

# CSCI 5408

## DATA MANAGEMENT AND WAREHOUSING

### Major Assignment - 1

Banner ID: B00957622

GitLab Assignment Link:

[https://git.cs.dal.ca/anagpal/csci5408\\_f23\\_b00957622\\_ashish\\_nagpal](https://git.cs.dal.ca/anagpal/csci5408_f23_b00957622_ashish_nagpal)

## Table of Contents

<b>Problem Statement 1:</b> .....	<b>3</b>
Research Paper .....	3
Similar Paper Review: .....	4
Critical Analysis .....	4
<b>Problem Statement 2:</b> .....	<b>5</b>
Overview: .....	5
Task Design: .....	6
Task A: .....	7
Task Data File: Novelty .....	10
Task B: Implementation of queries .....	11
Task C: Implementation of Transaction .....	18
<b>Test Cases:</b> .....	<b>23</b>
Test case 1: .....	23
Test case 2: .....	26
Test case 3: .....	27
<b>References:</b> .....	<b>29</b>

## Problem Statement 1:

### Research Paper

The research paper primarily focuses on transaction recovery in federated distributed systems, particularly in unreliable conditions. It places significant emphasis on enhancing the system's architecture with the introduction of a synchronization coordinator [1].

In the introductory section, the paper provides a foundational understanding of distributed database management systems and key terms. It outlines the challenges associated with this architecture, primarily centred on concurrency control and recovery [1].

The related work section introduces the concept of heterogeneous databases, adding an intermediary layer between users and databases. It highlights the benefits of distributed databases, such as improved reliability, availability, and scalability. The implementation of distributed systems is briefly touched upon, including distribution autonomy, data management, and database recovery [1].

The paper describes several characteristics of distributed databases, including distribution, autonomy, heterogeneity, semantic differences, and synchronous/asynchronous accessibility over networks. Issues with distributed database management systems, like distributed commits, redundant data items, node and network failures, and livelocks, are outlined [1].

The recovery section addresses different types of failures, like transaction errors and disk failures, and discusses techniques like write-ahead logging, checkpoints, and recoverability in federated database systems. It also introduces various recovery algorithms and their limitations [1].

The design and implementation section introduces key components like the Local Transaction Manager, Global Transaction Manager, GTM Algorithm, and Sync Coordinator. The use of Java and Windows for implementing these algorithms in a bank transfer scenario is discussed, and it is mentioned that Java was chosen due to its libraries [1].

Observations indicate that implementing an improved algorithm in banking databases can potentially reduce ATM cash-out fraud. The conclusion suggests further research, particularly on the sync coordinator at the wrapper level. Regarding the literature review, the related work section mentions levels of heterogeneity and autonomy and discusses the advantages of distributed databases [1].

However, the paper has shortcomings. It lacks concrete results and evidence of the research's success, such as practical outcomes or success rates for the defined algorithms and methodologies. Improvements could be made by providing a more detailed analysis of the algorithms, including their practical implementation [1].

## Similar Paper Review:

The research paper introduces an innovative distributed database solution built upon MySQL to address the limitations of traditional relational database management systems (RDBMS). The primary objective is to provide horizontal scalability for data storage while ensuring a seamless and user-friendly experience. In a rapidly evolving data landscape, traditional RDBMS architectures have struggled to keep pace [2].

The proposed solution draws inspiration from the concept of sharding, which involves dividing large databases into manageable "shards." By configuring specific files, this system allows application programmers to focus on their primary tasks while the infrastructure handles the intricacies of data distribution and management [2].

The paper outlines the key components of the system, including the MySQL communication protocol, connection pool, and SQL processor, which collectively facilitate efficient interaction with the database and data distribution. Additionally, it discusses vital technologies like input/output optimization, sharding algorithms, and distributed join strategies, all aimed at enhancing the system's performance, reliability, and robustness [2].

## Critical Analysis

The paper presents a promising solution for the challenges associated with traditional RDBMS, focusing on the implementation of a distributed database system based on MySQL. However, there are notable areas of concern that require further attention. Firstly, the paper lacks practical real-world examples or case studies demonstrating the successful implementation of the proposed system. Moreover, the absence of comprehensive performance trade-off discussions raises questions about potential drawbacks [2].

Additionally, the paper emphasizes distributed join strategies but ultimately discourages their practical use without offering alternative solutions. This leaves a gap in understanding how the proposed system might handle complex query scenarios effectively. To strengthen the paper's credibility, it would benefit from a more extensive comparison with alternative distributed database solutions and a more in-depth analysis of the advantages and disadvantages of various approaches. In summary, while the paper introduces an intriguing concept for addressing RDBMS limitations, further research, practical testing, and a broader exploration of potential issues are needed to fully evaluate its feasibility and applicability [2].

## Problem Statement 2:

**Prototype of a light-weight DBMS using Java programming language (no 3rd party libraries allowed). Objective is lo#2. Note: Here you are not using MySQL, you are expected to create a similar tool like MySQL**

### Overview:

To implement a light-weight DBMS using Java programming language, I have divided my project into different classes.

*Table 1: Database Overview*

Sno.	Class Name	Purpose
1.	Main	It is the entry point of the database which asks user for authentication input and perform database operation
2.	Authentication	This class handles the authentication logic of a user and interacts with User class
3.	Users	Class used to create files, read and write for users.
4.	DatabaseProcessor	Controller of database operation which performs create and select database along with input of query.
5.	Transaction	It handles the query processing in form of transaction
6.	QueryProcessor	This class defines the complete logic to process different types of queries such as - select, update, insert delete and create.
7.	DatabaseUpdate	Class that interact with file system to update the database
8.	DBUtil	Utility class for helper methods

The application starts from the Main class and follows the authentication process. Once the user is authenticated successfully, the application directs to the DatabaseProcessor class that handles database level processing. If the user selects query execution in that case the flow goes to Transaction class and the queries are executed with the help of QueryProcessor class.

I have written JavaDocs specification for each class, method and associated variables using @param and @return annotations.

## Task Design:

SOLID is a design principle that ensures that in object oriented programming the development code should follow - single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle.[3][4]

In my code, I have followed principles -

1. **Single Responsibility Principle** - Each class which I have defined has distinct responsibility.

*Table 2: Class and its single responsibility principle justification*

Sno.	Class Name	Reason
1.	Authentication	<ul style="list-style-type: none"><li>• Responsible for handling user authentication and related functionalities.</li><li>• Manages user login, validation, and password-related operations.</li></ul>
2.	DatabaseProcessor	<ul style="list-style-type: none"><li>• Responsible for managing different database operations, including creating, showing tables, executing queries, and database selection.</li><li>• Parses and processes SQL-like commands for database management.</li></ul>
3.	QueryProcessor	Contains methods for processing and executing each type of SQL Like query operation.
4.	DatabaseUpdate	Handles reading data from files and updating table files accordingly

2. **Open/Closed Principle** - I have defined classes in such a way that it can be extended for more functionality. Let's say if we have to implement an Alter table command in this, all we have to do is add a case for it in the switch control statement and define the functionality of the method in the QueryProcessor class.

## Task A:

### 1. Two factor user authentication

For user authentication I have created a class authentication that has instance variables - username, password and hashPassword. The class is used to take input parameters from the user based on the option they have selected from the main menu and take the action accordingly. The method definition is as follows -

*Table 3: Authentication class methods*

Sno.	Method Name	Params	Return Type	Description
1.	Authentication	None	void	Default constructor for Authentication Class
2.	Login	None	boolean	Validates user credential
3.	Signup	None	void	Registers a new user
4.	Authenticate	authOption	boolean	Authenticates the user for login or signup based on the authentication option. Entry point function called by Main class.

### 2. Multi user Support

For multiple user authentication I am storing the username and password in a text file as the persistent storage. The text file will contain all the previous users information and if a new user signup, then in that case the value is added to the users.txt file. For implementing this logic, I have created a Users class.

#### Users.class

This class deals with the file system to create a new user and validate the credentials of a user. For file writing I have used FileLock class which will lock the channel when the file is being updated.[5] This will make sure that the file is not updated simultaneously by two different users. The method definition is as follows -

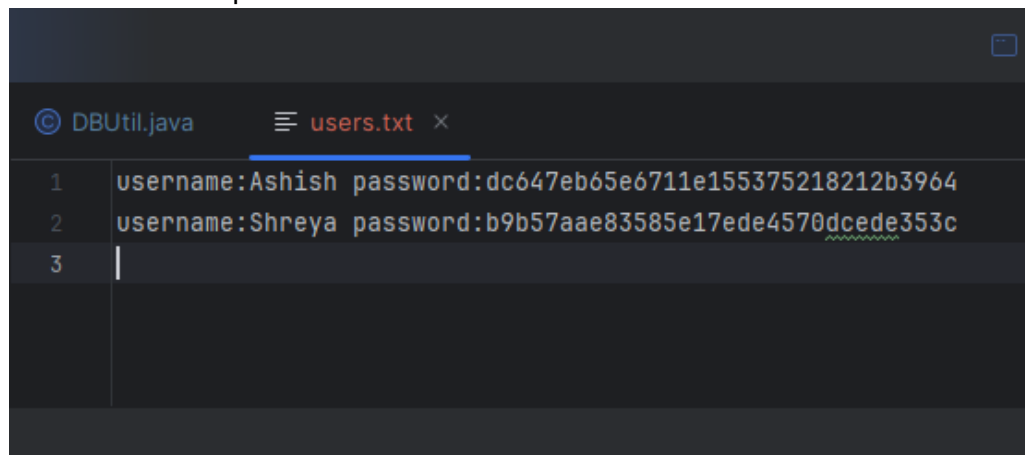
*Table 4: Users class methods*

Sno.	Method Name	Params	Return Type	Description
1.	exists	username	boolean	Checks whether the user exists in the user file.
2.	validateUser	username, hashPass	boolean	Validates a user by username and hashed password by reading users.txt file

3.	createUser	username, hashPass	None	Creates a new user with the given username and hashed password by updating users.txt file.
----	------------	-----------------------	------	--

### 3. Store password in the form of hash characters.

I have used the md5 Java library[6] to hash the password provided by the user and store it in users.txt file. DBUtil class contains a hashPassword method that converts the text password to hashed value.



```

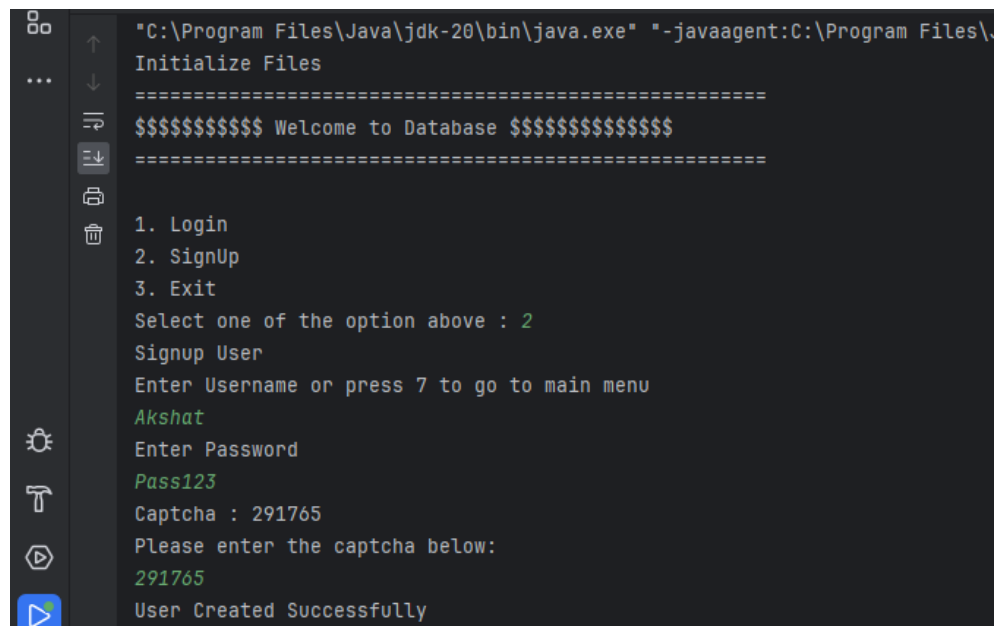
DBUtil.java  users.txt x
1 username:Ashish password:dc647eb65e6711e155375218212b3964
2 username:Shreya password:b9b57aae83585e17ede4570dced353c
3 |

```

Figure 1: Before users.txt file

## Demo

### a. Create a new user



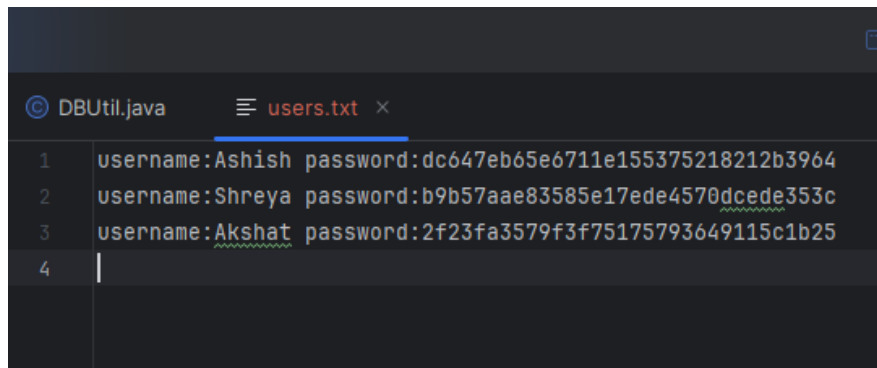
```

"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program Files\
Initialize Files
=====
$$$$$$$$$$ Welcome to Database $$$$$$$$$$$$
=====
1. Login
2. SignUp
3. Exit
Select one of the option above : 2
Signup User
Enter Username or press 7 to go to main menu
Akshat
Enter Password
Pass123
Captcha : 291765
Please enter the captcha below:
291765
User Created Successfully

```

Figure 2: SignUp user



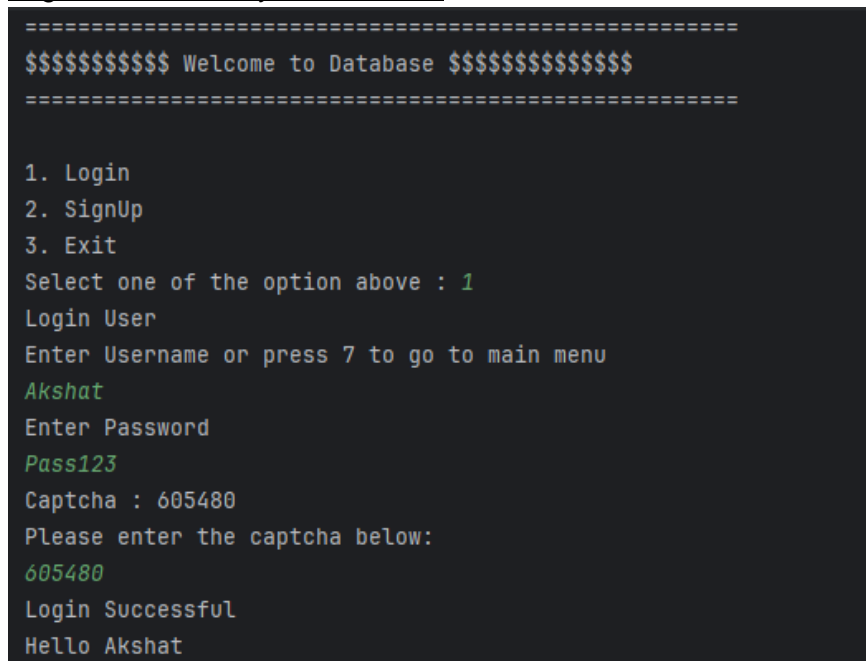


The screenshot shows a code editor with two tabs: 'DBUtil.java' and 'users.txt'. The 'users.txt' tab is active, displaying a list of three users in a table-like format. The first user is 'Ashish' with a long password. The second user is 'Shreya' with a long password. The third user is 'Akshat' with a long password. The fourth row is empty.

1	username:Ashish password:dc647eb65e6711e155375218212b3964
2	username:Shreya password:b9b57aae83585e17ede4570dc353c
3	username:Akshat password:2f23fa3579f3f75175793649115c1b25
4	

Figure 3: After user creation users.txt file

b. Login with the newly created user



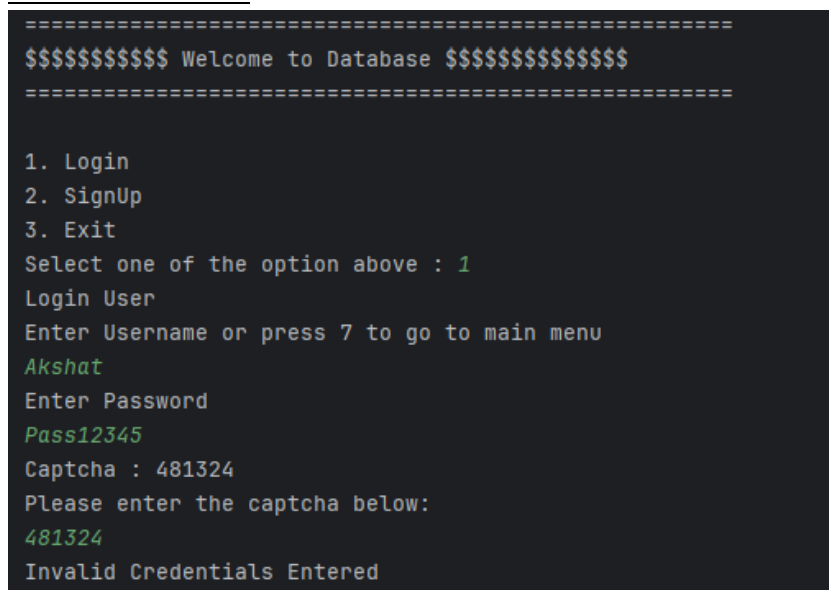
The screenshot shows a terminal window with a login menu. The menu options are 1. Login, 2. SignUp, and 3. Exit. The user selects option 1. The terminal prompts for 'Login User', 'Enter Username or press 7 to go to main menu', 'Enter Password', and 'Captcha : 605480'. The user enters 'Akshat' for the username, 'Pass123' for the password, and '605480' for the captcha. The terminal displays 'Login Successful' and 'Hello Akshat'.

```
=====
$$$$$$$$$$ Welcome to Database $$$$$$$$$$$$
=====

1. Login
2. SignUp
3. Exit
Select one of the option above : 1
Login User
Enter Username or press 7 to go to main menu
Akshat
Enter Password
Pass123
Captcha : 605480
Please enter the captcha below:
605480
Login Successful
Hello Akshat
```

Figure 4: Login user screen

c. Invalid Credentials



The screenshot shows a terminal window with a login menu. The menu options are 1. Login, 2. SignUp, and 3. Exit. The user selects option 1. The terminal prompts for 'Login User', 'Enter Username or press 7 to go to main menu', 'Enter Password', and 'Captcha : 481324'. The user enters 'Akshat' for the username, 'Pass12345' for the password, and '481324' for the captcha. The terminal displays 'Invalid Credentials Entered'.

```
=====
$$$$$$$$$$ Welcome to Database $$$$$$$$$$$$
=====

1. Login
2. SignUp
3. Exit
Select one of the option above : 1
Login User
Enter Username or press 7 to go to main menu
Akshat
Enter Password
Pass12345
Captcha : 481324
Please enter the captcha below:
481324
Invalid Credentials Entered
```

Figure 5: Login user - Invalid credentials

## Task Data File: Novelty

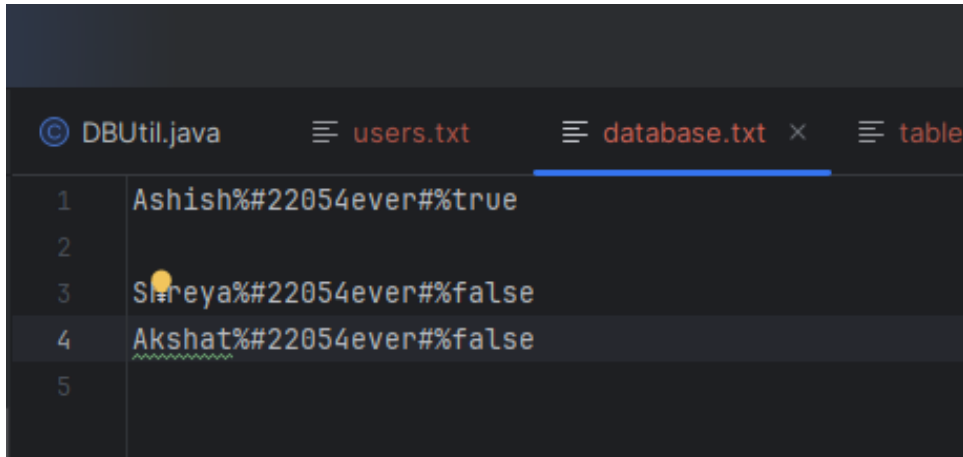
### 1. Persistent Storage

For storing data I have used simple text files. The text files will contain all the database's data like - users, tables, database, userLogin etc.

### 2. Delimiter

For storing data in text files I have created a delimiter - "%#22054ever#%" which will be added for data separation. Each row entry will be separated by a newline and the columns within a row will be differentiated by the delimiter.

### Demo



The screenshot shows a code editor with four tabs: DBUtil.java, users.txt, database.txt (selected), and table. The database.txt file contains three lines of data, each separated by a newline. The data is formatted as follows:

Line	Content
1	Ashish%#22054ever#%true
2	
3	Si#reya%#22054ever#%false
4	Akshat%#22054ever#%false
5	

Figure 6: database.txt file delimiter view

## Task B: Implementation of queries

For implementing queries in a database, I have created a QueryProcessor class. The task of this class is to select the appropriate query which is input by the user, execute the query and store the value into temporary storage. The temporary storage is basically a HashMap[7] which is an instance variable that is used to collect all the tables and data information. The instance variables and methods are as follows -

*Table 5: QueryProcessor class instance variables*

Sno.	Variable Name	Type	Description
1.	data	HashMap	It takes the table name as key and stores a list of Strings representing table rows.
2.	tables	HashMap	Stores the table name and column types associated with each table.
3.	tempData	HashMap	It is a temporary storage for data variable that stores similar data and is updated on every execution
4.	tempTables	HashMap	It is a temporary storage for tables variable that stores similar data and is updated on every execution
5.	util	DBUtil	It is used to execute methods provided by the DBUtil class.
6.	query	String	It stores the query provided by the user for execution.
7.	queryParts	String[]	Stores the parts of the query separated by space.

*Table 6: QueryProcessor class methods*

Sno.	Method Name	Params	Return Type	Description
1.	querySelector	None	boolean	Executes the specified query based on its operation.
2.	select	None	boolean	Selects and displays records from the table.
3.	insert	None	boolean	Inserts a new row into the tempTable hashMap.
4.	createTable	None	boolean	Creates a new table.

5.	update	None	boolean	Updates existing records in the table based on a specified condition.
6.	delete	None	boolean	Deletes records from the table based on a specified condition.

1. **QuerySelector** - This method consists of a switch case that decides the operation and calls the respective method for further processing.

```
public boolean querySelector() {

    String operation = queryParts[0].toLowerCase();
    switch (operation) {
        case "select":
            return select();
        case "insert":
            return insert();
        case "create":
            return createTable();
        case "update":
            return update();
        case "delete":
            return delete();
        case "commit":
        case "start":
        case "rollback":
            return true;
        default:
            break;
    }
    return false;
}
```

2. **Select** - Method to view all the rows specified in the table. It checks if the table doesn't exist, then it displays the corresponding message. Otherwise it iterates the entire List of string mapped with the table name and displays the rows.

```
boolean select() {
    String tableName = queryParts[3].toLowerCase();
    if (!tableExist(tableName)) {
        System.out.println("Table doesn't exist");
        return false;
    }
    List<String> tableValues = tempData.get(tableName);
    for (String row :
        tableValues) {
        System.out.println(row.replace(util.delimiter(), "\t"));
    }
    return true;
}
```

3. **Insert** - Method to add rows to a table. The values are first separated from the complete query and stored in a variable commaSepInsertValues. Then the actual values are taken into an array of strings using the split method[8] as the values are comma separated. Following this, the row is constructed with a delimiter and added to the table.

```
boolean insert(){
    String tableName = queryParts[2].toLowerCase();
    if(!tableExist(tableName)) {
        System.out.println("Table doesn't exist");
        return false;
    }

    String commaSepInsertValues = query.substring(query.lastIndexOf("(") + 1,
    query.lastIndexOf(")").trim());
    String[] values = commaSepInsertValues.split(",");
    StringBuilder output = new StringBuilder();
    for(int i=0; i<values.length; i++){
        output.append(values[i].replace("\"", "").trim().split("\"")[0]);
        if(!(i == values.length - 1) )
            output.append(util.delimiter());
    }
    List<String> tableEntries = new ArrayList<>(tempData.get(tableName));
    tableEntries.add(output.toString());
    tempData.put(tableName, tableEntries);

    return true;
}
```

4. **CreateTable** - This method first checks if the table which is proceeded for creation doesn't already exist. Following this, the values are constructed for the table data (tempData ) having the first row as columns and adding entry to tables as well with table name and columns as the mapping.

```
boolean createTable(){
    String tableName = queryParts[2].split("\\(")[0].toLowerCase();
    if(tableExist(tableName)) {
        System.out.println("Table exist with same name");
        return false;
    }

    String commaSepCreateValues = query.substring(query.indexOf("(") + 1,
    query.lastIndexOf(")").trim());
    String[] values = commaSepCreateValues.split(",");
    StringBuilder output = new StringBuilder();
    StringBuilder tableOutput = new StringBuilder();
    for(int i=0; i<values.length; i++) {
        String columnName = values[i].trim().split("\"")[0];
        String columnType = values[i].trim().split("\"")[1];
        output.append(columnName);
        tableOutput.append(columnName).append(util.delimiter()).append(columnType);
        if (i < values.length - 1) {
```

```

        output.append(util.delimiter());
        tableOutput.append(" ");
    }
}
List<String> dataList = new ArrayList<>();
dataList.add(output.toString());
tempData.put(tableName, dataList);
tempTables.put(tableName, tableOutput.toString());
return true;
}

```

**5. Update** - Similar to insert, update method constructs its entries for which column should be updated and with what value. These are stored in variables conditionValue and updateValue. The exact index of condition and update column are executed using the first row of the table.

After this, I have executed a for loop to iterate all the rows and splitted the rows with the delimiter. If the condition value satisfies with the current row, then the row is updated with new value and added to the updatedEntries list. Otherwise, the row is directly added.

Once the entire process is complete, the table is updated with new updatedEntries.

```

boolean update() {
    String tableName = queryParts[1].toLowerCase();
    if (!tableExist(tableName)) {
        System.out.println("Table doesn't exist ");
        return false;
    }
    List<String> tableEntries = tempData.get(tableName);
    List<String> updatedEntries = new ArrayList<>();
    updatedEntries.add(tableEntries.get(0));
    List<String> rowColumn = Arrays.asList(tableEntries.get(0).split(util.delimiter()));

    String condition = queryParts[queryParts.length - 1].replace("\"", "");
    int conditionColumnIndex = rowColumn.indexOf(condition.split("=")[0]);
    String conditionValue = condition.split("=")[1];
    String[] setParts = queryParts[3].split("=");
    int updateColumnIndex = rowColumn.indexOf(setParts[0].trim().replace("\"", ""));
    String updateValue = setParts[1].trim().replace("\"", "");

    for (int j = 1; j < tableEntries.size(); j++) {
        String[] row = tableEntries.get(j).split(util.delimiter());
        if (row[conditionColumnIndex].equalsIgnoreCase(conditionValue)) {
            StringBuilder updatedRow = new StringBuilder();
            for (int k = 0; k < row.length; k++) {
                if (k == updateColumnIndex) {
                    updatedRow.append(updateValue);
                } else {
                    updatedRow.append(row[k]);
                }
            }
            if (k < row.length - 1) {
                updatedRow.append(util.delimiter());
            }
        } else {
            updatedEntries.add(tableEntries.get(j));
        }
    }
    tempData.put(tableName, updatedEntries);
    return true;
}

```

```

    }
    }
    updatedEntries.add(updatedRow.toString());
}
else{
    updatedEntries.add(tableEntries.get(j));
}
}

tempData.put(tableName, updatedEntries);

return true;
}

```

6. **Delete** - Similar to update method, the delete function gets the condition index and whichever row matches the condition that row isn't added to the new updatedEntries list.

```

boolean delete() {
    String tableName = queryParts[2].toLowerCase();
    if (!tableExist(tableName)) {
        System.out.println("Table doesn't exist");
        return false;
    }

    List<String> tableEntries = tempData.get(tableName);
    List<String> updatedEntries = new ArrayList<>();
    updatedEntries.add(tableEntries.get(0));
    List<String> rowColumn = Arrays.asList(tableEntries.get(0).split(util.delimiter()));

    String condition = queryParts[4].replace("\"", "");
    int conditionColumnIndex = rowColumn.indexOf(condition.split(" ")[0]);
    String conditionValue = condition.split(" ")[1];


    for (int j = 1; j < tableEntries.size(); j++) {
        String[] row = tableEntries.get(j).split(util.delimiter());
        if (!row[conditionColumnIndex].equalsIgnoreCase(conditionValue)) {
            updatedEntries.add(tableEntries.get(j));
        }
    }
    tempData.put(tableName, updatedEntries);

    return true;
}

```

## Demo

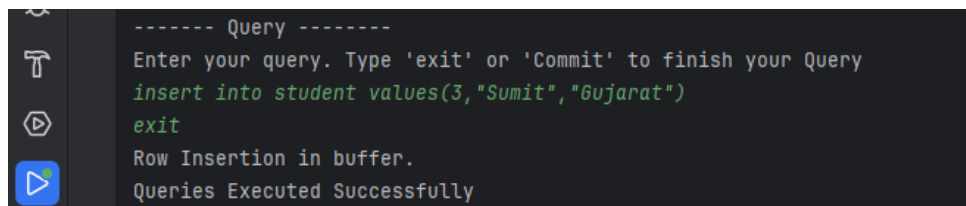
1. Select values from employee table.



```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
select * from employee
exit
id  name
8   Ashish
9   Shreya
10  Rajni
11  Rajat
13  Akshat
22  Abc
99  Ritu
69  RohanBro
98  Suchika
20  Surinder
102 HelloFrاند
Queries Executed Successfully
```

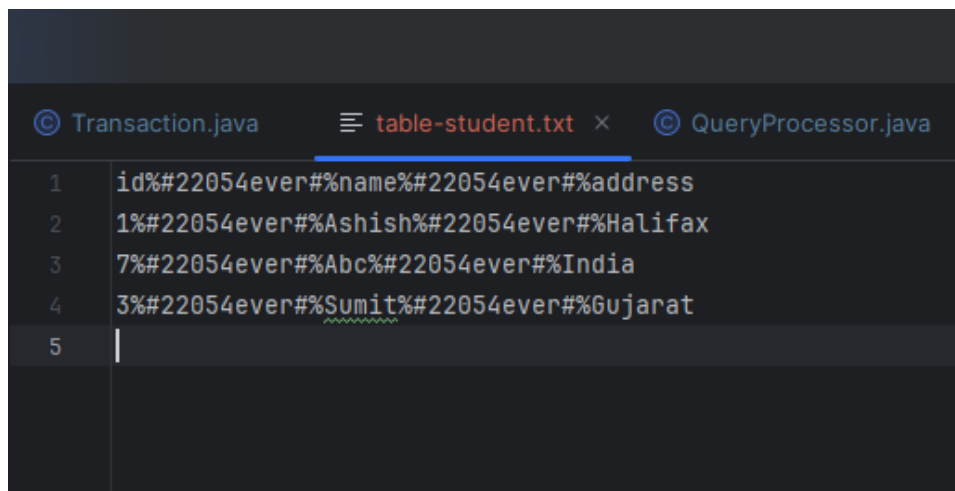
Figure 7: Select Query

2. Insert values into student table



```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
insert into student values(3,"Sumit","Gujarat")
exit
Row Insertion in buffer.
Queries Executed Successfully
```

Figure 8: Insert Query



id#22054ever#%name%#22054ever#%address
1#22054ever#%Ashish%#22054ever#%Halifax
7#22054ever#%Abc%#22054ever#%India
3#22054ever#%Sumit%#22054ever#%Gujarat
5

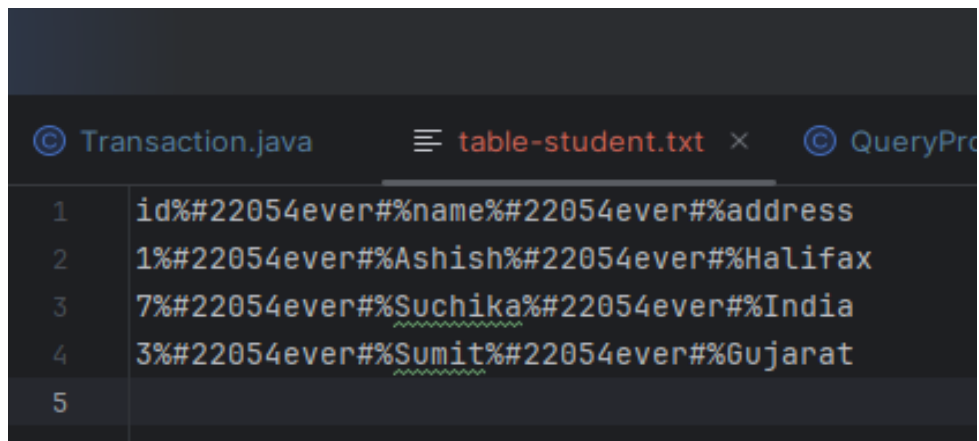
Figure 9: table-student.txt file



### 3. Update student table name from Abc to Suchika

```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
update student set name="Suchika" where id=7
exit
Rows updated --> buffer
Queries Executed Successfully
```

Figure 10: Update query



1	id#22054ever#%name%#22054ever#%address
2	1#22054ever#%Ashish%#22054ever#%Halifax
3	7#22054ever#%Suchika%#22054ever#%India
4	3#22054ever#%Sumit%#22054ever#%Gujarat
5	

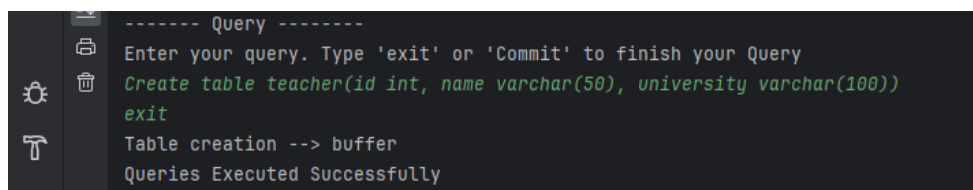
Figure 11: After update table-student.txt file

### 4. Delete

```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
Delete from student where id=3
exit
Rows Deleted --> buffer
Queries Executed Successfully
```

Figure 12: Delete Query

### 5. Create Table



```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
Create table teacher(id int, name varchar(50), university varchar(100))
exit
Table creation --> buffer
Queries Executed Successfully
```

Figure 13: Create table query

## Task C: Implementation of Transaction

For implementing transactions in the database, I have designed the database in such a way that all the queries are by default executed in the form of a transaction.[9] The user can start a transaction by using “Start transaction” keywords and end the transaction with “commit”. However, if they haven’t specified the transaction even then also the data is first updated into a buffer and then persistent storage is updated to maintain consistency in database logic.

A user can only rollback if they have started the transaction, otherwise the default behaviour of commit will happen even if the rollback is entered as a query.

For handling all the operations, I have created a class Transaction. The instance variables and methods are as follows -

*Table 7: Transaction class variables*

Sno.	Variable Name	Type	Description
1.	pendingQueriesCount	int	The count of pending queries within the transaction.
2.	inTransaction	boolean	Indicates whether a transaction is currently in progress.
3.	processQuery	QueryProcessor	The QueryProcessor instance responsible for processing queries.
4.	dbUpdate	DatabaseUpdate	DatabaseUpdate instance for managing database updates during transactions.

1. **beginTransaction** - This method takes all the queries which are input by the user and executes those one by one by calling the querySelector method in the QueryProcessor class. If the query is successful, it also displays the middle level message with the present in the buffer included. Once processing is done, it commits the queries if the user hasn’t explicitly defined a commit.

```
public void beginTransaction(List<String> queries) {
    for (String query : queries) {
        String[] queryParts = query.split(" ");
        String operation = queryParts[0].toLowerCase();
        processQuery.query = query;
        processQuery.queryParts = queryParts;
        boolean result = processQuery.querySelector();
        if(result) {
            pendingQueriesCount++;
            switch (operation) {
                case "start":
                    if(queryParts[1].equalsIgnoreCase("transaction")){
```

```

        inTransaction = true;
        System.out.println("Begin Transaction");
    }
    break;
case "select":
    break;
case "insert":
    System.out.println("Row Insertion in buffer.");
    break;
case "create":
    System.out.println("Table creation --> buffer");
    break;
case "update":
    System.out.println("Rows updated --> buffer");
    break;
case "delete":
    System.out.println("Rows Deleted --> buffer");
    break;
case "commit":
    commit();
    break;
case "rollback":
    rollback();
default:
    break;
    }
    }
}

if(pendingQueriesCount > 0){
    commit();
}
}

```

2. **endTransaction** - Method used to end the current ongoing transaction

```

public void endTransaction() {
    if (inTransaction) {
        inTransaction = false;
        System.out.println("End Transaction");
    }
}

```

3. **rollback** - This method clears the buffer that consists of the recent queries and rolls back the table state to previous state by copying the data and tables hashMaps.

```

void rollback(){
    if (!inTransaction) {

```

```

        System.out.println("Transaction not started. Cannot rollback");
        return;
    }
    processQuery.tempTables = new HashMap<>(processQuery.tables);
    processQuery.tempData = new HashMap<>(processQuery.data);
    pendingQueriesCount = 0;

    System.out.println("Transaction Rollback successful");
}

```

4. **commit** - Method used to commit the transaction by updating the persistent storage.

```

public void commit() {
    dbUpdate.updateDbTables(processQuery.tempTables);
    dbUpdate.insertDataIntoTables(processQuery.tempData);
    pendingQueriesCount = 0;

    if(inTransaction) {
        System.out.println("Transaction committed. Pending queries applied to
database");
    }
    else{
        System.out.println("Queries Executed Successfully");
    }
    endTransaction();
}

```

## Demo

1. Start a transaction and rollback after performing a few queries. Following this insert a value and then commit data.

	table-student.txt	table-teacher.txt	QueryProcessor.jav
1	id#22054ever#%name#22054ever#%address		
2	1#22054ever#%Ashish#22054ever#%Halifax		
3	7#22054ever#%Suchika#22054ever#%India		
4			

Figure 14: Before transaction table-student.txt file

```

----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
Start Transaction
insert into student values (5,"Sumit","Toronto")
update student set name="Su" where id=7
rollback
insert into student values(9,"Rajat","Faridabad")
commit
Row Insertion in buffer.
Rows updated --> buffer
Transaction Rollback successful
Row Insertion in buffer.
Transaction committed. Pending queries applied to database
Transaction ended.

```

Figure 15: Perform transaction

table-student.txt ×		table-teacher.txt	© QueryProcessor.js
1	id#22054ever#%name%#22054ever#%address		
2	1#22054ever#%Ashish%#22054ever#%Halifax		
3	7#22054ever#%Suchika%#22054ever#%India		
4	9#22054ever#%Rajat%#22054ever#%Faridabad		
5			

Figure 16: After transaction table-student.txt file

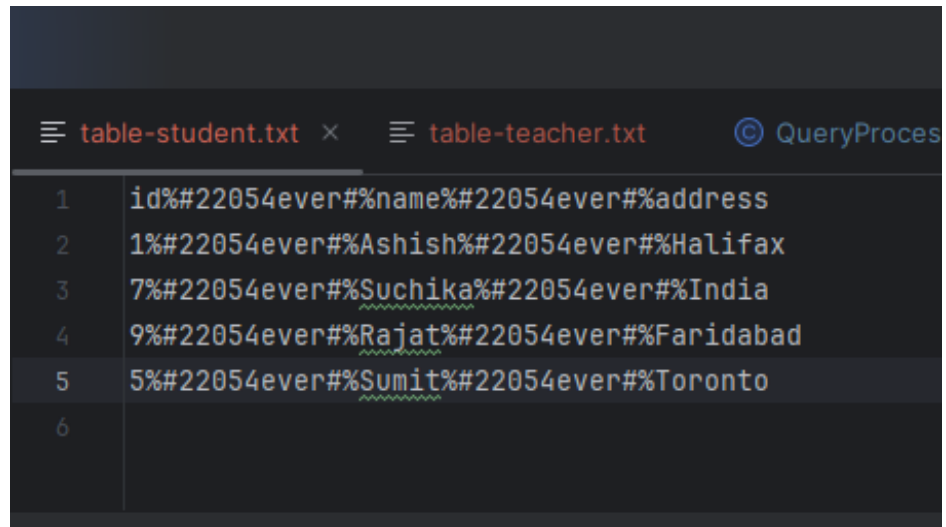
## 2. Rollback without a transaction

```

----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
insert into student values (5,"Sumit","Toronto")
rollback
commit
Row Insertion in buffer.
Transaction not started. Cannot rollback
Queries Executed Successfully

```

Figure 17: Rollback without transaction



1	id#22054ever#%name%#22054ever#%address
2	1#22054ever#%Ashish%#22054ever#%Halifax
3	7#22054ever#%Suchika%#22054ever#%India
4	9#22054ever#%Rajat%#22054ever#%Faridabad
5	5#22054ever#%Sumit%#22054ever#%Toronto
6	

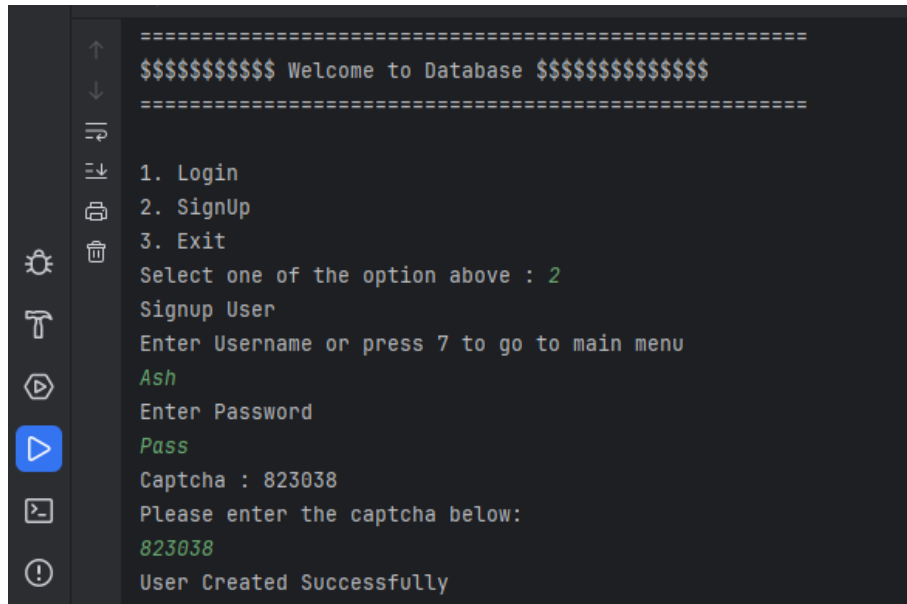
Figure 18: table-student.txt after transaction without rollback

## Test Cases:

### Test case 1:

Create user, login via user, create database, select database and show tables.

1. We have to create a new user using signup option

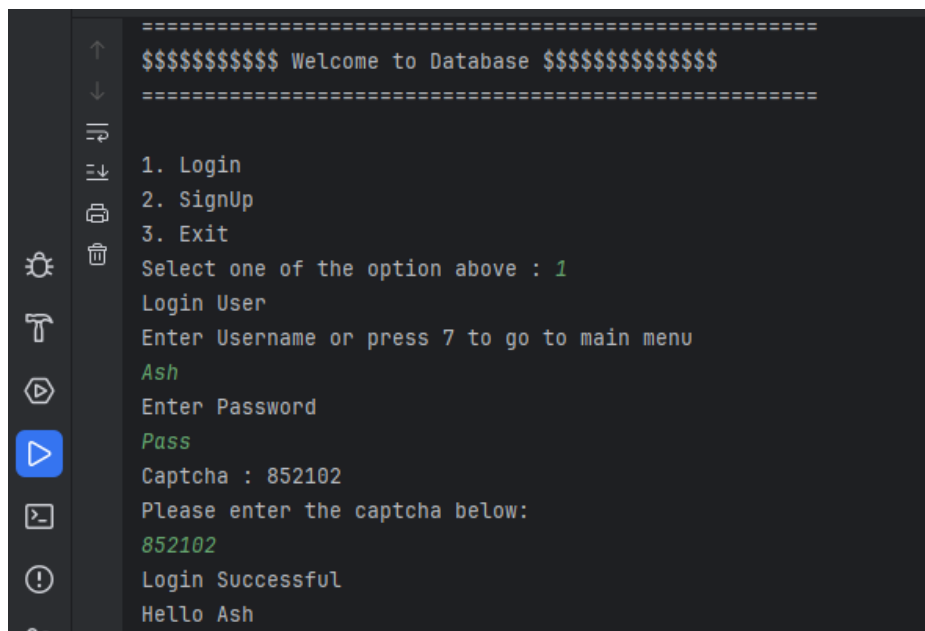


```
=====
$$$$$$$$$$ Welcome to Database $$$$$$$$$$$$
=====

1. Login
2. SignUp
3. Exit
Select one of the option above : 2
Signup User
Enter Username or press 7 to go to main menu
Ash
Enter Password
Pass
Captcha : 823038
Please enter the captcha below:
823038
User Created Successfully
```

Figure 19: User signUp

2. To login through a new user, we have to choose option 1 and then enter the credentials with correct captcha.

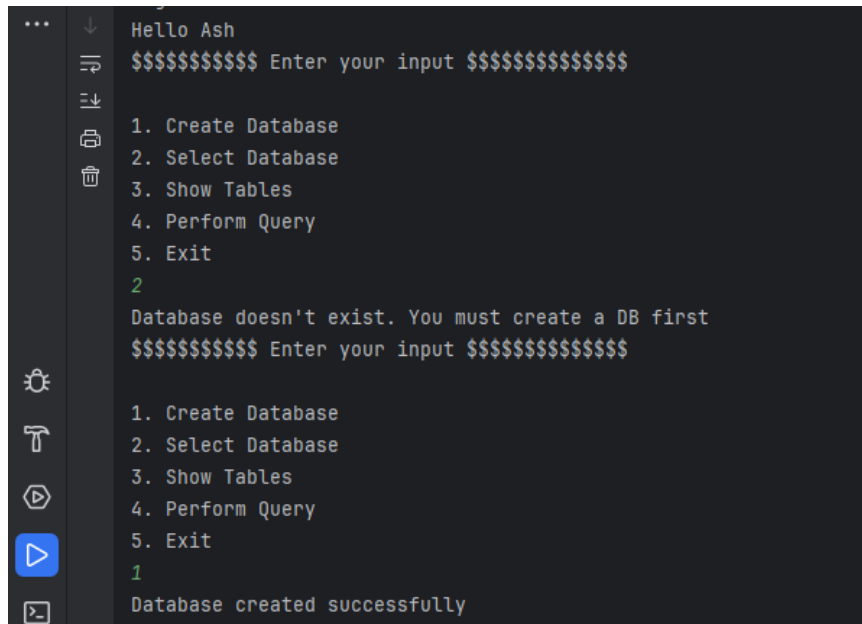


```
=====
$$$$$$$$$$ Welcome to Database $$$$$$$$$$$$
=====

1. Login
2. SignUp
3. Exit
Select one of the option above : 1
Login User
Enter Username or press 7 to go to main menu
Ash
Enter Password
Pass
Captcha : 852102
Please enter the captcha below:
852102
Login Successful
Hello Ash
```

Figure 20: User login

3. Create a database first. Before that, I have tried to select the database which shows the error that the database doesn't exist.

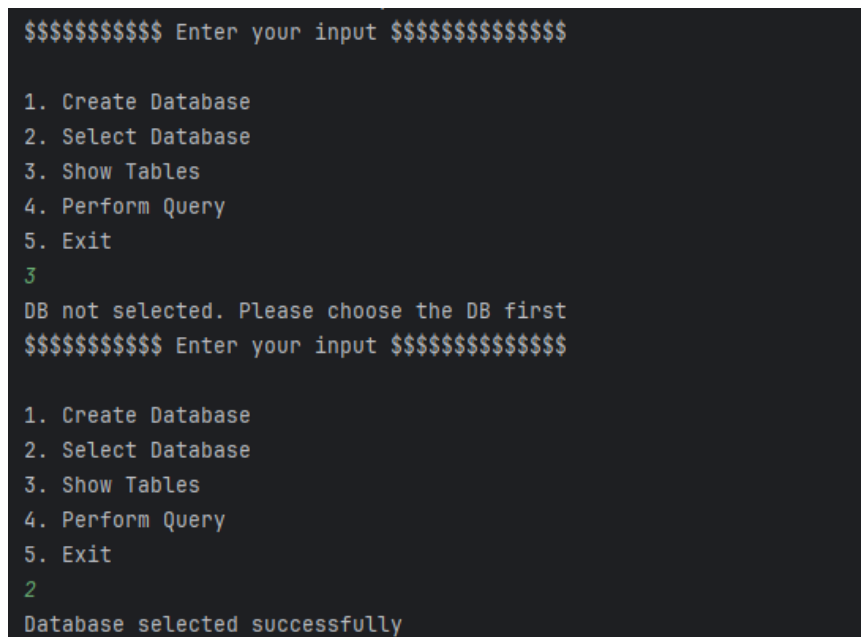


```
...  ↓ Hello Ash
      ⇨ $$$$$$$$$$ Enter your input $$$$$$$$$$$$$$
      ⇩
      🖨 1. Create Database
      🗑 2. Select Database
      3. Show Tables
      4. Perform Query
      5. Exit
      2
      Database doesn't exist. You must create a DB first
      $$$$$$$$$$ Enter your input $$$$$$$$$$$$$$

      🛠 1. Create Database
      📄 2. Select Database
      📁 3. Show Tables
      📄 4. Perform Query
      📄 5. Exit
      1
      Database created successfully
```

*Figure 21: Create Database*

4. Select the database before performing queries.



```
$$$$$$$$$$ Enter your input $$$$$$$$$$$$$$

1. Create Database
2. Select Database
3. Show Tables
4. Perform Query
5. Exit
3
DB not selected. Please choose the DB first
$$$$$$$$$$ Enter your input $$$$$$$$$$$$$$

1. Create Database
2. Select Database
3. Show Tables
4. Perform Query
5. Exit
2
Database selected successfully
```

*Figure 22: Select Database*

5. View all tables



```
...  ↓  $$$$$$$$$$ Enter your input $$$$$$$$$$$$$$  
      ||  1. Create Database  
      ||  2. Select Database  
      ||  3. Show Tables  
      ||  4. Perform Query  
      ||  5. Exit  
      ||  3  
      ||  -----  
      ||  | Tables |  
      ||  -----
```

Figure 23: Show Tables

6. Exit from the database will unselect the database of the user.

```
⚙️  $$$$$$$$$$ Enter your input $$$$$$$$$$$$$$  
🔧  1. Create Database  
📦  2. Select Database  
📦  3. Show Tables  
▶️  4. Perform Query  
▶️  5. Exit  
▶️  5  
▶️  Database unselected
```

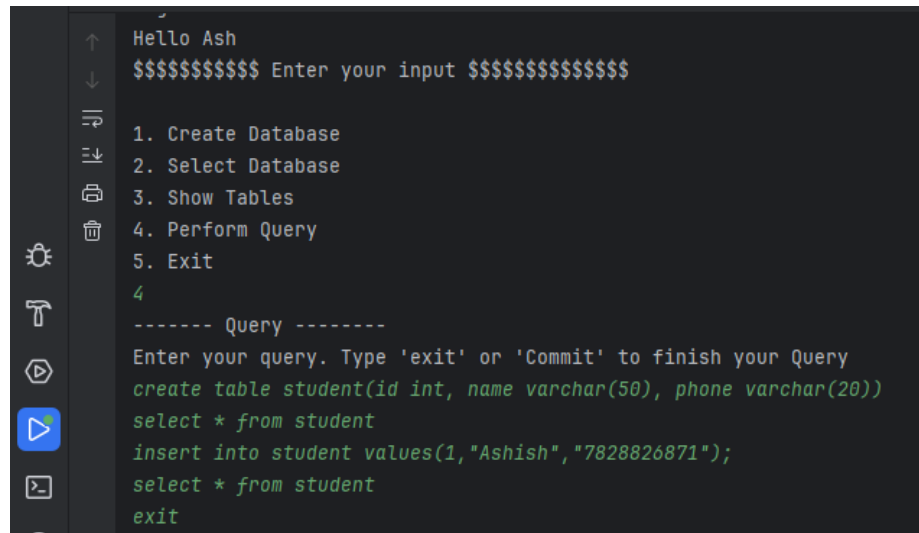
Figure 24: Unselect Database and exit

## Test case 2:

Perform Query - create table, insert data and select values from the table.

### 1. Query

```
create table student(id int, name varchar(50), phone varchar(20))
select * from student
insert into student values(1,"Ashish","7828826871");
select * from student
exit
```



```
Hello Ash
$$$$$$$$$$ Enter your input $$$$$$$$$$$$$$

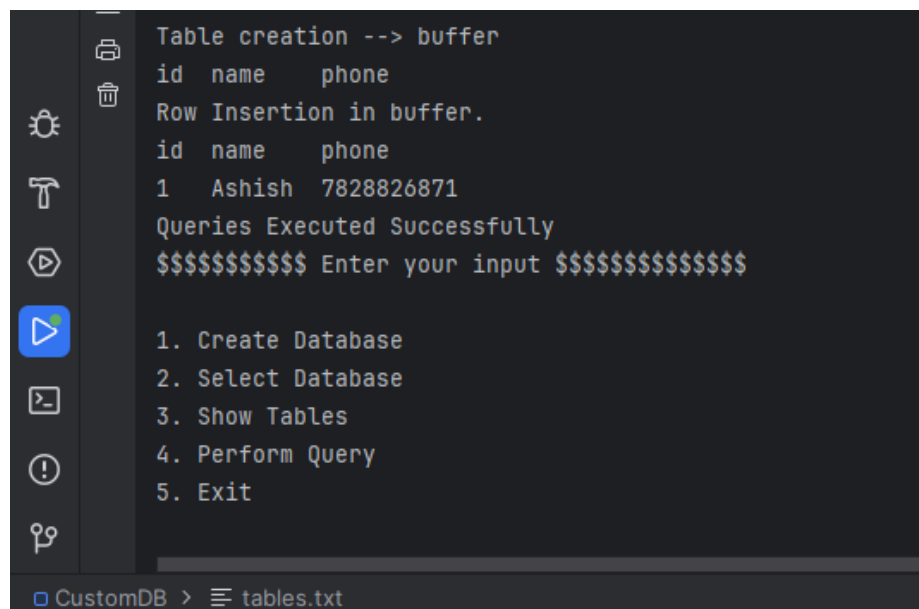
1. Create Database
2. Select Database
3. Show Tables
4. Perform Query
5. Exit

4

----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
create table student(id int, name varchar(50), phone varchar(20))
select * from student
insert into student values(1,"Ashish","7828826871");
select * from student
exit
```

Figure 25: Perform multiple queries

### 2. Output



```
Table creation --> buffer
id name phone
Row Insertion in buffer.
id name phone
1 Ashish 7828826871
Queries Executed Successfully
$$$$$$$$$$ Enter your input $$$$$$$$$$$$$$

1. Create Database
2. Select Database
3. Show Tables
4. Perform Query
5. Exit

CustomDB > tables.txt
```

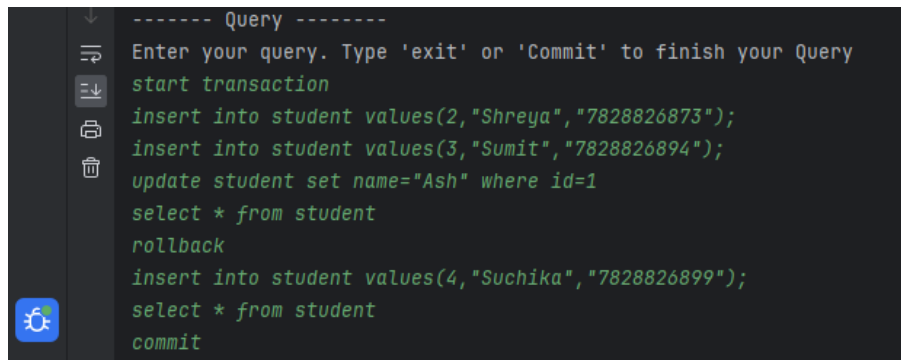
Figure 26: Queries Output

### Test case 3:

**Perform Transaction - insert and update data into student table. View all the contents of the table using the select command. Rollback the previous queries and then insert a new row. Show the current status of the table. Commit the transaction and show data again.**

#### 1. Query

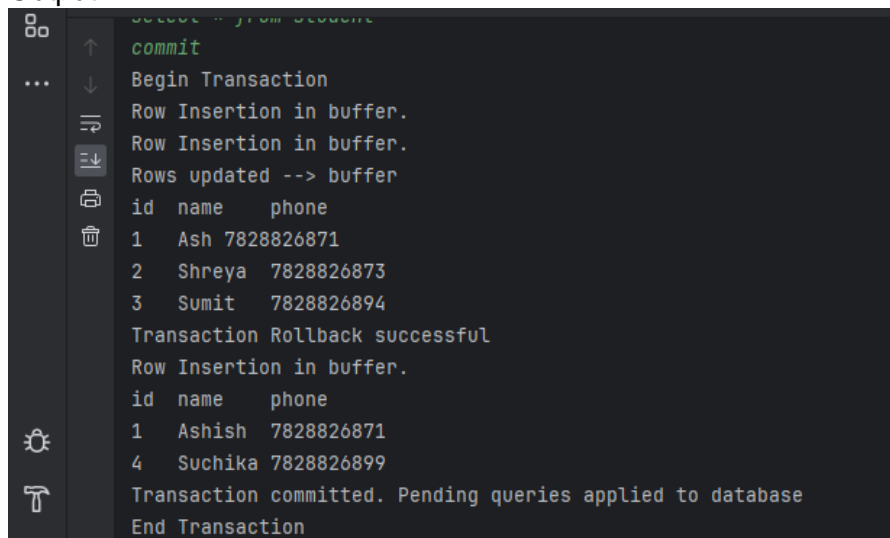
```
start transaction
insert into student values(2,"Shreya","7828826873");
insert into student values(3,"Sumit","7828826894");
update student set name="Ash" where id=1
select * from student
rollback
insert into student values(4,"Suchika","7828826899");
select * from student
commit
```

A screenshot of a SQL query editor with a dark background. The editor shows a series of SQL commands for a transaction. On the left side, there is a vertical toolbar with icons for undo, redo, save, and other functions. The SQL commands are as follows:

```
----- Query -----
Enter your query. Type 'exit' or 'Commit' to finish your Query
start transaction
insert into student values(2,"Shreya","7828826873");
insert into student values(3,"Sumit","7828826894");
update student set name="Ash" where id=1
select * from student
rollback
insert into student values(4,"Suchika","7828826899");
select * from student
commit
```

Figure 27: Transaction Query

#### 2. Output

A screenshot of a SQL query editor showing the output of the transaction queries. The output is displayed in a text area on the right side of the editor. The output text is as follows:

```
select * from student
commit
Begin Transaction
Row Insertion in buffer.
Row Insertion in buffer.
Rows updated --> buffer
id name phone
1 Ash 7828826871
2 Shreya 7828826873
3 Sumit 7828826894
Transaction Rollback successful
Row Insertion in buffer.
id name phone
1 Ashish 7828826871
4 Suchika 7828826899
Transaction committed. Pending queries applied to database
End Transaction
```

Figure 28: Transaction Output

```
----- Query -----  
Enter your query. Type 'exit' or 'Commit' to finish your Query  
select * from student  
exit  
id  name    phone  
1   Ashish  7828826871  
4   Suchika 7828826899  
Queries Executed Successfully
```

*Figure 29: Table after transaction*

## References:

- [1] D. Damoah, J. B. Hayfron-Acquah, S. Sebastian, E. Ansong, B. Agyemang and R. Villafane "A Transaction recovery in federated distributed database systems", Proceedings of IEEE International Conference on Conference on Computer Communication and Systems ICCCS14, Chennai, India, 2014, pp. 116-123, DOI: 10.1109/ICCCS.2014.7068178.
- [2] X. Dong and X. Li, "A Novel Distributed Database Solution Based on MySQL," 2015 7th International Conference on Information Technology in Medicine and Education (ITME), Huangshan, China, 2015, pp. 329-333, doi: 10.1109/ITME.2015.48.
- [3] BMC, "The Importance of SOLID Design Principles," *BMC Blogs*, [Online], June 15, 2020 Available: <https://www.bmc.com/blogs/solid-design-principles/>. [Accessed: November 03, 2023].
- [4] DigitalOcean, "SOLID: The First 5 Principles of Object Oriented Design," *DigitalOcean Community*, [Online], September 21, 2020 Available: <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design> [Accessed: November 03, 2023].
- [5] Oracle, "Methods for Channels and ByteBuffers," *Oracle Java SE Documentation*, [Online]. Available: <https://docs.oracle.com/javase/tutorial/essential/io/file.html#channels>. [Accessed: October 23, 2023].
- [6] Baeldung, "MD5 Hashing in Java," *Baeldung*, [Online], January 9, 2021 Available: <https://www.baeldung.com/java-md5>. [Accessed: October 23, 2023].
- [7] GeeksforGeeks, "HashMap in Java," *GeeksforGeeks*, [Online], September 6, 2023 Available: <https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/>. [Accessed: October 20, 2023].
- [8] GeeksforGeeks, "Split() String method in Java with examples," *GeeksforGeeks*, [Online], September 6, 2023 Available: <https://www.geeksforgeeks.org/split-string-java-examples/>. [Accessed: October 23, 2023].
- [9] S. Dey (October 5, 2023), "SQL statements of the transaction," Room 5260, Department of Psychology, Dalhousie University. [PowerPoint slides available: <https://dal.brightspace.com/d2l/le/content/284056/viewContent/3904477/View>]