CSCI 5408
Data Management and Warehousing

Sprint Report - 1

Group members:

Kenee Ashok Patel (B00969805)

Vraj Sunilkumar Shah (B00979965)

Shivani Uppe (B00976573)

GitLab Project Link: https://git.cs.dal.ca/kenee/dbms-builder-11

Table of Contents

Table of Figures

1. Background Research

1.1. Introduction

This project is about building a small DBMS in Java that processes user queries and stores data in files. The goal is to understand how a DBMS works and to create a simple version that can handle basic SQL-like commands. Through this project, we aim to bridge the theoretical knowledge and practical application, gaining insights on data processing, inner working of DBMSs and challenges involved in managing large data consistently and without error. For sprint 1, we managed to complete 3 modules i.e. DB Design document, Query Implementation and Transaction Processing which is detailed further.

1.2. Core Concepts

The tiny DBMS is built on several key ideas:

1. **Data Storage**: Data is organized in tables, with rows representing individual records. We use text files to store this data, which is simpler to implement but has some limitations compared to more advanced systems [1]. The base idea is to create separate package (folder) inside "Databases" package for every database created and have "tableName.txt" file(s) under that package for every table created in that database. With this, there will also be a file "databases.txt" that stores names of all databases created so far. The purpose of this is to load databases and their tables (without any row data) when program starts to avoid unnecessary file reading checks when user tries to create new table or database which might conflict with currently created table/database. The file structure will look something like this:

   ```
   Databases (package)
   ---- database1 (package)
         ---- table1.txt
         ---- table2.txt
   ---- database2 (package)
   ---- databases.txt (contains names of all databases created)
   ```

2. **SQL (Structured Query Language)**: The DBMS supports basic SQL commands to create databases and tables, insert data, query data, update records, and manage transactions. We won't be 100% replicating all SQL commands and will have some limitations regarding syntax which we will define later in this document.
3. **Transaction Management**: To keep data consistent and correct, the DBMS includes features like commit, rollback, and auto-commit. We have incorporated transaction in this sprint which lets users to group multiple instructions in single transaction [2]. Base idea

behind transaction is to introduce buffer area between the user interactions and actual database. This buffer area is like buffer data that are on hold inside program to be added to the database or removed from buffer as per the instruction received from the user. The buffer data (contains all data currently present in table) is only populated when any table has any kind of query incoming which will basically load all data from file to a local variable in program ensuring ACID property as to having databases in consistent state irrespective of transaction commit, reverted or failed. Once the transaction is committed, all buffer data is written back to file. As this operation is overwriting existing data in the file, we have a known bug for multiple users performing parallel transaction i.e. two users start transaction, one user commits transaction with few updates, other user won't have its buffer data updated and when it commits, it overwrites its buffer data to table file causing to data lost of first user. Even though, for single user environment, transaction is working as expected and follows ACID property.

1.3. Key Functionalities

The tiny DBMS can do several important things:

1. **Creating Databases**: Users can create new databases using SQL commands. This will also create a folder in the persistent storage and its metadata in a .txt file.
2. **Using Database**: Users will be able to change the current database which is being used for data and tables.
3. **Creating Table**: Users can create table(s) under any database using create table SQL command. Currently, we have support for data types int, string and double. We will extend support of more data types such as date in upcoming sprints. All the constraints given to the table columns are recorded while creating any table and stored in table.txt file.
4. **Inserting Data**: Users can add new records to tables. The system checks to make sure the data matches the table's structure. User needs to provide all column name(s) while inputting data to successfully insert data into table.
5. **Selecting Data**: Users can run SELECT queries to get data from tables. It can filter results based on conditions specified in the query. We have given support for selecting all columns as well as specific columns for a table.
6. **Updating Data**: Users can modify existing records in a table based on specified conditions. The system allows updating one or multiple columns for the selected records. This operation is essential for maintaining data integrity and ensuring that the stored information is accurate and relevant. For example, users can update a user's email address or modify the quantity of items in an inventory.
7. **Deleting Data**: Users can delete existing records based on conditions. This operation is allowing users to delete obsolete, redundant, or incorrect data entries. The DBMS verifies

conditions before deletion to ensure data integrity, preventing accidental or unauthorized data loss.

8. **Drop table**: Users can delete an entire table from the database using the DROP TABLE SQL command. This action permanently removes the table structure, all associated data, and any constraints defined during its creation. Dropping a table is irreversible. This feature is useful for removing obsolete tables that are no longer needed in the database schema.
9. **Transaction Control**: This project includes commands to start transactions, commit changes, and roll back changes. This ensures that data modifications are applied correctly and can be undone if necessary. Users can use standard SQL command for transaction control.

2. Architecture Diagram

2.1. Design and Implementation

The tiny DBMS is written in Java and uses its object-oriented features to create a modular system. The main parts include:

1. **Query Processor**: This part reads and executes user queries. It uses a map of handlers to manage different types of queries.
2. **Query Handler**: Base Interface for all types of query handlers. A query handler is a class that handles specific operation that is performed on table/database, for example create table query handler, drop table query handler, etc.
3. **Database Manager**: This handles all the operations that relates to the management of different databases, are handled through Database Manager. I encompass creation of database, keeping track of all databases, and changing the database that is in use.
4. **File Manager**: This handles all file operations, such as creating, reading, and writing table data. It ensures data is stored safely and can be accessed quickly.
5. **Table, Column and Database Classes**: These classes represent the main entities in the system. They manage the structure and behaviour of databases and tables, including creating tables, inserting data, and querying records.
6. **Transaction Manager**: This part manages transactions, making sure changes are applied correctly and can be rolled back if needed.

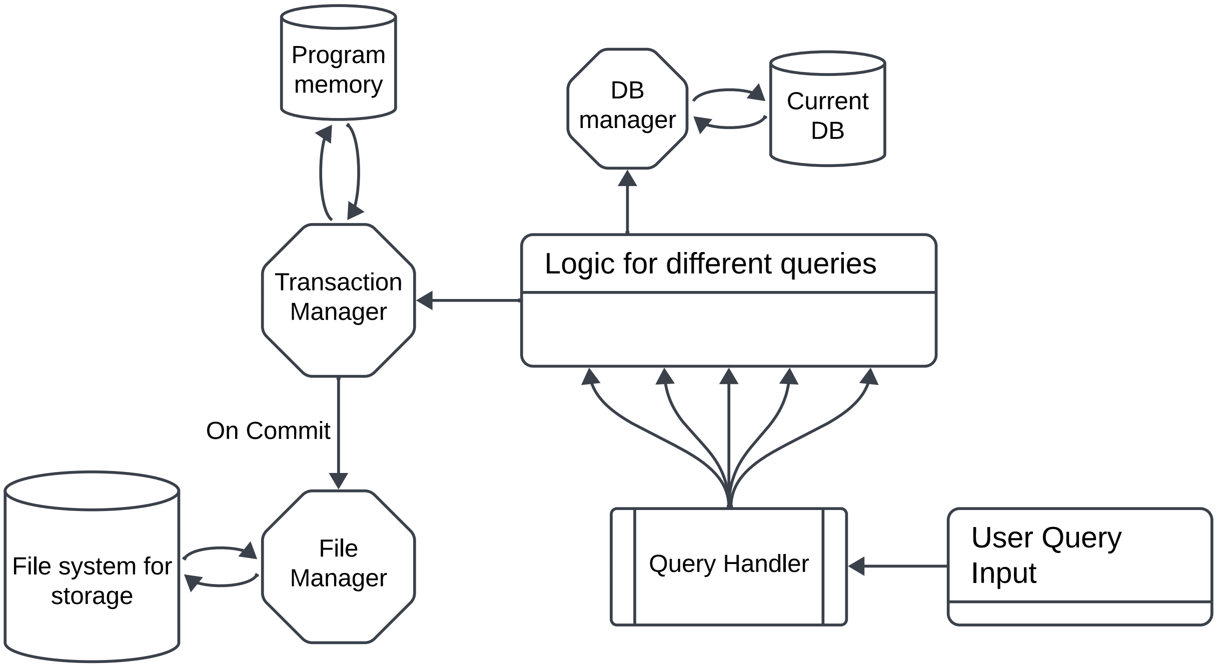


Figure 1: Architecture Diagram

3. Pseudocode

3.1. Create Database

```
function createDatabase(query):
    if query is not valid:
        throw "Invalid query"

    databaseName = extract database name from query
    if databaseName already exists:
        throw "Database already exists"

    update the databases with the new added database
    create directory with the name of database and save it databases.txt
    print "Database created: `databaseName`"
```

3.2. Use Database

```
function useDatabase(query):
    if query is not valid:
        throw "Invalid USE DATABASE query"

    databaseName = extract database name from query
    if databaseName does exist in the system:
        throw "Database does not exist"

    change the currently selected database to `databaseName` database
    print "Using database: `databaseName`"
```

3.3. Create table

```
function createTable(query):
    if query is not valid:
        throw "Invalid query"

    tableName = extract table name from query
    if tableName is invalid or already exists:
        throw "Table name is invalid, or table already exists"

    columnsData = extract column definitions from query
    columns = parse columnsData into column objects
    if columns are invalid:
        throw "Invalid column definition"

    create table object with tableName and columns
    save table structure to file
    print "Table created successfully"
```

3.4. Insert into table

```
function insertIntoTable(query):
    if query is not valid:
        throw "Invalid query"

    tableName = extract table name from query
    table = find table by tableName
    if table does not exist:
        throw "Table not found"
    columnsData = extract column names from query
    valuesData = extract values from query
    row = map columnsData to valuesData
    insert row into table

    if auto-commit is true:
        save row to file
    else:
        buffer row for transaction

    print "Row added successfully"
```

3.5. Select from table

```
function selectFromTable(query):
    if query is not valid:
        throw "Invalid query"

    columns = extract column names from query
    tableName = extract table name from query
    table = find table by tableName
    if table does not exist:
        throw "Table not found"

    condition = extract condition from query (if any)
    rows = get all rows from table
    if condition exists:
        filter rows based on condition

    if columns == "*":
        select all columns from rows
    else:
        select specified columns from rows

    print selected rows
```

3.6. Update table

```
function updateTable(query):
    if query is not valid:
        throw "Invalid query"
    tableName = extract table name from query
    table = find table by tableName
    if table does not exist:
        throw "Table not found"
    setClause = extract set clause from query
    condition = extract condition from query
    rows = get all rows from table
    affectedRows = 0
    for each row in rows:
    if condition is met for row:
            update row based on setClause
            affectedRows += 1
    if auto-commit is true:
        save updated rows to file
    else:
        buffer updated rows for transaction
    print affectedRows + " row(s) affected."
```

3.7. Delete from table

```
function deleteFromTable(query):
    if query is not valid:
        throw "Invalid query"
    tableName = extract table name from query
    table = find table by tableName
    if table does not exist:
        throw "Table not found"
    condition = extract condition from query
    rows = get all rows from table
      deletedRows = 0
    for each row in rows:
        if condition is met for row:
            delete row
            deletedRows += 1
    if auto-commit is true:
        save updated rows to file
    else:
        buffer updated rows for transaction
    print deletedRows + " row(s) deleted successfully."
```

3.8. Drop table

```
function dropTable(query):
    if query is not valid:
        throw "Invalid query"
    tableName = extract table name from query
    table = find table by tableName
    if table does not exist:
        throw "Table not found"
    delete table structure and data from file system
 remove table from database metadata
    print "Table dropped: " + tableName
```

3.9. Transaction

Start Transaction:

```
function startTransaction():
    for each database:
        for each table in database:
            table.saveBufferedDataToFile()
            table.clearBuffer()

    autoCommitStatusBeforeTransaction = auto-commit status
    set auto-commit to false
    set transaction in progress to true
    print "Transaction started"
```

Commit transaction:

```
function commitTransaction():
    if transaction in progress:
        for each database:
            for each table in database:
                table.saveBufferedDataToFile()


        set transaction in progress to false
        set auto-commit to autoCommitStatusBeforeTransaction
        print "Transaction committed"
```

Rollback transaction:

```
function rollbackTransaction():
    if transaction in progress:
        for each database:
            for each table in database:
                table.clearBuffer()


        set transaction in progress to false
        set auto-commit to autoCommitStatusBeforeTransaction
        print "Transaction rolled back"
```

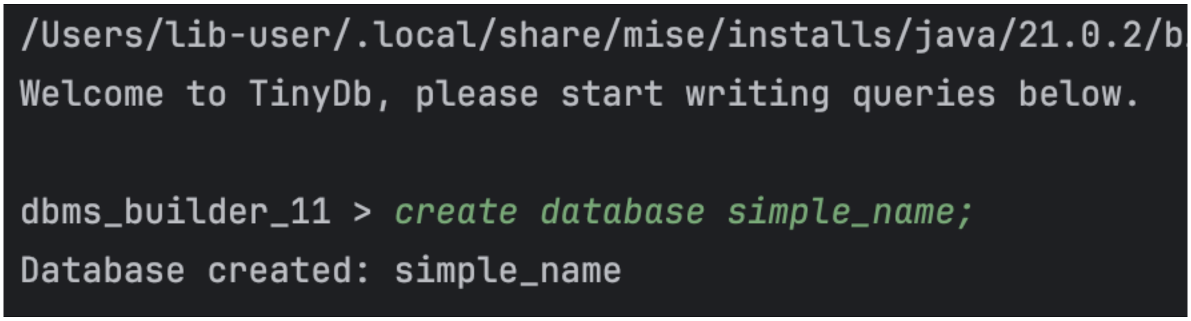
4. Git code repository link

Link: https://git.cs.dal.ca/kenee/dbms-builder-11

5. Test cases and evidence of testing

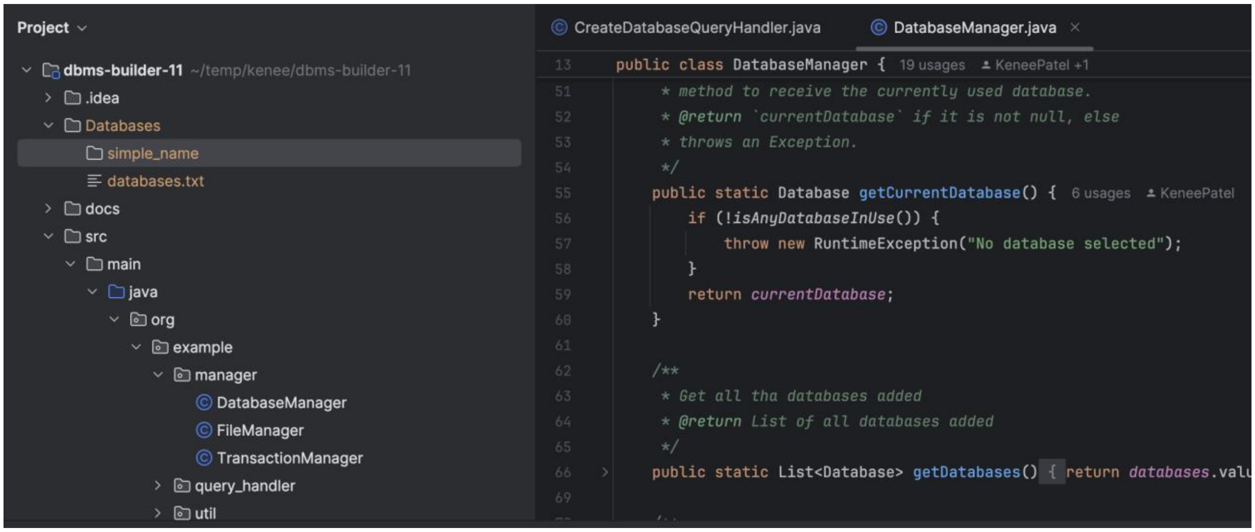
5.1. Create Database

Create a database



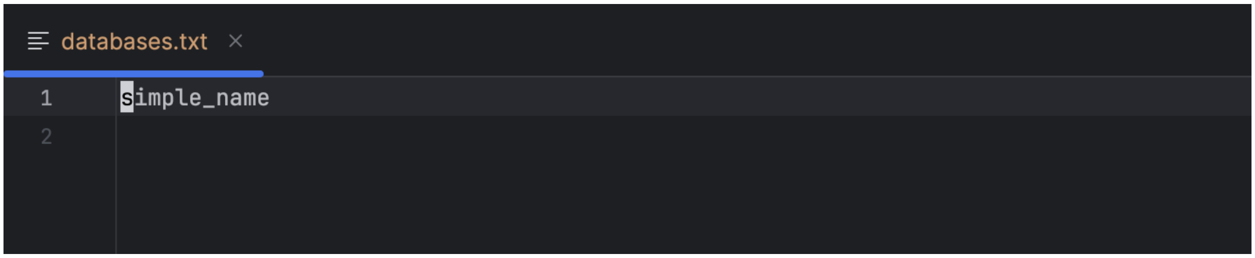
```
/Users/lib-user/.local/share/mise/installs/java/21.0.2/b
Welcome to TinyDb, please start writing queries below.

dbms_builder_11 > create database simple_name;
Database created: simple_name
```

Figure 2: CREATE TABLE demonstration by creating a table with name 'simple_name'



```
public class DatabaseManager {
     * method to receive the currently used database.
     * @return `currentDatabase` if it is not null, else
     * throws an Exception.
     */
    public static Database getCurrentDatabase() {
        if (!isAnyDatabaseInUse()) {
            throw new RuntimeException("No database selected");
        }
        return currentDatabase;
    }

    /**
     * Get all tha databases added
     * @return List of all databases added
     */
    public static List<Database> getDatabases() { return databases.valu
```

Figure 3: Folder creation succession after creating the database

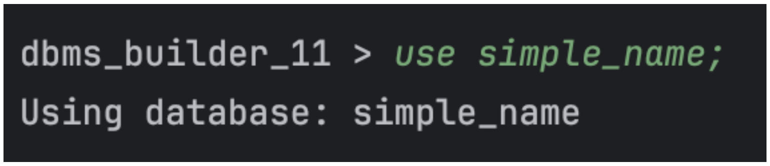


```
simple_name
```

Figure 4: Contents of databases.txt

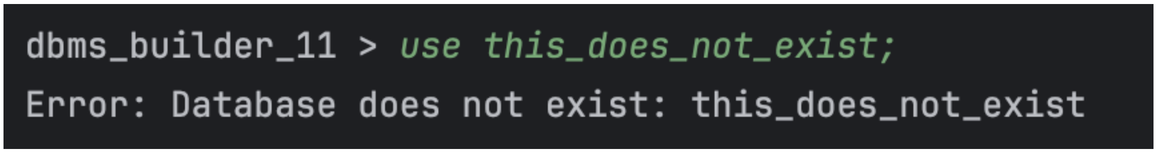
5.2. Use Database

Using a database which was created earlier with create database statement



```
dbms_builder_11 > use simple_name;
Using database: simple_name
```

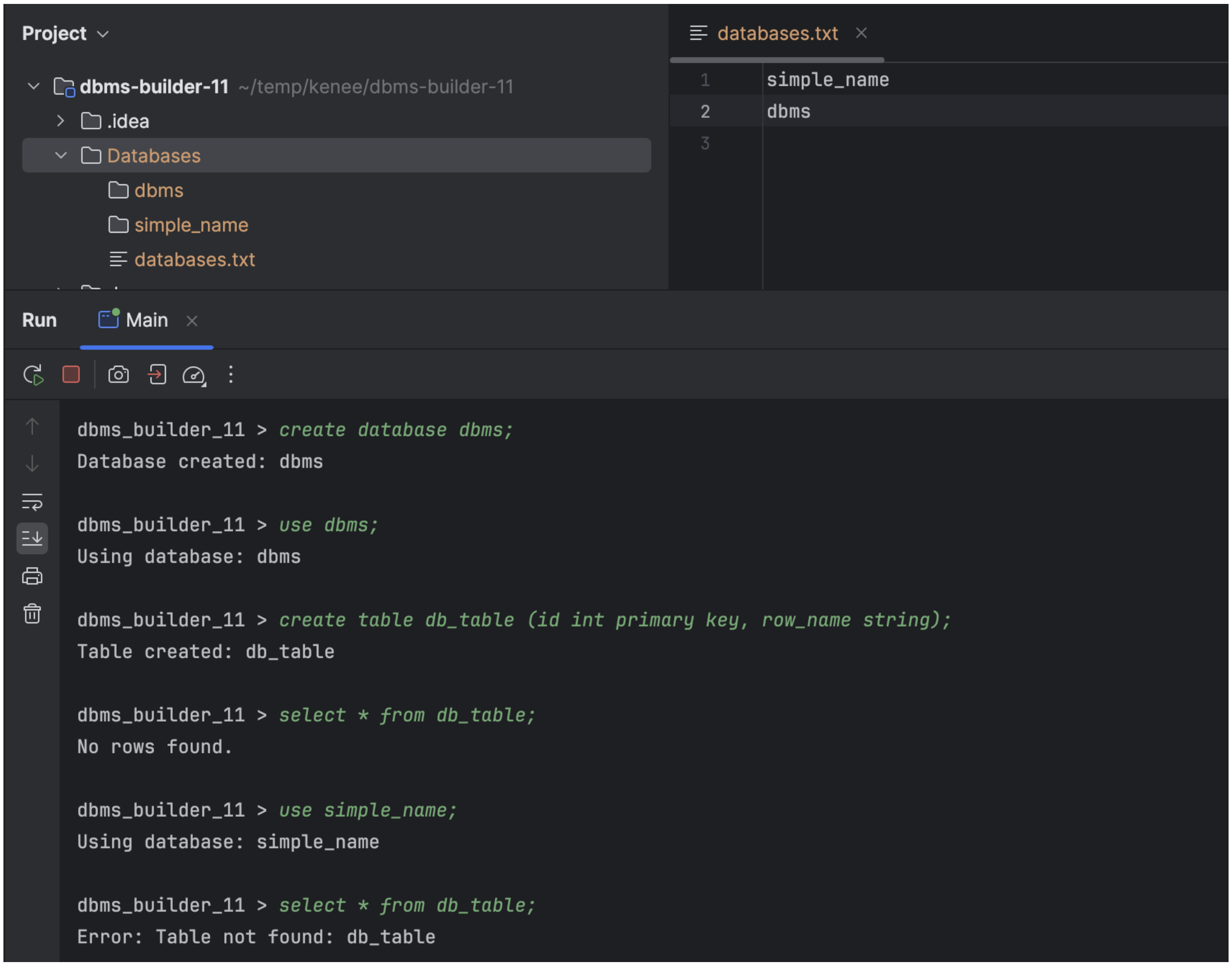
Figure 5: USE `DATABASE_NAME` demonstration by using simple_name database



```
dbms_builder_11 > use this_does_not_exist;
Error: Database does not exist: this_does_not_exist
```

Figure 6: Demonstration of USE `DATABASE_NAME` failing due to the database not existing

For the main functionality of using the correct database for table insertion, we will be creating a database named `dbms`, create a table named `db_table` in it and select * from it. Then we will try selecting everything from `db_table` but from `simple_name` database which should fail as there is no table named `db_table` in that.



databases.txt

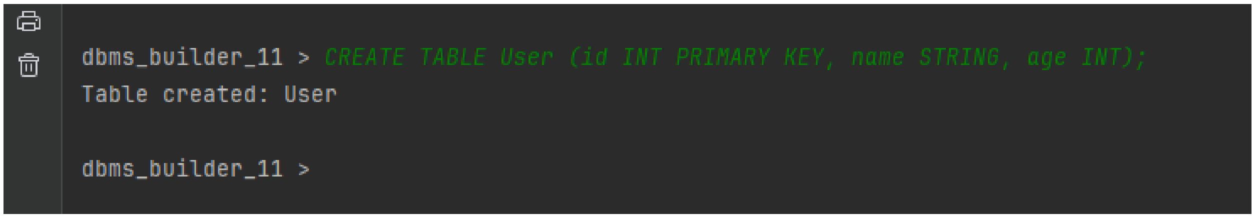
```
simple_name
dbms
```

```
dbms_builder_11 > create database dbms;
Database created: dbms

dbms_builder_11 > use dbms;
Using database: dbms

dbms_builder_11 > create table db_table (id int primary key, row_name string);
Table created: db_table

dbms_builder_11 > select * from db_table;
No rows found.

dbms_builder_11 > use simple_name;
Using database: simple_name

dbms_builder_11 > select * from db_table;
Error: Table not found: db_table
```

Figure 7: Demonstration of using database management

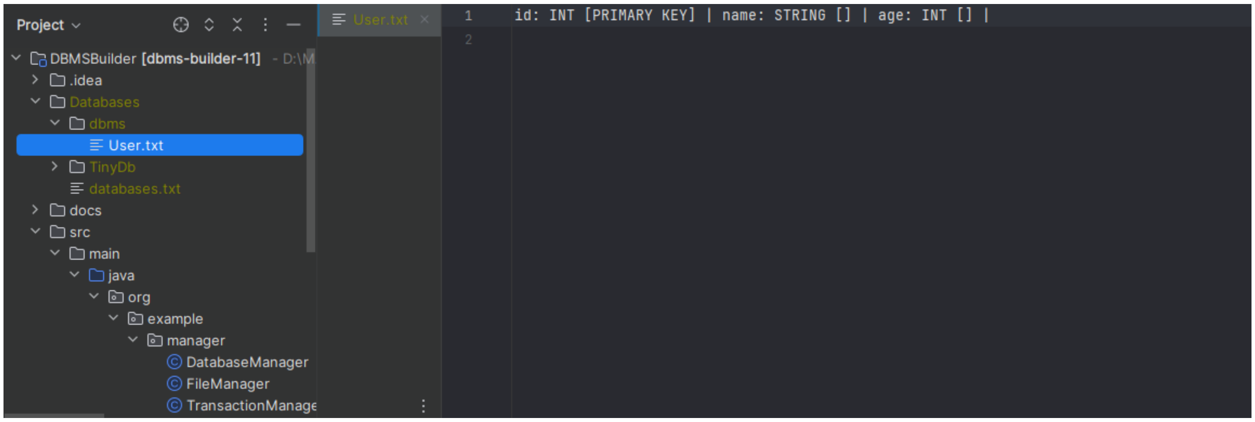
5.3. Create table

User table with 3 attributes i.e. id, name and age.



```
dbms_builder_11 > CREATE TABLE User (id INT PRIMARY KEY, name STRING, age INT);
Table created: User

dbms_builder_11 >
```

Figure 8: CREATE TABLE demonstration using User table

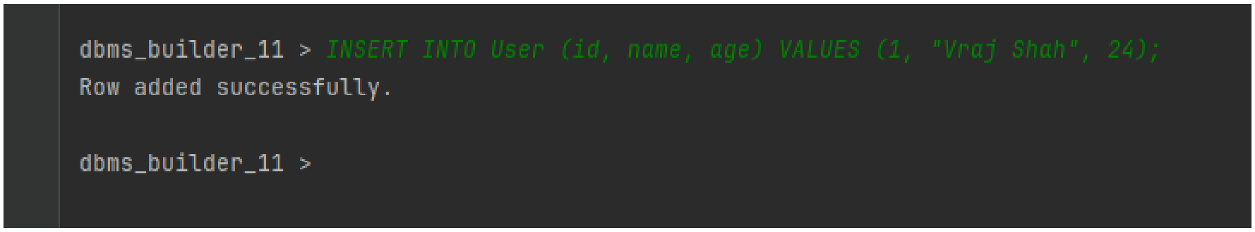


```
id: INT [PRIMARY KEY] | name: STRING [] | age: INT [] |
```

Figure 9: File creation succession after creating table

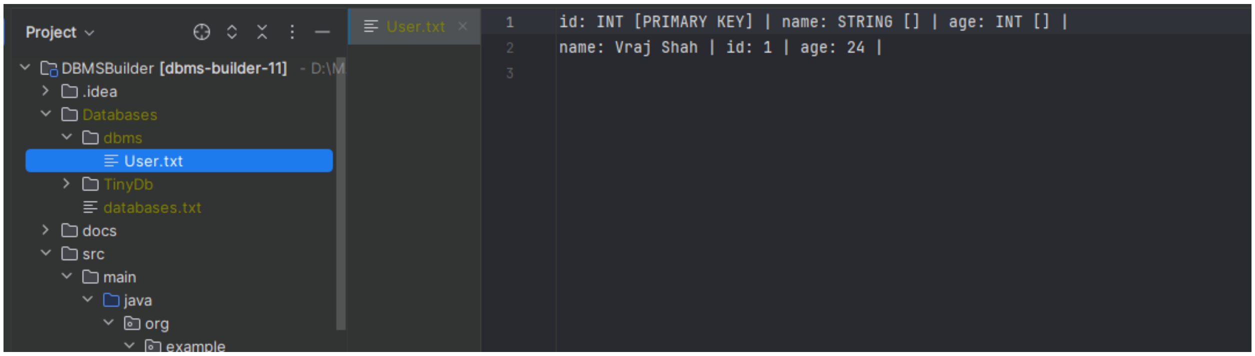
5.4. Insert into table

Insert a row into user table



```
dbms_builder_11 > INSERT INTO User (id, name, age) VALUES (1, "Vraj Shah", 24);
Row added successfully.

dbms_builder_11 >
```

Figure 10: INSERT INTO demonstration by inserting a row into User

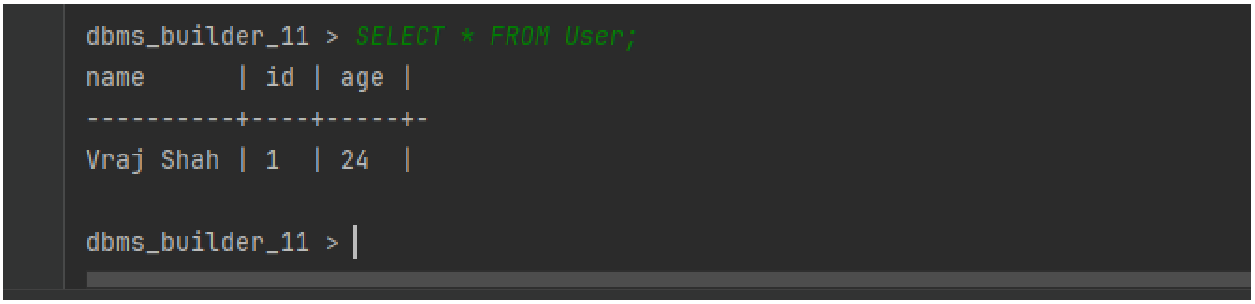


```
id: INT [PRIMARY KEY] | name: STRING [] | age: INT [] |
name: Vraj Shah | id: 1 | age: 24 |
```

Figure 11: File row addition successful after inserting data

5.5. Select from table

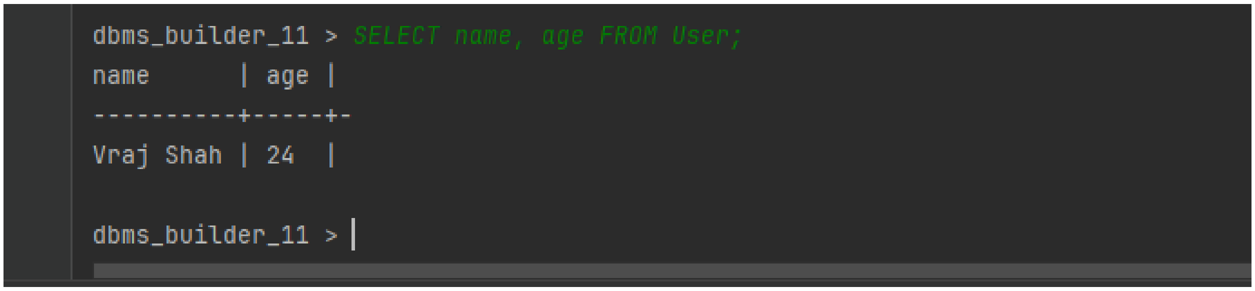
Select all columns from User.



```
dbms_builder_11 > SELECT * FROM User;
name       | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > 
```

Figure 12: Selecting all columns from User table

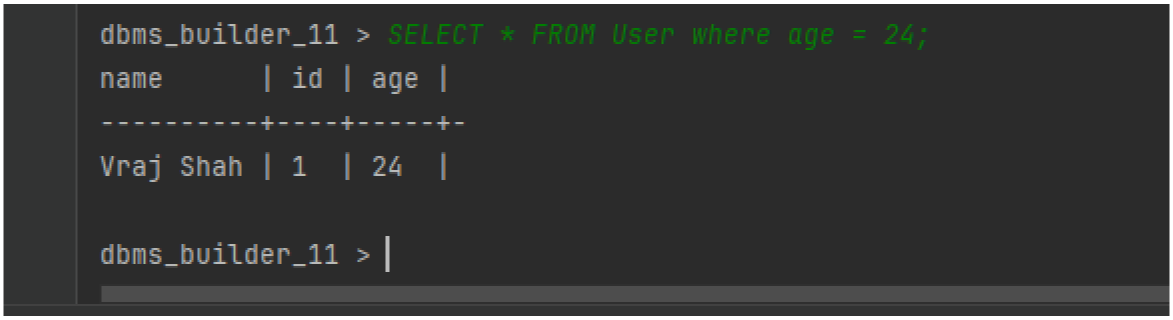
Selecting specific columns from User.



```
dbms_builder_11 > SELECT name, age FROM User;
name       | age |
----------+-----+-
Vraj Shah | 24  |

dbms_builder_11 > 
```

Figure 13: Select specific columns from User table

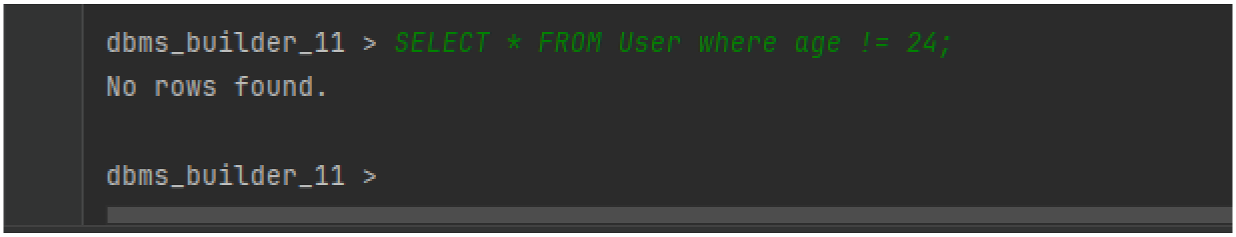
Selecting columns from User table where age = 24



```
dbms_builder_11 > SELECT * FROM User where age = 24;
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > 
```

Figure 14: Select from User where age is 24

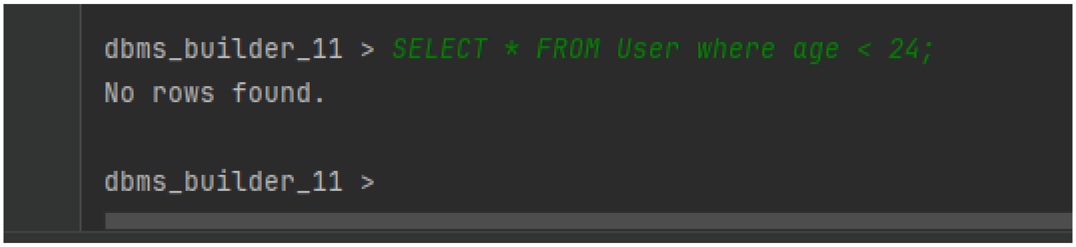
Selecting columns from User table where age != 24



```
dbms_builder_11 > SELECT * FROM User where age != 24;
No rows found.

dbms_builder_11 >
```

Figure 15: Select from User where age is not equal to 24

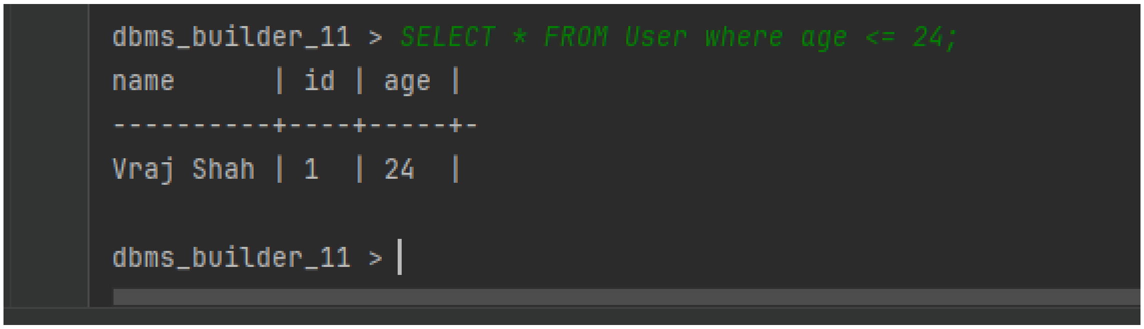
Selecting columns from User table where age < 24



```
dbms_builder_11 > SELECT * FROM User where age < 24;
No rows found.

dbms_builder_11 >
```

Figure 16: Select from User where age is less than 24

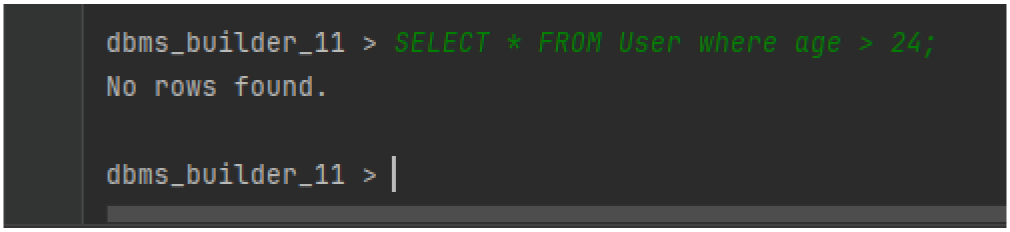
Selecting columns from User table where age <= 24



```
dbms_builder_11 > SELECT * FROM User where age <= 24;
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > 
```

Figure 17: Select from User where age is less than or equal to 24

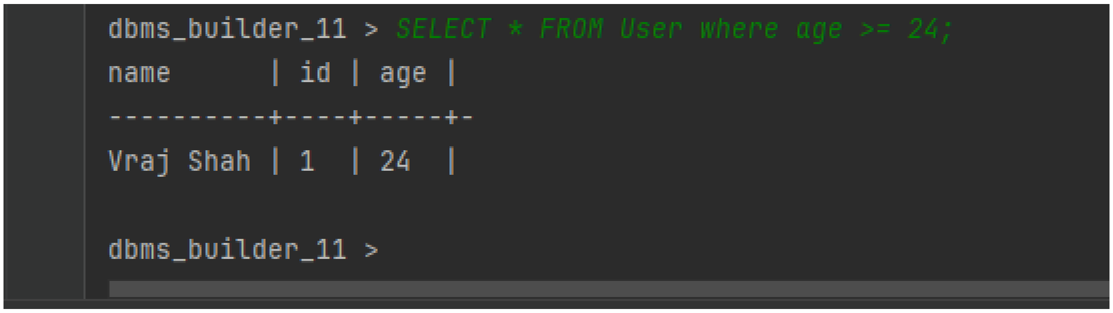
Selecting columns from User table where age > 24



```
dbms_builder_11 > SELECT * FROM User where age > 24;
No rows found.

dbms_builder_11 > 
```

Figure 18: Select from User where age is greater than 24

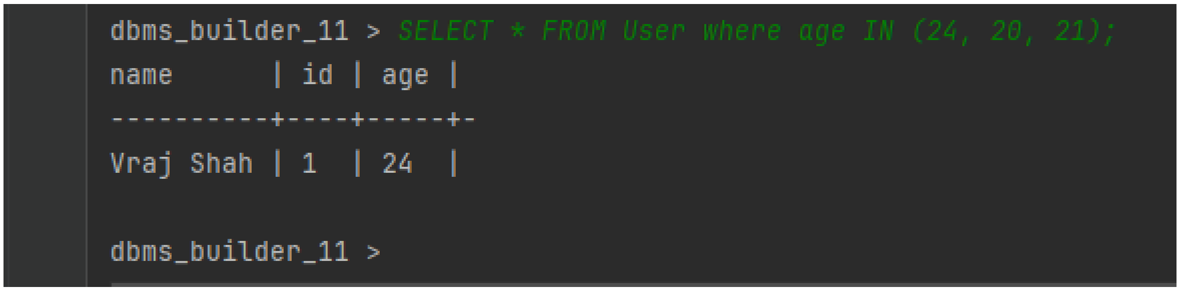
Selecting columns from User table where age >= 24



```
dbms_builder_11 > SELECT * FROM User where age >= 24;
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > 
```

Figure 19: Select from User where age is greater than or equal to 24

Selecting columns from User table where age IN 24, 20, 21

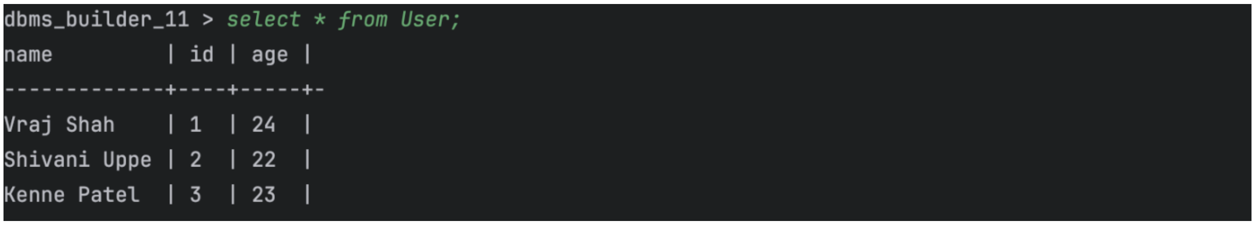


```
dbms_builder_11 > SELECT * FROM User where age IN (24, 20, 21);
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > 
```

Figure 20: Select from User where age is either 24 or 20 or 21

5.6. Update table

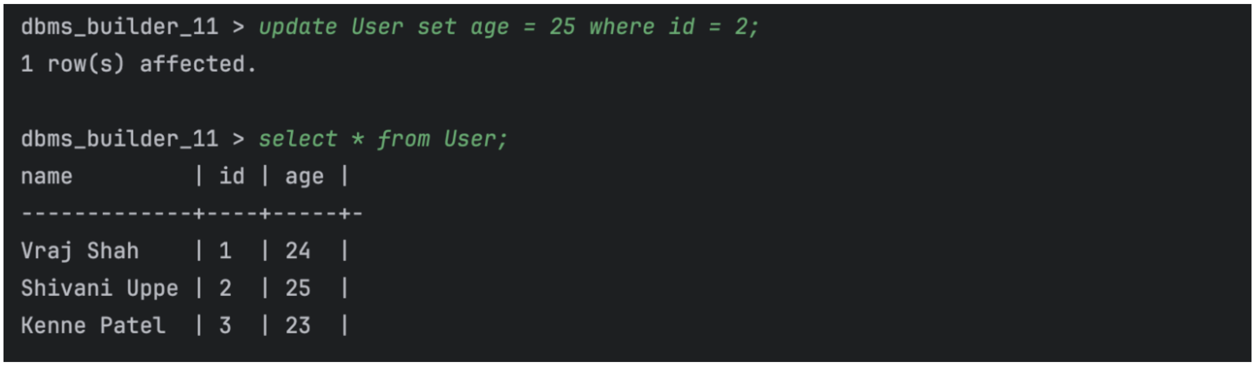
Table before updating



```
dbms_builder_11 > select * from User;
name         | id | age |
-------------+----+-----+-
Vraj Shah    | 1  | 24  |
Shivani Uppe | 2  | 22  |
Kenne Patel  | 3  | 23  |
```

Figure 21: Table before performing update

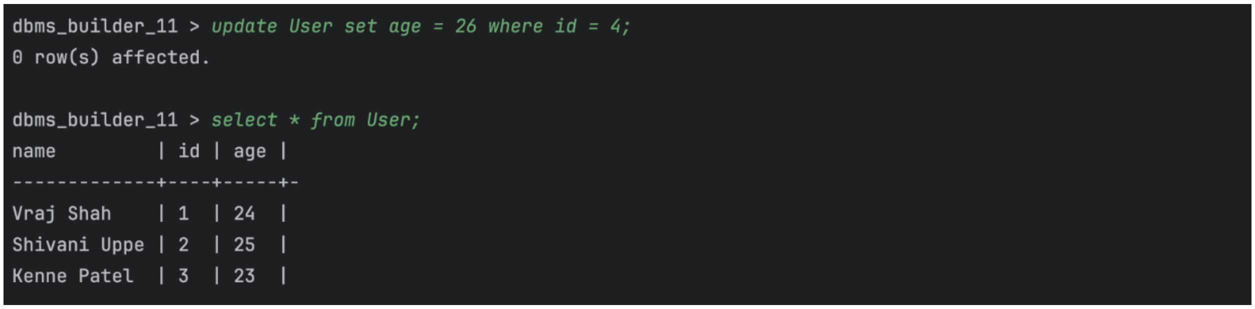
Updating User where id = 2



```
dbms_builder_11 > update User set age = 25 where id = 2;
1 row(s) affected.

dbms_builder_11 > select * from User;
name         | id | age |
-------------+----+-----+-
Vraj Shah    | 1  | 24  |
Shivani Uppe | 2  | 25  |
Kenne Patel  | 3  | 23  |
```

Figure 22: Updating age where id = 2

Updating User who does not exist

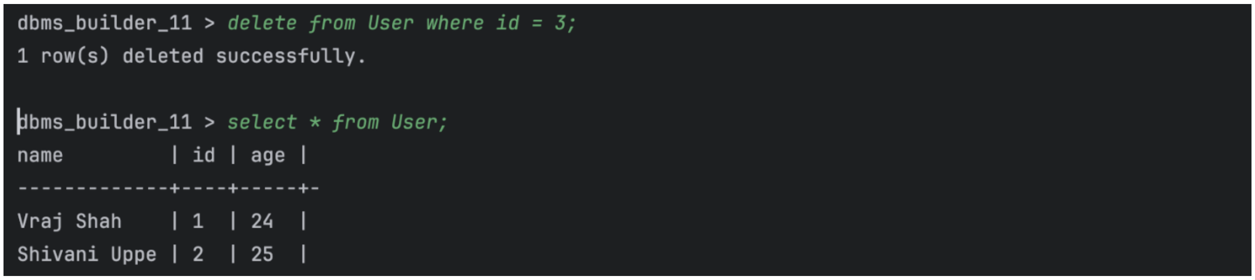


```
dbms_builder_11 > update User set age = 26 where id = 4;
0 row(s) affected.

dbms_builder_11 > select * from User;
name         | id | age |
-------------+----+-----+-
Vraj Shah    | 1  | 24  |
Shivani Uppe | 2  | 25  |
Kenne Patel  | 3  | 23  |
```

Figure 23: Updating age for an id which does not exist

5.7. Delete from table

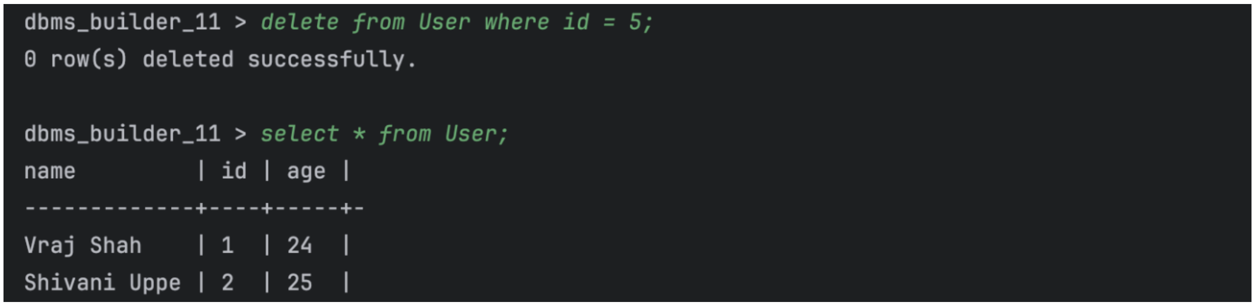
Deleting from User where id = 3



```
dbms_builder_11 > delete from User where id = 3;
1 row(s) deleted successfully.

dbms_builder_11 > select * from User;
name         | id | age |
-------------+----+-----+-
Vraj Shah    | 1  | 24  |
Shivani Uppe | 2  | 25  |
```

Figure 24: Delete from User where id = 3

Deleting from User where id does not exist

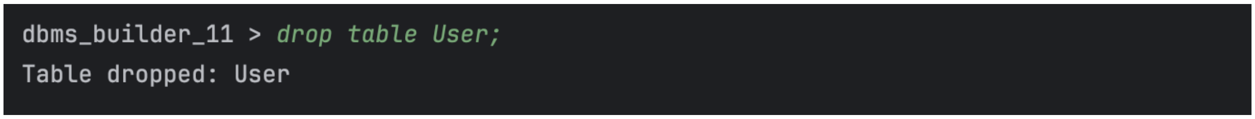


```
dbms_builder_11 > delete from User where id = 5;
0 row(s) deleted successfully.

dbms_builder_11 > select * from User;
name         | id | age |
-------------+----+-----+-
Vraj Shah    | 1  | 24  |
Shivani Uppe | 2  | 25  |
```

Figure 25: Deleting a User with an id which does not exist

5.8. Drop table

Dropping Table

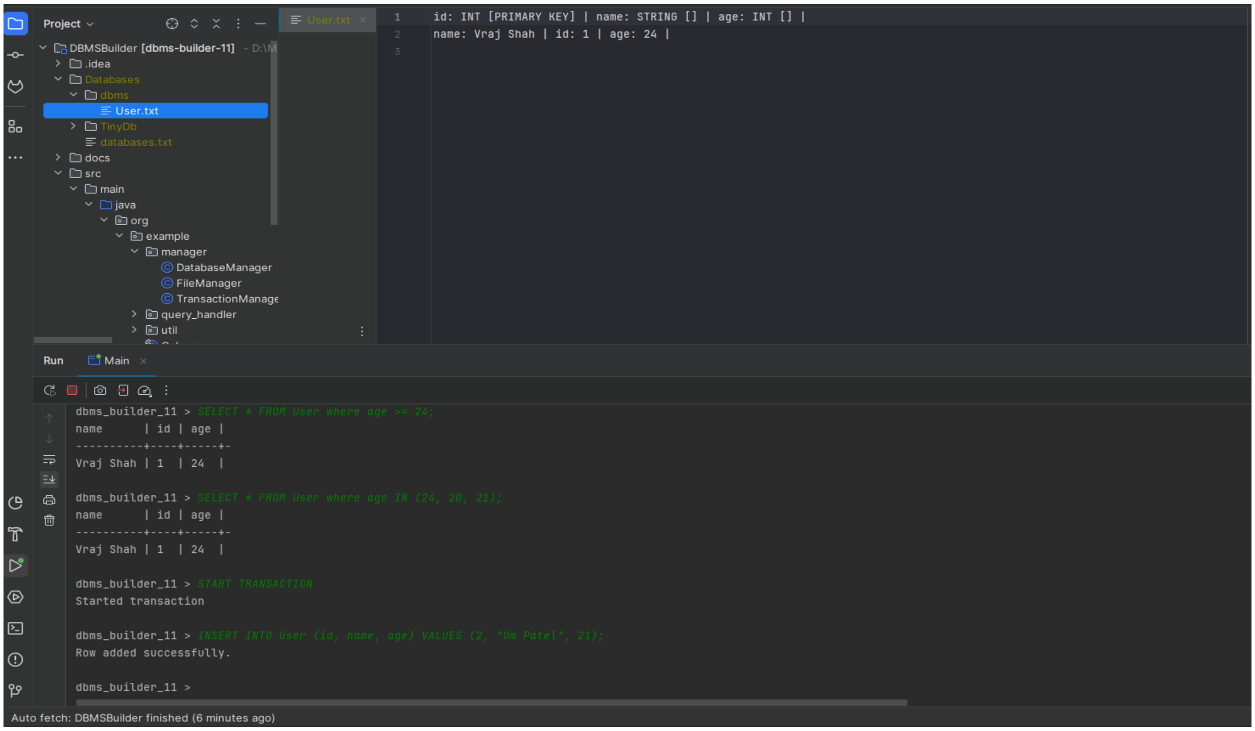


```
dbms_builder_11 > drop table User;
Table dropped: User
```

Figure 26: Dropping table

5.9.Transaction

Start transaction, insert row

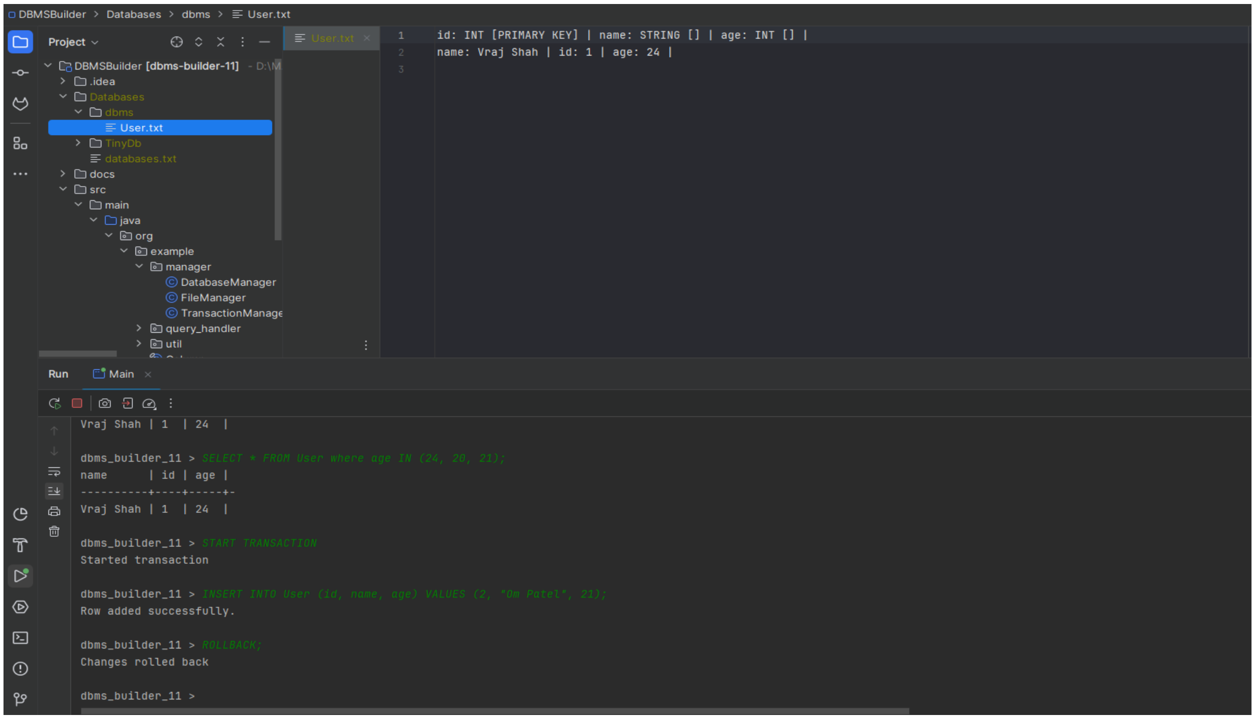


```
id: INT [PRIMARY KEY] | name: STRING [] | age: INT [] |
name: Vraj Shah | id: 1 | age: 24 |
```

```
dbms_builder_11 > SELECT * FROM User where age >= 24;
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > SELECT * FROM User where age IN (24, 20, 21);
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > START TRANSACTION
Started transaction

dbms_builder_11 > INSERT INTO User (id, name, age) VALUES (2, "Om Patel", 21);
Row added successfully.

dbms_builder_11 >
```

Figure 27: Starting transaction and inserting a row

Rollback changes

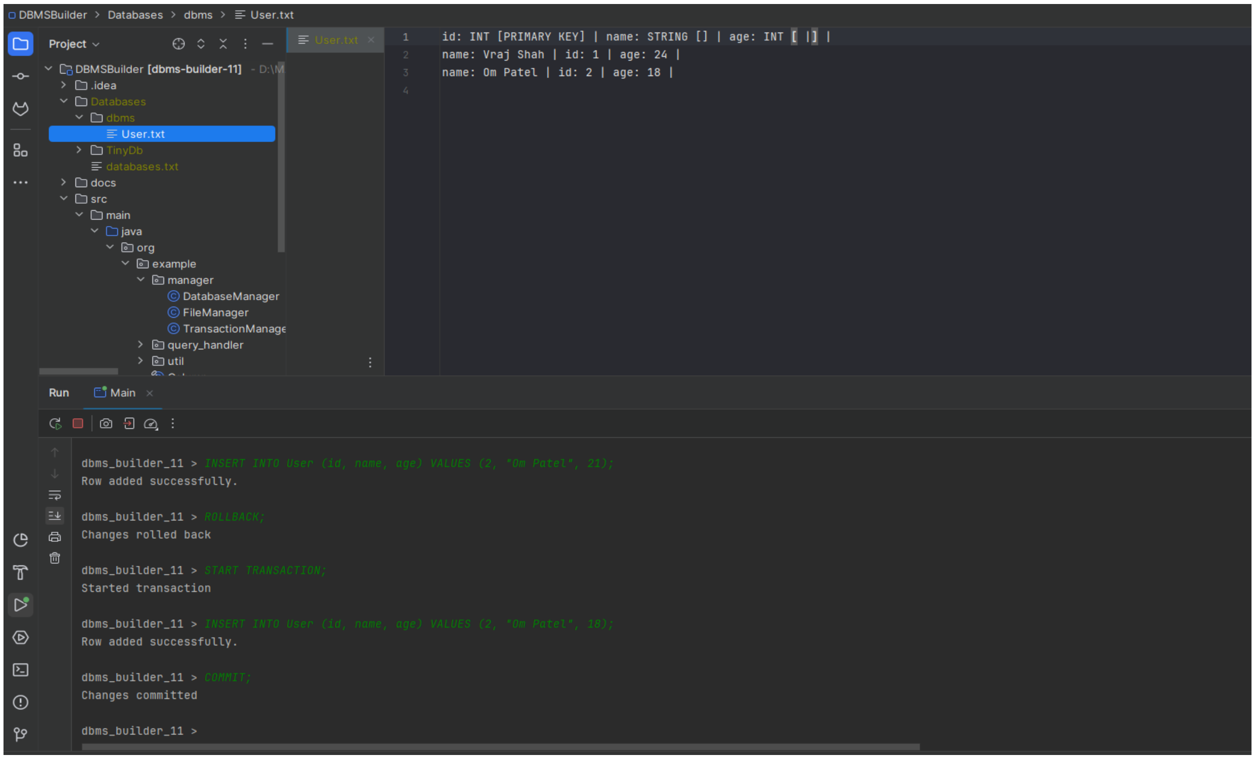


```
id: INT [PRIMARY KEY] | name: STRING [] | age: INT [] |
name: Vraj Shah | id: 1 | age: 24 |
```

```
Vraj Shah | 1  | 24  |

dbms_builder_11 > SELECT * FROM User where age IN (24, 20, 21);
name      | id | age |
----------+----+-----+-
Vraj Shah | 1  | 24  |

dbms_builder_11 > START TRANSACTION
Started transaction

dbms_builder_11 > INSERT INTO User (id, name, age) VALUES (2, "Om Patel", 21);
Row added successfully.

dbms_builder_11 > ROLLBACK;
Changes rolled back

dbms_builder_11 >
```

Figure 28: Rollback transaction and discard buffer data

Commit changes



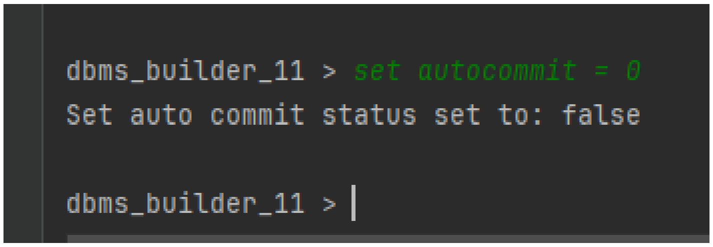
```
id: INT [PRIMARY KEY] | name: STRING [] | age: INT [] |
name: Vraj Shah | id: 1 | age: 24 |
name: Om Patel | id: 2 | age: 18 |
```

```
dbms_builder_11 > INSERT INTO User (id, name, age) VALUES (2, "Om Patel", 21);
Row added successfully.

dbms_builder_11 > ROLLBACK;
Changes rolled back

dbms_builder_11 > START TRANSACTION;
Started transaction

dbms_builder_11 > INSERT INTO User (id, name, age) VALUES (2, "Om Patel", 18);
Row added successfully.

dbms_builder_11 > COMMIT;
Changes committed

dbms_builder_11 >
```

Figure 29: Commit changes to User table

6. Novelty

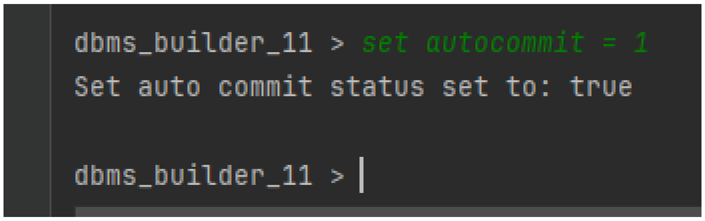
Apart from given requirements, we have also covered setting auto commit status. This allows user flexibility to commit changes instantly into file or to store in buffer before writing to file.



```
dbms_builder_11 > set autocommit = 0
Set auto commit status set to: false

dbms_builder_11 >
```

Figure 30: Setting auto commit to 0

After setting auto commit to 0, all the operations performed will be performed on local variable in program i.e. the buffer data. This data won't be written to file.



```
dbms_builder_11 > set autocommit = 1
Set auto commit status set to: true

dbms_builder_11 >
```

Figure 31: Setting auto commit to 1

After setting auto commit to 1, all the operations performed will be performed directly to the file instead of performing it on local variable in program i.e. buffer data.

With this introduction, there is a known issue in the program which lets user write data to file directly when in a transaction. For example, user starts a transaction, program automatically sets the auto commit to 0 and doesn't write directly to file. Now if user sets the auto commit to 1, the queries written after wards are directly written to file despite of being in a transaction. We can fix this issue in future work.

References

[1] "Are databases basically data stored in .txt files?," [Online]. Available: https://www.reddit.com/r/AskComputerScience/comments/k3608d/comment/ge1tr60/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button. [Accessed 28 June 2024].

[2] "Transaction Management - GeeksforGeeks," [Online]. Available: https://www.geeksforgeeks.org/transaction-management/. [Accessed 24 June 2024].