CSCI 5408

DATA MANAGEMENT AND WAREHOUSING

ASSIGNMENT - 1

Banner ID: B00952865

GitLab Assignment Link:
https://git.cs.dal.ca/apurohit/CSCI5408_F23_B00952865_AdityaMaheshbhai_Purohit/-/tree/main/A1

Table of Contents

Problem1:

Research Paper 1: TRANSACTION RECOVERY IN FEDERATED DISTRIBUTED DATABASE SYSTEMS

Central Theme:

The focus of the paper is on how to recover the federated distributed database systems in case of transaction failures. It explores on the architecture as well as algorithm for solving this issue. It highlights the complexity of recovery and concurrency control in such distributed systems as compared to the centralized systems. It discuses various algorithms used in this area of federated distributed database systems [1].

Their implementation focuses on the Local Transaction Agent, Global Transaction Agent and its algorithm made by different research, Sync Co-ordinator and its algorithm. Out of all the components focus to achieve this recovery process, they seem to be extremely focused on sync co-ordinator component in their existing research [1].

They also conclude that others should proceed in the direction of improvement of the sync co-ordinator component in the architecture and algorithmic areas of the recovery system [1].

Problem Addressed:

The problem is how to recover to a consistent state after transaction failure, in federated database systems. The complexity problem is address due to presence of local and global databases and presence of root transactions as well as sub-transactions [1].

About Literature Review:

Yes, there are several other papers that the authors have referred to. They started with referring to various definitions like Database System (DBS), Database Management System (DBMS), Distributed Database (DDB), etc. Then reviewed some characterises of Distributed Databases as per previous papers. Mainly they focused on the literature review of FDBS Recovery algorithm concepts. There were 6 such points raised from various different sources [1].

Success of Research:

The research has got some successful results but is not fully successful yet, thus the authors have provided the direction for future successful research. They concluded to focus on the sync co-ordinator to get good results [1].

Shortcomings & Room of improvements:

The research seems to be over-focused on literature review part and very less focused on the design and implementation part [1].

The solution describing the integration of various algorithms of federated distributed database system’s transaction recovery is also not detailed enough. The results of the java program experiment is quite abstract and just concludes with the direction of future research into the area of sync-coordinator [1].

The research was unable to provide a concrete implementation of the sync-coordinator. The room of improvement lies both in the architecture and the algorithm for the architecture to be used [1].

Summary:

Overall, the paper focuses on the recovery of federated distributed database systems in case of transaction failures which may lead or dirty reads or inconsistent database states. It highlights the complexity of recovery and concurrency control in such systems as compared to centralized systems [1].

The authors examine various existing algorithms used in federated distributed database systems, along with topics such as the Local Transaction Agent, Global Transaction Agent, and Sync Co-ordinator, and suggest that further research should be done to improve the sync co-ordinator component in order to achieve better results [1].

Research Paper 2: Evaluation of federated database for distributed applications in e-government

Summary:

This study is about the importance of efficient data retrieval in government, mainly in e-government services. These services cover government-to-government (G2G), government-to-business (G2B), government-to-citizens (G2C), and government-to-enterprise (G2E) [2].

Municipal administrative units are often scattered across various locations and need access to data from their own servers and external department servers. The federated database approach aims to merge data from different sources but often falls short in addressing real-time needs. Cloud technology is suggested to enhance security and scalability [2].

Real-time responses are crucial in today's environment for both in-person and mobile users. To address this, the study tests three approaches in a real e-government context. The first relies on MySQL's federated database, the second on OpenLink Virtuoso, and the third on a specialized software architecture for data federation within a cloud infrastructure. The goal is to identify the best approach for achieving real-time data retrieval in an e-government setting [2].

Critical Analysis:

The paper is well structured and tries to implement a new method to solve a problem. The introduction is detailed and also has a comprehensive background but there is a need for trimming the content and conciseness. To maintain readability summarizing certain points is needed [2].

The technologies and approaches used should also be cited which was found missing. The abbreviations used should be clearly defined which is missing from the paper. Case studies are not properly given for real-life examples to make it more relatable to the users [2].

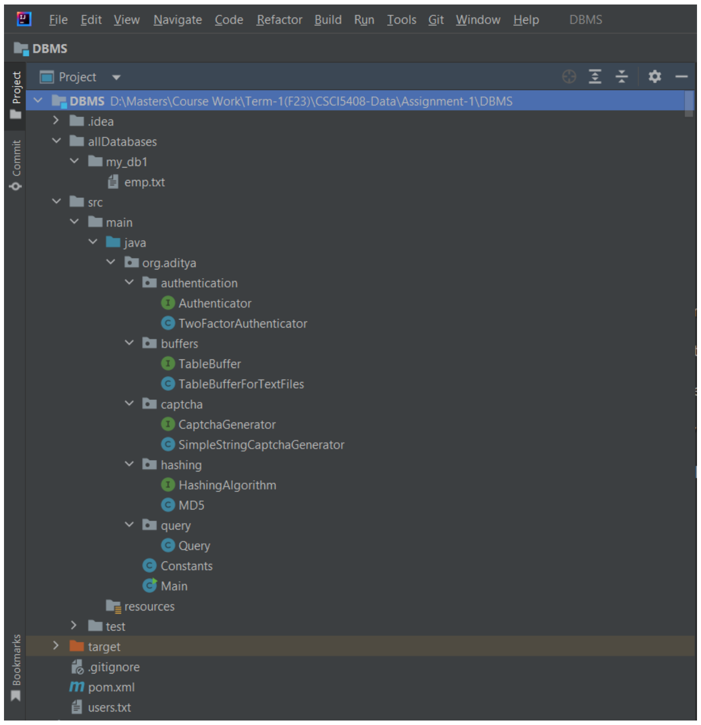
The custom solution proposed showed high performance but specific metrics and comparisons should have made the argument stronger. It would have been helpful if scenarios of this custom solution were given and outperformed the federated database approach [2].

There is no proper conclusion or room for future talks at last the methodologies and experiments should be discussed in more detail [2].

Problem 2:

Project Design & Structure:

Project Structure


File Edit View Navigate Code Refactor Build Run Tools Git Window Help DBMS
DBMS
Project
Commit
Bookmarks

```
DBMS D:\Masters\Course Work\Term-1(F23)\CSCI5408-Data\Assignment-1\DBMS
  > .idea
  v allDatabases
    v my_db1
        emp.txt
  v src
    v main
      v java
        v org.aditya
          v authentication
              Authenticator
              TwoFactorAuthenticator
          v buffers
              TableBuffer
              TableBufferForTextFiles
          v captcha
              CaptchaGenerator
              SimpleStringCaptchaGenerator
          v hashing
              HashingAlgorithm
              MD5
          v query
              Query
            Constants
            Main
        resources
    > test
  > target
    .gitignore
    pom.xml
    users.txt
```

Figure 1: Java Project Structure [3][4][5].

The project is divided across 5 sub-packages (authentication, buffers, captcha, hashing, query) and 1 main package (org.aditya).

1. **authentication** package has interface & class related to user signup and login functionality.
2. **buffers** package has interface & class which supports loading and saving of table data from file to data structure.
3. **captcha** package has interface and class which supports simple string captcha generation and validation.
4. **hashing** package has interface and class which supports string hashing. It's used for password hashing.
5. **query** package is a like core package that has a class containing different methods to execute different queries.
6. **org.aditya** package has a **Constants** class which host the path configuration for database and user details. It also has the **Main** class which is the entry point for the DBMS console

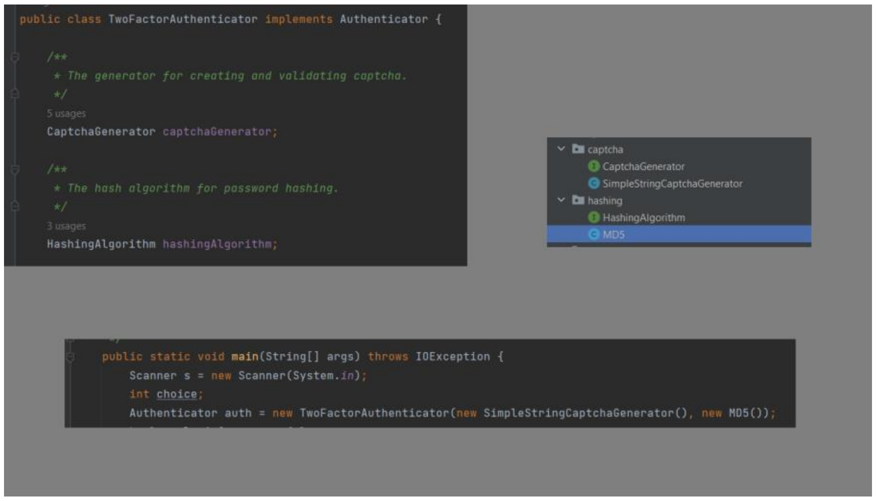
application. It controls the user flow and calls appropriate methods as needed, depending on user input.

Project Design

The project follows SOLID Design principles as described below:

1. **Single Responsibility [6] [7]:** Each class has <u>only 1 responsibility</u> here in the project.
    a. **TwoFactorAuthentication class:** It only handles user management task. Login & Sign-up.
    b. **TableBufferForTextFiles class:** It only handles buffer. It loads and saves the ArrayList buffer for text files.
    c. **SimpleStringCaptchaGenerator class:** It only manages captcha. It generates & validates the captcha.
    d. **MD5 class:** It only provides MD5 hashing functionality.
    e. **Query class:** It only handles a line of query. It executes various types of query.
    f. **Constants class:** It only manages the constants used for project configuration.

2. **Open-Closed Principle [6] [7]:** The modules are open for extension but closed for modification here in the project due to interfaces. For example, the TwoFactorAuthenticator class refers to the interfaces of hashing and captcha. So, if we need to used a different hashing algorithm instead of MD5 or different captcha instead of SimpleString, we can extend the functionality without any modifications in the TwoFactorAuthenticator class.



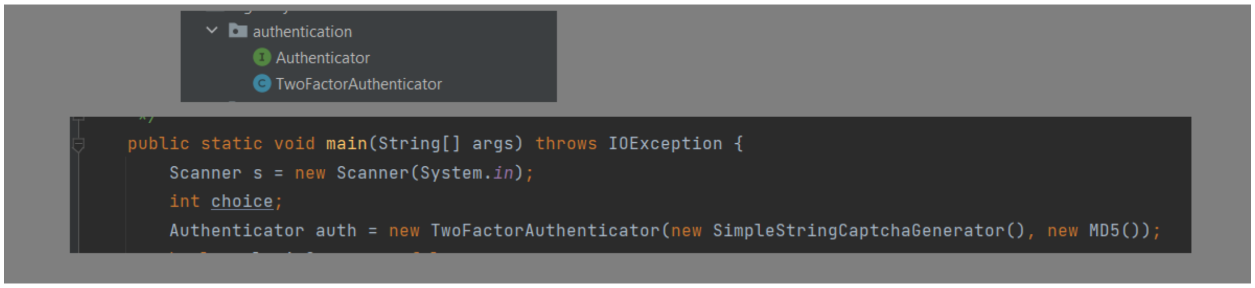
```
public class TwoFactorAuthenticator implements Authenticator {

    /**
     * The generator for creating and validating captcha.
     */
    5 usages
    CaptchaGenerator captchaGenerator;

    /**
     * The hash algorithm for password hashing.
     */
    3 usages
    HashingAlgorithm hashingAlgorithm;
```

```
    public static void main(String[] args) throws IOException {
        Scanner s = new Scanner(System.in);
        int choice;
        Authenticator auth = new TwoFactorAuthenticator(new SimpleStringCaptchaGenerator(), new MD5());
```

Figure 2: Usage of interface in class. Using specific class instance while calling constructor [3][4][5].

3. **Liskov Substitution Principle [6] [7]:** The project supports substitution of the subclass, wherever the super class or super type reference is there. This is also achieved using the

interfaces. The Authenticator interface's auth reference variables currently points to the TwoFactorAuthenticator class, but it can be replaced with a ThreeFactorAuthenticator class without changing the reference variable of interface here.



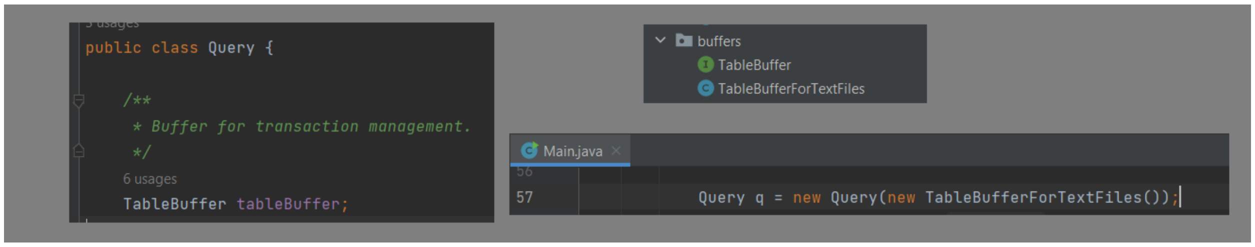
authentication
Authenticator
TwoFactorAuthenticator

```
public static void main(String[] args) throws IOException {
    Scanner s = new Scanner(System.in);
    int choice;
    Authenticator auth = new TwoFactorAuthenticator(new SimpleStringCaptchaGenerator(), new MD5());
```

Figure 3: Reference variable of interface points to a Sub-class Object [3][4][5].

4. **Interface Segregation [6] [7]:** There are no large interfaces in this project yet which has lot of methods in it so no segregation is needed yet. All interfaces are small and created in such a way that the implementing classes can focus on only those methods that are related to the class and don't need to implement un-necessary methods.

5. **Dependency Inversion [6] [7]:** In the project, high-level dependency (eg: Query class) is not depending on the low-level dependency (eg: TableBufferForTextFiles class) directly but instead depends on the interface/abstraction (eg: TableBuffer ineterface).



```
public class Query {

    /**
     * Buffer for transaction management.
     */
    6 usages
    TableBuffer tableBuffer;
```

buffers
TableBuffer
TableBufferForTextFiles

Main.java

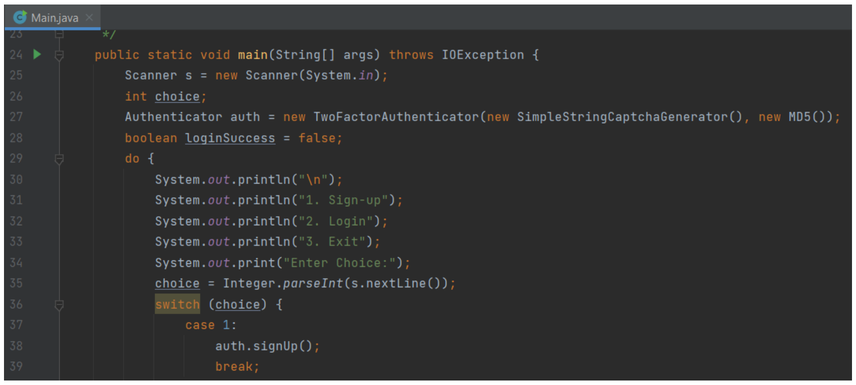
```
Query q = new Query(new TableBufferForTextFiles());
```

Figure 4: Query depends on Table buffer interface, instead of the implementing class [3][4][5].

Task A: User Authentication and Sign-up

Implementation of Sign-up

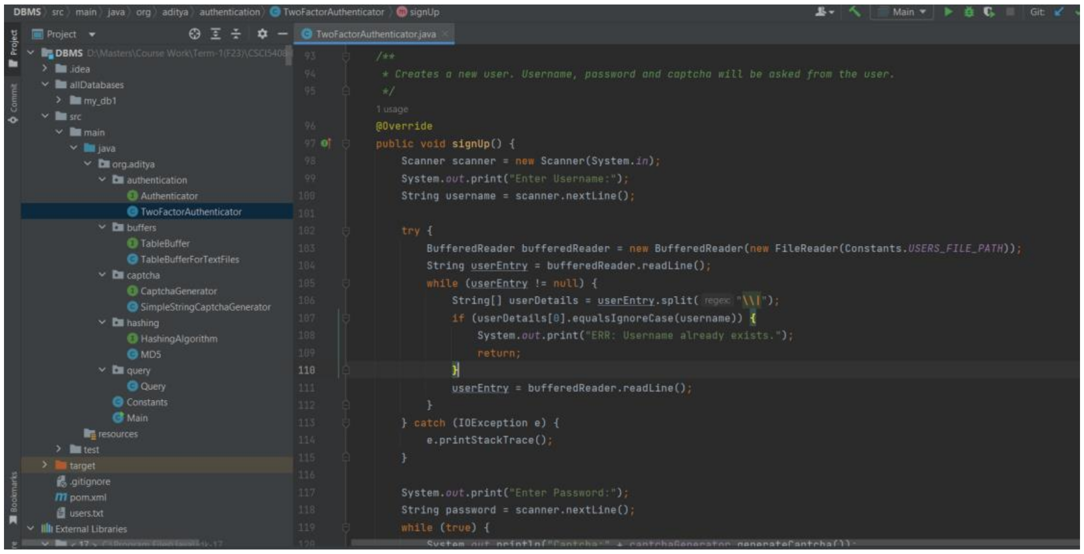
The menu is displayed using the main method which allows to call the sign-up functionality given in the TwoFactorAuthenticator class.



```java
    */
    public static void main(String[] args) throws IOException {
        Scanner s = new Scanner(System.in);
        int choice;
        Authenticator auth = new TwoFactorAuthenticator(new SimpleStringCaptchaGenerator(), new MD5());
        boolean loginSuccess = false;
        do {
            System.out.println("\n");
            System.out.println("1. Sign-up");
            System.out.println("2. Login");
            System.out.println("3. Exit");
            System.out.print("Enter Choice:");
            choice = Integer.parseInt(s.nextLine());
            switch (choice) {
                case 1:
                    auth.signUp();
                    break;
```

Figure 5: Menu in main method that shows the sign-up option [3][4][5].

The signUp() method then creates the user by taking username, password as input. Also it displays a random string captcha. The user details are then stored in the .txt file with the format of “username|hashed_password”. MD5 hashing is used for storing passwords.

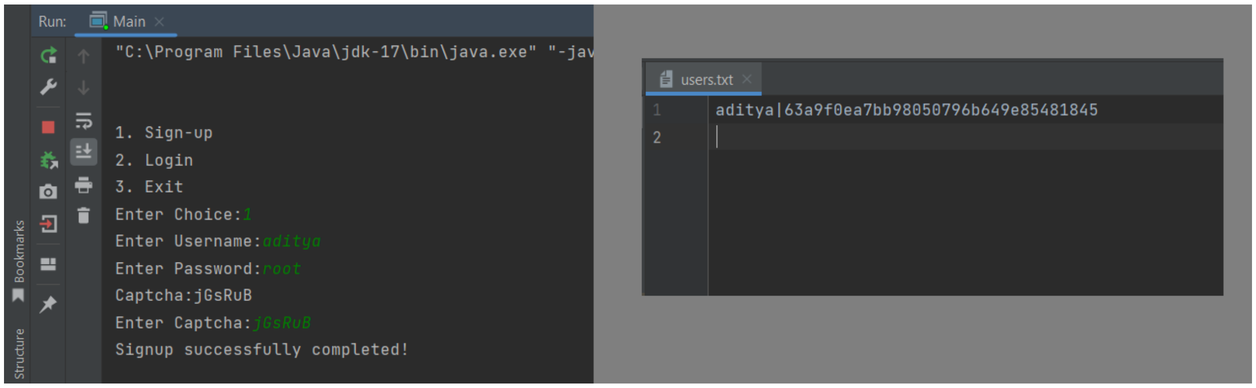


```java
    /**
     * Creates a new user. Username, password and captcha will be asked from the user.
     */
    @Override
    public void signUp() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Username:");
        String username = scanner.nextLine();

        try {
            BufferedReader bufferedReader = new BufferedReader(new FileReader(Constants.USERS_FILE_PATH));
            String userEntry = bufferedReader.readLine();
            while (userEntry != null) {
                String[] userDetails = userEntry.split( regex: "\\|");
                if (userDetails[0].equalsIgnoreCase(username)) {
                    System.out.print("ERR: Username already exists.");
                    return;
                }
                userEntry = bufferedReader.readLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.print("Enter Password:");
        String password = scanner.nextLine();
        while (true) {
            System.out.println("Captcha:" + captchaGenerator.generateCaptcha());
```

Figure 6: signUp() method of TwoFactorAuthenticator class [3][4][5].

Testing of Sign-up

T1 - Sign-up successful: When user enters a fresh username, some password and valid captcha, the user should be created in the .txt file. Actual Output: entry appears in .txt file.

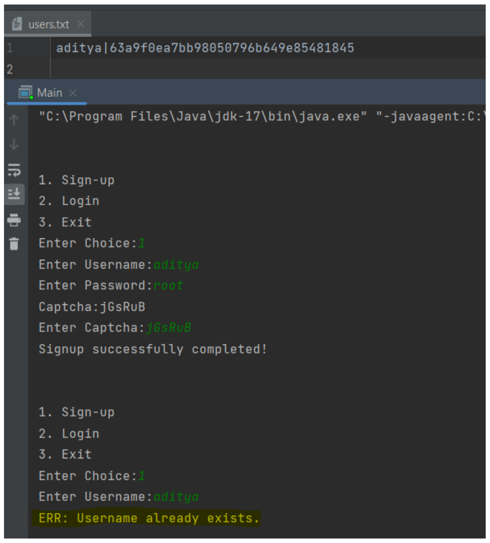


```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-jav

1. Sign-up
2. Login
3. Exit
Enter Choice:1
Enter Username:aditya
Enter Password:root
Captcha:jGsRuB
Enter Captcha:jGsRuB
Signup successfully completed!
```

```
aditya|63a9f0ea7bb98050796b649e85481845
```

Figure 7: Sign-up success [3][4][5].

T2 - Sign-up failed for duplicate user: When user enters an existing username, error should appear. Actual Output: Error appears indicating username already exists.

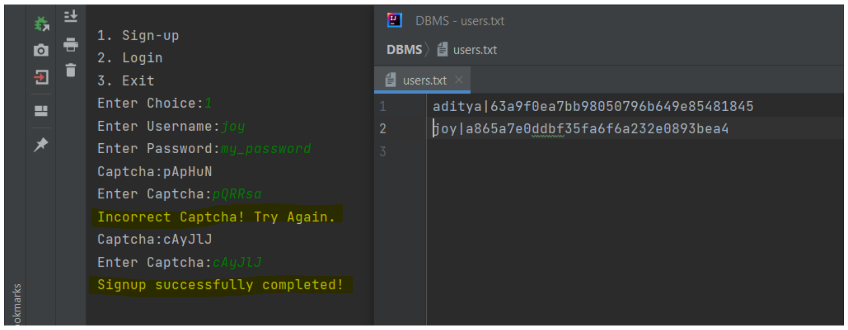


```
aditya|63a9f0ea7bb98050796b649e85481845
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\

1. Sign-up
2. Login
3. Exit
Enter Choice:1
Enter Username:aditya
Enter Password:root
Captcha:jGsRuB
Enter Captcha:jGsRuB
Signup successfully completed!

1. Sign-up
2. Login
3. Exit
Enter Choice:1
Enter Username:aditya
ERR: Username already exists.
```

Figure 8: Sign-up failed, due to duplicate username [3][4][5].

T3 - Sign-up success with 2nd captcha attempt: When user enters a fresh username, some password but invalid captcha first but valid later, the user should be created in the .txt file. Actual Output: captcha error appears once but then user entry appears in .txt file.



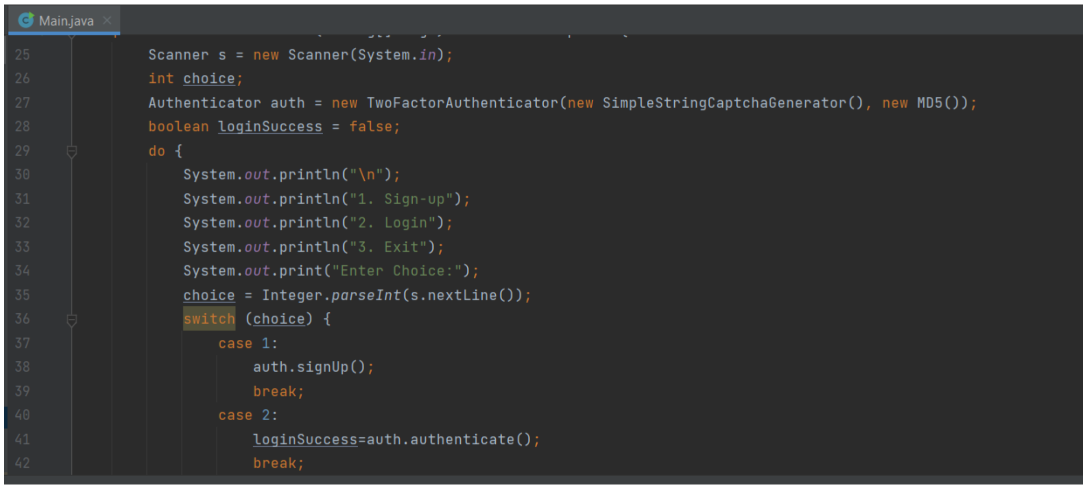
```
1. Sign-up
2. Login
3. Exit
Enter Choice:1
Enter Username:joy
Enter Password:my_password
Captcha:pApHuN
Enter Captcha:pQRRsa
Incorrect Captcha! Try Again.
Captcha:cAyJlJ
Enter Captcha:cAyJlJ
Signup successfully completed!
```

```
aditya|63a9f0ea7bb98050796b649e85481845
joy|a865a7e0ddbf35fa6f6a232e0893bea4
```

Figure 9: Sign-up success with 2nd captcha attempt [3][4][5].

Implementation of Authentication (Login)

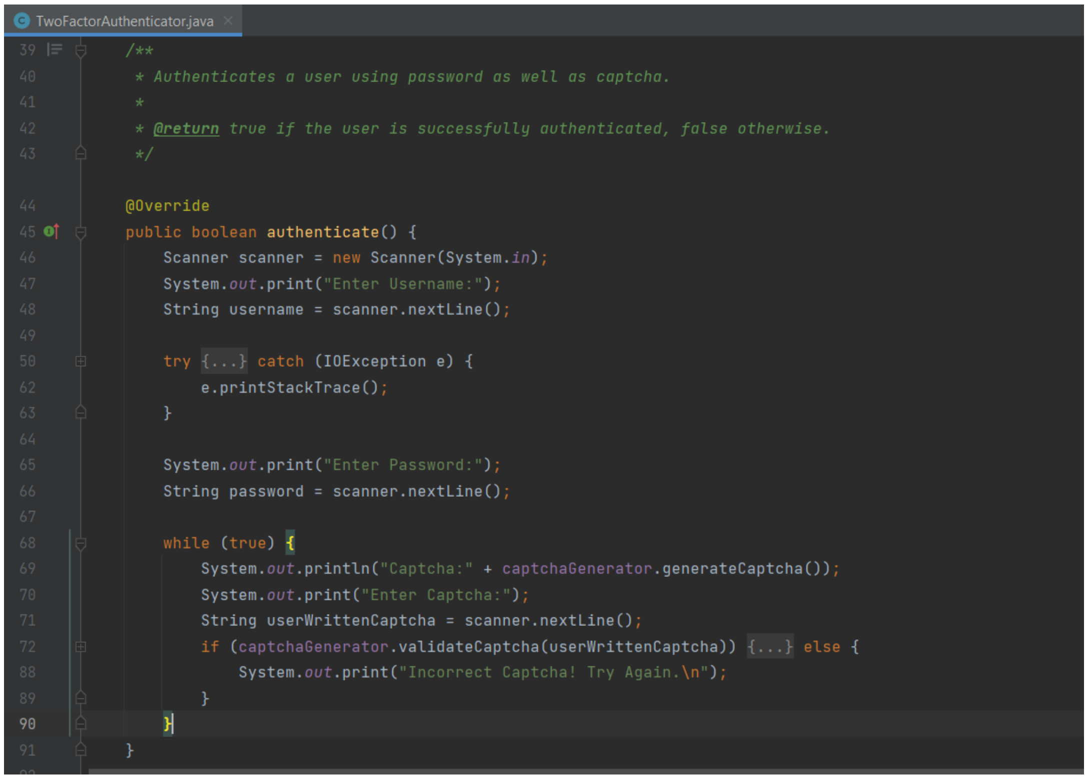
The menu is displayed using the main method which allows to call the login functionality given in the TwoFactorAuthenticator class.



```java
        Scanner s = new Scanner(System.in);
        int choice;
        Authenticator auth = new TwoFactorAuthenticator(new SimpleStringCaptchaGenerator(), new MD5());
        boolean loginSuccess = false;
        do {
            System.out.println("\n");
            System.out.println("1. Sign-up");
            System.out.println("2. Login");
            System.out.println("3. Exit");
            System.out.print("Enter Choice:");
            choice = Integer.parseInt(s.nextLine());
            switch (choice) {
                case 1:
                    auth.signUp();
                    break;
                case 2:
                    loginSuccess=auth.authenticate();
                    break;
```

Figure 10: Menu in main method that shows the login option [3][4][5].

The authenticate() method then takes the username, password as input. Also it displays a random string captcha. If the credentials are valid, then login is successful but if username is invalid or password is incorrect, the login fails.

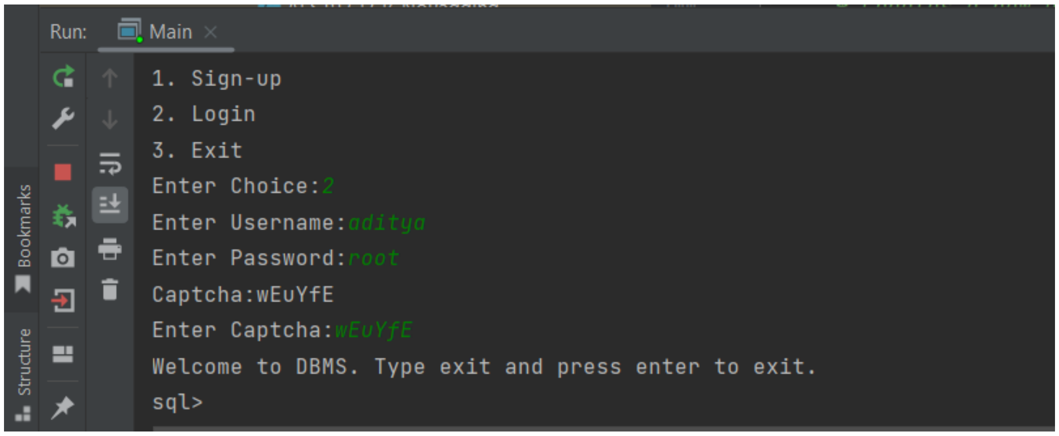


```java
/**
 * Authenticates a user using password as well as captcha.
 *
 * @return true if the user is successfully authenticated, false otherwise.
 */

@Override
public boolean authenticate() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter Username:");
    String username = scanner.nextLine();

    try {...} catch (IOException e) {
        e.printStackTrace();
    }

    System.out.print("Enter Password:");
    String password = scanner.nextLine();

    while (true) {
        System.out.println("Captcha:" + captchaGenerator.generateCaptcha());
        System.out.print("Enter Captcha:");
        String userWrittenCaptcha = scanner.nextLine();
        if (captchaGenerator.validateCaptcha(userWrittenCaptcha)) {...} else {
            System.out.print("Incorrect Captcha! Try Again.\n");
        }
    }
}
```

Figure 11: authenticate() method of TwoFactorAuthenticator class [3][4][5].

Testing of Authenticate(login)

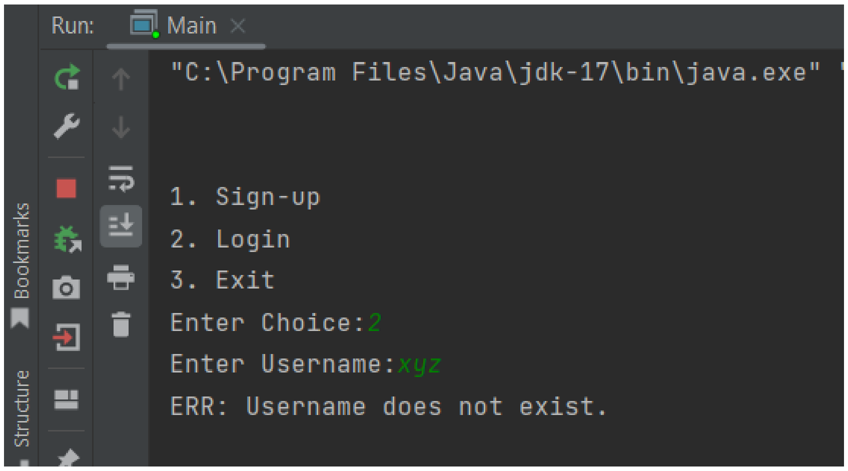
T1 - Login successful: When user enters a valid username and password, login should be successful and sql prompt to appear Actual Output: sql prompt appears.



```
1. Sign-up
2. Login
3. Exit
Enter Choice:2
Enter Username:aditya
Enter Password:root
Captcha:wEuYfE
Enter Captcha:wEuYfE
Welcome to DBMS. Type exit and press enter to exit.
sql>
```

Figure 12: login success [3][4][5].

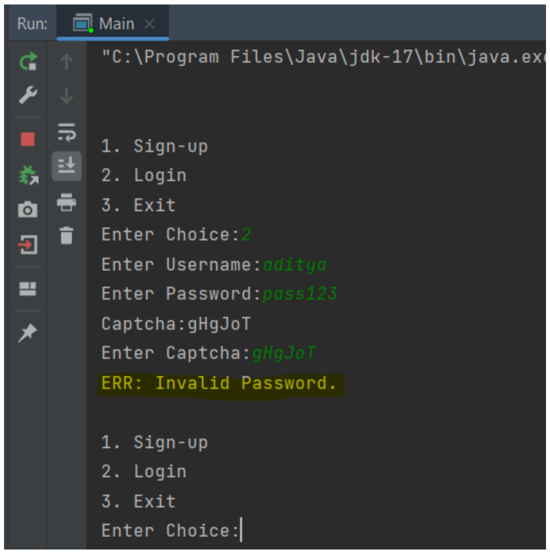
T2 - Login failed for incorrect user: When user enters a username which is not registered, error should appear. Actual Output: Error appears indicating username does not exist.



```
"C:\Program Files\Java\jdk-17\bin\java.exe"


1. Sign-up
2. Login
3. Exit
Enter Choice:2
Enter Username:xyz
ERR: Username does not exist.
```

Figure 13: login failed, due to invalid username [3][4][5].

T3 - Login failed for incorrect password: When user enters a valid username but incorrect password, error should appear. Actual Output: Error appears indicating incorrect password.

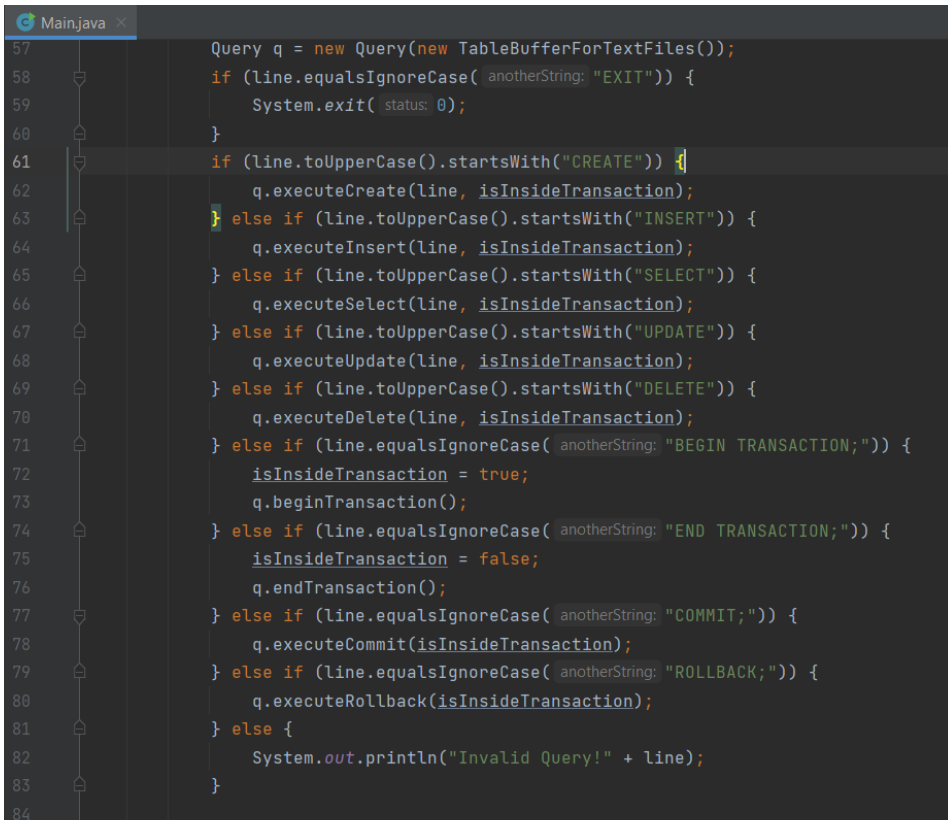


```
"C:\Program Files\Java\jdk-17\bin\java.ex


1. Sign-up
2. Login
3. Exit
Enter Choice:2
Enter Username:aditya
Enter Password:pass123
Captcha:gHgJoT
Enter Captcha:gHgJoT
ERR: Invalid Password.

1. Sign-up
2. Login
3. Exit
Enter Choice:
```

Figure 14: login failed, due to invalid password [3][4][5].

Task B: Queries (DDL & DML)

The main method calls the specific query execution method based on the starting keyword.

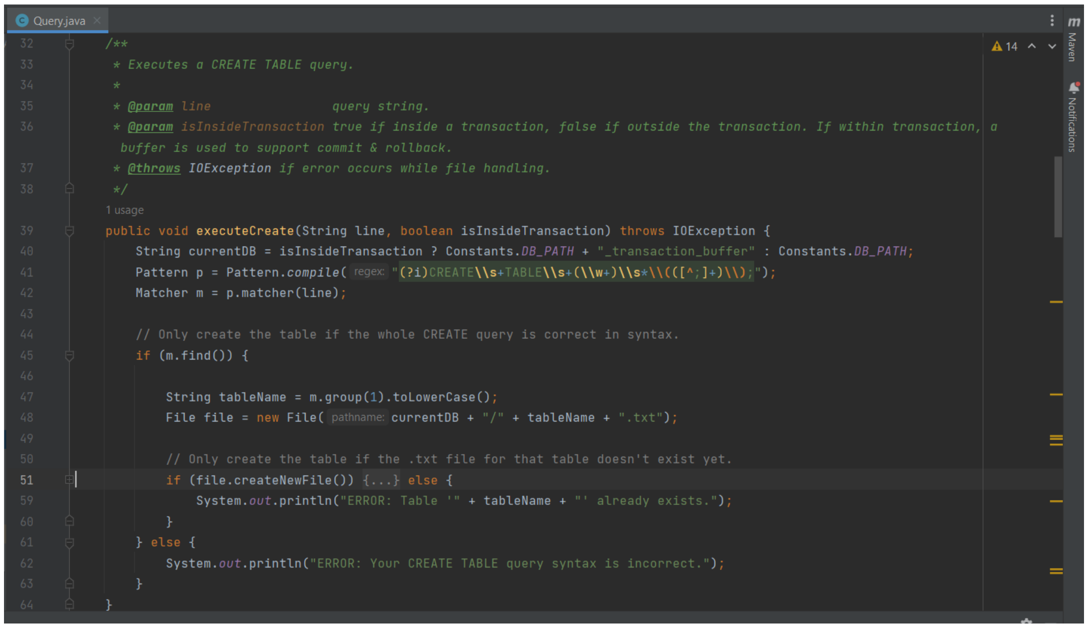


Main.java

```java
            Query q = new Query(new TableBufferForTextFiles());
            if (line.equalsIgnoreCase( anotherString: "EXIT")) {
                System.exit( status: 0);
            }
            if (line.toUpperCase().startsWith("CREATE")) {
                q.executeCreate(line, isInsideTransaction);
            } else if (line.toUpperCase().startsWith("INSERT")) {
                q.executeInsert(line, isInsideTransaction);
            } else if (line.toUpperCase().startsWith("SELECT")) {
                q.executeSelect(line, isInsideTransaction);
            } else if (line.toUpperCase().startsWith("UPDATE")) {
                q.executeUpdate(line, isInsideTransaction);
            } else if (line.toUpperCase().startsWith("DELETE")) {
                q.executeDelete(line, isInsideTransaction);
            } else if (line.equalsIgnoreCase( anotherString: "BEGIN TRANSACTION;")) {
                isInsideTransaction = true;
                q.beginTransaction();
            } else if (line.equalsIgnoreCase( anotherString: "END TRANSACTION;")) {
                isInsideTransaction = false;
                q.endTransaction();
            } else if (line.equalsIgnoreCase( anotherString: "COMMIT;")) {
                q.executeCommit(isInsideTransaction);
            } else if (line.equalsIgnoreCase( anotherString: "ROLLBACK;")) {
                q.executeRollback(isInsideTransaction);
            } else {
                System.out.println("Invalid Query!" + line);
            }
```

Figure 15: Main method calling specific query execution methods [3][4][5].

Implementation of CREATE
The executeCreate() method inside the Query class has the implementation of the create query. The main method calls this method only when the query starts with CREATE keyword. Regex is used to identify the correct syntax. The executeCreate() method creates a text file with the column names in the first line. Error is thrown if the .txt file(i.e. Table Name) already exists. The column names are separated by pipe delimiter '|'.

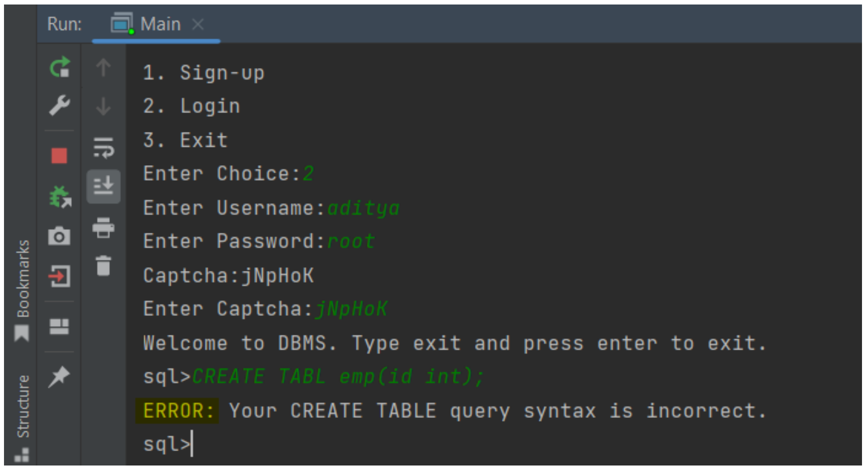


```java
/**
 * Executes a CREATE TABLE query.
 *
 * @param line                query string.
 * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, a
  buffer is used to support commit & rollback.
 * @throws IOException if error occurs while file handling.
 */
1 usage
public void executeCreate(String line, boolean isInsideTransaction) throws IOException {
    String currentDB = isInsideTransaction ? Constants.DB_PATH + "_transaction_buffer" : Constants.DB_PATH;
    Pattern p = Pattern.compile( regex: "(?i)CREATE\\s+TABLE\\s+(\\w+)\\s*\\(([^;]+)\\);");
    Matcher m = p.matcher(line);

    // Only create the table if the whole CREATE query is correct in syntax.
    if (m.find()) {

        String tableName = m.group(1).toLowerCase();
        File file = new File( pathname: currentDB + "/" + tableName + ".txt");

        // Only create the table if the .txt file for that table doesn't exist yet.
        if (file.createNewFile()) {...} else {
            System.out.println("ERROR: Table '" + tableName + "' already exists.");
        }
    } else {
        System.out.println("ERROR: Your CREATE TABLE query syntax is incorrect.");
    }
}
```

Figure 16: Implementation of executeCreate()[3][4][5].

Testing of CREATE

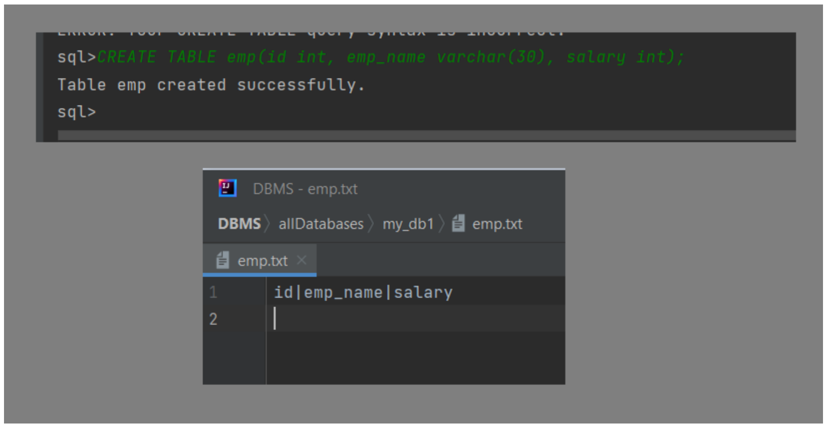
T1 – Create Table unsuccessful due to invalid syntax: When user enters a Create Table query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
1. Sign-up
2. Login
3. Exit
Enter Choice:2
Enter Username:aditya
Enter Password:root
Captcha:jNpHoK
Enter Captcha:jNpHoK
Welcome to DBMS. Type exit and press enter to exit.
sql>CREATE TABL emp(id int);
ERROR: Your CREATE TABLE query syntax is incorrect.
sql>
```

Figure 17: create query with spelling mistake [3][4][5].

T2 - Create Table successful: When user enters a Create Table query with no syntax error, a table should be created (.txt file). Actual Output: Table is created (.txt file is created).



```
ERROR: Your CREATE TABLE query syntax is incorrect.
sql>CREATE TABLE emp(id int, emp_name varchar(30), salary int);
Table emp created successfully.
sql>
```

DBMS - emp.txt

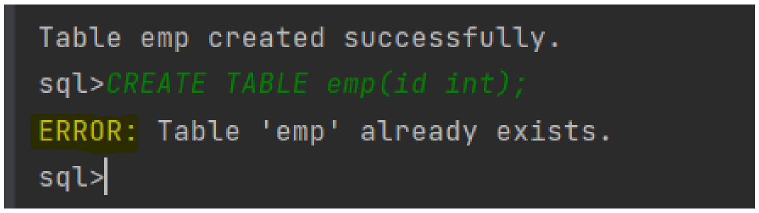
DBMS 〉 allDatabases 〉 my_db1 〉 emp.txt

emp.txt

```
id|emp_name|salary

```

Figure 18: create table successful [3][4][5].

T3 – Create Table unsuccessful due to duplicate table name: When user enters a Create Table query with a table name which is already present. Actual Output: table name error is displayed.

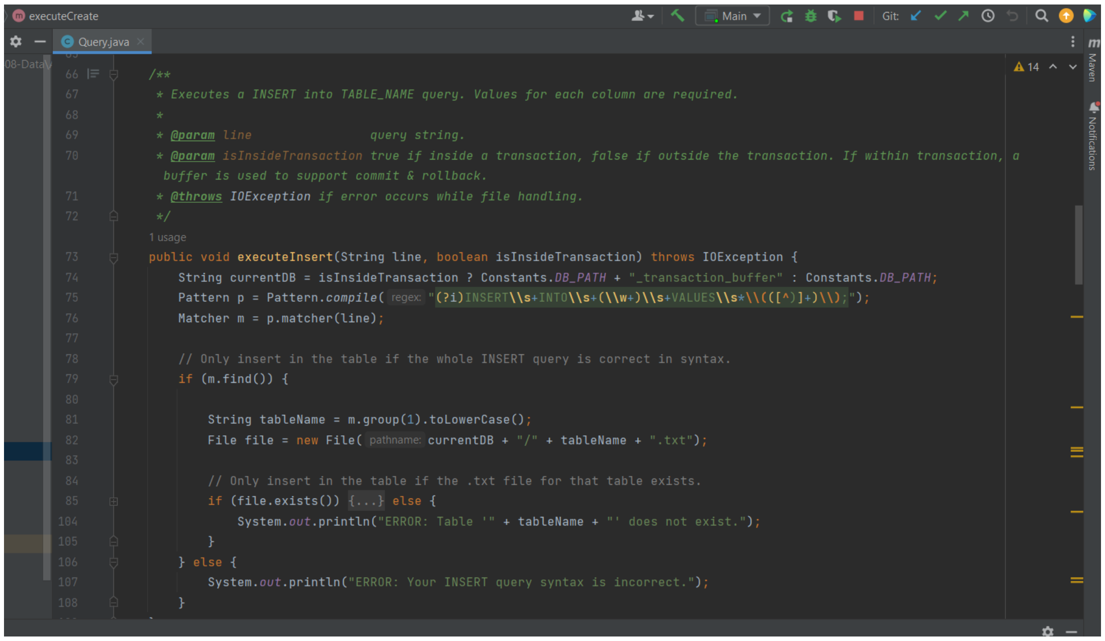


```
Table emp created successfully.
sql>CREATE TABLE emp(id int);
ERROR: Table 'emp' already exists.
sql>
```

Figure 19: create table failed, due to duplicate table name [3][4][5].

Implementation of INSERT

The executeInsert() method inside the Query class has the implementation of the insert query. The main method calls this method only when the query starts with INSERT keyword. Regex is used to identify the correct syntax. The executeInsert() method creates a new row in the table by adding a new line in the .txt file of that table. The row values are separated by pipe delimiter '|'.

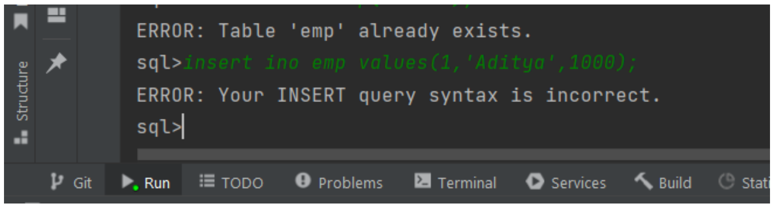


```java
/**
 * Executes a INSERT into TABLE_NAME query. Values for each column are required.
 *
 * @param line                  query string.
 * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, a
 buffer is used to support commit & rollback.
 * @throws IOException if error occurs while file handling.
 */
public void executeInsert(String line, boolean isInsideTransaction) throws IOException {
    String currentDB = isInsideTransaction ? Constants.DB_PATH + "_transaction_buffer" : Constants.DB_PATH;
    Pattern p = Pattern.compile( regex: "(?i)INSERT\\s+INTO\\s+(\\w+)\\s+VALUES\\s*\\(([^)]+)\\);");
    Matcher m = p.matcher(line);

    // Only insert in the table if the whole INSERT query is correct in syntax.
    if (m.find()) {

        String tableName = m.group(1).toLowerCase();
        File file = new File( pathname: currentDB + "/" + tableName + ".txt");

        // Only insert in the table if the .txt file for that table exists.
        if (file.exists()) {...} else {
            System.out.println("ERROR: Table '" + tableName + "' does not exist.");
        }
    } else {
        System.out.println("ERROR: Your INSERT query syntax is incorrect.");
    }
```

Figure 20: Implementation of executeInsert() [3][4][5].

Testing of INSERT

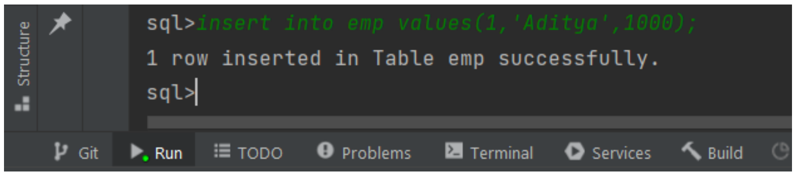
T1 – Insert unsuccessful due to invalid syntax: When user enters Insert Table query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
ERROR: Table 'emp' already exists.
sql>insert ino emp values(1,'Aditya',1000);
ERROR: Your INSERT query syntax is incorrect.
sql>
```

Figure 21: insert query with spelling mistake [3][4][5].

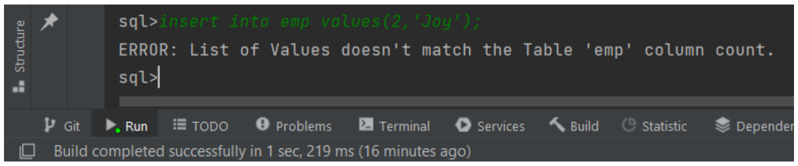
T2 - Insert successful: When user enters Insert query with no syntax error, a row should be created in the table (.txt file). Actual Output: row is created in table (.txt file).



```
sql>insert into emp values(1,'Aditya',1000);
1 row inserted in Table emp successfully.
sql>
```

Figure 22: row inserted successfully [3][4][5].

T3 – Insert unsuccessful due to column count mismatch: When user enters Insert query with number of values les or more than column count, error should appear. Actual Output: column count error is displayed.

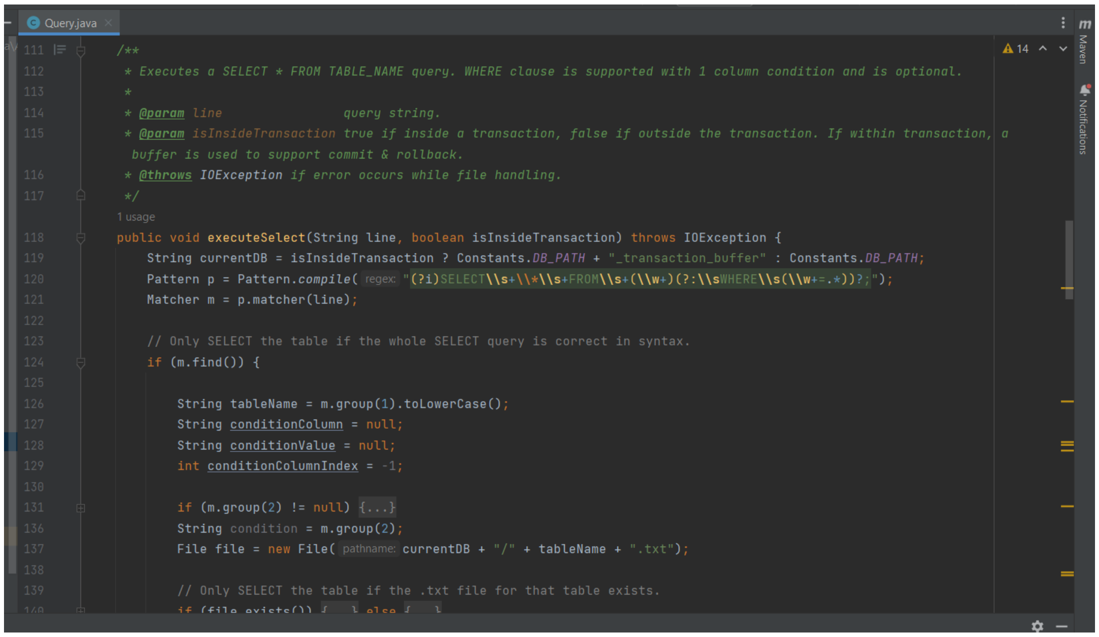


```
sql>insert into emp values(2,'Joy');
ERROR: List of Values doesn't match the Table 'emp' column count.
sql>
```

Figure 23: insert row failed, due to value and column mismatch [3][4][5].

Implementation of SELECT

The executeSelect() method inside the Query class has the implementation of the select query. The main method calls this method only when the query starts with SELECT keyword. Regex is used to identify the correct syntax. The executeSelect() method loads the table data into a buffer first and then displays it on the console. ArrayList of String ArrayList is used as buffer. Where each entry in parent stores rows and the child ArrayList stores rows values in strings. WHERE clause is optional and if used, only supports 1 condition.



```java
/**
 * Executes a SELECT * FROM TABLE_NAME query. WHERE clause is supported with 1 column condition and is optional.
 *
 * @param line                  query string.
 * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, a
  buffer is used to support commit & rollback.
 * @throws IOException if error occurs while file handling.
 */
1 usage
public void executeSelect(String line, boolean isInsideTransaction) throws IOException {
    String currentDB = isInsideTransaction ? Constants.DB_PATH + "_transaction_buffer" : Constants.DB_PATH;
    Pattern p = Pattern.compile("(?i)SELECT\\s+\\*\\s+FROM\\s+(\\w+)(?:\\sWHERE\\s(\\w+=.*))?;");
    Matcher m = p.matcher(line);

    // Only SELECT the table if the whole SELECT query is correct in syntax.
    if (m.find()) {

        String tableName = m.group(1).toLowerCase();
        String conditionColumn = null;
        String conditionValue = null;
        int conditionColumnIndex = -1;

        if (m.group(2) != null) {...}
        String condition = m.group(2);
        File file = new File(currentDB + "/" + tableName + ".txt");

        // Only SELECT the table if the .txt file for that table exists.
        if (file.exists()) {...} else {...}
```

Figure 24: Implementation of executeSelect() [3][4][5].

Testing of SELECT

T1 – Select unsuccessful due to invalid syntax: When user enters Select query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.

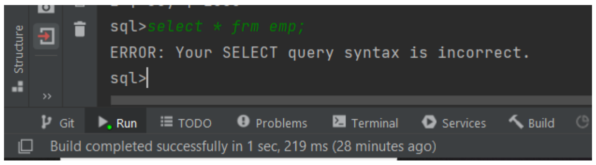

```
sql>select * frm emp;
ERROR: Your SELECT query syntax is incorrect.
sql>
```
Build completed successfully in 1 sec, 219 ms (28 minutes ago)

Figure 25: select query with spelling mistake [3][4][5].

T2 - Select successful: When user enters select query with no syntax error, without where clause, all rows from of the table should be displayed. Actual Output: All rows are displayed.

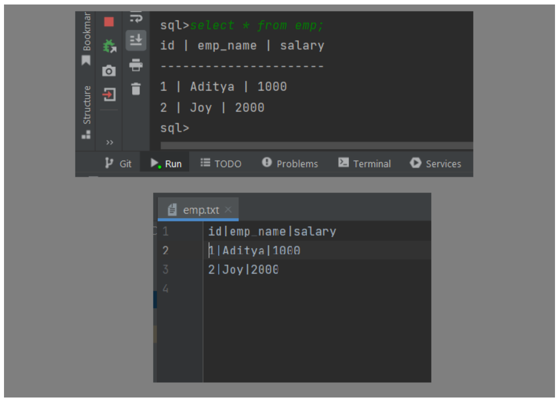
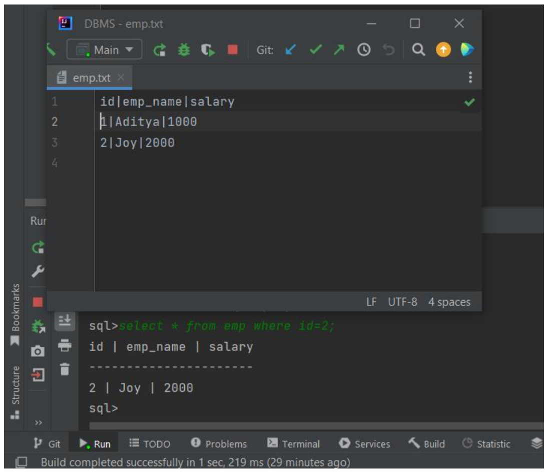

```
sql>select * from emp;
id | emp_name | salary
---------------------
1 | Aditya | 1000
2 | Joy | 2000
sql>
```
emp.txt
```
id|emp_name|salary
1|Aditya|1000
2|Joy|2000
```

Figure 26: Select: All rows are displayed [3][4][5].

T3 – Select successful with WHERE clause: When user enters select query with no syntax error, with where clause, filtered rows from of the table should be displayed. Actual Output: Rows matching the condition are displayed.

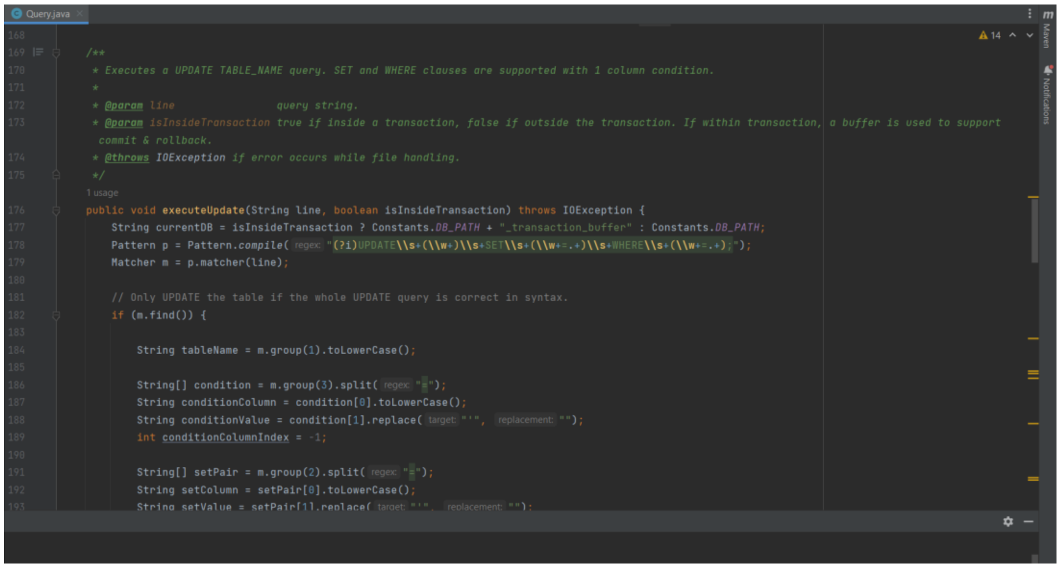

DBMS - emp.txt

emp.txt
```
id|emp_name|salary
1|Aditya|1000
2|Joy|2000
```
```
sql>select * from emp where id=2;
id | emp_name | salary
---------------------
2 | Joy | 2000
sql>
```
Build completed successfully in 1 sec, 219 ms (29 minutes ago)

Figure 27: Select: some rows are displayed [3][4][5].

Implementation of UPDATE

The executeUpdate() method inside the Query class has the implementation of the update query. The main method calls this method only when the query starts with UPDATE keyword. Regex is used to identify the correct syntax. The executeUpdate() method loads the table data into a buffer first, updates the particular values in buffer and saves the buffer back to text file. WHERE clause is supported and based on the given condition buffer is filtered.

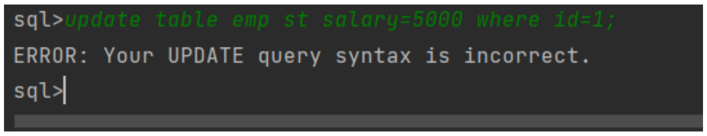


```java
/**
 * Executes a UPDATE TABLE_NAME query. SET and WHERE clauses are supported with 1 column condition.
 *
 * @param line                 query string.
 * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, a buffer is used to support
  commit & rollback.
 * @throws IOException if error occurs while file handling.
 */
1 usage
public void executeUpdate(String line, boolean isInsideTransaction) throws IOException {
    String currentDB = isInsideTransaction ? Constants.DB_PATH + "_transaction_buffer" : Constants.DB_PATH;
    Pattern p = Pattern.compile( regex: "(?i)UPDATE\\s+(\\w+)\\s+SET\\s+(\\w+=.+)\\s+WHERE\\s+(\\w+=.+);");
    Matcher m = p.matcher(line);

    // Only UPDATE the table if the whole UPDATE query is correct in syntax.
    if (m.find()) {

        String tableName = m.group(1).toLowerCase();

        String[] condition = m.group(3).split( regex: "=");
        String conditionColumn = condition[0].toLowerCase();
        String conditionValue = condition[1].replace( target: "'",  replacement: "");
        int conditionColumnIndex = -1;

        String[] setPair = m.group(2).split( regex: "=");
        String setColumn = setPair[0].toLowerCase();
        String setValue = setPair[1].replace( target: "'",  replacement: "");
```

Figure 28: Implementation of executeUpdate() [3][4][5].

Testing of UPDATE

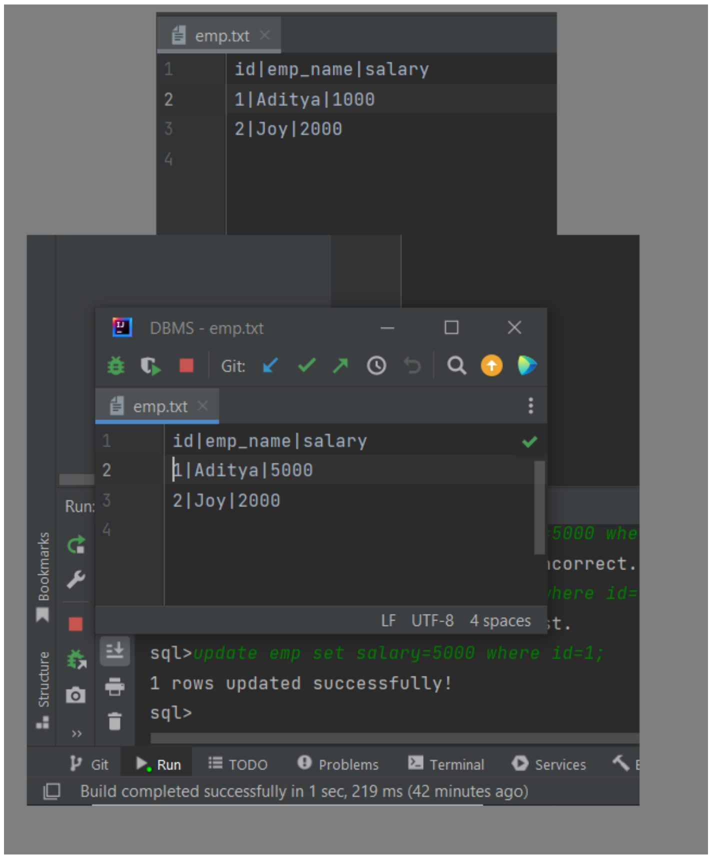
T1 – Update unsuccessful due to invalid syntax: When user enters Update query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
sql>update table emp st salary=5000 where id=1;
ERROR: Your UPDATE query syntax is incorrect.
sql>
```

Figure 29: update query with spelling mistake [3][4][5].

T2 - Update successful: When user enters update query with no syntax error, with set & where clause, those rows from of the table should be updated. Actual Output: applicable rows are updated.



```
emp.txt
id|emp_name|salary
1|Aditya|1000
2|Joy|2000
```

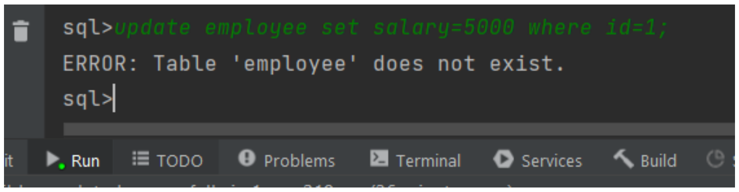
```
DBMS - emp.txt
emp.txt
id|emp_name|salary
1|Aditya|5000
2|Joy|2000
```

```
sql>update emp set salary=5000 where id=1;
1 rows updated successfully!
sql>
```

```
Build completed successfully in 1 sec, 219 ms (42 minutes ago)
```

Figure 30: Update: some rows updated [3][4][5].

T3 – Update unsuccessful due to invalid table name: When user enters update query on invalid table, error should be displayed. Actual Output: Table doesn't exist error appears.

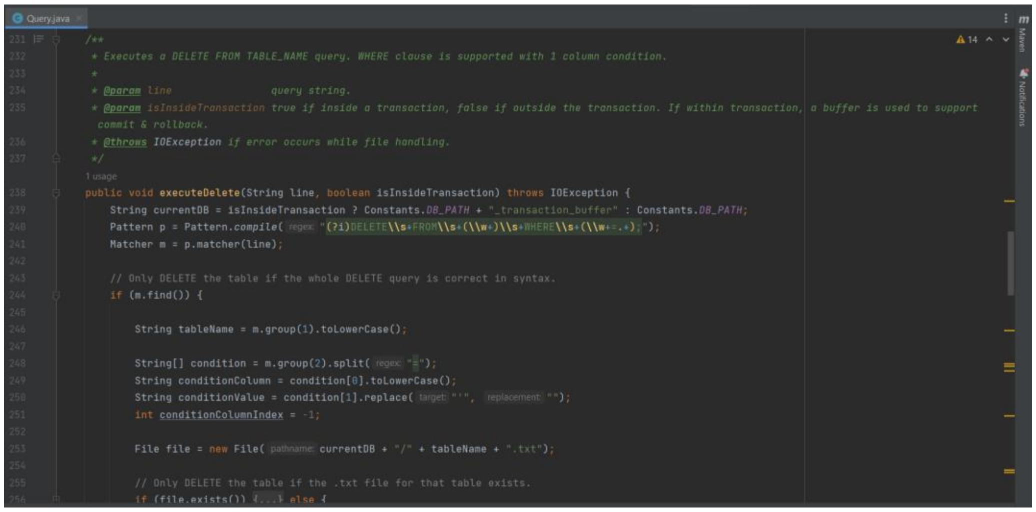


```
sql>update employee set salary=5000 where id=1;
ERROR: Table 'employee' does not exist.
sql>
```

Figure 31: Update: invalid table name [3][4][5].

Implementation of DELETE

The executeDelete() method inside the Query class has the implementation of the delete query. The main method calls this method only when the query starts with DELETE keyword. Regex is used to identify the correct syntax. The executeDelete() method loads the table data into a buffer first, deletes the particular rows in buffer and saves the buffer back to text file. WHERE clause is supported and based on the given condition buffer rows are deleted.

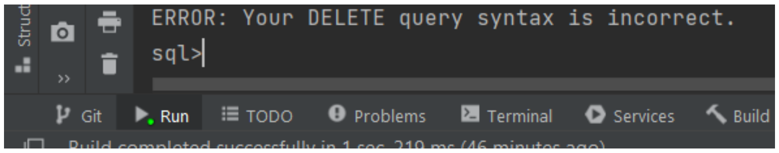


```java
/**
 * Executes a DELETE FROM TABLE_NAME query. WHERE clause is supported with 1 column condition.
 *
 * @param line                query string.
 * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, a buffer is used to support
 *  commit & rollback.
 * @throws IOException if error occurs while file handling.
 */
public void executeDelete(String line, boolean isInsideTransaction) throws IOException {
    String currentDB = isInsideTransaction ? Constants.DB_PATH + "_transaction_buffer" : Constants.DB_PATH;
    Pattern p = Pattern.compile("(?i)DELETE\\s+FROM\\s+(\\w+)\\s+WHERE\\s+(\\w+=.+);");
    Matcher m = p.matcher(line);

    // Only DELETE the table if the whole DELETE query is correct in syntax.
    if (m.find()) {

        String tableName = m.group(1).toLowerCase();

        String[] condition = m.group(2).split("=");
        String conditionColumn = condition[0].toLowerCase();
        String conditionValue = condition[1].replace("'", "");
        int conditionColumnIndex = -1;

        File file = new File(currentDB + "/" + tableName + ".txt");

        // Only DELETE the table if the .txt file for that table exists.
        if (file.exists()) {...} else {
```

Figure 32: Implementation of executeDelete() [3][4][5].

Testing of DELETE

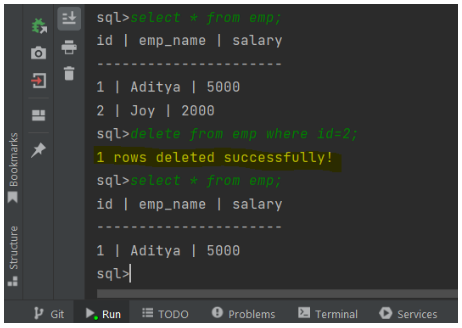
T1 – Delete unsuccessful due to invalid syntax: When user enters delete query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
ERROR: Your DELETE query syntax is incorrect.
sql>
```

Figure 33: Delete query with spelling mistake [3][4][5].

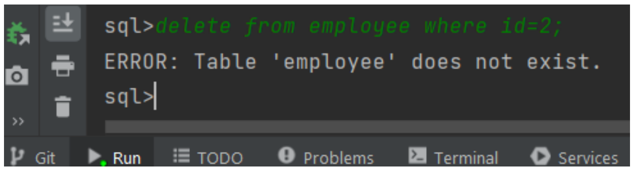
T2 - Delete successful: When user enters delete query with no syntax error, with where clause, those rows from of the table should be deleted. Actual Output: applicable rows are deleted.



```
sql>select * from emp;
id | emp_name | salary
----------------------
1 | Aditya | 5000
2 | Joy | 2000
sql>delete from emp where id=2;
1 rows deleted successfully!
sql>select * from emp;
id | emp_name | salary
----------------------
1 | Aditya | 5000
sql>
```

Figure 34: Delete: some rows deleted [3][4][5].

T3 – Delete unsuccessful due to invalid table name: When user enters delete query on invalid table, error should be displayed. Actual Output: Table doesn't exist error appears.



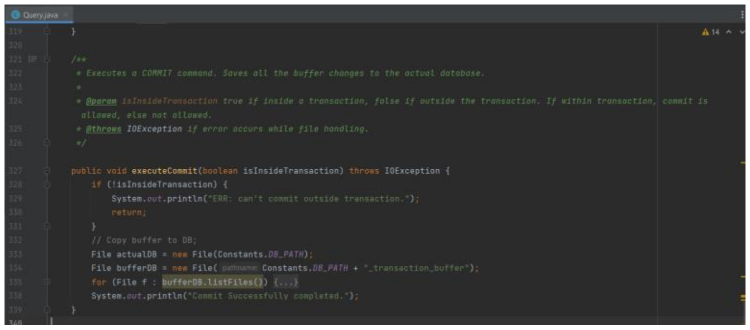
```
sql>delete from employee where id=2;
ERROR: Table 'employee' does not exist.
sql>
```

Figure 35: delete: invalid table name [3][4][5].

Task C: Transaction - commit & rollback.

The transaction is control using "Begin Transaction" and "End Transaction" commands. Commit and rollback can be used only within the transaction otherwise error is thrown. A buffer database is used to perform all the operations when inside the transaction.

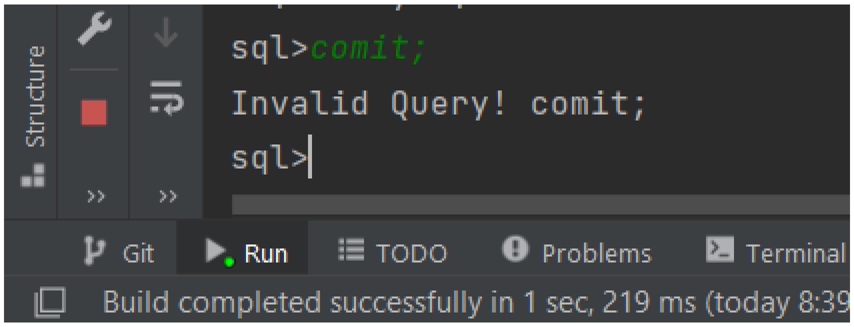
Implementation of COMMIT
The executeCommit() method inside the Query class has the implementation of the commit query. The main method calls this method only when the query is COMMIT keyword. The executeCommit() method pushes all the data from buffer database to the actual database.



```java
    }

    /**
     * Executes a COMMIT command. Saves all the buffer changes to the actual database.
     *
     * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, commit is
     allowed, else not allowed.
     * @throws IOException if error occurs while file handling.
     */

    public void executeCommit(boolean isInsideTransaction) throws IOException {
        if (!isInsideTransaction) {
            System.out.println("ERR: can't commit outside transaction.");
            return;
        }
        // Copy buffer to DB;
        File actualDB = new File(Constants.DB_PATH);
        File bufferDB = new File( pathname: Constants.DB_PATH + "_transaction_buffer");
        for (File f : bufferDB.listFiles()) {...}
        System.out.println("Commit Successfully completed.");
    }
```

Figure 36: Implementation of executeCommit()[3][4][5].

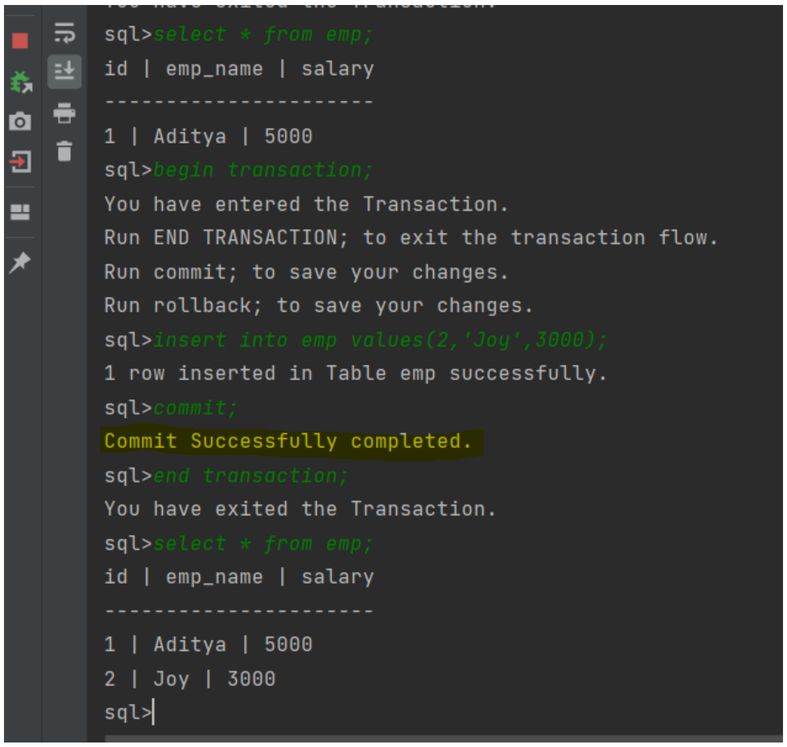
Testing of Commit
T1 – Commit unsuccessful due to invalid syntax: When user enters commit query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
sql>comit;
Invalid Query! comit;
sql>
```

Figure 37: commit query with spelling mistake [3][4][5].

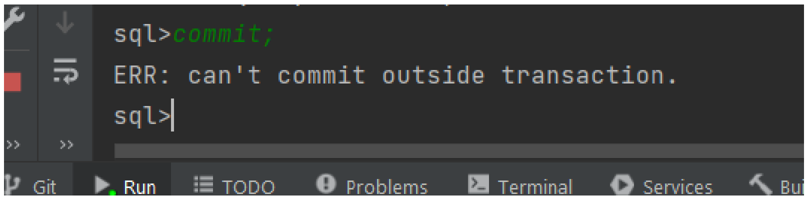
T2 - Commit successful: When user enters commit query with no syntax error the changes should be applied to actual database. Actual Output: actual database is updated.



```
sql>select * from emp;
id | emp_name | salary
----------------------
1 | Aditya | 5000
sql>begin transaction;
You have entered the Transaction.
Run END TRANSACTION; to exit the transaction flow.
Run commit; to save your changes.
Run rollback; to save your changes.
sql>insert into emp values(2,'Joy',3000);
1 row inserted in Table emp successfully.
sql>commit;
Commit Successfully completed.
sql>end transaction;
You have exited the Transaction.
sql>select * from emp;
id | emp_name | salary
----------------------
1 | Aditya | 5000
2 | Joy | 3000
sql>
```

Figure 38: commit inside transaction [3][4][5].

T3 – commit unsuccessful due to outside transaction: When user enters commit query outside transaction error should appear. Actual Output: Can't commit error appears.

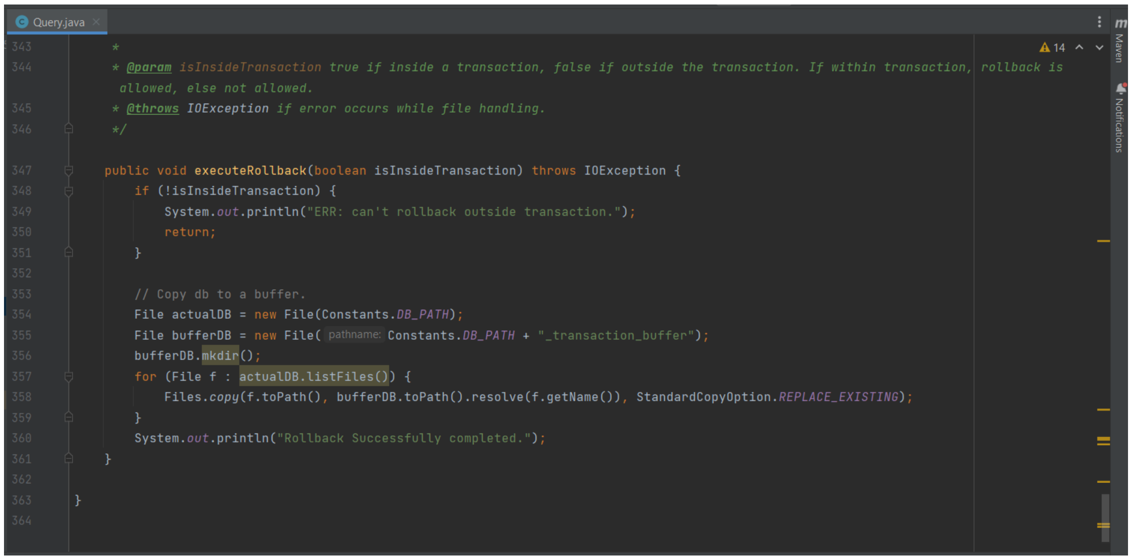


```
sql>commit;
ERR: can't commit outside transaction.
sql>
```

Figure 39: commit outside transaction [3][4][5].

Implementation of ROLLBACK

The executeRollback() method inside the Query class has the implementation of the rollback query. The main method calls this method only when the query is ROLLBACK keyword. The executeRollback() method overwrites the buffer with the actual database to dump un-committed changes.

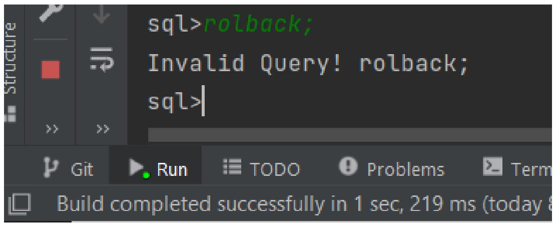


```java
     *
     * @param isInsideTransaction true if inside a transaction, false if outside the transaction. If within transaction, rollback is
      allowed, else not allowed.
     * @throws IOException if error occurs while file handling.
     */

    public void executeRollback(boolean isInsideTransaction) throws IOException {
        if (!isInsideTransaction) {
            System.out.println("ERR: can't rollback outside transaction.");
            return;
        }

        // Copy db to a buffer.
        File actualDB = new File(Constants.DB_PATH);
        File bufferDB = new File( pathname: Constants.DB_PATH + "_transaction_buffer");
        bufferDB.mkdir();
        for (File f : actualDB.listFiles()) {
            Files.copy(f.toPath(), bufferDB.toPath().resolve(f.getName()), StandardCopyOption.REPLACE_EXISTING);
        }
        System.out.println("Rollback Successfully completed.");
    }

}
```

Figure 40: Implementation of executeRollback()[3][4][5].

Testing of Rollback

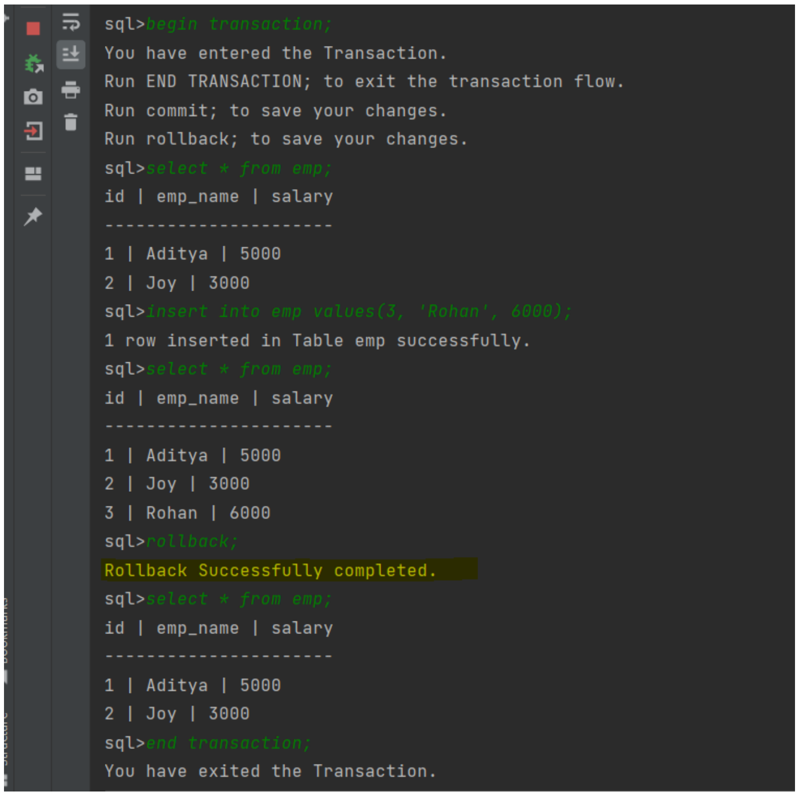
T1 – Rollback unsuccessful due to invalid syntax: When user enters rollback query with some spelling mistake or syntax error, error should be displayed. Actual Output: error is displayed.



```
sql>rolback;
Invalid Query! rolback;
sql>
```

Figure 41: rollback query with spelling mistake [3][4][5].

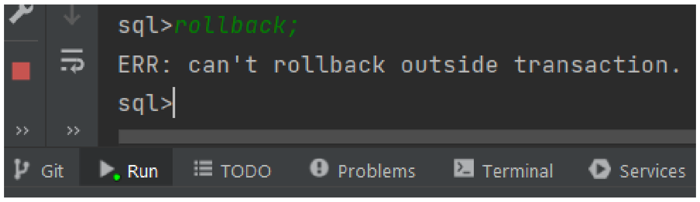
T2 - Rollback successful: When user enters rollback query with no syntax error, the un-committed changes are discarded. Actual Output: uncommitted changes are discarded.



```
sql>begin transaction;
You have entered the Transaction.
Run END TRANSACTION; to exit the transaction flow.
Run commit; to save your changes.
Run rollback; to save your changes.
sql>select * from emp;
id | emp_name | salary
---------------------
1 | Aditya | 5000
2 | Joy | 3000
sql>insert into emp values(3, 'Rohan', 6000);
1 row inserted in Table emp successfully.
sql>select * from emp;
id | emp_name | salary
---------------------
1 | Aditya | 5000
2 | Joy | 3000
3 | Rohan | 6000
sql>rollback;
Rollback Successfully completed.
sql>select * from emp;
id | emp_name | salary
---------------------
1 | Aditya | 5000
2 | Joy | 3000
sql>end transaction;
You have exited the Transaction.
```

Figure 42: rollback inside transaction [3][4][5].

T3 – Rollback unsuccessful due to outside transaction: When user enters rollback query outside transaction error should appear. Actual Output: Can't rollback error appears.

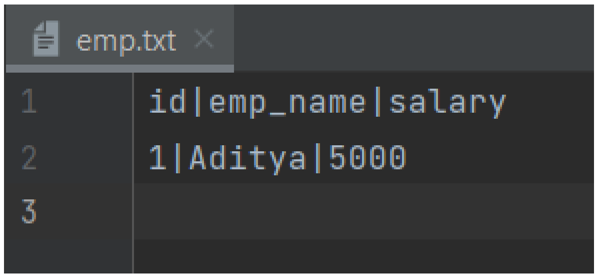


```
sql>rollback;
ERR: can't rollback outside transaction.
sql>
```

Figure 43: rollback outside transaction [3][4][5].

Novelty:

Own delimiter pipe operator is used in text files. '|'

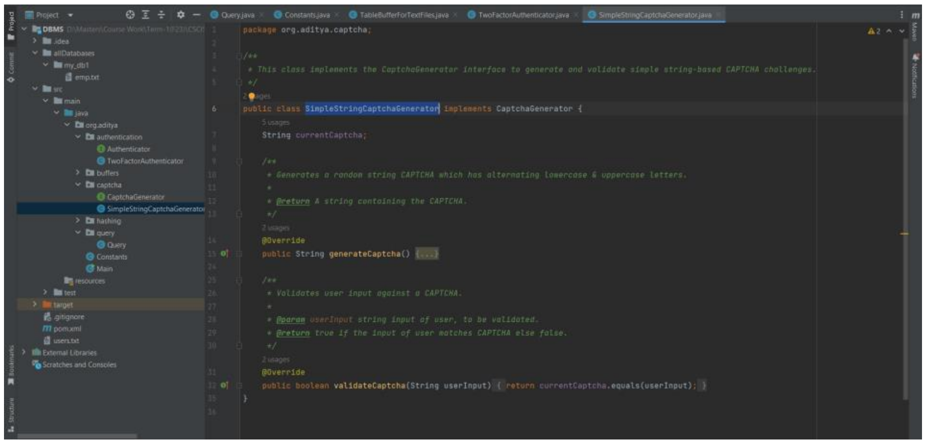


```
id|emp_name|salary
1|Aditya|5000
```

Figure 44: Delimiter '|' in text file [3][4][5].

JavaDocs specification comments:

All classes, interfaces, methods and data members have been given JavaDocs specification comments. Used @param, @throws, @return tags.



```java
package org.aditya.captcha;

/**
 * This class implements the CaptchaGenerator interface to generate and validate simple string-based CAPTCHA challenges.
 */
public class SimpleStringCaptchaGenerator implements CaptchaGenerator {
    String currentCaptcha;

    /**
     * Generates a random string CAPTCHA which has alternating lowercase & uppercase letters.
     *
     * @return A string containing the CAPTCHA.
     */
    @Override
    public String generateCaptcha() {...}

    /**
     * Validates user input against a CAPTCHA.
     *
     * @param userInput string input of user, to be validated.
     * @return true if the input of user matches CAPTCHA else false.
     */
    @Override
    public boolean validateCaptcha(String userInput) { return currentCaptcha.equals(userInput); }
}
```

Figure 45: JavaDocs Specification comments for SimpleStringCaptchaGenertor class [3][4][5].

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] C. Pino, S. Ravidá and S. Scibilia, "Evaluation of federated database for distributed applications in e-government," 2013 7th International Conference on Application of Information and Communication Technologies, Baku, Azerbaijan, 2013, pp. 1-5, doi: 10.1109/ICAICT.2013.6722791.

[3] “Java | Oracle,” Java.com, https://www.java.com/en/ (accessed Oct. 28, 2023).

[4] “IntelliJ IDEA – the leading Java and Kotlin Ide,” JetBrains, https://www.jetbrains.com/idea/ (accessed Oct. 28, 2023).

[5] B. Porter, J. van Zyl, and O. Lamy, “Welcome to Apache Maven,” Maven, https://maven.apache.org/ (accessed Oct. 28, 2023).

[6] W. by: S. Millington, “A solid guide to solid principles,” Baeldung, https://www.baeldung.com/solid-principles (accessed Oct. 31, 2023).

[7] “Solid design principles in java application development,” JRebel by Perforce, https://www.jrebel.com/blog/solid-principles-in-java (accessed Oct. 31, 2023).