



CSCI5408 | PROJECT REPORT

Instructor – Dr. Saurabh Dey

Dhrumil Amish Shah (B00857606)
Jaspreet Kaur Gill (B00879409)
Samiksha Narendra Salgaonkar (B00865423)

RELATIONAL DATABASE MANAGEMENT SYSTEM

Problem Statement

The requirement of the project is to develop a simple relational database along with its management system. Our team has to explore the concepts of the data structures that need to be used for the implementation of the system. Multi-factor authentication must be implemented with passwords stored in the encrypted format. Users should be able to execute general SQL queries, generate ERD & SQL dumps, logs should be stored in files, proper locking mechanism needs to be developed to perform transaction concurrency control. Functionalities of an efficient relational database management system through a command-line interface should be provided to the user without showcasing all the working details.

Understanding of the problem

The first and foremost focus is on developing a user interface that provides a better user experience and usability. Therefore, it is considered to have a simple and clear console interface that directs the user to perform specific relational database operations such as query execution, SQL dumps or ERD generation, creation of data-dictionary and metadata, and implementing transaction concurrency control. Also, the proper query validations that pertain to syntax and semantic checks should be done before query execution. Another important aspect is to store the data efficiently to implement a robust and efficient relational database management system.

Team Coordination –

We have worked on the implementation of this project as a team and used to meet frequently to discuss the approach for its development. As a team, we have designed the architecture, and developed the project. It is ensured that each member of the group would get a chance to work on the end-to-end flow of the system to learn and understand the relational database management system completely. Our aim is to develop an efficient system that operates with optimal performance.

Individual Contribution –

1) Dhrumil Amish Shah (B00857606)

Tasks Completed:

- Proposed and developed the project structure and setup for the relational database management system. Suggested and implemented best programming practices for the development of the project.
- Designed and developed the logic for multifactor user authentication and authorization.
- Designed and implemented the SQL dump generator.
- Explored and developed ERD generator with efficiency.
- Analysed and implemented the transaction management module with a full-proof design.
- Implemented the concurrency control module with the help of shared and exclusive locks.

- Designed and developed a system to import SQL dumps in the application.

New Learnings:

- Learnt how the relational database management system works internally.
- Learnt and developed user authentication while maintaining the session.
- Learnt the working of SQL dump generator.
- Got an understanding of the ERD generation feature.
- Understood the working of locks and their use to control concurrent transactions.
- Explored methods of implementing transaction management to conclude on the best approach.

2) Jaspreet Kaur Gill (B00879409)

Tasks Completed:

- Analysed, designed and implemented the logic to gather query triggered by each user along with the query execution time, stored and maintained as query logs.
- Designed and developed the code for storing all the generic operations such as query execution, ERD or SQL dump generation, performed based on currently logged in user and the timestamp of their execution in the form of general logs.
- Built event logs to capture the failed operations based on logged in user and the time of event occurrence.
- Explored and coded metadata generation.

New Learnings:

- Analyzed and understood the end to end working of database management system.
- Learnt the query execution mechanism to capture query logs.
- Understood the errors that can occur due to incorrect inputs from the user to log crashed events.
- Learnt the working of session management and all the operations of DBMS like metadata, data dictionary, query parsing and execution to maintain general logs based on the logged in user.
- Learnt to generate metadata with efficient data storage.

3) Samiksha Narendra Salgaonkar (B00865423)

Tasks completed:

- Analyzed, designed and implemented the database query parser to parse the queries for syntactical checks.
- Discussed the approach of implementing the database query processor within the team.
- Analyzed, designed and developed the query processor module for the SQL queries implemented as a part of this project.
- Implemented code for custom exceptions to handle errors reported in the form of exceptions in the project.

- Integrated data-dictionary generator considering efficient data storage.

New Learnings:

- Learnt and understood overall working of the database management system.
- Had to come up with ways to develop a query parser and processor for the project.
- Learnt the usage of regex to parse and implement query processor.
- Learnt and understood the functionality of exceptions to develop our own exceptions to provide user-friendly exception messages.
- Learnt the use of data-dictionary for its' implementation.

Meeting Logs:

Meeting 1 – Introduction | Meet & Greet

Figure 2 and 3 displays the meeting logs and Minutes of the meeting (MoM) of our first meeting respectively.

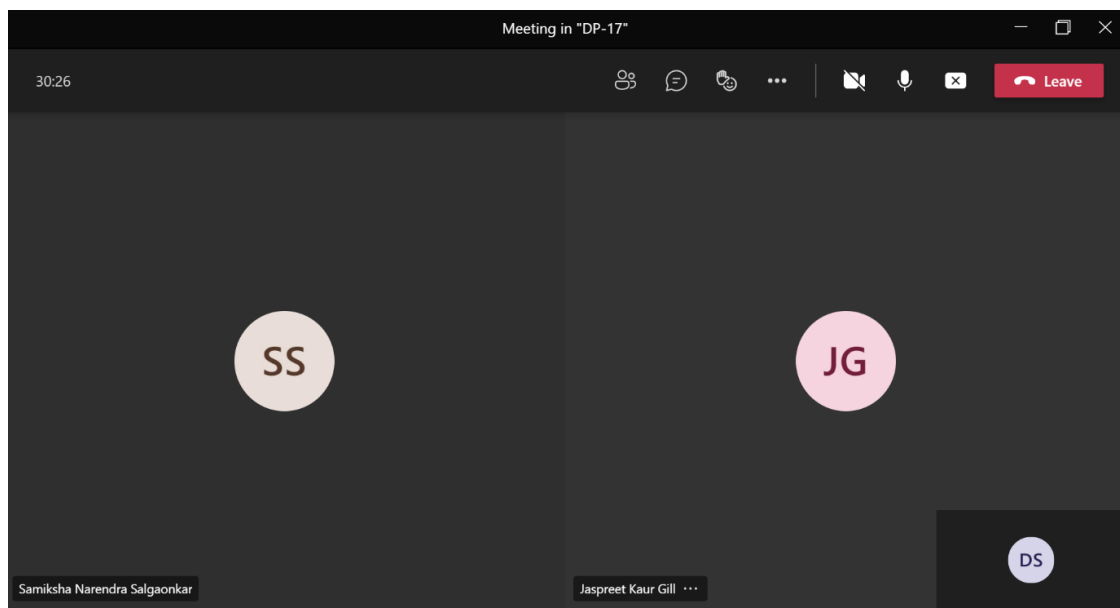


Figure 1: Meeting 1 - "Introduction | Meet & Greet"

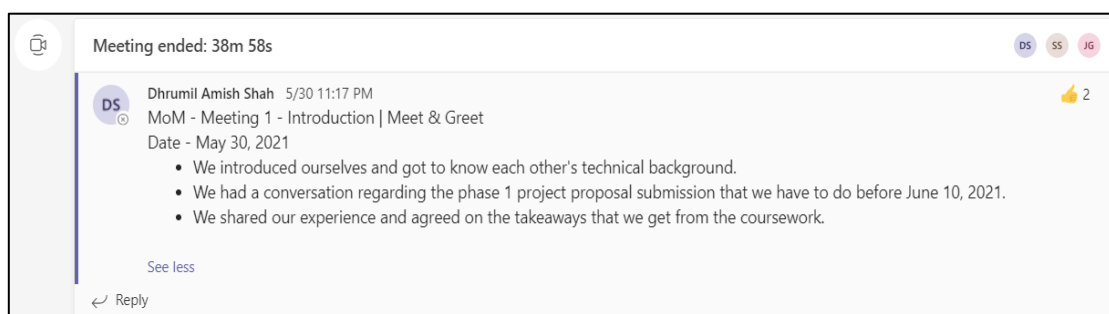


Figure 2: Meeting 1 – MOM

Meeting 2 – Discussion on Project Requirements

Figure 4 and 5 displays the meeting logs and Minutes of the meeting (MoM) of our second meeting respectively.

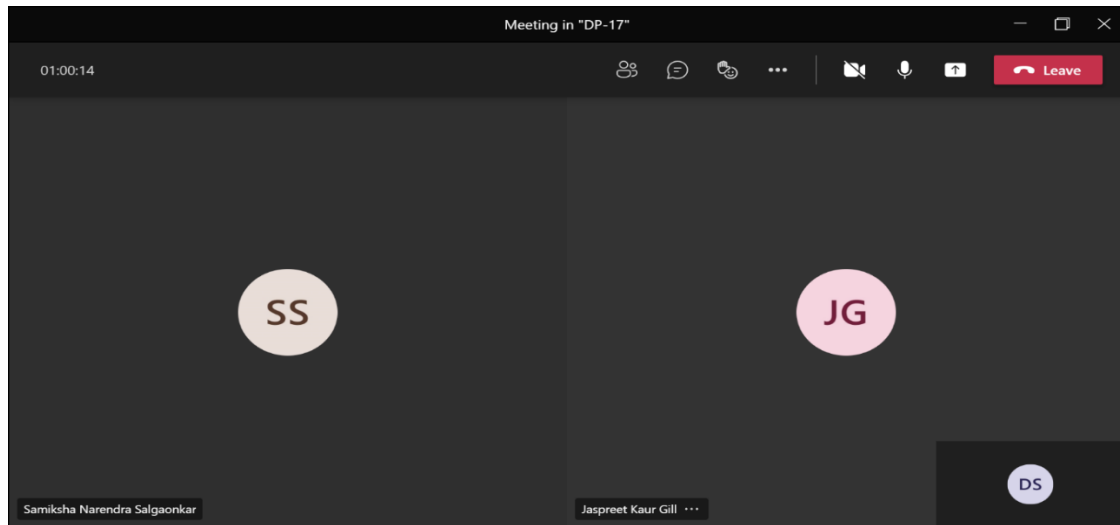


Figure 3: Discussion on Project Requirements

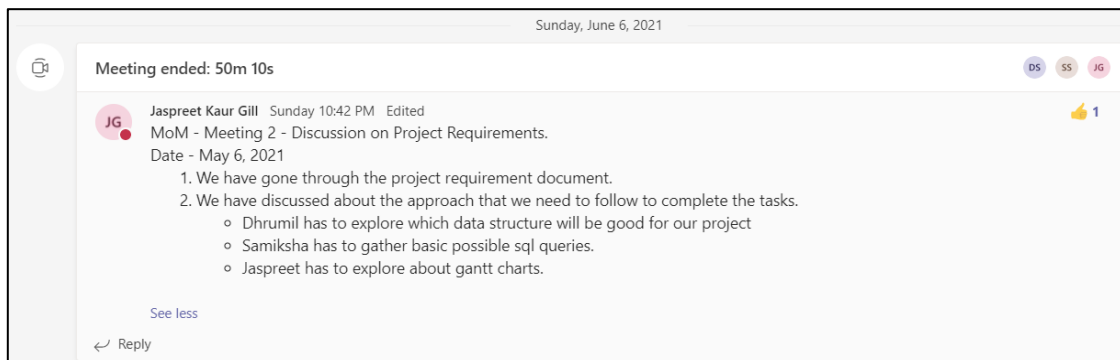


Figure 4: Meeting 2 - MoM

Meeting 3 – Feasibility Analysis and Report Progress Check

Figure 7 displays the Minutes of the meeting (MoM) of our third meeting.

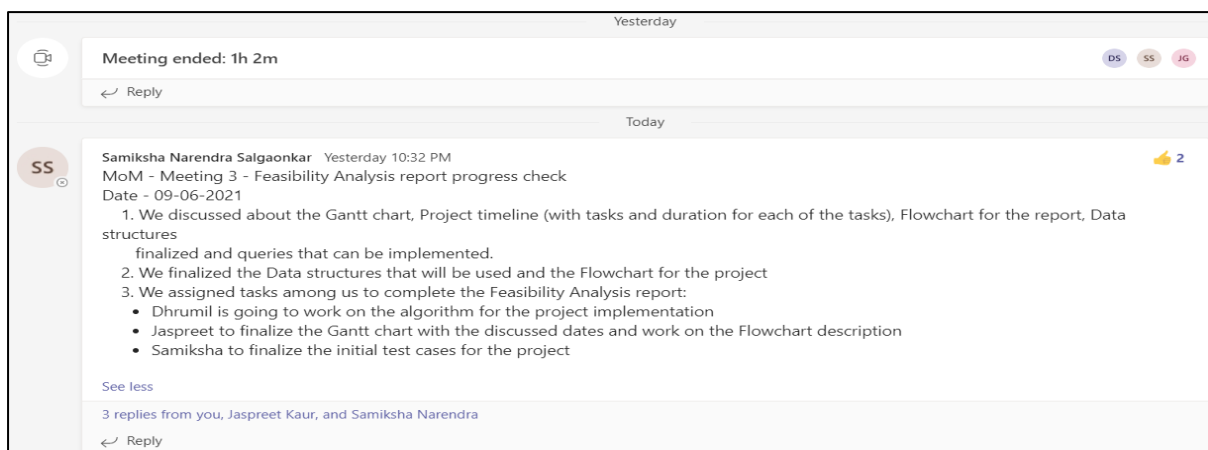


Figure 5: Meeting 3 – MoM

Meeting 4 – Report Meeting

Figure 8 displays the meeting logs of our fourth meeting.

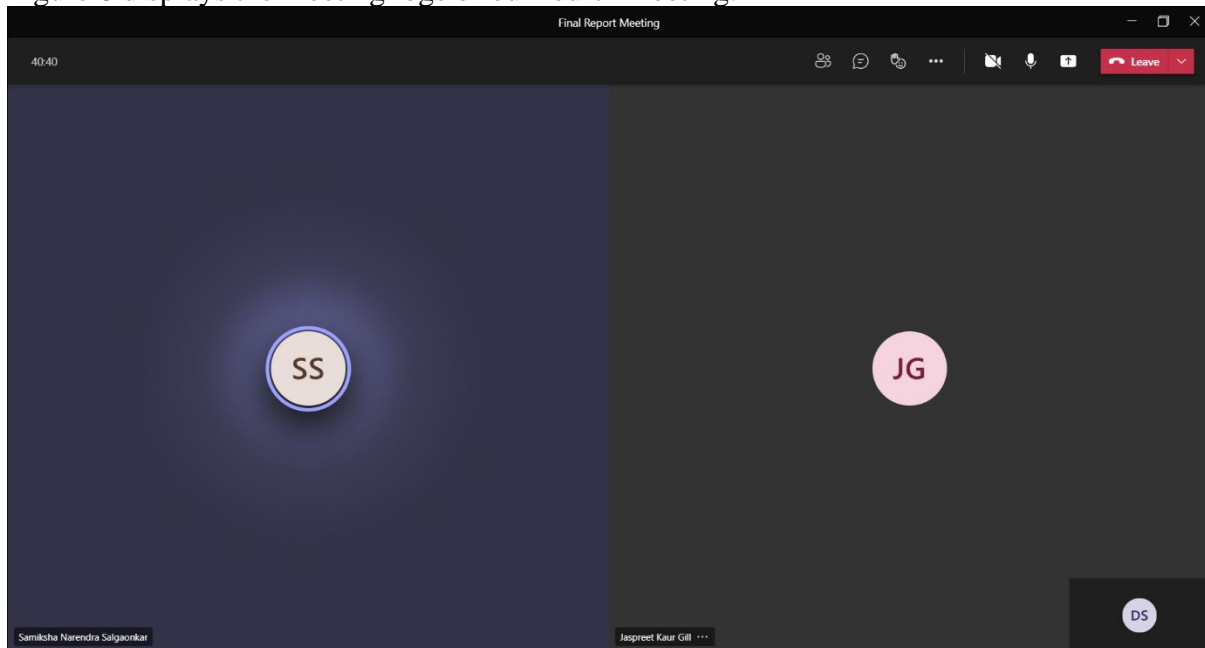


Figure 6: Document edit

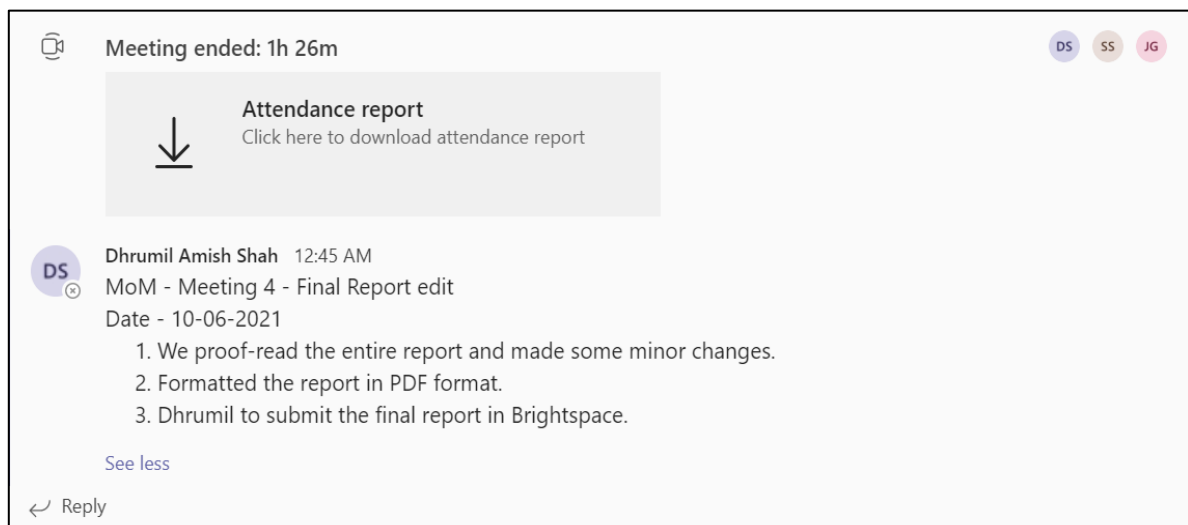


Figure 7: Meeting 4 – MoM

Implementation Details

We have implemented a Relational Database Management System (RDBMS) using Java Collection APIs that provide the best time complexity with their operations. This would ensure the implementation of efficient in-memory operations for the database. This section covers the usage of data structures, IDE, process flow, system architecture, implemented modules, and algorithms for the queries parsed and processed.

- **Data Structure Details** – We have used Array, List, Map, and Set data structure as the list is used to store data in a structured format, maps store data in key-value

format, set helps retain unique values, and array stores data of all types of fixed length. As we have used core Java, these data structures are feasible to use and offer an advantage as we can implement the combinations of arrays, lists, maps, and sets.

- **Implementation Environment** – Entire project is developed using Java with version 8. No external or third-party libraries are used. IntelliJ IDEA is used as an integrated development environment. For team collaboration, we have decided to make use of GitLab [1].
- **Block Diagram** – Figure 8 given below represents the design of our relational database management system.

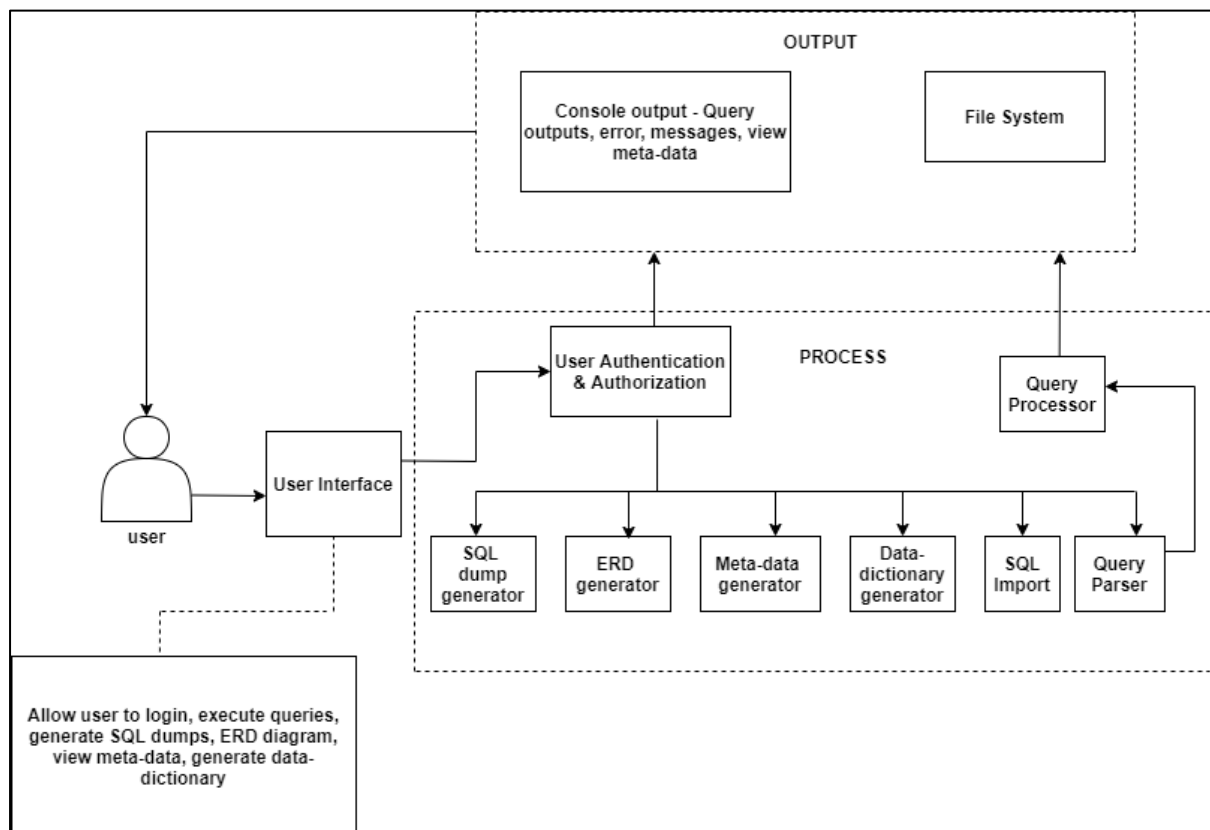


Figure 8: Block Diagram [2]

For implementing various modules of the relational database management system, we have made use of File class of Java, along with some of its most commonly used methods to process the queries we have implemented. We have implemented this database management system for four data types i.e. INT, FLOAT, TEXT, and BOOLEAN.

In an attempt to implement good programming practices, we have designed the project structure to contain four primary components:

backend: This component handles the logic pertaining to all the features of the relational database management system.

frontend: This component handles the presentation layer of the application where user interaction takes place.

database: This component stores all the databases, tables and data created in the application.

output: This component contains the logs, ERD, metadata, data-dictionary, and SQL dumps for the application.

For every error that is reported, we have designed it to be handled using custom Exception classes that we have created for each of our features.

- **Initial Task Flow diagram** – Figure 9 represents the flow of the database management system that we have proposed.

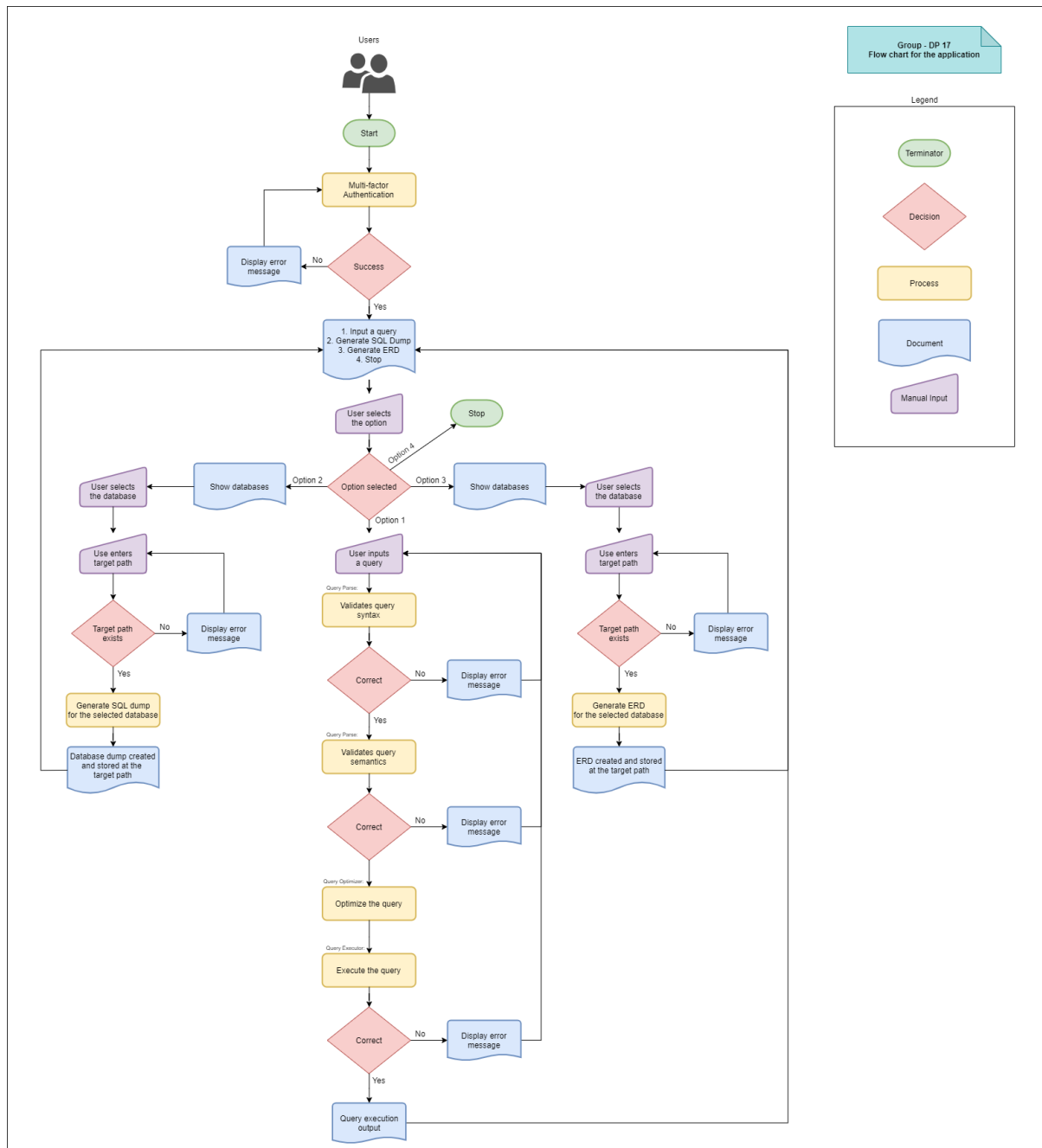


Figure 9: Task flow of the application [2]

- **Algorithm –**

The algorithm for the implementation of the system is described below-

Step 1: Start

Step 2: User performs authentication

Step 3: If user authentication fails, display an error message, and go to step 2

Step 4: If user authentication succeeds, go to step 5

Step 5: Display the below options:

1. Execute SQL Query
2. Generate SQL Dump
3. Generate ERD
4. Generate Data Dictionary
5. View Meta Data
6. Import SQL Dump
7. Logout

Step 6: User selects an option from step 5

If option 1 is selected, go to step 7

If option 2 is selected, go to step 19

If option 3 is selected, go to step 23

If option 4 is selected, go to step 27

If option 5 is selected, go to step 31

If option 6 is selected, go to step 38

If option 7 is selected, go to step 42

Step 7: User inputs a query

Step 8: Query parser validates the syntax of the query

Step 9: If query validation fails, display an error message, and go to step 7

Step 10: If query validation succeeds, go to step 11

Step 11: Query parser validates the semantics of the query

Step 12: If query validation fails, display an error message, and go to step 7

Step 13: If query validation succeeds, go to step 14

Step 14: Query optimizer optimizes the query

Step 15: Query executor executes the query

Step 16: If query execution fails, display an error message, and go to step 7

Step 17: If query execution succeeds, go to step 18

Step 18: Display the query execution output and go to step 5

Step 19: User enters database name

Step 20: If database name validation fails, display an error message, and go to step 19

Step 21: If database name validation succeeds and the database exists, then go to step 22

Step 22: SQL dump will be generated, stored in the form of a text file at a specific path, and go to step 5

Step 23: User enters database name

Step 24: If database name validation fails, display an error message, and go to step 23

Step 25: If database name validation succeeds and the database exists, then go to step 26

Step 26: SQL ERD will be generated, stored in the form of a text file at a specific path, and go to step 5

Step 27: User enters database name

Step 28: If database name validation fails, display an error message, and go to step 27
 Step 29: If database name validation succeeds and the database exists, then go to step 30
 Step 30: Data dictionary will be generated, stored in the form of a text file at a specific path, and go to step 5
 Step 31: User enters database name
 Step 32: If database name validation fails, display an error message, and go to step 31
 Step 33: If database name validation succeeds and the database exists, then go to step 34
 Step 34: User enters table name
 Step 35: If table name validation fails, display an error message, and go to step 31
 Step 36: If table name validation succeeds and table exists, then go to step 37
 Step 37: The system will display the metadata of the selected table on the console and go to step 5
 Step 38: User enters .sql file to import SQL dump
 Step 39: If the file is not valid, display an error message, and go to step 38
 Step 40: If the file is valid then go to step 41
 Step 41: SQL dump is imported successfully and go to step 5
 Step 42: Stop

The project has several modules, and this section displays those implemented modules.

Multifactor Authentication – While registering, the user will be asked to set username, email, password and answer four security questions. While login, the user will be asked to enter a username or email, password and must answer randomly asked one of the four security questions. Also, password is stored in encrypted format using the SHA-256 hashing algorithm.

Query Parser – Query parser validates the syntax of the query entered by the user. If validation fails, then it returns an error to the user otherwise passes the query to the query processor module.

Query Processor – Query processor performs some validations to test the semantics of the entered query and then executes the query to displays the result of the query to the user via console.

SQL Dump Generator – It generates the SQL dump in the form of a .sql file for the database entered by the user if the database is valid, otherwise it will display an error on the console.

ERD Generator – It generates the ERD of the database entered by the user and stores the ERD in a text file if the database is valid. On the other hand, if the database is not valid, then it will throw an error to the user.

Data Dictionary Generator – Data dictionary generator prompts the user to enter the database name. If the entered database name is incorrect (the database is not present in the database), then it will display an error message to the user. But, if the entered database name is correct, then it will generate a data dictionary and stores it at the specific path.

Meta Data Generator – Metadata generator generates the metadata automatically when the “CREATE TABLE” query is triggered by the user and stores it in the text file at the specified path. The user is also provided with an option to view the metadata in the main menu. The user is asked to enter the database name and table name. If the entered details are valid then the user can view the metadata on the console for the mentioned table of the database. But, if the entered details such as database or table name are not correct, then the user will be notified with an error message.

Import SQL Dump – The user can also import the SQL dump. Users can choose the option to import the SQL dump and then must enter the .sql file to import it. If the .sql file is valid, then the database and tables inside it are created. But if the file is not valid, then the user will be notified via an error message.

Transaction Management – The user can also execute a set of queries at a time in the form of transactions. We have implemented transaction management with the help of the following three queries:

START TRANSACTION: This query indicates the starting point of transaction.

COMMIT: This query tells the system that the transaction should be committed.

ROLLBACK: This query tells the system that the transaction should be rolled back.

As in the Query parser, the transaction management module also parses the entered queries for syntactical correctness. We have implemented transaction management system by creating a temporary copy of the database in use into a in-memory location. The changes will be committed or discarded to the primary database server based on the operations that the user executes i.e. COMMIT or ROLLBACK.

Concurrency control - When two transactions are executing at the same time and perform a read operation on the same table then the shared lock will be applied, and transactions can read the data at the same time. But, if two transactions are executing at the same time and are writing any data into the table, then an exclusive lock will be applied to the table by the transaction which enters the system first and the lock will be released when the transaction commits or is rolled back.

Log Generator – Logs are generated and are stored in text files. Three types of logs are generated namely event logs, general logs, and query logs.

1. **Query Logs** – Query logs are used to store the query and the timestamp at which that query is executed. Logs are stored based on the currently logged-in user.
2. **General Logs** – It is used to store the general operations performed by the user at the timestamp. Operations such as generating SQL dumps, ERD, creating databases and tables, and the list goes on.
3. **Event Logs** – Event logs store the crash events that are the errors that are raised when the user is performing operations based on the current user.

Use Cases:

- New user registration using username, email, password, and security questions.
- Login already registered user with multi-factor authentication.
- Perform SQL queries –
 - Create database
 - Use database
 - Drop database
 - Create table
 - Drop table
 - Truncate table
 - Select all query
 - Select distinct query
 - Insert data into the table
- Generate SQL dump.
- Generate ERD.
- Create metadata automatically when create query is triggered by the user.
- View metadata of the table on the console.
- Generate data-dictionary.
- Import SQL dump.
- Perform concurrent transactions.
- Generate query, general and event logs.
- Exit application.

Technical details

The relational database management application supports multiple features listed in . The pseudocodes for each of these features are described in detail in this section:

Multi-factor Authentication –

The multi-factor authentication feature encompasses two modules namely User registration and User login.

User registration:

1. Start
2. Accept input from the user for setting username, email, password, and answers to the four security questions.
3. Validate the entered username, email, password, and answers for correctness against the allowed values for each of the fields.
4. If the value entered for any of the fields is incorrect, throw a custom `UserAuthenticationException` for the encountered errors and go to step 13.
5. Validate if the user is registered already by checking the username or email entered is stored in the application for a user registered previously.
6. If the username or email is registered already, throw a custom `UserAuthenticationException` stating that the user already exists and go to step 13.
7. Store the userID entered by the user in a string object to be returned.
8. Concatenate the entered username into the string object to be returned.

9. Hash the entered password using SHA-256 algorithm and **getSHA256Hash()** method and concatenate the hashed password to the string object to be returned.
10. Concatenate the entered answers to the security questions to the string object to be returned.
11. Write the string object containing all the details to a file using an instance of **FileWriter** class.
12. If any error is encountered while writing to file, throw a custom **UserAuthenticationException** stating the error and go to step 13.
13. Stop

User login:

1. Start
2. Accept input from the user for username or email for logging into the application.
3. Accept input from the user for password for logging into the application.
4. Generate a security question randomly using the **Random** class of Java.
5. Validate the entered username or email, password, and security answer for correctness against the allowed values for each of the fields.
6. If the value entered for any of the fields is incorrect, throw a custom **UserAuthenticationException** for the encountered errors and go to step 13.
7. Hash the entered password using SHA-256 algorithm and **getSHA256Hash()** method and validate it against the stored hashed password during registration.
8. If the hashed values of the passwords do not match, throw a custom **UserAuthenticationException** stating the error and go to step 13.
9. Validate if the answer for the security question matches with the answer stored during registration.
10. If the answers do not match, throw a custom **UserAuthencnticationException** stating the error and go to step 13.
11. Set a variable **loggedInUser** of type **User** to store the session of the logged in user.
12. If an error was encountered while logging in user and session, throw a custom **UserAuthencnticationException** stating the error and go to step 13.
13. Stop

Queries –

There are nine queries implemented for this database management system which include **CREATE DATABASE, USE DATABASE, CREATE TABLE, DROP DATABASE, DROP TABLE, SELECT ALL, SELECT DISTINCT, INSERT INTO, and TRUNCATE TABLE.**

CREATE DATABASE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom **QueryParserException** stating the error and go to step 12.
4. Instantiate the **Instant** class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.

6. Process the query to extract the name of the database from the query using split() method of string. Split on a single whitespace character.
7. Validate if the directory already exists using the isDirectory() method of the File class.
8. If the directory already exists, throw a custom QueryProcessorException stating the error and go to step 12.
9. Create a directory on the database path with the database name specified by the user in the query using the mkdir() method of the File class.
10. If the directory was created successfully, return an instance of QueryProcessResponse with true parameter and a success message.
11. If an error encountered while creating the directory, throw a custom QueryProcessorException stating the error and go to step 12.
12. Stop

USE DATABASE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom QueryParserException stating the error and go to step 14.
4. Instantiate the Instant class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using split() method of string. Split on a single whitespace character.
7. Validate if the directory already exists using the isDirectory() method of the File class.
8. If the directory does not exist on the specified path, throw a custom QueryProcessorException stating the error and go to step 14.
9. List all the files on the database server path (consisting of all the databases i.e. directories) and store it in an array **files** of type File.
10. If the **files** array is null, throw a custom QueryProcessorException stating the error and go to step 14.
11. Iterate through the array **files** and check if the database name specified in the user query matches any of the directory names mentioned in the array.
12. If the name of the current file in iteration matches the database name specified by the user, assign the file name as value of the global variable **useDatabaseName**.
13. If the useDatabaseName was set successfully, return an instance of QueryProcessResponse with true parameter and a success message.
14. Stop.

DROP DATABASE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom QueryParserException stating the error and go to step 16.
4. Instantiate the Instant class to get the current timestamp and store it in a variable.

5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using split() method of string. Split on a single whitespace character.
7. Validate if the directory already exists using the isDirectory() method of the File class.
8. If the directory does not exist on the specified path, throw a custom QueryProcessorException stating the error and go to step 16.
9. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type File.
10. If the **allTables** array is null, throw a custom QueryProcessorException stating the error and go to step 16.
11. Iterate through the array **allTables** and delete the text file in that database using the delete() method of the File class.
12. If an error encountered while deleting the table i.e. the text file, throw a custom QueryProcessorException stating the error and go to step 16.
13. If the tables were deleted successfully, then delete the database i.e. the directory using the delete() method after all the tables within the database are deleted.
14. If an error encountered while deleting the table i.e. the text file, throw a custom QueryProcessorException stating the error and go to step 16.
15. If the database was deleted successfully, return an instance of QueryProcessResponse with true parameter and a success message.
16. Stop.

CREATE TABLE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom QueryParserException stating the error and go to step 24.
4. Instantiate the Instant class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using split() method of string. Split on a single whitespace character.
7. Validate if the useDatabaseName is set to a value other than null or an empty string.
8. If useDatabaseName is null or empty, throw a custom QueryProcessorException stating the error and go to step 24.
9. Validate if the directory already exists using the isDirectory() method of the File class.
10. If the directory does not exist on the specified path, throw a custom QueryProcessorException stating the error and go to step 24.
11. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type File.
12. If the **allTables** array is null, throw a custom QueryProcessorException stating the error and go to step 24.

13. Iterate through the array **allTables** and check if the table name specified in the user query matches any of the text files' names mentioned in the array.
14. If any files' name matches the table specified by the user, throw a custom `QueryProcessorException` stating the error and go to step 24.
15. If none of the array elements' names match the table name specified by the user, then create a text file with the file name as the table name specified by the user in the query,
16. Process the query further to extract the column names, primary or foreign key constraints, data types assigned for each of the fields in the table and split the columns by a , character.
17. Create a `StringBuilder` instance to build a string defining the columns in the table where each column is separate by the delimiter `$$@ @|||@ @$$`.
18. Trim `$$@ @|||@ @$$` from the end of the string builder instance and append a new line to it.
19. Write the String value of the string builder instance to the created text file as the first line that defines the columns of the table in the database.
20. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 24.
21. If the string was written to the text file successfully, create a text file with the database name, table name and column definition formatted in a string instance and written to a separate file as the metadata file for the table.
22. If the metadata file for the created table was built successfully, return an instance of `QueryProcessResponse` with true parameter and a success message.
23. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 24.
24. Stop.

SELECT ALL:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom `QueryParserException` stating the error and go to step 24.
4. Instantiate the `Instant` class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using `split()` method of string. Split on a single whitespace character.
7. Validate if the **useDatabaseName** is set to a value other than null or an empty string.
8. If **useDatabaseName** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 24.
9. Validate if the directory already exists using the `isDirectory()` method of the `File` class.
10. If the directory does not exist on the specified path, throw a custom `QueryProcessorException` stating the error and go to step 24.
11. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type `File`.

12. If the **allTables** array is null, throw a custom `QueryProcessorException` stating the error and go to step 24.
13. Iterate through the array **allTables** and check if the table name specified in the user query matches any of the text files' names mentioned in the array.
14. If any files' name does not match the table specified by the user, throw a custom `QueryProcessorException` stating the error and go to step 24.
15. If any of the array elements' names match the table name specified by the user, then create a `FileReader` object to read the contents in the text file i.e. the table of the database.
16. Create a string builder object **selectStringBuilder** which contains the tokens extracted. Extract only the column names and values of the those columns in the text file.
17. Read the current line of the file by creating tokens by splitting the read line using the delimiter `$$@ @|||@ @$$` and store it in an array of type `String`.
18. Append every token with `|` to separate the column names and values by the separator.
19. Append a new line to the created string builder instance.
20. Repeat step 17, 18 and 19 for every line read in the text file.
21. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 24.
22. If the string builder instance was created successfully for all lines of the file, return an instance of `QueryProcessResponse` with `true` parameter and a success message.
23. Print the string value of the string builder object on the console.
24. Stop.

SELECT DISTINCT:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom `QueryParserException` stating the error and go to step 29.
4. Instantiate the `Instant` class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character `;` from the entered query.
6. Process the query to extract the name of the database from the query using `split()` method of `String`. Split on a single whitespace character.
7. Validate if the **useDatabaseName** is set to a value other than null or an empty string.
8. If **useDatabaseName** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 29.
9. Validate if the directory already exists using the `isDirectory()` method of the `File` class.
10. If the directory does not exist on the specified path, throw a custom `QueryProcessorException` stating the error and go to step 29.
11. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type `File`.
12. If the **allTables** array is null, throw a custom `QueryProcessorException` stating the error and go to step 29.

13. Iterate through the array **allTables** and check if the table name specified in the user query matches any of the text files' names mentioned in the array.
14. If any files' name does not match the table specified by the user, throw a custom `QueryProcessorException` stating the error and go to step 29.
15. If any of the array elements' names match the table name specified by the user, then create a `FileReader` object to read the contents in the text file i.e. the table of the database.
16. Create a string builder object **selectStringBuilder** which contains the tokens extracted. Extract only the column names and values of those columns in the text file.
17. Set an integer variable **columnIndexInInterest** and initialize the value to -1 to track the column whose distinct value is queried. Set a Boolean variable **isHeading** and initialize it to true.
18. Iterate through the `rawColumns` array and perform steps 19-23 for every element in the array.
19. Read the current line of the file by creating tokens by splitting the read line using the delimiter `$$@ @|||@ @$` and store it in an array **rawColumns** of type `String`.
20. If the value of any of the tokens created match the column queried by the user, set the **columnIndexInInterest** to the index in iteration.
21. Append every token with `|` to separate the column names and values by the separator.
22. Append a new line to the created string builder instance.
23. Add the value of the `rawColumns` at the **columnIndexInInterest** index in a `LinkedHashSet` **uniqueElements** to maintain the insertion order.
24. Set the **isHeading** Boolean variable to false after reading the first line during iteration of **rawColumns**.
25. If the value of **columnIndexInInterest** is -1 after the first iteration, throw a custom `QueryProcessorException` stating the error and go to step 29.
26. Iterate through the `LinkedHashSet` **uniqueElements** to create a string builder instance of distinct elements.
27. If the string builder instance was created successfully for all lines of the file, return an instance of `QueryProcessResponse` with true parameter and a success message.
28. Print the string value of the string builder object on the console.
29. Stop.

INSERT INTO:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom `QueryParserException` stating the error and go to step 33.
4. Instantiate the `Instant` class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character `;` from the entered query.
6. Process the query to extract the name of the database from the query using `split()` method of string. Split on a single whitespace character.
7. Validate if the **useDatabaseName** is set to a value other than null or an empty string.
8. If **useDatabaseName** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 33.

32. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 33.
33. Stop.

TRUNCATE TABLE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.
3. If an error was encountered while validating query syntax, throw a custom `QueryParserException` stating the error and go to step 21.
4. Instantiate the `Instant` class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using `split()` method of string. Split on a single whitespace character.
7. Validate if the **useDatabaseName** is set to a value other than null or an empty string.
8. If **useDatabaseName** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 21.
9. Validate if the directory already exists using the `isDirectory()` method of the `File` class.
10. If the directory does not exist on the specified path, throw a custom `QueryProcessorException` stating the error and go to step 21.
11. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type `File`.
12. If the **allTables** array is null, throw a custom `QueryProcessorException` stating the error and go to step 21.
13. Iterate through the array **allTables** and check if the table name specified in the user query matches any of the text files' names mentioned in the array.
14. If any files' name does not match the table specified by the user, throw a custom `QueryProcessorException` stating the error and go to step 21.
15. If any of the array elements' names match the table name specified by the user, then create a `FileWriter` and a `FileReader` instance to read from the file in use.
16. Read the first line of the file in use and store it in a string literal named **writeToFile**.
17. If **writeToFile** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 21.
18. If **writeToFile** is not null and non-empty value, then open the file with append mode set to false, write the **writeToFile** to the file and close the file instance in use.
19. If the file was closed successfully, return an instance of `QueryProcessResponse` with true parameter and a success message.
20. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 21.
21. Stop.

DROP TABLE:

1. Start
2. Validate the user entered query for syntax correctness using regular expressions.

3. If an error was encountered while validating query syntax, throw a custom `QueryParserException` stating the error and go to step 18.
4. Instantiate the `Instant` class to get the current timestamp and store it in a variable.
5. Extract the substring of the query by removing the last character ; from the entered query.
6. Process the query to extract the name of the database from the query using `split()` method of string. Split on a single whitespace character.
7. Validate if the **useDatabaseName** is set to a value other than null or an empty string.
8. If **useDatabaseName** is null or empty, throw a custom `QueryProcessorException` stating the error and go to step 18.
9. Validate if the directory already exists using the `isDirectory()` method of the `File` class.
10. If the directory does not exist on the specified path, throw a custom `QueryProcessorException` stating the error and go to step 18.
11. List all the files on the database server's database path (consisting of all the tables in the databases i.e. text files) and store it in an array **allTables** of type `File`.
12. If the **allTables** array is null, throw a custom `QueryProcessorException` stating the error and go to step 18.
13. Iterate through the array **allTables** and check if the table name specified in the user query matches any of the text files' names mentioned in the array.
14. If any files' name does not match the table specified by the user, throw a custom `QueryProcessorException` stating the error and go to step 18.
15. If any of the array elements' names match the table name specified by the user, then delete the file instance using the `delete()` method of the `File` class.
16. If the file instance was deleted successfully, return an instance of `QueryProcessResponse` with true parameter and a success message.
17. If an error was encountered while writing to the text file, throw a custom `QueryProcessorException` stating the error and go to step 18.
18. Stop.

Test Cases

As a part of testing, the unit and integrated modules of the project, we have created JUnit test cases and placed the files at location -

/src/test/java/backend/query_parser/controller/QueryParserController.java.

Multi-factor Authentication –

This module will involve test cases for user registration and user login.

User registration:

1. User registration for a user that is already a registered user of the system

```

Run: CSCI5408DP17EntryView
Welcome to CSCI5408 DP17 Project
*****
1. User registration.
2. User login.
3. Exit.
Select an option:
Enter username(Must be alphanumeric characters only)
samiksha
Enter email(Example: johndoe@gmail.com)
samiksha@gmail.com
Enter password (1 small letter, 1 capital letter, 1 special char, and 1 number and min length 8 and max length 20)
samiksha@123
What was your name of first pet?
Kio
What is your birth year?
1999
What are the last 4 digits of your telephone number?
1234
What is your favorite sport?
Hockey
UserAuthenticationException{errorMessage='User exists already!'}
1. User registration.
2. User login.
3. Exit.
Select an option:

```

Figure 1: User registration for a registered user

2. User login attempt with wrong credentials

```

Run: CSCI5408DP17EntryView
C:\Users\Samiksha\jdk8\openjdk-16.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2020.3.2\lib\idea_rt.jar=53947:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2020.3.2\bin" -Dfile.encoding=UTF-8
Welcome to CSCI5408 DP17 Project
*****
1. User registration.
2. User login.
3. Exit.
Select an option:
2
Enter username/email
samiksha
Enter password
samiksha@123
What was your name of first pet?
Kio
Logged in as: samiksha
*****
Main Menu
*****
1. Execute SQL Query.
2. Generate SQL Dump.
3. Generate ERD.
4. Generate Data Dictionary.
5. View Meta Data.
6. Import SQL Dump.
7. Logout.
Select an option:
1

```

Figure 2: User login with invalid credentials

Queries:

This section covers the test cases for all the implemented queries.

CREATE DATABASE:

1. Creating a database that already exists

```

CSCI5408DP17EntryView
Logged in as: samiksha
*****
Main Menu
*****
1. Execute SQL Query.
2. Generate SQL Dump.
3. Generate ERD.
4. Generate Data Dictionary.
5. View Meta Data.
6. Import SQL Dump.
7. Logout.
Select an option:

*****
Execute SQL Query
*****
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE DATABASE del;
QueryProcessorException{errorMessage='Error: Database del already exists! | Execution Time: 0ms'}
1. Execute SQL Query.
2. Go Back.
Select an option:

```

Figure 3: Creating a database that already exists

2. Creating a database with invalid syntax

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE DATABASE ;
QueryParserException{errorMessage='Invalid CREATE DATABASE query!'}
1. Execute SQL Query.
2. Go Back.
Select an option:

```

Figure 4: Creating a database with invalid syntax

3. Creating a database with valid syntax

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE DATABASE del;
Database del created successfully! | Execution Time: 0ms
1. Execute SQL Query.
2. Go Back.
Select an option:

```

Figure 5: Creating a database with valid syntax

USE DATABASE:

1. Using a database that does not exist

```

1. Execute SQL Query.
2. Go Back.
Select an option:
2
Enter SQL Query:
USE DATABASE employee;
QueryProcessorException{errorMessage='Error: Database employee does not exist! | Execution Time: 0ms'}

```

Figure 6: Using a database that does not exist

2. Executing an invalid USE DATABASE command

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
USE DATABASE ;
QueryParserException{errorMessage='Invalid USE DATABASE query!'}
```

Figure 7: Executing an invalid USE DATABASE command

3. Valid USE DATABASE command

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
USE DATABASE dal;
Database dal in use! | Execution Time: 0ms
1. Execute SQL Query.
2. Go Back.
Select an option:
|
```

Figure 8: Valid USE DATABASE command

CREATE TABLE

1. Invalid CREATE TABLE query

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE TABLE ;
QueryParserException{errorMessage='Invalid CREATE TABLE query!'}
```

Figure 9: Invalid CREATE TABLE query

2. Creating table without USE DATABASE query being executed prior.

```
*****
Execute SQL Query
*****
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE TABLE course (id INT PRIMARY KEY, name TEXT, description TEXT);
QueryProcessorException{errorMessage='Error: No database set! | Execution Time: 0ms'}
```

Figure 10: Creating table without USE DATABASE query being executed prior.

3. Valid CREATE table query

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
USE DATABASE dal;
Database dal in use! | Execution Time: 0ms
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE TABLE course (id INT PRIMARY KEY, name TEXT, description TEXT);
Table course created successfully! | Execution Time: 0ms

```

Figure 11: Valid CREATE table query

4. Creating table with foreign key constraint

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE TABLE student (id INT PRIMARY KEY, name TEXT, email TEXT, courseID INT REFERENCES course(id));
Table student created successfully! | Execution Time: 4ms

```

Figure 12: Creating table with foreign key constraint

5. Creating a table that already exists

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
CREATE TABLE course (id INT PRIMARY KEY, name TEXT, description TEXT);
QueryProcessorException{errorMessage='Error: Table course already exists! | Execution Time: 0ms'}
1. Execute SQL Query.
2. Go Back.
Select an option:
1

```

Figure 13: Creating a table that already exists

INSERT INTO

1. Invalid INSERT INTO query

```

*****
Execute SQL Query
*****
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
INSERT INTO ;
QueryParserException{errorMessage='Invalid INSERT query!'}

```

Figure 14: Invalid INSERT INTO query

2. Inserting data in table without USE DATABASE query being executed prior.

```
1. Execute SQL Query.
2. Go Back.
Select an option:
|
Enter SQL Query:
INSERT INTO course (id, name, description) VALUES (1, 'Course 1', 'Description 1');
QueryProcessorException{errorMessage='Database not set!'}
```

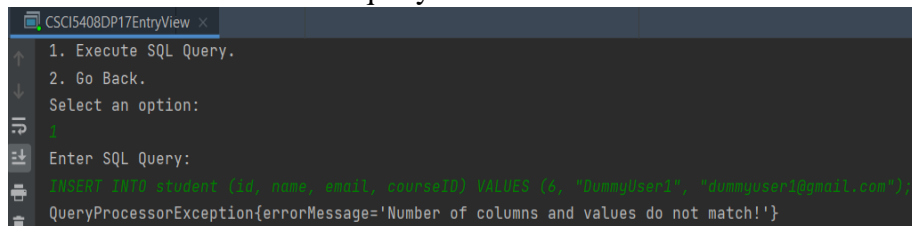
Figure 15: Inserting data in table without USE DATABASE query being executed prior.

3. Inserting data in a table that does not exist

```
1. Execute SQL Query.
2. Go Back.
Select an option:
|
Enter SQL Query:
INSERT INTO employee (id, name, email, courseID) VALUES (1, 'Chris', 'chris@gmail.com', 1);
QueryProcessorException{errorMessage='Table does not exists!'}
1. Execute SQL Query.
2. Go Back.
Select an option:
|
```

Figure 16: Inserting data in a table that does not exist

4. Invalid INSERT INTO query – number of columns and values do not match



```
1. Execute SQL Query.
2. Go Back.
Select an option:
|
Enter SQL Query:
INSERT INTO student (id, name, email, courseID) VALUES (4, 'DummyUser1', 'dummyuser1@gmail.com');
QueryProcessorException{errorMessage='Number of columns and values do not match!'}
```

Figure 17: Invalid INSERT INTO query – number of columns and values do not match

5. Invalid INSERT INTO query – fields in the query do not match with the table definition.

```
1. Execute SQL Query.
2. Go Back.
Select an option:
|
Enter SQL Query:
INSERT INTO student (id, name, email, courseID, courseName) VALUES (4, 'DummyUser1', 'dummyuser1@gmail.com', 3, 'Course 3');
QueryProcessorException{errorMessage='Number of fields in the table do not match!'}
```

Figure 18: Invalid INSERT INTO query – fields in the query do not match with the table definition.

6. Invalid INSERT INTO query – sequence of columns in the table and query do not match

```
1. Execute SQL Query.
2. Go Back.
Select an option:
|
Enter SQL Query:
INSERT INTO student (id, name, courseID, email) VALUES (4, 'DummyUser1', 3, 'dummyuser1@gmail.com');
QueryProcessorException{errorMessage='Sequence of columns does not match!'}
```

Figure 19: Invalid INSERT INTO query – sequence of columns in the table and query do not match

7. Invalid INSERT INTO query – data type of the columns of the table do not match

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
INSERT INTO student (id, name, email, courseID) VALUES (1, "Chris", "chris@gmail.com", "Course 1");
QueryProcessorException{errorMessage='For input string: "Course 1"'}

```

Figure 20: Invalid INSERT INTO query – data type of the columns of the table do not match

8. Valid INSERT INTO table query

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
USE DATABASE dal;
Database dal in use! | Execution Time: 15ms
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
INSERT INTO course (id, name, description) VALUES (2, "Course 2", "Description 2");
Data inserted in table course successfully!
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
INSERT INTO course (id, name, description) VALUES (3, "Course 3", "Description 3");
Data inserted in table course successfully!
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
INSERT INTO course (id, name, description) VALUES (4, "Course 4", "Description 4");
Data inserted in table course successfully!

```

Figure 21: Valid INSERT INTO table query

SELECT ALL

1. Invalid SELECT ALL query

```

Select an option:
1
*****
Execute SQL Query
*****
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT * FROM ;
QueryParserException{errorMessage='Invalid SELECT ALL query!'}

```

Figure 22: Invalid SELECT ALL query

2. Executing SELECT ALL query without USE DATABASE query executed in prior

```

Select an option:
1
Enter SQL Query:
SELECT * FROM course;
QueryProcessorException{errorMessage='Error: No database set! | Execution Time: 0ms'}

```

Figure 23: Executing SELECT ALL query without USE DATABASE query executed in prior

3. SELECT ALL query on a table that does not exist

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT * FROM employee;
QueryProcessorException{errorMessage='Error: Table employee does not exists! | Execution Time: 1ms'}
1. Execute SQL Query.
2. Go Back.
Select an option:
|

```

Figure 24: SELECT ALL query on a table that does not exist

4. VALID SELECT ALL query

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
use database dal;
Database dal in use! | Execution Time: 2ms
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT * FROM course;
| id | name | description |
| 1 | Course 1 | Description 1 |
| 2 | Course 2 | Description 2 |
| 3 | Course 3 | Description 3 |
| 4 | Course 4 | Description 4 |

1. Execute SQL Query.
2. Go Back.
Select an option:

```

Figure 25: VALID SELECT ALL query

SELECT DISTINCT

1. Invalid SELECT DISTINCT query

```

1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT DISTINCT FROM ;
QueryParserException{errorMessage='Invalid SELECT DISTINCT query!'}

```

Figure 26: Invalid SELECT DISTINCT query

2. Executing SELECT DISTINCT query without USE DATABASE query executed in prior

```
*****
Execute SQL Query
*****
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT DISTINCT id FROM course;
QueryProcessorException{errorMessage='Error: No database set! | Execution Time: 0ms'}
```

Figure 27: Executing *SELECT DISTINCT* query without *USE DATABASE* query executed in prior

3. SELECT DISTINCT query on a table that does not exist

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT DISTINCT id FROM employee;
QueryProcessorException{errorMessage='Error: Table employee does not exists! | Execution Time: 0ms'}
1. Execute SQL Query.
2. Go Back.
Select an option:
1
```

Figure 28: *SELECT DISTINCT* query on a table that does not exist

4. VALID SELECT DISTINCT query

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
use database dal;
Database dal in use! | Execution Time: 3ms
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
SELECT DISTINCT id FROM course;
| id |
| 1 |
| 2 |
| 3 |
| 4 |
|
```

Figure 29: *VALID SELECT DISTINCT* query

TRUNCATE TABLE

1. Truncating table that does not exist

```
Select an option:
1
Enter SQL Query:
TRUNCATE TABLE employee;
QueryProcessorException{errorMessage='Table does not exists!'}
```

Figure 30: Truncating table that does not exist

2. Valid TRUNCATE TABLE query

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
TRUNCATE TABLE course;
Table course truncated successfully!
```

Figure 31: Valid TRUNCATE TABLE query

DROP TABLE

1. Drop table that does not exist

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
DROP TABLE employee;
QueryProcessorException{errorMessage='Error: Table employee does not exists! | Execution Time: 0ms'}
```

Figure 32: Drop table that does not exist

2. Valid DROP TABLE query

```
1. Execute SQL Query.
2. Go Back.
Select an option:
1
Enter SQL Query:
DROP TABLE course;
Table course dropped successfully! | Execution Time: 15ms
1. Execute SQL Query.
2. Go Back.
Select an option:
```

Figure 33: Valid DROP TABLE query

Viewing metadata

1. Viewing metadata of a table that does not exist

```
Select an option:
1
*****
View Meta Data
*****
1. View Meta Data.
2. Go Back.
Select an option:
1
Enter database name to generate Meta Data:
dal
Enter table name to generate Meta Data:
employee
MetadataException{errorMessage='Error: Invalid table path ./src/main/java/database/database_server/dal/employee.txt or database does not exists!'}
1. View Meta Data.
2. Go Back.
Select an option:
1
```

Figure 34: Viewing metadata of a table that does not exist

2. Viewing metadata of a valid table

```

Enter database name to generate Meta Data:
dal
Enter table name to generate Meta Data:
student
Database: dal
Table: student
Columns:
  Column 1: id (Datatype: INT | Key: PRIMARY KEY)
  Column 2: name (Datatype: TEXT)
Database: dal
Table: student
Columns:
  Column 1: id (Datatype: INT | Key: PRIMARY KEY)
  Column 2: name (Datatype: TEXT)
  Column 3: email (Datatype: TEXT)
  Column 4: courseID (Datatype: INT | Key: FOREIGN KEY | Reference Table: course | Reference Column: id)

1. View Meta Data.
2. Go Back.
Select an option:

```

Figure 35: Viewing metadata of a valid table

Limitations and Future work

- Limitations** – This system has some limitations. We have developed a relational database management system that executes simple SQL queries. We have not implemented complex SQL queries such as joins, nested queries, and subqueries. This system also does not support aggregate functions. **Delete from table** query is not implemented. Constraints such as not null, auto-increment are not provided. We have implemented the simple SQL select queries but the “WHERE” clause is not implemented to perform operations on specific records by providing the conditions.
- Future work** – In the future, the system will be more user-friendly and efficient providing some additional features such as performing operations on specific records of the table by providing the conditions using the “WHERE” clause, nested queries, subqueries, and joins. The system will also support aggregate functions, not null and auto-increment. The option of graphic ERD generation can also be considered.

However, we have additionally implemented the SQL dump import to replicate or create a database using the SQL dumps generated from an existing or historical database.

References

- [1] "DalFCS Git," [Online]. Available: <https://git.cs.dal.ca/dashah/csci-5408-s2021-dp17>. [Accessed 7 August 2021].
- [2] diagrams.net, "Diagram Software and Flowchart Maker," [Online]. Available: <https://app.diagrams.net/>. [Accessed 6 August 2021].