CSCI 5408
DATA MANAGEMENT
AND
WAREHOUSING

ASSIGNMENT - 1

Submitted By: Kenil Shaileshkumar Patel
(kenil.patel@dal.ca)
Banner ID: B00954251
Submitted On: October 28, 2023

Gitlab Repository Link

https://git.cs.dal.ca/kenil/csci5408_f23_b00954251_kenil_patel/-/tree/main/A1

Table of Contents

1. Problem Statement 1

Perform a systematic literature review and document your findings with critical analysis.

1.1 Research Paper Analysis

The research paper's main idea is to address the issues of recoverability in federated database systems (FDBS) [1]. FDBS are collections of cooperating, heterogeneous, and autonomous database systems that are distributed across a computer network [1]. When there is a failure, the system faces unique challenges in terms of ensuring basic Database Management System properties such as data consistency and recovery. The purpose of the paper is to propose an enhanced algorithm and architecture for dealing with the recoverability issue in a complex structure [1].

Problem Addressed: The difficulties of preserving data integrity, consistency, and availability within federated database systems is the fundamental issue addressed in this research paper [1]. Multiple databases, typically geographically scattered, collaborate while being potentially independent and heterogeneous. In such scenarios, ensuring data dependability and rapid recovery from errors is a significant task.

Literature Review: The paper references various works related to the field, but it does not provide an in-depth analysis or synthesis of existing literature on the topic. In the paper," Related Work" section is present which provides some background information related to the topic [1]. This section mentions the basics of federated databases and, the benefits of using distributed databases. Instead, it primarily presents the proposed solution and its implementation.

Success and Shortcomings: The study successfully provides the difficulties posed by federated database systems. It proposes an improved algorithm and architecture to tackle a complex issue faced by the system [1]. However, it is important that analysing the success of the research and identification of shortcomings require actual research findings and results, which are not mentioned in the paper.

Without having empirical proof, it is difficult to determine the degree of success of the proposed algorithm or to find the potential shortcomings. A thorough examination of the research methodology, practical results, and explanations of experimental implications of the algorithm would be required for analysis of the research performance and limitations

Room for Improvement: There is still much potential for development in terms of actual implementation and numerical evaluation of the suggested algorithm. They have only given one real-world case study but it can be expanded through various different cases which can be used to validate the effectiveness and applicability of the proposed methodologies.

The paper follows a structured flow, beginning with an introduction that emphasizes the significance of the problem and ending with a discussion of related work [1]. It then examines distributed database characteristics, recovery techniques, and the design and implementation of the proposed solution, which includes components such as the Local Transaction Manager (LTA) and Sync Coordinator [1]. Moreover, to improve its credibility and practical usefulness, it should consider empirical evidence and the incorporation of more recent database advancements. This would ensure that the proposed solutions stay relevant and successful as database technology evolves [1].

1.2 Summary and Critical Analysis of Research Article or Proposal

Summary:

The research paper, titled "Roll Back Mechanism in Multi Version Locking in Distributed Database" [2], explores the domain of distributed databases and tackles the challenge of concurrency control via a multi-version locking protocol. The paper's main concern is the possibility of transaction rollbacks, which can impair database performance and reliability. The authors investigate the use of timestamp-based multi-version locking and propose solutions to avoid such rollbacks.

The main idea of the research is to enhance concurrency control in distributed databases by employing multi-version locking. In this approach, the system allows read requests to access older versions of data while write operations are ongoing, thereby increasing concurrency [2]. The paper specifically focuses on the timestamp-based multi-version locking protocol, which assigns unique timestamps to transactions, ensuring proper synchronization [2].

The paper provides valuable insights into multi-version locking and presents possible solutions to the rollback problem. While the proposed improvements are promising, the success of their implementation would depend on practical application and testing [2]. Therefore, further experimentation and validation are necessary to assess the effectiveness of these enhancements in real-world scenarios.

The paper establishes the groundwork for more efficient multi-version locking in distributed databases, with the goal of reducing transaction rollbacks [2]. The proposed improvements have the potential to be helpful, but their practical sustainability remains to be seen.

Critical Analysis:

The research presented in this paper addresses a crucial issue in the area of multi-version locking within distributed databases [2]. Rollbacks can be a significant obstacle in achieving high levels of concurrency and transaction efficiency. The proposed alternatives to the existing methodology are a promising step toward reducing rollbacks.

However, there are certain limitations to the research paper that should be considered. While the proposed approach is unique, it needs empirical support. The authors discuss the "future scope" of their work in order to validate the proposed method. A more thorough validation via experimental or case studies would bolster the paper's findings [2].

Furthermore, the study would benefit from a more thorough literature review. While it acknowledges relevant publications in the field, a more thorough examination of related research could give a more comprehensive context for the suggested improvements [2].

In the end, this study presents a novel approach to tackling the issue of rollbacks in multi-version locking within distributed databases [2]. While the presented alternatives show promise, additional validation and a more complete literature analysis are advised to cement the paper's contributions to the area. The results of this study can be used as a foundation for future research in multi-version concurrency control and distributed database management.

2. Problem Statement 2

Prototype of a lightweight DBMS using Java programming language (no 3rd party libraries allowed). Note: Here you are not using MySQL, you are expected to create a similar tool MySQL.

2.1 Overview

The objective of the project is to create a lightweight DBMS and implement it using the JAVA programming language. All database operations, including table creation, deletion, and updating, are managed by the DBMS KenDB, which ensures that users have a simple and easy experience. It also handles transactions and allows the user to conduct DBMS operations. The technique is so simple that even when executing complex tasks, the user has no difficulties.

2.2 Design

While working on the challenge, I used my knowledge of OOPS ideas and the Java programming language. I utilized text files to hold persistent data and then used multiple classes to conduct the queries as the user required.

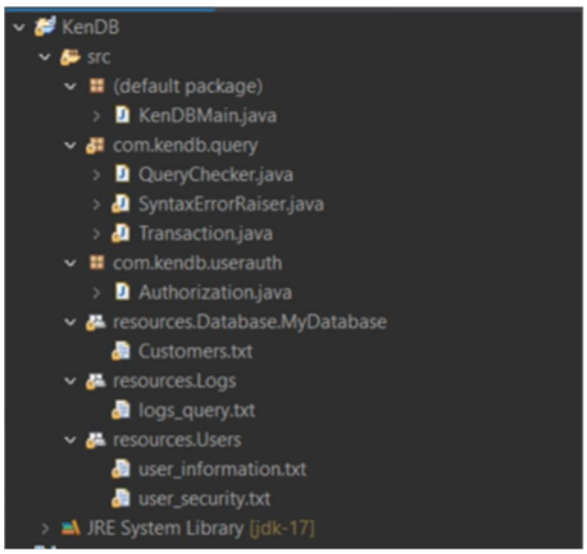


Figure 1: Folder Structure of the KenDB application

2.3 Authentication

The user will be given two options: create a new account or login using an existing username and password. Security questions for the user to ensure two-factor authentication. I used MD5 to encrypt the user login information. Below is the more detailed functionality of the Authentication module:

- **User Registration**: New users can register by entering their User ID and Password. They also select a security question and provide an answer.
- **User Data Storage**: The user's registration information is saved in separate files, including the encrypted User ID and Password.
- **Sign-In:** Existing users can sign in using the username and password.

- **Verifying:** The system validates the username and password by comparing encrypted credentials stored in the data files.
- **Two-factor Authentication:** If the user's credentials are correct, the system will prompt them to answer their chosen security question as a second way of authentication. And on the correct answer, they will be given access to the KenDB application.
- **User Interface:** On successful login, the user can interact with the database by submitting queries or exiting.
- **Security & Privacy:** For user security, the application stores the hashed credential using the MD5 in the file.

Now let's create a user in the KenDB application.

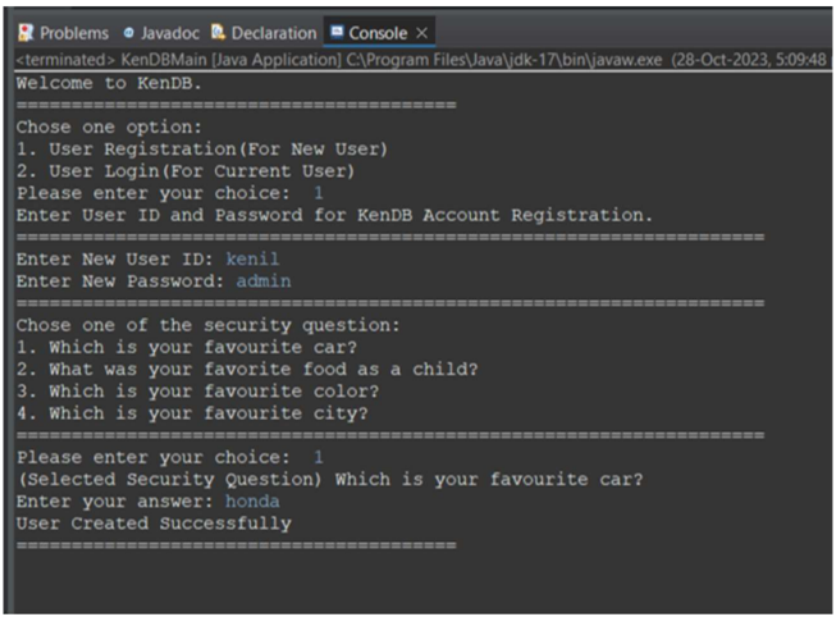


Figure 2: User registration for KenDB.

On user registration, it creates the “user_information.txt” which stores an encrypted username and password. It also stores the security question and answers for all the users in “user_security.txt”.

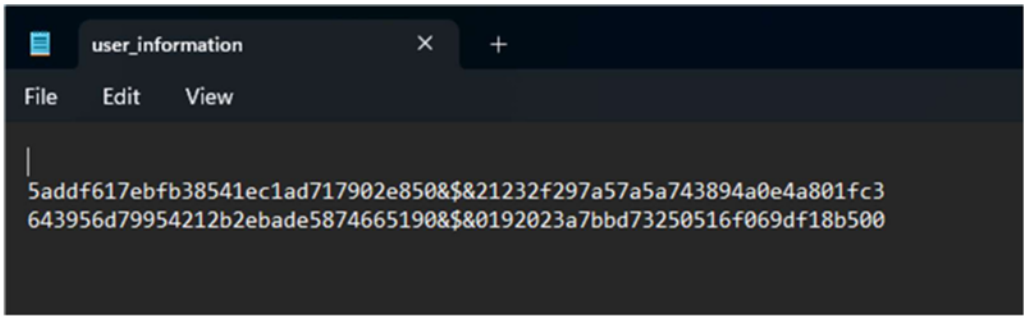


Figure 3: User encrypted information in txt file.

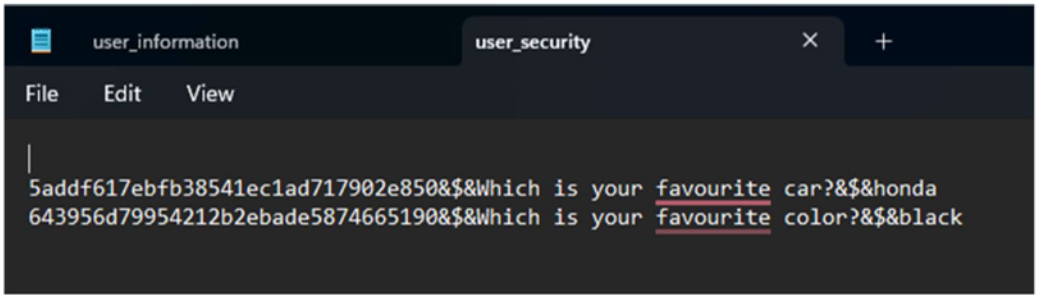


Figure 4: Storing security question and answer.

After successful registration, perform sign-in and get access to the KenDB application. It validates the user's input of username and password in the user_information and matches with the encrypted credentials. There are multiple users for a single database. So now let's perform the login and get access to enter queries.

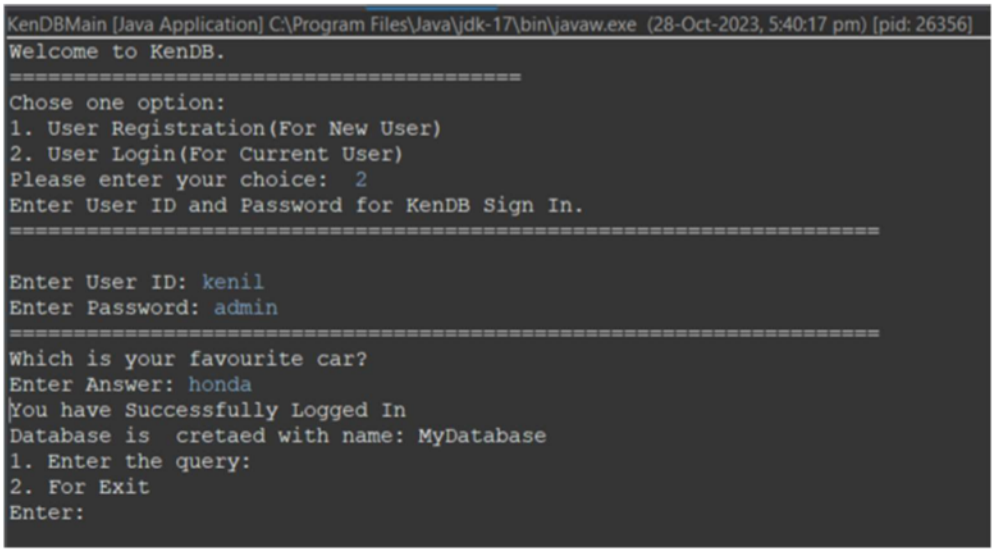


Figure 5: Sign In Successful in the Application.

Test Cases

Test case 1: Login
Scenario: Enter the wrong password.
Expected Results: It will give the error and not ask the security question.

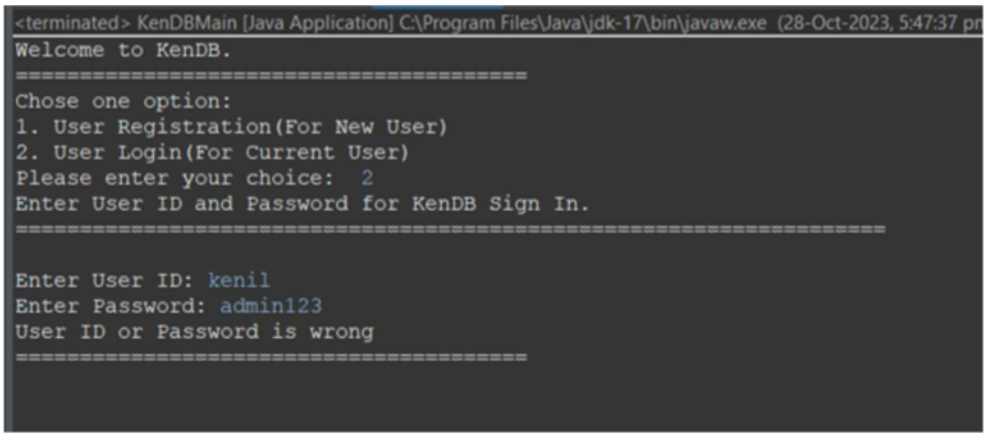


Figure 6: Entering the wrong password gives an error.

Test case 2: Option Choice
Scenario: After Login, Choose the option that is invalid.
Expected Results: It will again ask for the correct input.

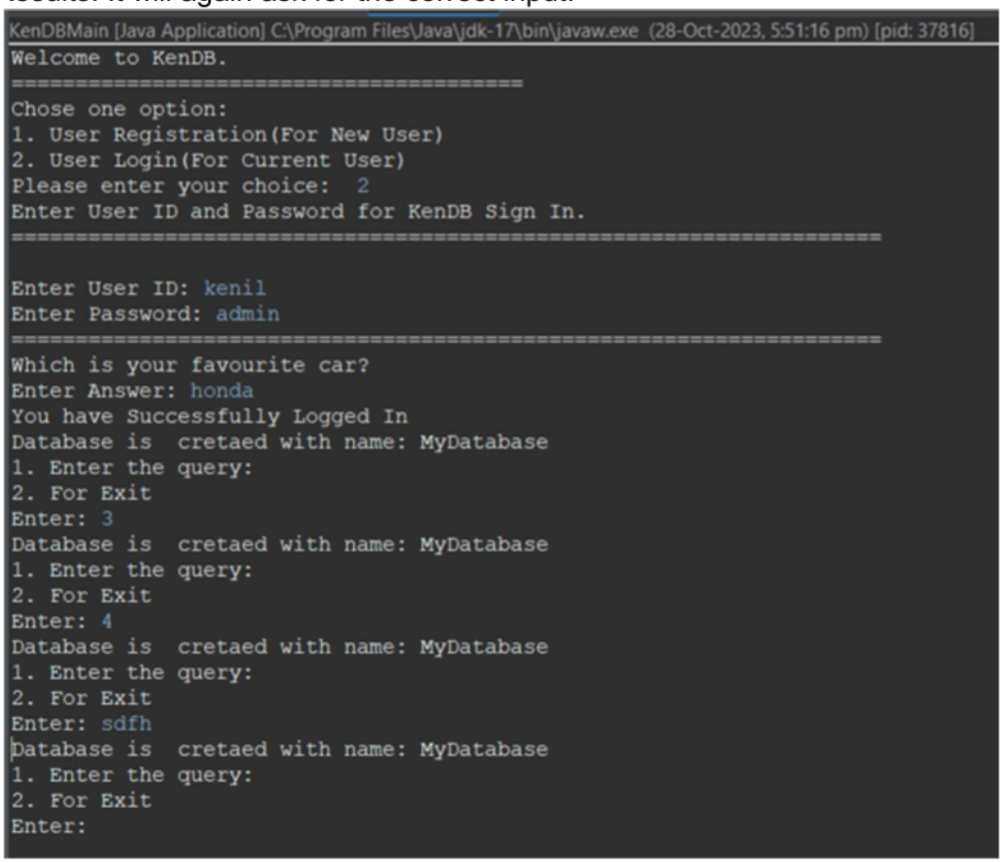


Figure 7: Choosing an invalid option.

2.4 Query Processing

The “QueryChecker” class is responsible for processing and validating the queries including creating, dropping, inserting, selecting, updating, and deleting data within database tables. It performs various tasks such as:

- **Queries Pattern**: It has various regular expressions using the Pattern class to match SQL queries for creating tables, dropping tables, inserting data, selecting data, updating data, and beginning transactions.
- **Database Directory**: It creates the database directory named “MyDatabase” automatically on the first user login. This is the default database.
- **Query Processing**: The traverseQuery method processes SQL queries by matching them to the defined Queries Pattern. It also handles transactions and checks the database is not locked before any query execution.

Table Creation

If a query matches the "Create table" regex, the code creates a new table file and stores the table's metadata, including column names, data types, primary keys, and foreign keys.

```
// check the create regex
matcher = CREATE.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag &&  transactionFlag) || (!databaseLockFlag &&  !transactionFlag)) {

        String tableName = matcher.group(1);
        String[] tableMetaData = matcher.group(2).split(",");
        HashMap<String, String> tableColumnData = new HashMap<>();
        List<String> primaryKeys = new ArrayList<>();
        List<String> foreignKeys = new ArrayList<>();

        for (String data : tableMetaData) {
            String[] column = data.trim().split(" ");
            if (column.length == 2) {
                tableColumnData.put(column[0], column[1]);
            } else if (column[0].equalsIgnoreCase("PRIMARY")) {
                primaryKeys.add(column[2]);
            } else if (column[0].equalsIgnoreCase("FOREIGN")) {
                foreignKeys.add(column[4]);
            } else {
                throw new SyntaxErrorRaiser("Invalid syntax");
            }
        }

        StringBuilder metadataLine = new StringBuilder(tableName + "(");
        for (Map.Entry<String, String> entry : tableColumnData.entrySet()) {
            metadataLine.append(entry.getKey()).append(":").append(entry.getValue()).append(",");
        }
        if (!primaryKeys.isEmpty()) {
            metadataLine.append("PRIMARY_KEY:");
            for (String primaryKey : primaryKeys) {
                metadataLine.append(primaryKey).append(",");
            }
        }
        if (!foreignKeys.isEmpty()) {
            metadataLine.append("FOREIGN_KEY:");
            for (String foreignKey : foreignKeys) {
                metadataLine.append(foreignKey).append(",");
            }
        }
        metadataLine.deleteCharAt(metadataLine.length() - 1);
        metadataLine.append(")");

        try {
            queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase).append(" ] [Table: ")
            .append(tableName).append(" ] [Query: ").append(query).append("] [Query Type: Valid]").append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");
            File tableFile = new File(DATABASE_ROOT_PATH + activeDatabase + "/" + tableName + ".txt");
            if (tableFile.createNewFile()) {
                System.out.println("Table is created successfully : " + tableName);
                updateTableTxt(tableName, metadataLine.toString());
            } else {
                System.out.println("Table exists");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        System.out.println("Database Locked");
    }
```

Figure 8: Code for Table Creation

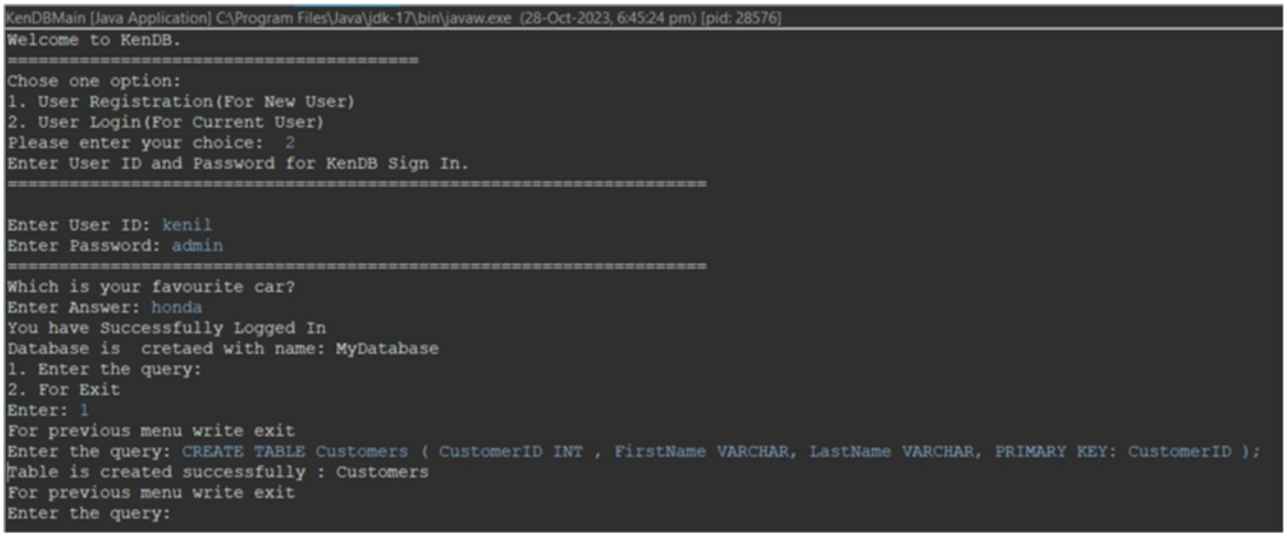


Figure 9: Customer Table Created.

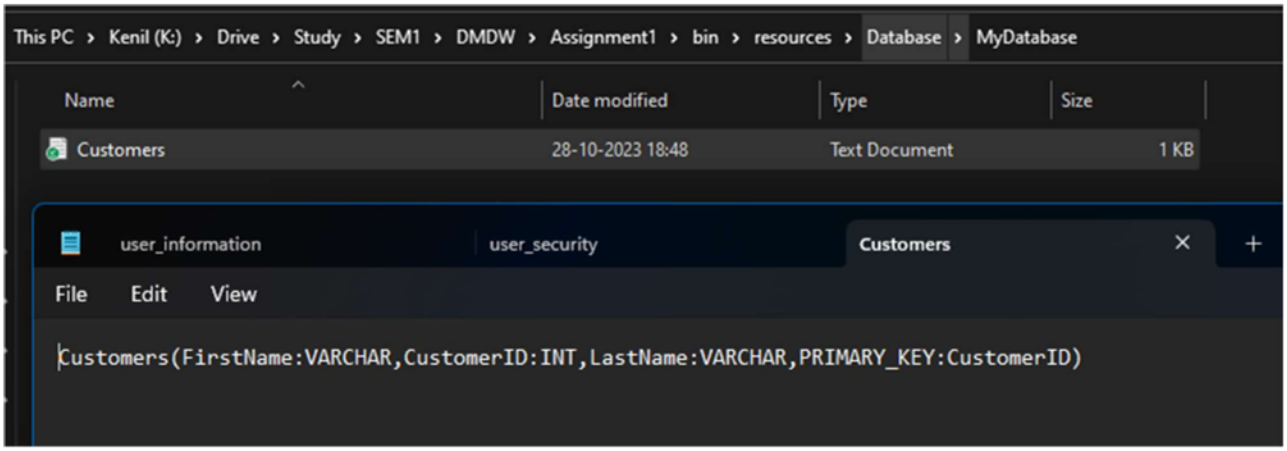


Figure 10: Customers table file generated with metadata.

Data Insertion: When an "Insert into" query is matched, the code inserts data into the specified table.

```java
// check the insert query regex
matcher = INSERT.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag &&  transactionFlag) || (!databaseLockFlag &&  !transactionFlag)) {
        String tableName = matcher.group(1);
        Path tableFile = Paths.get(DATABASE_ROOT_PATH, activeDatabase, tableName + ".txt");
        queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase).append(" ] [Table: ").append(tableName)
        .append(" ] [Query: ").append(query).append("] [Query Type: Valid]").append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");

        try {
            List<String> tableData = Files.readAllLines(tableFile);
            if (tableData.isEmpty()) {
                throw new SyntaxErrorRaiser("Table does not exist.");
            }
            else {
                String[] colNames = getColNames(tableName);
                String[] rowData = matcher.group(5).split(",");
                HashMap<String, String> tableRowData = new HashMap<>();
                if (colNames.length == rowData.length) {
                    for (int i = 0; i < colNames.length; i++) {
                        tableRowData.put(colNames[i].trim(), rowData[i].trim());
                    }
                }
                else {
                    throw new SyntaxErrorRaiser("Invalid column names");
                }

                if (!tableRowData.isEmpty()) {
                    if (tableRowData.size() == colNames.length) {
                        System.out.println("Inserted 1 row into " + tableName);
                        List<String> rowsData = new ArrayList<>();
                        for (String col : colNames) {
                            rowsData.add(tableRowData.get(col.trim()));
                        }

                        tableData.add(String.join("|", rowsData));

                        Files.write(tableFile, tableData, StandardOpenOption.TRUNCATE_EXISTING);
                    } else {
                        System.out.println("Values are missing in the query");
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    else {
        System.out.println("Database Locked");
    }
}
```

Figure 11: Code for Data Insertion.

```
KenDBMain [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (28-Oct-2023, 7:18:24 pm) [pid: 22752]
Welcome to KenDB.
=====================================
Chose one option:
1. User Registration(For New User)
2. User Login(For Current User)
Please enter your choice:  2
Enter User ID and Password for KenDB Sign In.
=================================================================

Enter User ID: kenil
Enter Password: admin
=================================================================
Which is your favourite car?
Enter Answer: honda
You have Successfully Logged In
Database is  cretaed with name: MyDatabase
1. Enter the query:
2. For Exit
Enter: 1
For previous menu write exit
Enter the query: Create table Customers ( CustomerID INT,  FirstName VARCHAR, LastName VARCHAR, PRIMARY KEY: CustomerID );
Table is created successfully : Customers
For previous menu write exit
Enter the query: Insert into Customers (CustomerID , FirstName , LastName ) values (1, 'Kenil', 'Patel');
Customers(FirstName:VARCHAR,CustomerID:INT,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
Inserted 1 row into Customers
For previous menu write exit
Enter the query: Insert into Customers (CustomerID , FirstName , LastName ) values (1, 'John', 'Charles');
Customers(FirstName:VARCHAR,CustomerID:INT,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
Inserted 1 row into Customers
For previous menu write exit
Enter the query:
```

Figure 12: Data inserted into the customers table.

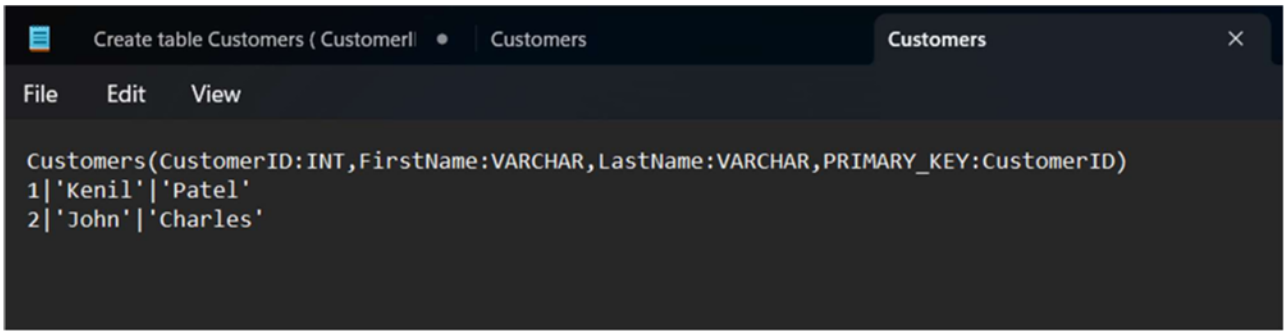


Figure 13: Customer table File with data.

Data Selection: It supports both selecting all data from a table and specific rows that match a condition in a "where" clause.

```
// check the select query regex
matcher = SELECT_WHERE.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag &&  transactionFlag) || (!databaseLockFlag &&  !transactionFlag)) {
        String tableName = matcher.group(1);
        String columnNameToGet = matcher.group(2);
        String columnValueToGet = matcher.group(3);
        queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase).append(" ] [Table: ")
        .append(tableName).append(" ] [Query: ").append(query).append("] [Query Type: Valid]").append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");

        try {
            Path tableFile = Paths.get(DATABASE_ROOT_PATH, activeDatabase, tableName + ".txt");
            List<String> tableData = Files.readAllLines(tableFile);
            String[] colNames = getColNames(tableName);
            int columnNameToGetIndex = -1;
            for (int i = 0; i < colNames.length; i++) {
                if (colNames[i].equalsIgnoreCase(columnNameToGet)) {
                    columnNameToGetIndex = i;
                    break;
                }
            }

            if (columnNameToGetIndex == -1) {
                System.out.println("Condition column not found: " + columnNameToGet);
            }
            else {
                for (String col : colNames) {
                    System.out.print(col.split(":")[0] + "\t\t");
                }
                System.out.println();
                for (int i = 1; i < tableData.size(); i++) {
                    String rowFull = tableData.get(i);
                    String[] rowDataSeparate = rowFull.split("\\|");
                    if (rowDataSeparate.length > columnNameToGetIndex &&
                            (rowDataSeparate[columnNameToGetIndex].replace("'", "")).equalsIgnoreCase(columnValueToGet)) {
                        for (String colValue : rowDataSeparate) {
                            System.out.print(colValue + "\t\t");
                        }
                        System.out.println();
                    }
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        System.out.println("Database Locked");
    }
```

Figure 14: Code for displaying table

```
Enter the query:  Select * from Customers;
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
+------------+-----------+-----------+
| CustomerID | FirstName | LastName  |
+------------+-----------+-----------+
| 1          | 'Kenil'   | 'Patel'   |
| 2          | 'John'    | 'Charles' |
+------------+-----------+-----------+
```

Figure 15: Display all data from the customer Table.

```
| 2          | 'John'     | 'Charles' |
+------------+------------+-----------+
For previous menu write exit
Enter the query: Select * from customers where CustomerID=1;
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
CustomerID              FirstName               LastName
1               'Kenil'         'Patel'
```

Figure 16: Display only one row that has customerID=1.

Data Updating: If a query matches the "Update" regex, it updates specific values in rows that meet a given condition.

```
// check the update value row query regex
matcher = UPDATE_WHERE.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag &&  transactionFlag) || (!databaseLockFlag &&  !transactionFlag)) {
        String tableName = matcher.group(1);
        String columnToUpdate = matcher.group(2);
        String valueToUpdate = matcher.group(3);
        String columnNameToGet = matcher.group(4);
        String columnValueToGet = matcher.group(5);
        queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase).append(" ] [Table: ").append(tableName)
        .append(" ] [Query: ").append(query).append("] [Query Type: Valid]").append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");

        List<String> tableDataFull = Files.readAllLines(Paths.get(DATABASE_ROOT_PATH, activeDatabase, tableName + ".txt"));
        if (tableDataFull.isEmpty()) {
            System.out.println("Table " + tableName + " does not exist.");
        }
        else {
            String[] colNames = getColNames(tableName);
            int columnIndexToUpdate = -1;
            for (int i = 0; i < colNames.length; i++) {
                if (colNames[i].equalsIgnoreCase(columnToUpdate)) {
                    columnIndexToUpdate = i;
                    break;
                }
            }

            if (columnIndexToUpdate == -1) {
                System.out.println("Column to update not found: " + columnToUpdate);

            }

            int columnNameToGetIndex = -1;
            for (int i = 0; i < colNames.length; i++) {
                if (colNames[i].equalsIgnoreCase(columnNameToGet)) {
                    columnNameToGetIndex = i;
                    break;
                }
            }

            if (columnNameToGetIndex == -1) {
                System.out.println("Condition column not found: " + columnNameToGet);
            }

            for (int i = 1; i < tableDataFull.size(); i++) {
                String rowDataFull = tableDataFull.get(i);
                String[] rowDataSeparate = rowDataFull.split("\\|");

                if (rowDataSeparate.length > columnNameToGetIndex &&
                        rowDataSeparate[columnNameToGetIndex].equalsIgnoreCase(columnValueToGet)) {
                    rowDataSeparate[columnIndexToUpdate] = valueToUpdate;
                    tableDataFull.set(i, String.join("|", rowDataSeparate));
                }
            }
            Files.write(Paths.get(DATABASE_ROOT_PATH, activeDatabase, tableName + ".txt"), tableDataFull);
            System.out.println("Table data updated in " + tableName);
        }
    }
    else {
        System.out.println("Database Locked");
    }
```

Figure 17: Code for updating values in the table.

```
For previous menu write exit
Enter the query: update customers set FirstName=Mathew where CustomerId=1;
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
Table data updated in customers
For previous menu write exit
Enter the query:
```

Figure 18: Updating new values in the customer table.

```
For previous menu write exit
Enter the query: update customers set FirstName=Mathew where CustomerId=1;
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
Table data updated in customers
For previous menu write exit
Enter the query: Select * from Customers;
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
+------------+-----------+-----------+
| CustomerID | FirstName | LastName  |
+------------+-----------+-----------+
| 1          | Mathew    | 'Patel'   |
| 2          | 'John'    | 'Charles' |
+------------+-----------+-----------+
For previous menu write exit
Enter the query:
```

Figure 19: Updated Customers table.

Data Deletion: Queries matching the "Delete from" regex allow for the removal of specific rows based on a condition in the "where" clause.

```
// check the Delete row query regex
matcher = DELETE_WHERE.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag && transactionFlag) || (!databaseLockFlag && !transactionFlag)) {
        String tableName = matcher.group(1);
        String columnNameToGet = matcher.group(2);
        String columnValueToGet = matcher.group(3);

        queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase).append(" ] [Table: ").append(tableName)
        .append(" ] [Query: ").append(query).append("] [Query Type: Valid]").append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");

        try {
            Path tableFile = Paths.get(DATABASE_ROOT_PATH, activeDatabase, tableName + ".txt");
            List<String> tableDataFull = Files.readAllLines(tableFile);

            if (tableDataFull.isEmpty()) {
                System.out.println("Table " + tableName + " is empty.");
            } else {

                String firstLineTable = tableDataFull.get(0);
                String[] colNames = getColNames(tableName);

                int columnNameToGetIndex = -1;
                for (int i = 0; i < colNames.length; i++) {
                    if (colNames[i].equalsIgnoreCase(columnNameToGet)) {
                        columnNameToGetIndex = i;
                        break;
                    }
                }

                if (columnNameToGetIndex == -1) {
                    System.out.println("Condition column not found: " + columnNameToGet);
                } else {
                    List<String> newTableData = new ArrayList<>();
                    newTableData.add(firstLineTable);

                    int count = 0;

                    for (int i = 1; i < tableDataFull.size(); i++) {
                        String rowDataFull = tableDataFull.get(i);
                        String[] rowDataSeparate = rowDataFull.split("\\|");
                        if (rowDataSeparate.length > columnNameToGetIndex) {
                            String conditionColumnValue = rowDataSeparate[columnNameToGetIndex].replace("'", "");
                            if (conditionColumnValue.equals(columnValueToGet)) {
                                count++;
                            } else {
                                newTableData.add(rowDataFull);
                            }
                        }
                    }
                    Files.write(tableFile, newTableData);
                    System.out.println("Total " + count + " row(s) are deleted in" + tableName);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    else {
        System.out.println("Database Locked");
    }
```

Figure 20: Code for deletion.

```
KenDBMain [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (28-Oct-2023, 8:01:39 pm) [pid: 33664]
Welcome to KenDB.
=======================================
Chose one option:
1. User Registration(For New User)
2. User Login(For Current User)
Please enter your choice:  2
Enter User ID and Password for KenDB Sign In.
=========================================================================

Enter User ID: kenil
Enter Password: admin
=========================================================================
Which is your favourite car?
Enter Answer: honda
You have Successfully Logged In
Database is  cretaed with name: MyDatabase
1. Enter the query:
2. For Exit
Enter: 1
For previous menu write exit
Enter the query: Delete from customers where FirstName=John;
Total 1 row(s) are deleted incustomers
For previous menu write exit
Enter the query:
```

Figure 21: Deleting row where FirstName=John.

```
For previous menu write exit
Enter the query: select * from customers;
+------------+-----------+----------+
| CustomerID | FirstName | LastName |
+------------+-----------+----------+
| 1          | Mathew    | 'Patel'  |
+------------+-----------+----------+
For previous menu write exit
Enter the query:
```

Figure 22: Customer Table after deletion.

Table Dropping: For queries matching the "Drop table" regex, the code deletes the corresponding table file.

```
// check the drop query regex
matcher = DROP.matcher(query);
if (matcher.find()) {
    queryInvalid = false;
    if ((databaseLockFlag &&  transactionFlag) || (!databaseLockFlag &&  !transactionFlag)) {
        String tableName = matcher.group(1);
        if (matcher.group(1).contains(" ")) {
            throw new SyntaxErrorRaiser("Table name has a white space");
        }

        queryLogs.append("[User: ").append(Username).append(" ]").append("[Database: ").append(activeDatabase)
        .append(" ] [Table: ").append(tableName).append(" ] [Query: ").append(query).append("] [Query Type: Valid]")
        .append("[Timestamp: ").append(String.valueOf(ts)).append(" ]\n");

        File tableFile = new File(DATABASE_ROOT_PATH + activeDatabase + "/" + tableName + ".txt");
        if (tableFile.exists()) {
            if (tableFile.delete()) {
                System.out.println("Table named " + tableName + " is Dropped." );
            } else {
                System.out.println("Error in dropping table " + tableName);
            }
        } else {
            System.out.println("Table named " + tableName + " does not exist in the database");
        }
    }
    else {
        System.out.println("Database Locked");
    }

}
```

Figure 23: Code for Drop Table.

Creating a dummy table for dropping it. A new file for a dummy table is generated.

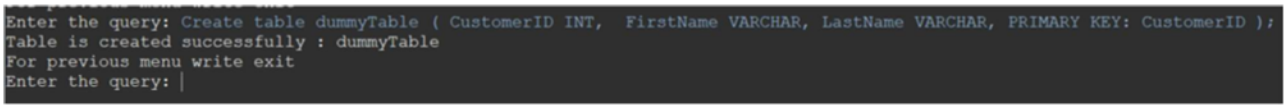


Figure 24: Creating a dummy table.

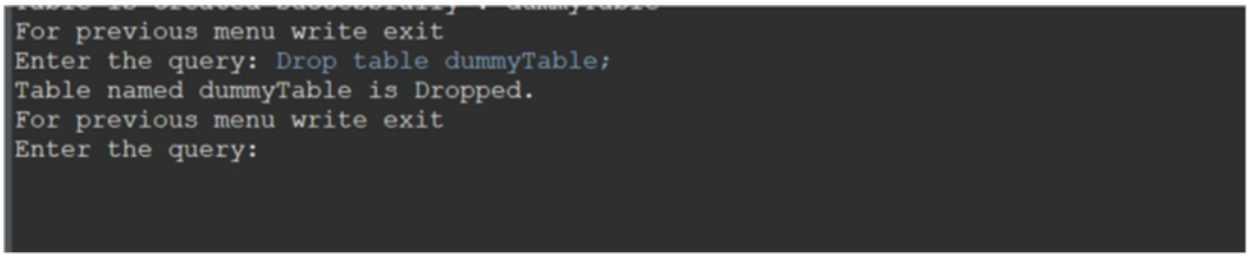


Figure 25: Dropped the dummy table.

Test Cases:

Test case 1: Wrong Query
Scenario: Enter the wrong query which is without ";".
Expected Results: It will give Invalid Query.

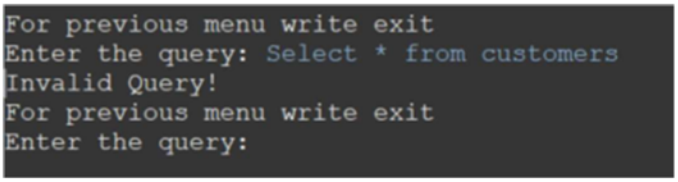


Figure 26: Entered wrong query.

Test case 2: Query with the wrong table name.
Scenario: Enter the query with a table name that does not exist.
Expected Results: It will give a table doesn't exist.

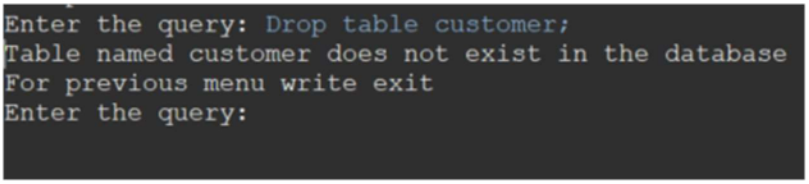


Figure 27: Wrong table name.

2.5 Logging and Delimiters (Novelty Task)

It stores all the queries in the "queryLogs.txt". The timestamps, database name, username, query, and valid status are also stored in it. Logs help to track all the previous operations on the database. The logs are stored in a file automatically as soon as the query runs and if the query is valid it has type valid if the query is entered wrong then it will have Invalid type.

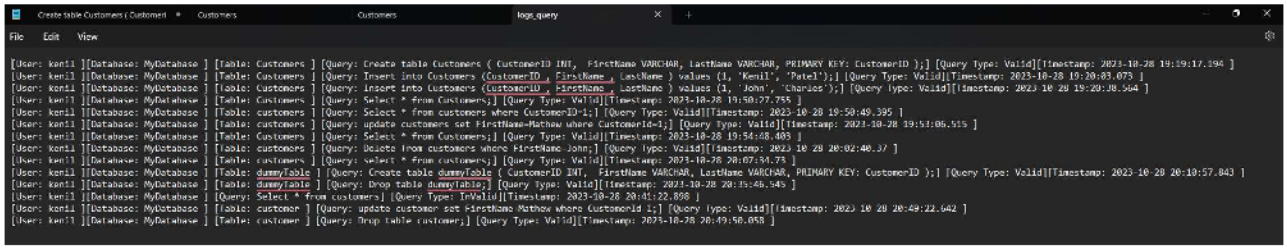


Figure 28: Logs of query.

To store the data in the txt file delimiter " | " is used to separate the data types from other.

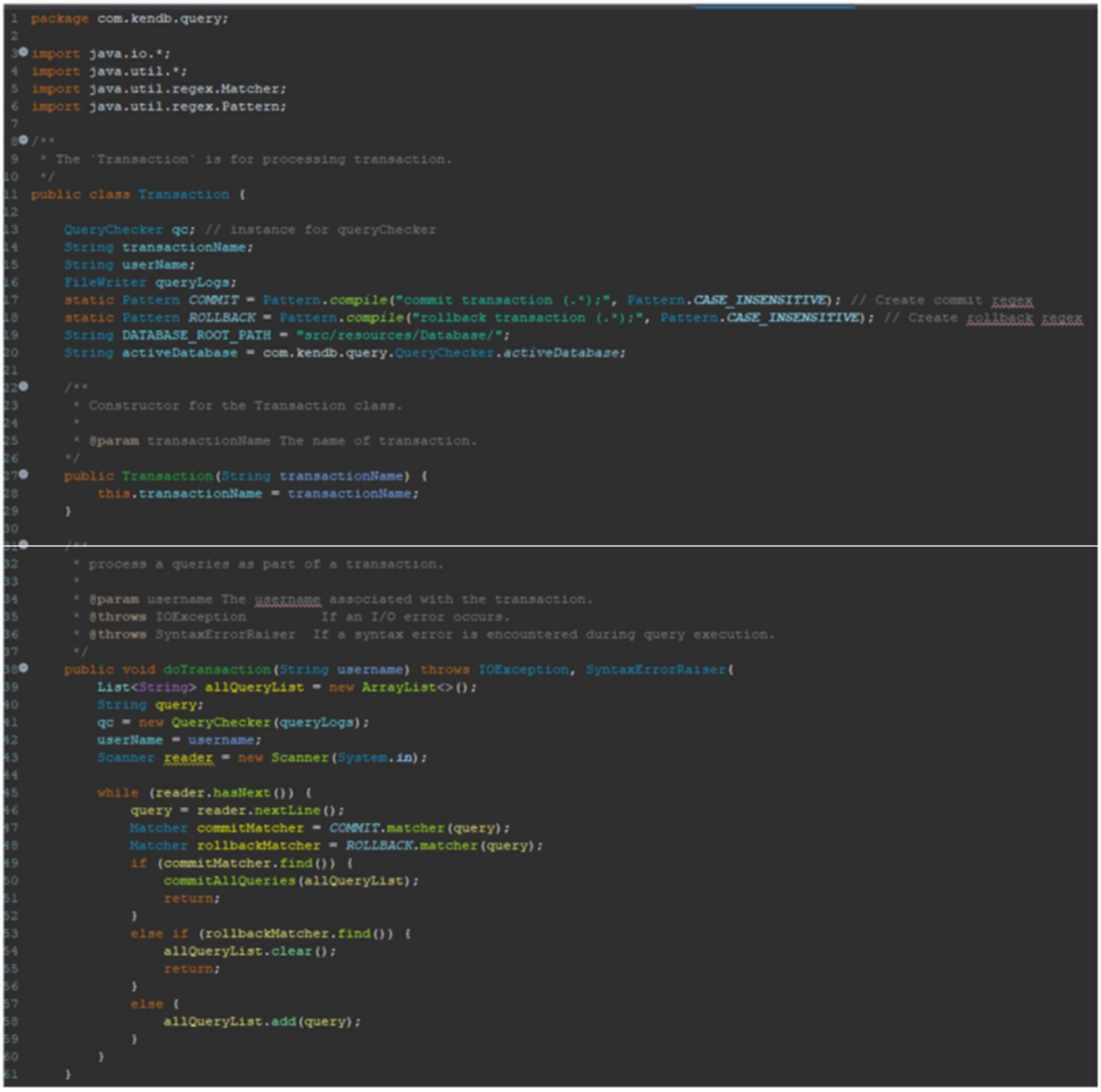
```
Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
1|'Kenil'|'Patel'
2|'John'|'Charles'
```

Figure 29: Delimiters

2.6 Transactions

The transactions provide a way to group a series of SQL queries, ensuring that either all queries are successfully executed and committed or none of them are applied if the rollback occurs.
Working of Transaction module:

- It contains patterns and methods for handling transaction-related SQL statements, specifically "commit" and "rollback" statements.
- Transactions are associated with a particular database, and the code ensures exclusive access to the database during a transaction by creating a "lockfile.lock" in the database folder.
- When a "commit" statement is encountered, all the queries in the transaction are executed and permanently saved to the database.
- When a "rollback" statement is encountered, all queries in the current transaction are discarded, and no changes are made to the database.



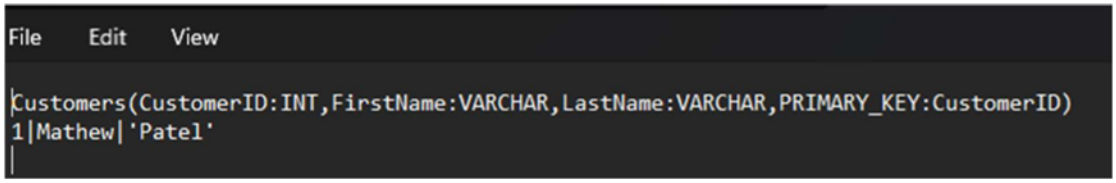
```java
package com.kendb.query;

import java.io.*;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * The `Transaction` is for processing transaction.
 */
public class Transaction {

    QueryChecker qc; // instance for queryChecker
    String transactionName;
    String userName;
    FileWriter queryLogs;
    static Pattern COMMIT = Pattern.compile("commit transaction (.*);", Pattern.CASE_INSENSITIVE); // Create commit regex
    static Pattern ROLLBACK = Pattern.compile("rollback transaction (.*);", Pattern.CASE_INSENSITIVE); // Create rollback regex
    String DATABASE_ROOT_PATH = "src/resources/Database/";
    String activeDatabase = com.kendb.query.QueryChecker.activeDatabase;

    /**
     * Constructor for the Transaction class.
     *
     * @param transactionName The name of transaction.
     */
    public Transaction(String transactionName) {
        this.transactionName = transactionName;
    }

    /**
     * process a queries as part of a transaction.
     *
     * @param username The username associated with the transaction.
     * @throws IOException          If an I/O error occurs.
     * @throws SyntaxErrorRaiser  If a syntax error is encountered during query execution.
     */
    public void doTransaction(String username) throws IOException, SyntaxErrorRaiser{
        List<String> allQueryList = new ArrayList<>();
        String query;
        qc = new QueryChecker(queryLogs);
        userName = username;
        Scanner reader = new Scanner(System.in);

        while (reader.hasNext()) {
            query = reader.nextLine();
            Matcher commitMatcher = COMMIT.matcher(query);
            Matcher rollbackMatcher = ROLLBACK.matcher(query);
            if (commitMatcher.find()) {
                commitAllQueries(allQueryList);
                return;
            }
            else if (rollbackMatcher.find()) {
                allQueryList.clear();
                return;
            }
            else {
                allQueryList.add(query);
            }
        }
    }
```

Figure 30: Code for Transaction.

```java
    /**
     * Perform execution of queries.
     *
     * @param allQueryList List of queries to be executed and committed.
     * @throws SyntaxErrorRaiser  If a syntax error is encountered during query execution.
     * @throws IOException        If an I/O error occurs.
     */
    private void performCommitAllQueries(List<String> allQueryList) throws SyntaxErrorRaiser, IOException {
        for (String q : allQueryList) {
            qc.traverseQuery(q, userName, true);
        }
    }

    /**
     * Execute a lock and commit a all queries in a transaction.
     *
     * @param allQueryList List of queries to be executed and committed.
     * @throws SyntaxErrorRaiser  If a syntax error is encountered during query execution.
     * @throws IOException        If an I/O error occurs.
     */
    private void commitAllQueries(List<String> allQueryList) throws SyntaxErrorRaiser, IOException{
        String databasePath = DATABASE_ROOT_PATH + activeDatabase;
        try {
            lockDatabaseFolder(databasePath);
            performCommitAllQueries(allQueryList);
        } finally {
            unlockDatabaseFolder(databasePath);
        }
    }

    /**
     * Unlock the database folder allowing other to access it.
     *
     * @param databasePath Path to the database folder.
     */
    private void unlockDatabaseFolder(String databasePath) {
        File folderLockFile = new File(databasePath, "lockfile.lock");
        if (folderLockFile.exists() && folderLockFile.delete()) {
            System.out.println("Databse lock removed");
        } else {
            System.out.println("Database lock not removed");
        }
    }

    /**
     * Lock the database folder to give exclusive access.
     *
     * @param databasePath Path to the database folder.
     */
    private void lockDatabaseFolder(String databasePath) {
        File folderLockFile = new File(databasePath, "lockfile.lock");
        try {
            if (folderLockFile.createNewFile()) {
                System.out.println("Database is now locked.");
            } else {
                System.out.println("Database lock exists");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Figure 31: Functions of Transaction Class.

Let's insert data using Transaction in the customer table.



```
File    Edit    View

Customers(CustomerID:INT,FirstName:VARCHAR,LastName:VARCHAR,PRIMARY_KEY:CustomerID)
1|Mathew|'Patel'
```

Figure 32: Table file before transaction.

Transaction is completed if commit transaction t1 is entered as a query.

```
Database is  cretaed with name: MyDatabase
1. Enter the query:
2. For Exit
Enter: 1
For previous menu write exit
Enter the query: Begin Transaction t1;
Insert into Customers (CustomerID , FirstName , LastName ) values (2, 'Rajesh', 'Parmar');
Insert into Customers (CustomerID , FirstName , LastName ) values (3, 'Mahesh', 'Parmar');
Commit transaction t1;
Database is now locked.
Inserted 1 row into Customers
Inserted 1 row into Customers
Databse lock removed
For previous menu write exit
Enter the query:
```

Figure 33: Committing Transaction.

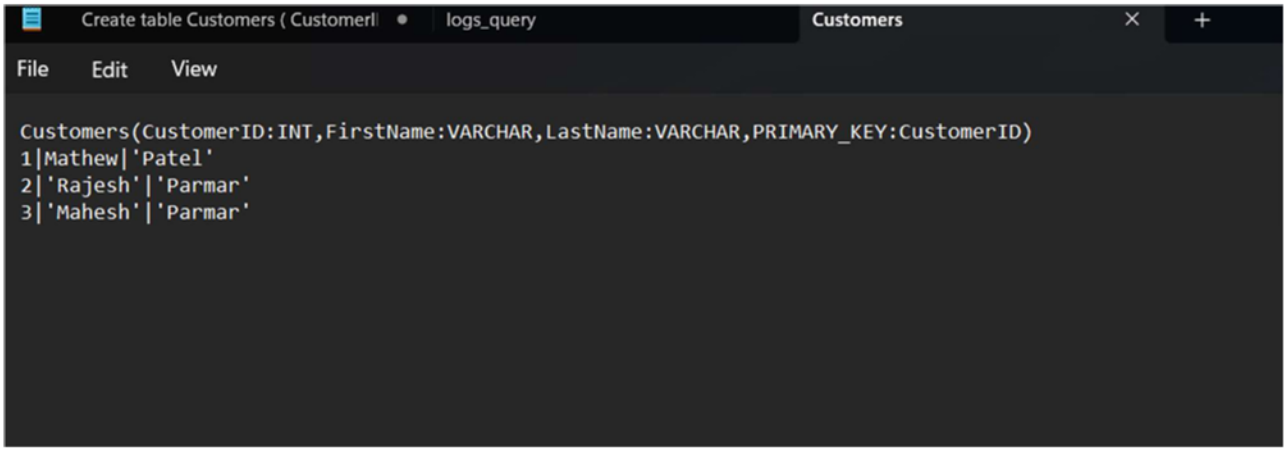


Figure 34: Table file after Transaction.

Rollback will not make any changes to the table.

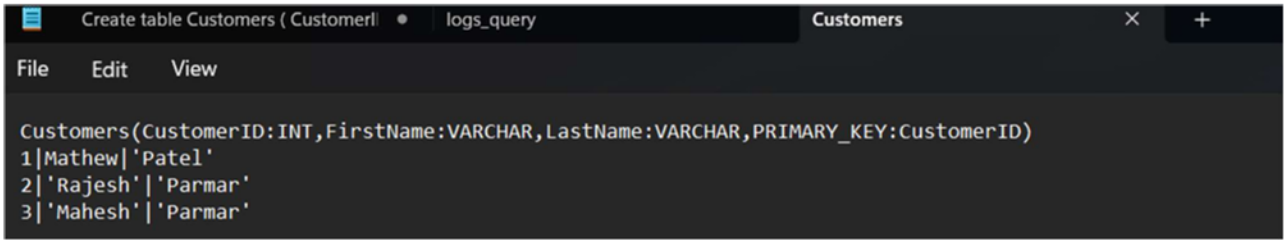


Figure 35: Table before Transaction.

```
Enter: 1
For previous menu write exit
Enter the query: Begin Transaction t1;
update customers set FirstName=Kamlesh where CustomerId=2;
rollback transaction t1;
For previous menu write exit
Enter the query:
```

Figure 36: No update on rollback.

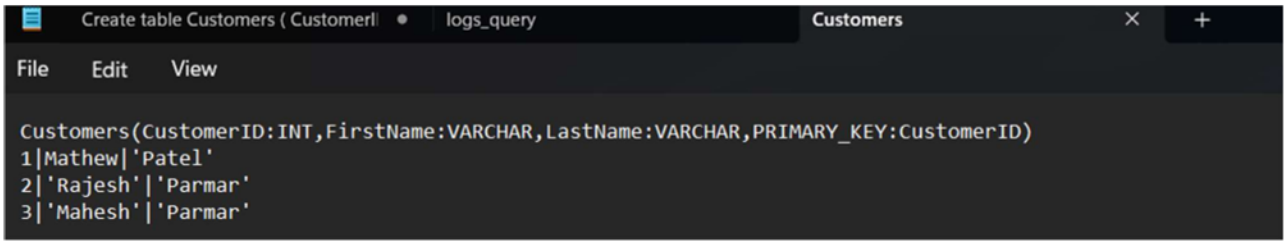


Figure 37: No change in table after transaction.

2.7 Conclusion

In the end, I can say that it was an amazing opportunity to learn. I was able to get a normal database running using JAVA. It helped me in gaining a better understanding of our database principles and improving our critical thinking skills.

References

[1] D. Damoah, J. B. Hayfron-Acquah, S. Sebastian, E. Ansong, B. Agyemang and R. Villafane, "Transaction recovery in federated distributed database systems," Proceedings of IEEE International Conference on Computer Communication and Systems ICCCS14, Chennai, India, 2014, pp. 116-123, DOI: 10.1109/ICCCS.2014.7068178.

[2] Swati and S. B. Bajaj, "Roll Back Mechanism in Multi Version Locking in Distributed Database," 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2018, pp. 1593-1597, DOI: 10.1109/ICECA.2018.8474678.

[3] Java.com. [Online]. Available: https://www.java.com/en/. [Accessed: 29-Oct-2023].

[4] "SQL," GeeksforGeeks, 06-Nov-2017. [Online]. Available: https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/. [Accessed: 29-Oct-2023].

[5] Eclipse Foundation, "Eclipse IDE," eclipseide.org. [Online]. Available: https://eclipseide.org/. [Accessed: 29-Oct-2023].

[6] *Baeldung.com*. [Online]. Available: https://www.baeldung.com/java-md5. [Accessed: 29-Oct-2023].