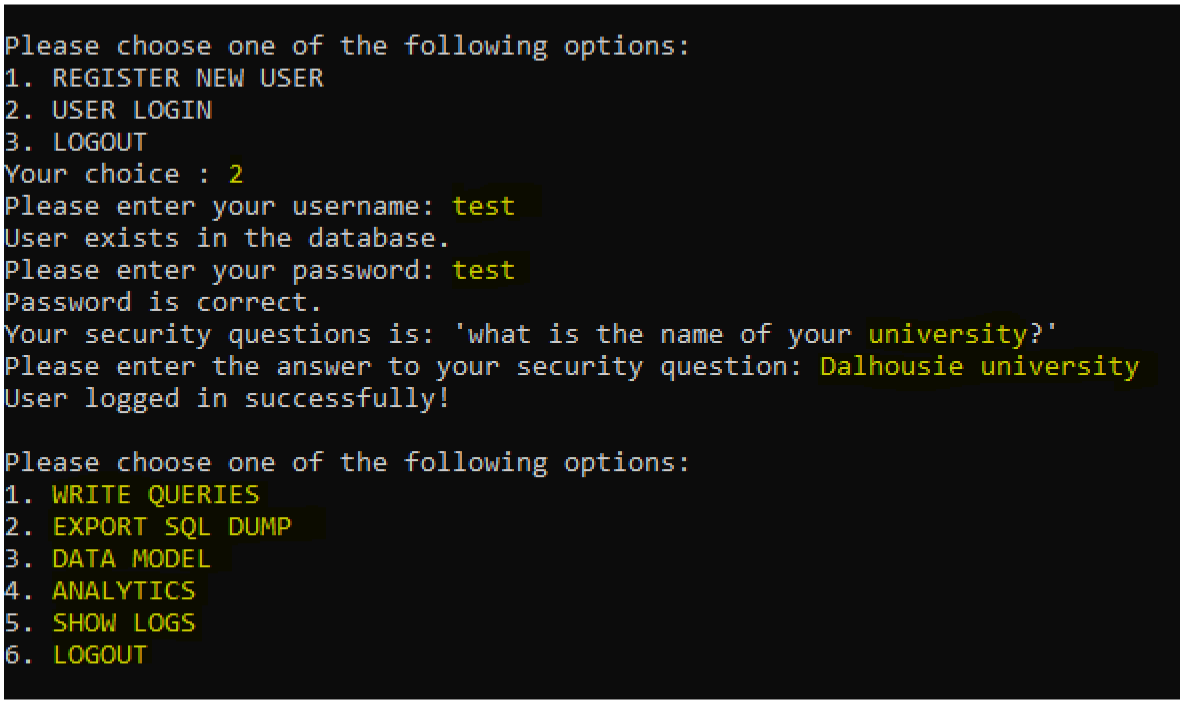
```
Please choose one of the following options:
1. REGISTER NEW USER
2. USER LOGIN
3. LOGOUT
Your choice : 2
Please enter your username: test
User exists in the database.
Please enter your password: abc
Username and password doesn't match. Retry again!
```

If the user provides the wrong username or password then it will display the error message. Additionally, If the user entered the wrong answer to the security question, then it will prevent the user from login.



```
Please choose one of the following options:
1. REGISTER NEW USER
2. USER LOGIN
3. LOGOUT
Your choice : 2
Please enter your username: test
User exists in the database.
Please enter your password: test
Password is correct.
Your security questions is: 'what is the name of your university?'
Please enter the answer to your security question: Dal
Security answer is incorrect. Retry logging in again!
```

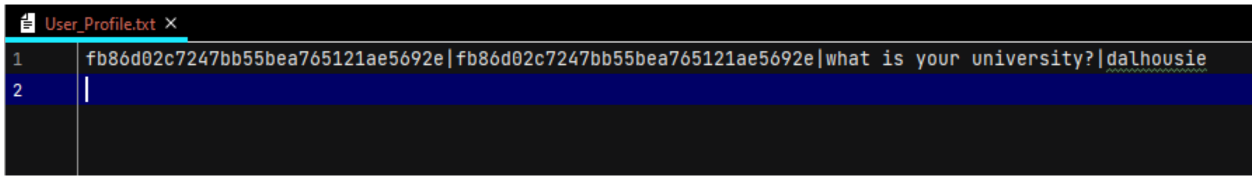
If every input login is correct then it will allow the user to go inside the database and show 6 different options.



```
Please choose one of the following options:
1. REGISTER NEW USER
2. USER LOGIN
3. LOGOUT
Your choice : 2
Please enter your username: test
User exists in the database.
Please enter your password: test
Password is correct.
Your security questions is: 'what is the name of your university?'
Please enter the answer to your security question: Dalhousie university
User logged in successfully!

Please choose one of the following options:
1. WRITE QUERIES
2. EXPORT SQL DUMP
3. DATA MODEL
4. ANALYTICS
5. SHOW LOGS
6. LOGOUT
```

For the security of the user, we are encrypting not only passwords but also usernames. Here we have used a hashing algorithm for encryption. When a user enters the username and password, the code will encrypt them and then compare them with the existing encrypted username and password. If a match is found then it will allow the user to go further in the application. Otherwise, it will give an error message.

An overview of how the user profile file looks is as below.



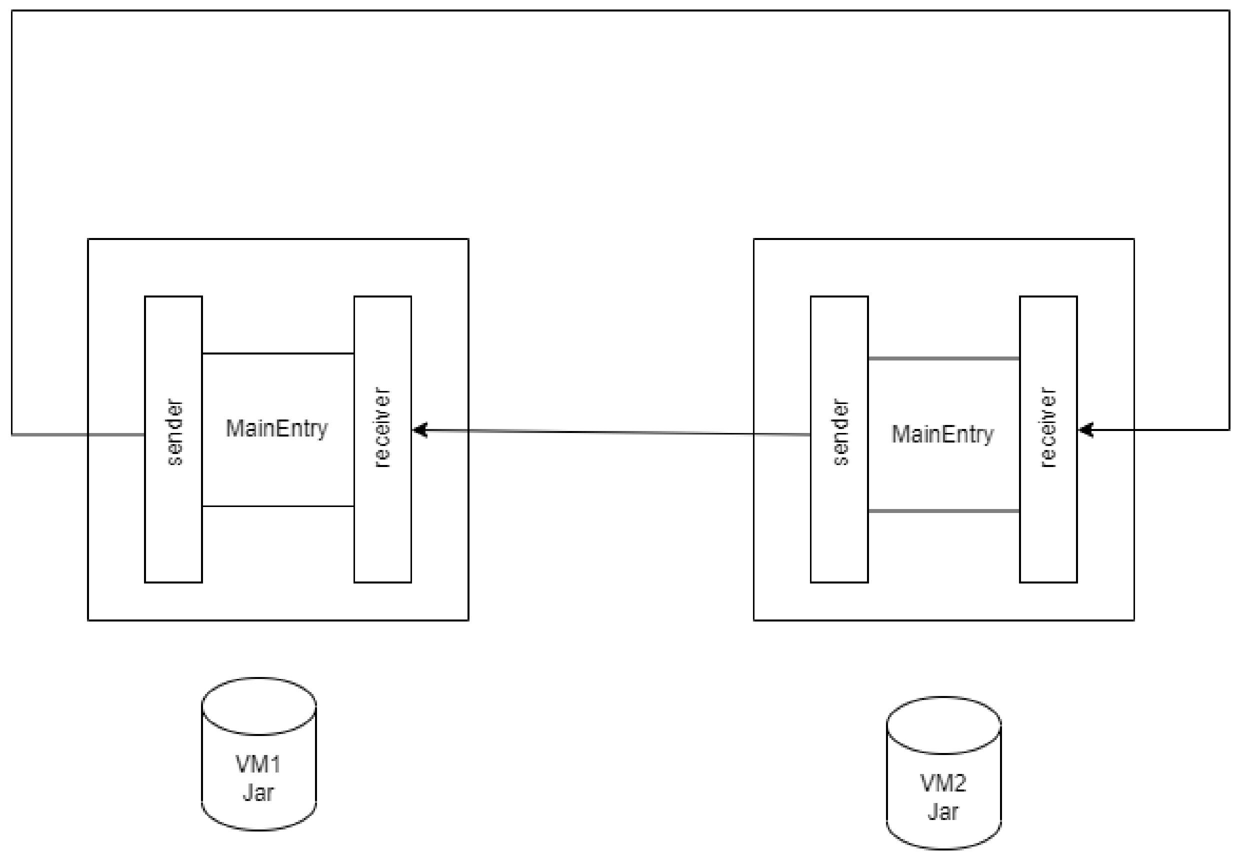
```
fb86d02c7247bb55bea765121ae5692e|fb86d02c7247bb55bea765121ae5692e|what is your university?|dalhousie

```

Conclusion

This program, like the true database management system, allows users to write and view the database. The user can create queries, execute analysis checks, read logs, export data dictionaries, and extract data models once they have successfully logged in. Parsing for various input kinds, including insert, update, delete, create, and drop, maybe done effortlessly with the help of a query parser. The input distinguishes between a transaction and an ordinary query. Logs are generated every time a database operation is done. To facilitate portability, the database can be exported as a SQL dump. The system also allows for database-specific ERD creation.

Future Scope

Communication using JAR file



We tried to communicate between two instances using JAR files.

Execution Flow

1. Create two identical jar files. One for the first virtual machine and another for the second virtual machine.
2. ssh into a preferred virtual machine
3. Execute the query on the jar file
4. If no files are present, generate the base file structure
5. If required files are present on the current virtual machine, execute the query
6. If files are not present on the current virtual machine, establish a connection with the second virtual machine and then execute the query
7. Return the result back to the calling virtual machine

Implementation

This follows a sender and receiver approach where two identical JAR files communicate with each other in two instances. When we ssh into a first instance and try to execute a query on the custom database system, we first investigate the file systems which are available on the current instance. If the required information is found, the query is executed, and the result is returned. Now, if the required files are not found on the current instance, we do not stop the execution immediately, and instead, we try to give a second try on another instance available. To perform this operation, we have defined the sender and receiver class and a single point of entry for the main class. The query is always prefixed with an identifier which indicates whether the call is from another jar or direct from the user. If the query is from the user, we follow the normal login flow and let the user decide the next operation. In case of search operation from another jar, we bypass the initial authentication process and directly execute the module which is related to the query being executed.

To achieve this, we have added sender and receiver classes. The sender class is used to transfer the query from one virtual machine to another using the external IP address. We transmit the command with an identifier to let another instance that the query is being sent from another virtual machine and should be given direct access to the module concerned. After execution is complete, the sender class of the second instance will be used to transmit back to the caller jar the results of the query which was being executed.

Approach

- Connection was established using JSch
- It required setting up of JSch dependency in the POM file
- External IP address of receiver virtual machine
- Password for receiver virtual machine
- Port 22 was exposed for connection

Achieved Task

The following tasks were successful

- Establish connection with first virtual machine
- Establish a connection with a second virtual machine
- Execute query on a first virtual machine using ssh
- Create basic file structure on the first virtual machine with a return response
- Execute query on second virtual machine using ssh
- Create a basic file structure on second virtual machine with a return response

Failed Task

- Establish connection between from sender and receiver jar

Meeting Logs

We had several internal meetings which are both online and offline along with one meeting which we had with our TA and another one with the professor.

2/2/22:

- Introduction
- Discussed project requirements.
- Divided modules amongst everyone for in-depth requirement gathering.

5/2/22:

- Had a call where all the teammates explained all the requirements and contents that are needed to execute the assigned modules.
- Decided on the data structure and also the file structure.
- For the feasibility report, we have divided the contents among the individuals.

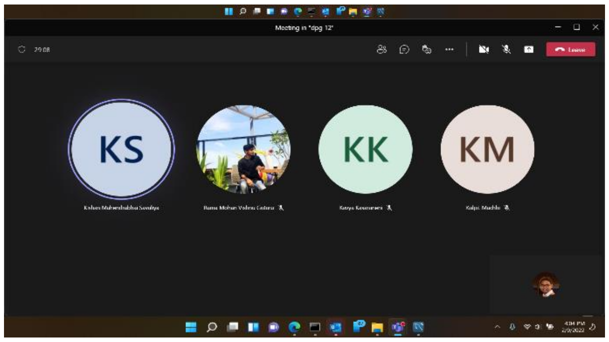
15/2/22:

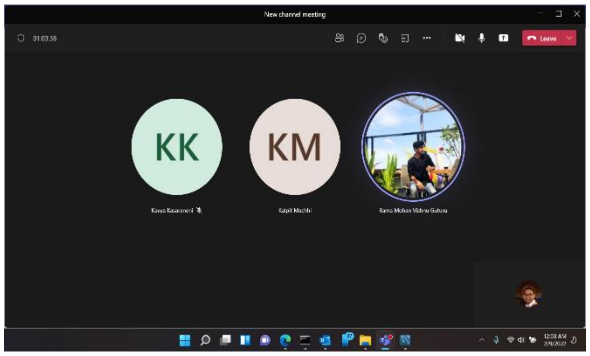
- Had an internal meeting and briefed the remaining doubts.
- Requested both TA and professor for meetings.

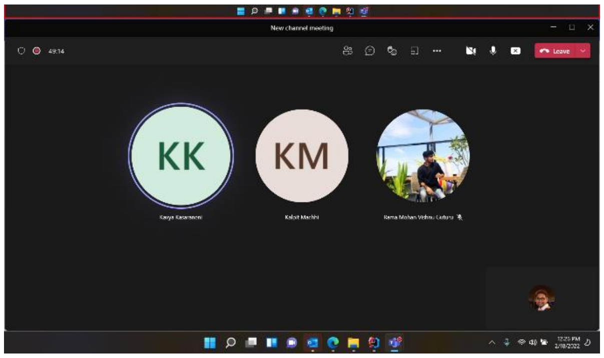
11/3/21:

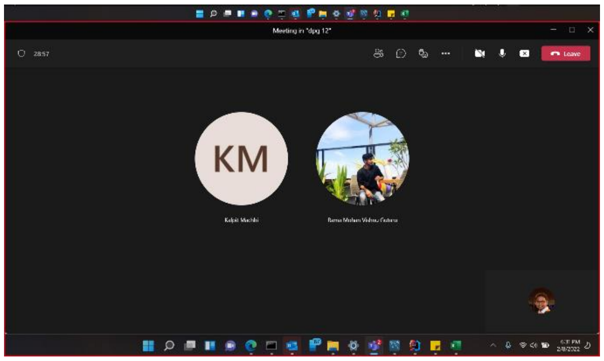
- Had Meeting with professor and finalized all the requirements needed.
- Divided all the modules amongst everyone and started the project implementation.

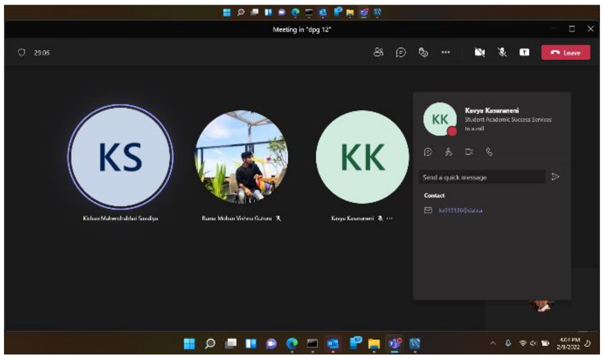
Meeting in "dpg 12"
28:48
Leave
KM
Kalpit Machhi
Rama Mohan Vishnu Guturu


Meeting in "dpg 12"
29:08
Leave
KS
KK
KM
Kishan Maheshbhai Sasidiya
Rama Mohan Vishnu Guturu
Kavya Kasaraneni
Kalpit Machhi


New channel meeting
01:03:38
Leave
KK
KM
Kavya Kasaraneni
Kalpit Machhi
Rama Mohan Vishnu Guturu


New channel meeting
49:14
Leave
KK
KM
Kavya Kasaraneni
Kalpit Machhi
Rama Mohan Vishnu Guturu


Meeting in "dpg 12"
28:57
Leave
KM
Kalpit Machhi
Rama Mohan Vishnu Guturu


Meeting in "dpg 12"
29:06
Leave
KS
KK
Kishan Maheshbhai Sasidiya
Rama Mohan Vishnu Guturu
Kavya Kasaraneni
Kavya Kasaraneni
Student Academic Success Services
In a call
Send a quick message
Contact