CSCI 5408
DATA MANAGEMENT AND WAREHOUSING

Tiny DB Sprint 2 Report

GitLab Assignment Link: https://git.cs.dal.ca/mkalathiya/tinydb

Submitted by:
Aditya Chaudhary (B00971587)
Rutvik Mansukhbhai Vaghani(B00978840)
Mansi Kalathiya(B00979173)

Table of Contents

Pseudocode for Transaction Manager

Function TransactionManager:

```
FILE_NAME = "transaction_buffer.txt"
isTransactionActive = False
function isTransactionActive():
    return isTransactionActive
function beginTransaction():
    if not isTransactionActive:
        isTransactionActive = True
        clearBufferFile()
        print("Transaction started.")
        logTransaction("BEGIN", "Transaction started.")
    else:
        print("Transaction already in progress.")
function execute(tokens, input):
    command = tokens[0].upper()
    if command == "SELECT":
        SelectCommand.execute(tokens, input)
    else if command in ["INSERT", "DELETE", "UPDATE"]:
        if validateCommand(tokens, input):
            appendToBufferFile(input)
            print("Operation added to transaction: " + input)
            logTransaction("EXECUTE", "Operation added to transaction: " + input)
        else:
            print("Error: Invalid " + command + " statement format: " + input)
    else:
        appendToBufferFile(input)
        print("Operation added to transaction: " + input)
        logTransaction("EXECUTE", "Operation added to transaction: " + input)

function executeTransactionCommand(command):
    if not isTransactionActive:
        raise Exception("No active transaction.")
    if command.upper() == "COMMIT":
        commitTransaction()
    else if command.upper() == "ROLLBACK":
        rollbackTransaction()
    else:
        raise Exception("Invalid transaction command: " + command)

function commitTransaction():
    try:
        operations = readBufferFile()
        for operation in operations:
            executeOperation(operation)
        isTransactionActive = False
        clearBufferFile()
        print("Transaction committed.")
        logTransaction("COMMIT", "Transaction committed.")
    except Exception as e:
        rollbackTransaction()
        raise Exception("Transaction failed, rolled back. Error: " + e.message)

function executeOperation(operation):
```

```
tokens = operation.split("\\s+")
command = tokens[0].upper()
switch command:
case "CREATE":
CreateCommand.execute(tokens, operation)
break
case "INSERT":
InsertCommand.execute(tokens, operation)
break
case "UPDATE":
UpdateCommand.execute(operation)
break
case "DELETE":
DeleteCommand.execute(tokens)
break
case "DROP":
DropCommand.execute(tokens)
break
default:
raise Exception("Invalid operation in transaction: " + operation)

function rollbackTransaction():
isTransactionActive = False
clearBufferFile()
print("Transaction rolled back.")
logTransaction("ROLLBACK", "Transaction rolled back.")

function validateCommand(tokens, input):
```

```
command = tokens[0].upper()
switch command:
    case "INSERT":
        return input.matches("INSERT INTO [a-zA-Z0-9_]+\\s*\\([a-zA-Z0-9_,\\s]*\\)\\s*VALUES\\s*\\([a-zA-Z0-9_,\\s\"']*\\);?")
    case "DELETE":
        return input.matches("DELETE FROM [a-zA-Z0-9_]+\\s*WHERE\\s*[a-zA-Z0-9_\\s=]*;?")
    case "UPDATE":
        return input.matches("UPDATE [a-zA-Z0-9_]+\\s*SET\\s*[a-zA-Z0-9_\\s=,\"]+\\s*WHERE\\s*[a-zA-Z0-9_\\s=]*;?")
    default:
        return True


function appendToBufferFile(operation):
    try:
        with open(FILE_NAME, 'a') as file:
            file.write(operation + '\n')
    except IOException as e:
        print("Error writing to buffer file: " + e.message)


function readBufferFile():
    operations = []
    try:
        with open(FILE_NAME, 'r') as file:
            for line in file:
                operations.append(line.strip())
    except IOException as e:
        print("Error reading from buffer file: " + e.message)
    return operations
```

```
function clearBufferFile():
try:
with open(FILE_NAME, 'w') as file:
file.close()
except IOException as e:
print("Error clearing buffer file: " + e.message)
```

Pseudocode for TinyDBExporter

Function TinyDBExporter:

```
function getColumnNamesAndTypes(metaFilePath):
metaPath = Paths.get(metaFilePath)
tempMetaPath = Paths.get(metaFilePath.replace("_meta.txt", "_temp_meta.txt"))

if not Files.exists(metaPath):
if Files.exists(tempMetaPath):
metaPath = tempMetaPath
else:
raise IOException("Metadata file not found: " + metaFilePath)

columnDetails = LinkedHashMap()
lines = Files.readAllLines(metaPath)
if lines.size() > 1:
columns = lines.get(1).split(":")[1].split(",")
for column in columns:
parts = column.trim().split(" ")
columnDetails.put(parts[0], parts[1])
return columnDetails

function getTableData(tableFilePath):
data = []
try (BufferedReader reader = new BufferedReader(new FileReader(tableFilePath))):
line = reader.readLine()
while line is not None:
row = line.split(",")
```

```
data.add(row)
line = reader.readLine()
return data

function exportToSQL(databaseName):
basePath = "tinydb/databases"
exportFilePath = "export.sql"

baseDir = new File(basePath)
if not baseDir.exists() or not baseDir.isDirectory():
raise IllegalArgumentException("Invalid base path: " + basePath)

try (BufferedWriter writer = new BufferedWriter(new FileWriter(exportFilePath))):
databaseDir = new File(baseDir, databaseName.toUpperCase())
if not databaseDir.exists() or not databaseDir.isDirectory():
raise IllegalArgumentException("Database directory not found: " + databaseDir.getPath())

tableFiles = databaseDir.listFiles((dir, name) => name.endsWith(".txt") and not
name.endsWith("_meta.txt"))
if tableFiles is null:
raise IllegalArgumentException("No table files found in database directory: " +
databaseDir.getPath())

for tableFile in tableFiles:
tableName = tableFile.getName().replace(".txt", "")
metaFilePath = tableFile.getPath().replace(".txt", "_meta.txt")

print("Processing table: " + tableName)
print("Meta file path: " + metaFilePath)
```

```
try:
columnDetails = getColumnNamesAndTypes(metaFilePath)
tableData = getTableData(tableFile.getPath())

writer.write("CREATE TABLE " + tableName + " (")
firstColumn = True
for entry in columnDetails.entrySet():
if not firstColumn:
writer.write(", ")
writer.write(entry.getKey() + " " + entry.getValue())
firstColumn = False
writer.write(");\n")

for row in tableData:
writer.write("INSERT INTO " + tableName + " VALUES (")
firstValue = True
for value in row:
if not firstValue:
writer.write(", ")
writer.write("'" + value.trim() + "'")
firstValue = False
writer.write(");\n")

writer.write("\n")
except IOException as e:
print("Error processing table " + tableName + ": " + e.getMessage())
print("Data exported successfully to " + exportFilePath)
catch IOException as e:
print("Error exporting data: " + e.getMessage())
```

Pseudocode for Logs

Function LogManager:

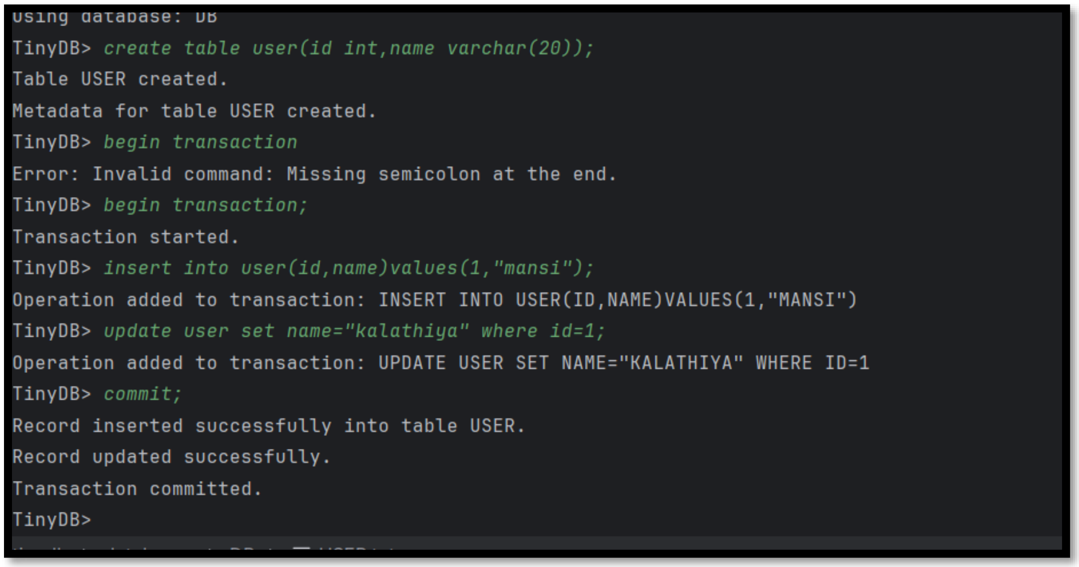
```
function logGeneral(message, state):
writeLog("general_log.json", createLogEntry(message, state))

function logEvent(eventType, description):
writeLog("event_log.json", createLogEntry(eventType, description))

function logQuery(query):
writeLog("query_log.json", createLogEntry("Query", query))

function logTransaction(transactionType, transactionDetail):
writeLog("transaction_log.json", createLogEntry(transactionType, transactionDetail))

function logUserActivity(activityType, username):
writeLog("user_activity_log.json", createLogEntry(activityType, username))

function writeLog(logFileName, logEntry):
logFilePath = "logs/" + logFileName
createLogDirectoryIfNotExists("logs")

try (FileWriter fileWriter = new FileWriter(logFilePath, true)):
fileWriter.write(logEntry + "\n")
catch IOException as e:
print("Error writing log: " + e.getMessage())

function createLogDirectoryIfNotExists(logDirectoryPath):
```

```
logDirectory = new File(logDirectoryPath)
if not logDirectory.exists():
logDirectory.mkdirs()


function createLogEntry(key, value):
logEntry = HashMap()
logEntry.put("timestamp", LocalDateTime.now().toString())
logEntry.put(key, value)


return toJson(logEntry)


function toJson(map):
jsonBuilder = StringBuilder("{")
for entry in map.entrySet():
jsonBuilder.append("\"").append(entry.getKey()).append("\":
\"").append(entry.getValue()).append("\", ")
jsonBuilder.delete(jsonBuilder.length() - 2, jsonBuilder.length()) // Remove trailing comma and
space
jsonBuilder.append("}")
return jsonBuilder.toString()
```

Testing Evidence

Testing of the Transaction :

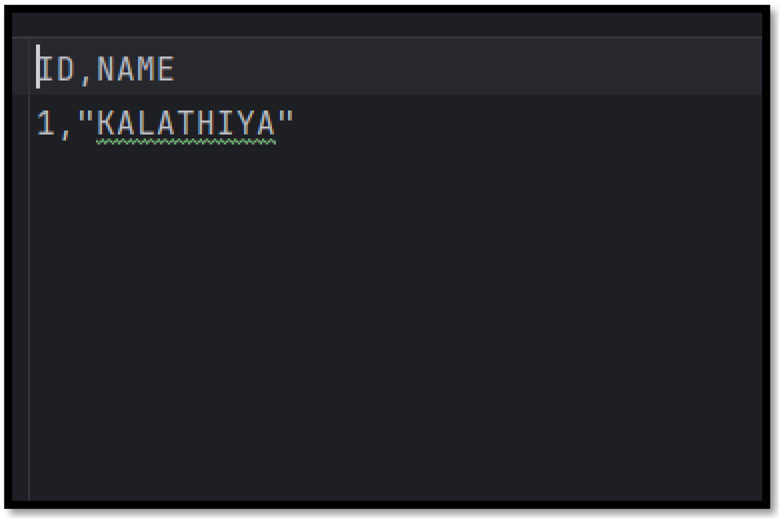
- After begin transaction it store all the command in transaction buffer file and it also validate the command that we are give is right or wrong and after commit it clear the transaction buffer file and exceute all commnad in user.txt file which is our table name.



```
using database: DB
TinyDB> create table user(id int,name varchar(20));
Table USER created.
Metadata for table USER created.
TinyDB> begin transaction
Error: Invalid command: Missing semicolon at the end.
TinyDB> begin transaction;
Transaction started.
TinyDB> insert into user(id,name)values(1,"mansi");
Operation added to transaction: INSERT INTO USER(ID,NAME)VALUES(1,"MANSI")
TinyDB> update user set name="kalathiya" where id=1;
Operation added to transaction: UPDATE USER SET NAME="KALATHIYA" WHERE ID=1
TinyDB> commit;
Record inserted successfully into table USER.
Record updated successfully.
Transaction committed.
TinyDB>
```

Figure 1 : Begin Transaction

- After commit all the command execute properly in file as you can see.



```
ID,NAME
1,"KALATHIYA"
```

Figure 2 : Commit done successfully

- After commit the transaction buffer file got clear.

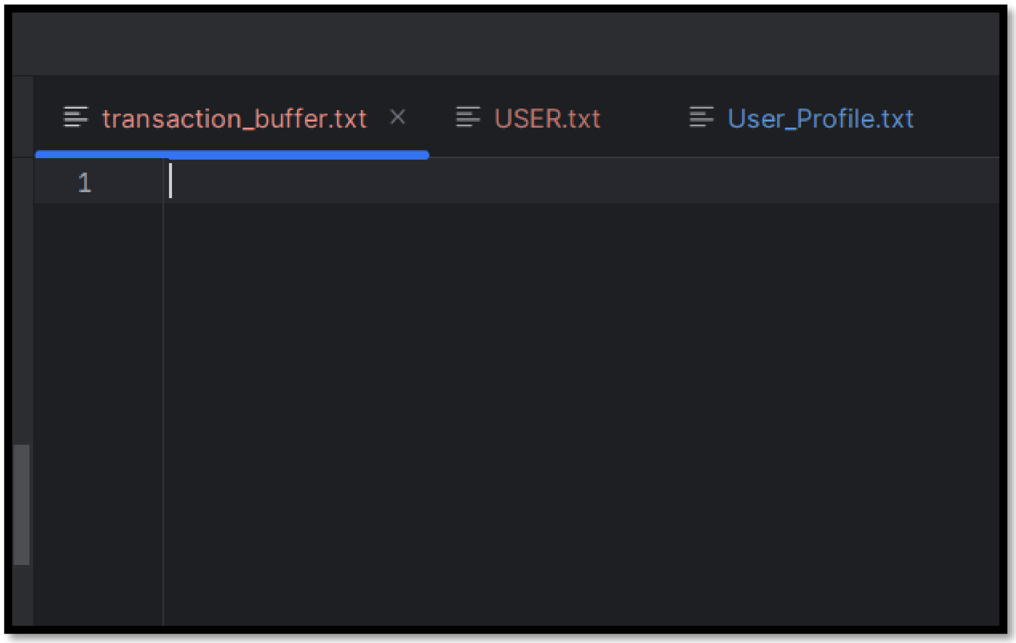
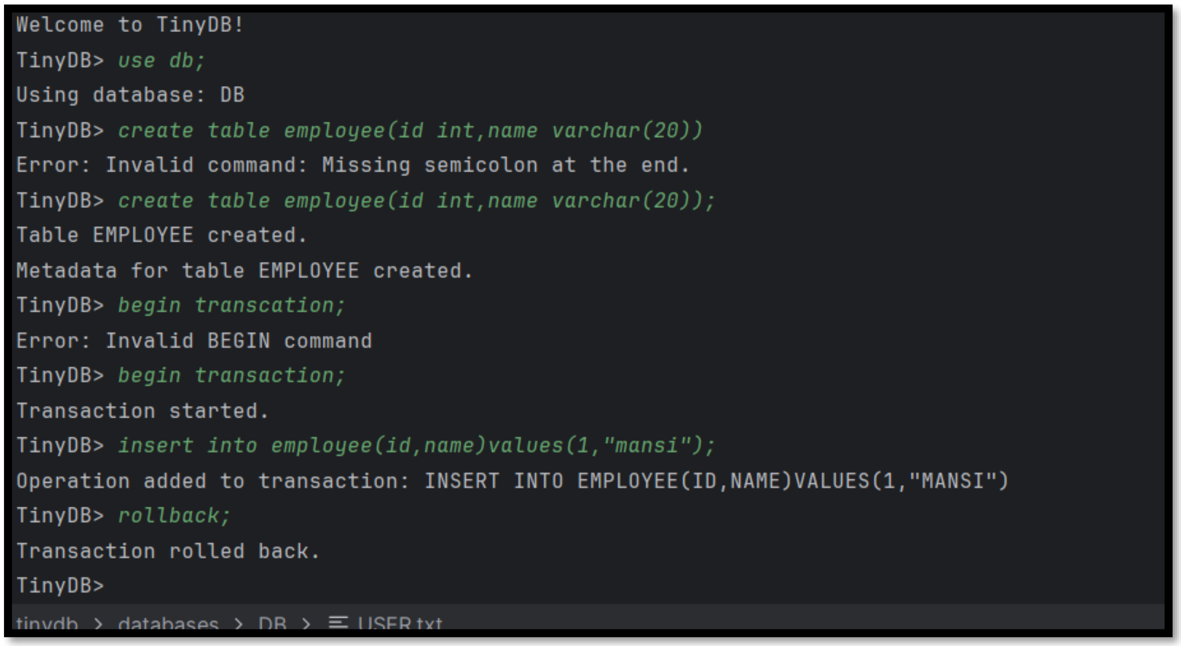


Figure 3 : Transaction buffer file clear successfully

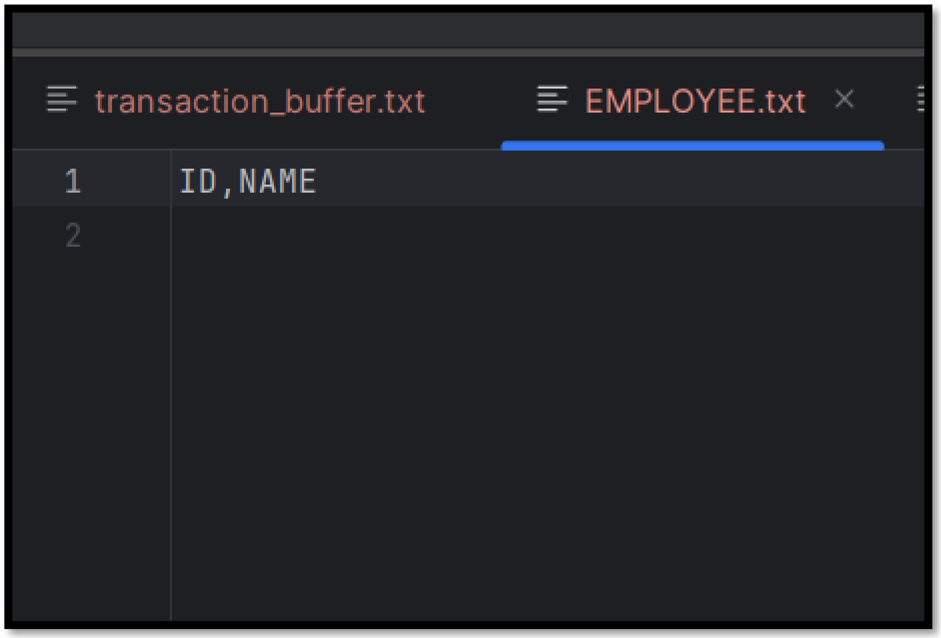
- After begin transaction it store all the command in transaction buffer file and it also validate the command that we are give is right or wrong and after rollback it clear the transaction buffer file and there is no change table file because of rollback.



```
Welcome to TinyDB!
TinyDB> use db;
Using database: DB
TinyDB> create table employee(id int,name varchar(20))
Error: Invalid command: Missing semicolon at the end.
TinyDB> create table employee(id int,name varchar(20));
Table EMPLOYEE created.
Metadata for table EMPLOYEE created.
TinyDB> begin transcation;
Error: Invalid BEGIN command
TinyDB> begin transaction;
Transaction started.
TinyDB> insert into employee(id,name)values(1,"mansi");
Operation added to transaction: INSERT INTO EMPLOYEE(ID,NAME)VALUES(1,"MANSI")
TinyDB> rollback;
Transaction rolled back.
TinyDB>
```

Figure 4 : Begin Transaction

- After rollback it clear the buffer file and there is no change employee.txt file



```
ID,NAME
```

Figure 5 : Rollback done successfully

- After rollback transaction buffer file is clear

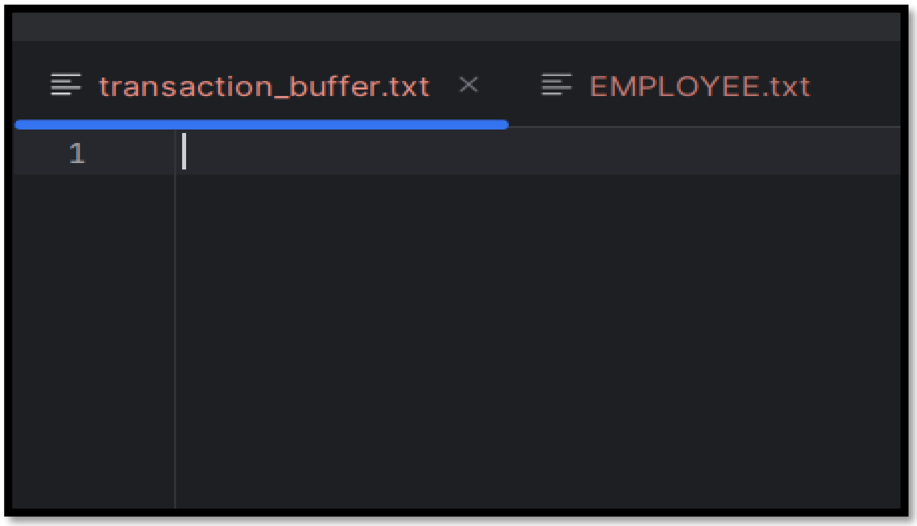
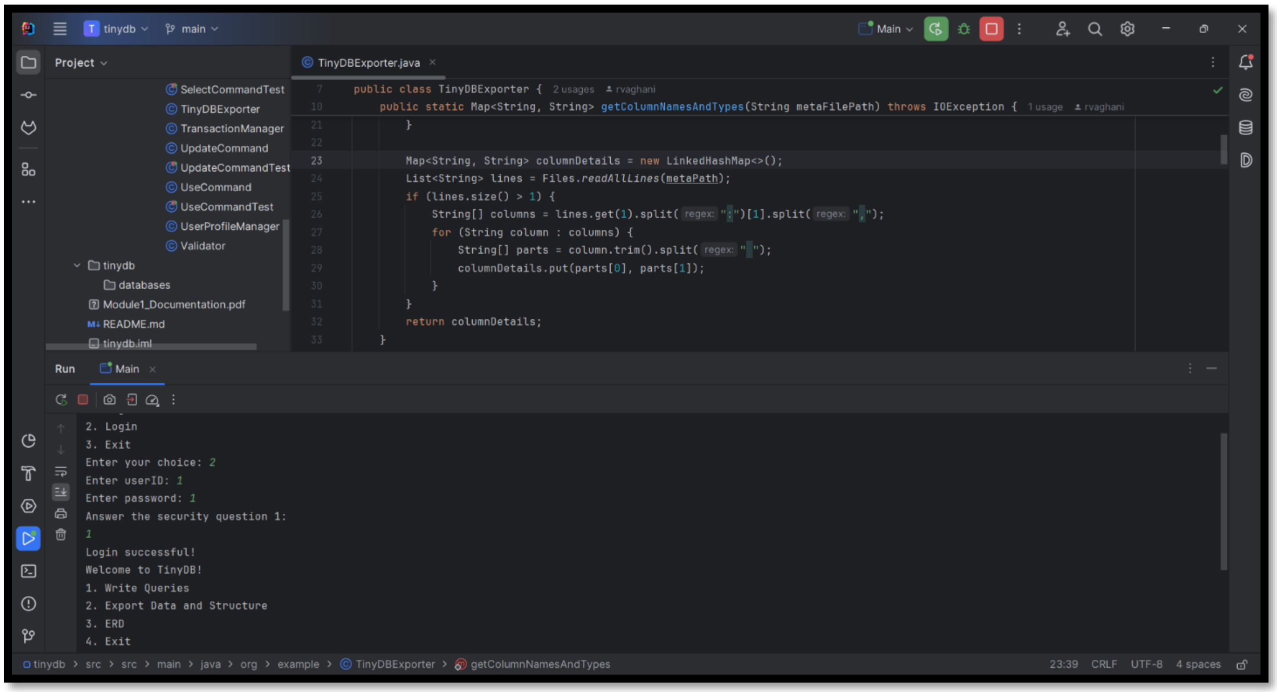


Figure 6 : Transaction buffer file clear successfully

Testing of the Export of the Data and Structures:

- After login user get the 4 options like query, export, ER diagram, and exit.

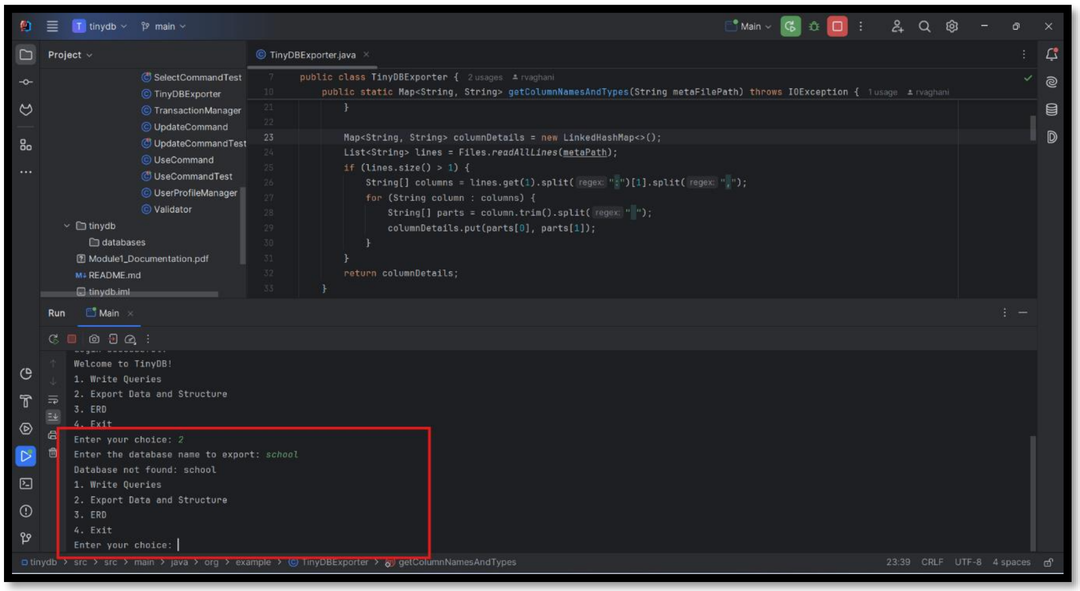


```java
public class TinyDBExporter {
    public static Map<String, String> getColumnNamesAndTypes(String metaFilePath) throws IOException {
        }

        Map<String, String> columnDetails = new LinkedHashMap<>();
        List<String> lines = Files.readAllLines(metaPath);
        if (lines.size() > 1) {
            String[] columns = lines.get(1).split(":")[1].split(",");
            for (String column : columns) {
                String[] parts = column.trim().split(" ");
                columnDetails.put(parts[0], parts[1]);
            }
        }
        return columnDetails;
    }
```

```
2. Login
3. Exit
Enter your choice: 2
Enter userID: 1
Enter password: 1
Answer the security question 1:
1
Login successful!
Welcome to TinyDB!
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
```

Figure 7 : list of opinions after login in tinydb

- Check that the if user want to export the database which is not created till now then user not able to export the database.

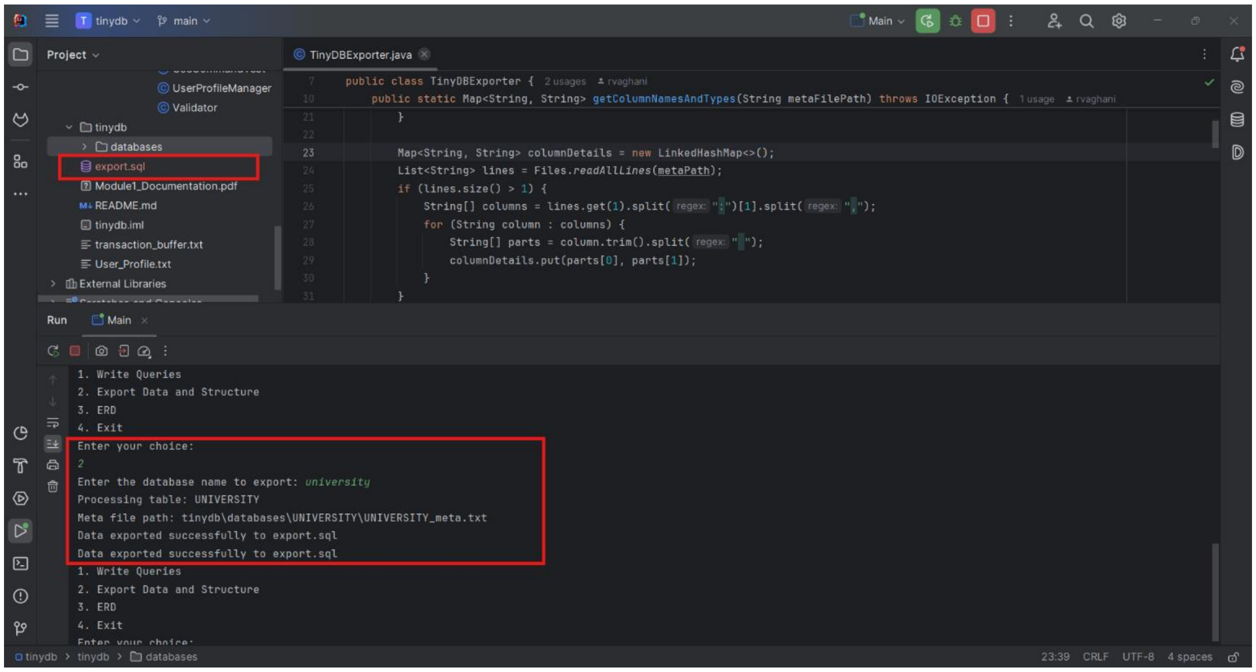


```java
public class TinyDBExporter {
    public static Map<String, String> getColumnNamesAndTypes(String metaFilePath) throws IOException {
        }

        Map<String, String> columnDetails = new LinkedHashMap<>();
        List<String> lines = Files.readAllLines(metaPath);
        if (lines.size() > 1) {
            String[] columns = lines.get(1).split(":")[1].split(",");
            for (String column : columns) {
                String[] parts = column.trim().split(" ");
                columnDetails.put(parts[0], parts[1]);
            }
        }
        return columnDetails;
    }
```

```
Welcome to TinyDB!
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice: 2
Enter the database name to export: school
Database not found: school
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice: 
```

Figure 8 : Entering the database which is not created

- After giving the correct database name export.sql fill will be created.

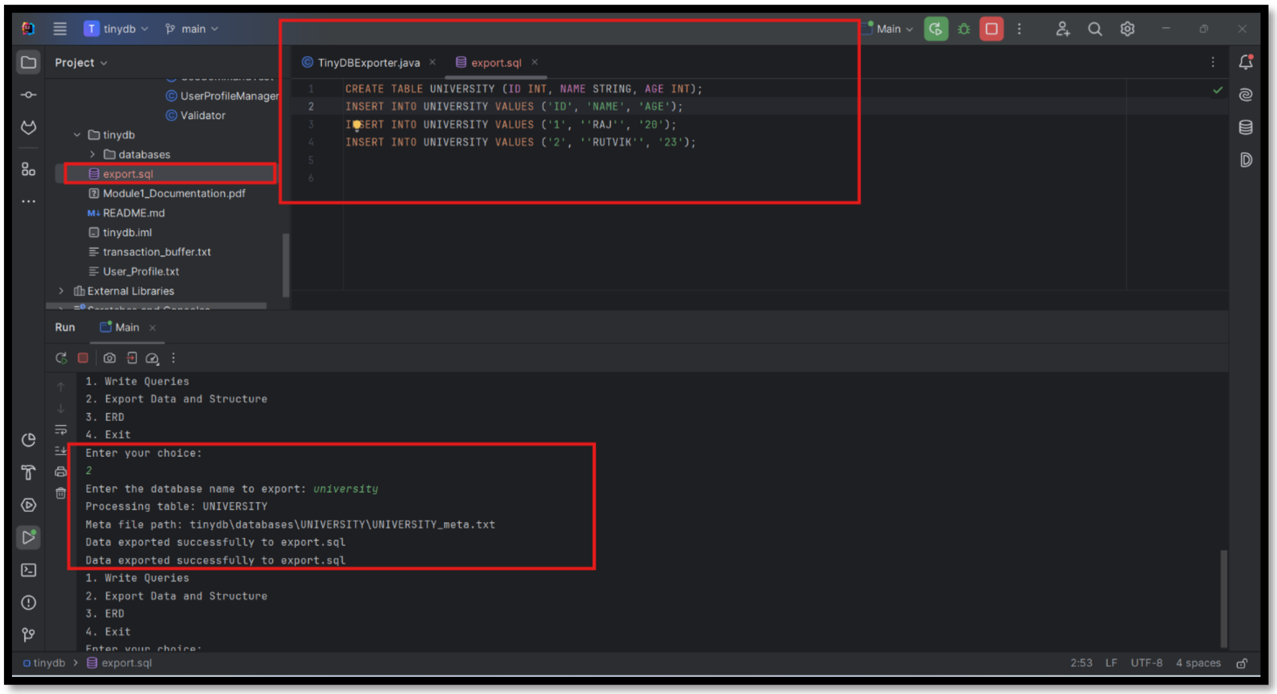


```
public class TinyDBExporter {
    public static Map<String, String> getColumnNamesAndTypes(String metaFilePath) throws IOException {
        }

        Map<String, String> columnDetails = new LinkedHashMap<>();
        List<String> lines = Files.readAllLines(metaPath);
        if (lines.size() > 1) {
            String[] columns = lines.get(1).split(":")[1].split(",");
            for (String column : columns) {
                String[] parts = column.trim().split(" ");
                columnDetails.put(parts[0], parts[1]);
            }
        }
```

```
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice:
2
Enter the database name to export: university
Processing table: UNIVERSITY
Meta file path: tinydb\databases\UNIVERSITY\UNIVERSITY_meta.txt
Data exported successfully to export.sql
Data exported successfully to export.sql
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice:
```

Figure 9 : entering the correct the database name and export the database

- Export the database table structure and the data with only one table in the database.

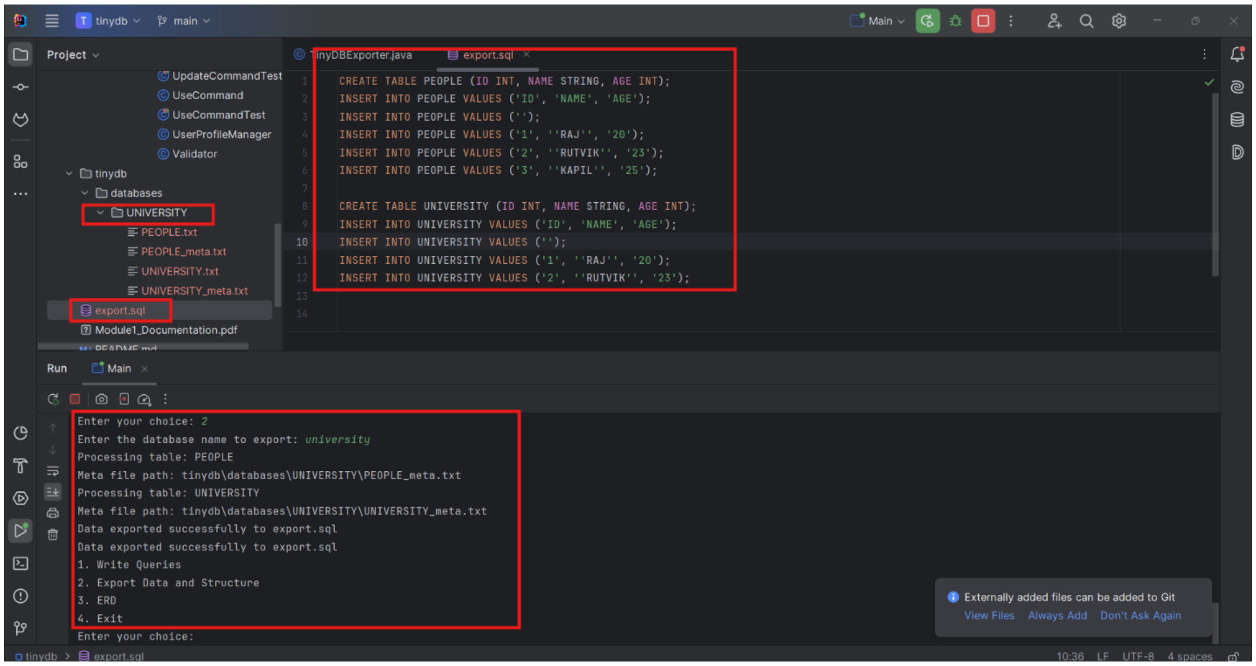


```
CREATE TABLE UNIVERSITY (ID INT, NAME STRING, AGE INT);
INSERT INTO UNIVERSITY VALUES ('ID', 'NAME', 'AGE');
INSERT INTO UNIVERSITY VALUES ('1', ''RAJ'', '20');
INSERT INTO UNIVERSITY VALUES ('2', ''RUTVIK'', '23');
```

```
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice:
2
Enter the database name to export: university
Processing table: UNIVERSITY
Meta file path: tinydb\databases\UNIVERSITY\UNIVERSITY_meta.txt
Data exported successfully to export.sql
Data exported successfully to export.sql
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice:
```

Figure 10 : export the database with the export file with the one table

- Export the database with the table structure and the data with two table tables.

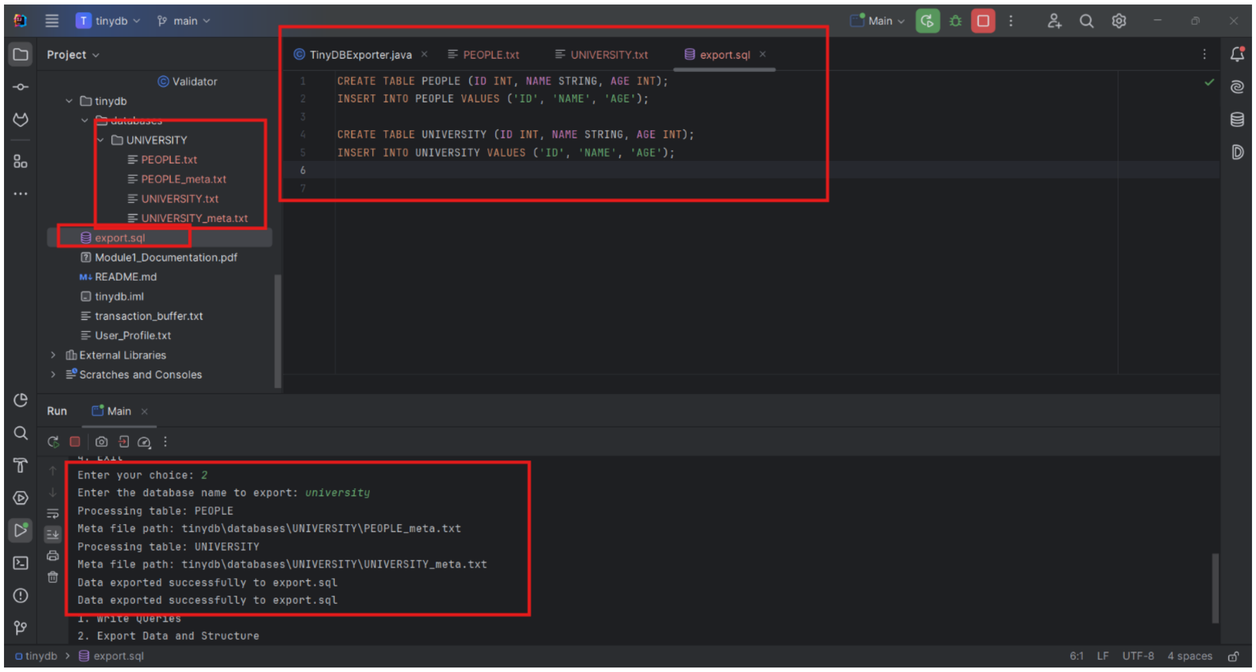


```sql
CREATE TABLE PEOPLE (ID INT, NAME STRING, AGE INT);
INSERT INTO PEOPLE VALUES ('ID', 'NAME', 'AGE');
INSERT INTO PEOPLE VALUES ('');
INSERT INTO PEOPLE VALUES ('1', ''RAJ'', '20');
INSERT INTO PEOPLE VALUES ('2', ''RUTVIK'', '23');
INSERT INTO PEOPLE VALUES ('3', ''KAPIL'', '25');

CREATE TABLE UNIVERSITY (ID INT, NAME STRING, AGE INT);
INSERT INTO UNIVERSITY VALUES ('ID', 'NAME', 'AGE');
INSERT INTO UNIVERSITY VALUES ('');
INSERT INTO UNIVERSITY VALUES ('1', ''RAJ'', '20');
INSERT INTO UNIVERSITY VALUES ('2', ''RUTVIK'', '23');
```

```
Enter your choice: 2
Enter the database name to export: university
Processing table: PEOPLE
Meta file path: tinydb\databases\UNIVERSITY\PEOPLE_meta.txt
Processing table: UNIVERSITY
Meta file path: tinydb\databases\UNIVERSITY\UNIVERSITY_meta.txt
Data exported successfully to export.sql
Data exported successfully to export.sql
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Enter your choice:
```

Figure 11 : export the database with the two tables in the database

- Export the database table structure only without having the data in the export only the table structure.



```sql
CREATE TABLE PEOPLE (ID INT, NAME STRING, AGE INT);
INSERT INTO PEOPLE VALUES ('ID', 'NAME', 'AGE');

CREATE TABLE UNIVERSITY (ID INT, NAME STRING, AGE INT);
INSERT INTO UNIVERSITY VALUES ('ID', 'NAME', 'AGE');
```

```
Enter your choice: 2
Enter the database name to export: university
Processing table: PEOPLE
Meta file path: tinydb\databases\UNIVERSITY\PEOPLE_meta.txt
Processing table: UNIVERSITY
Meta file path: tinydb\databases\UNIVERSITY\UNIVERSITY_meta.txt
Data exported successfully to export.sql
Data exported successfully to export.sql
1. Write Queries
2. Export Data and Structure
```

Figure 12 : Export the database with the structure of the table without the table data

Testing of the Logging:

UserActivity logs:

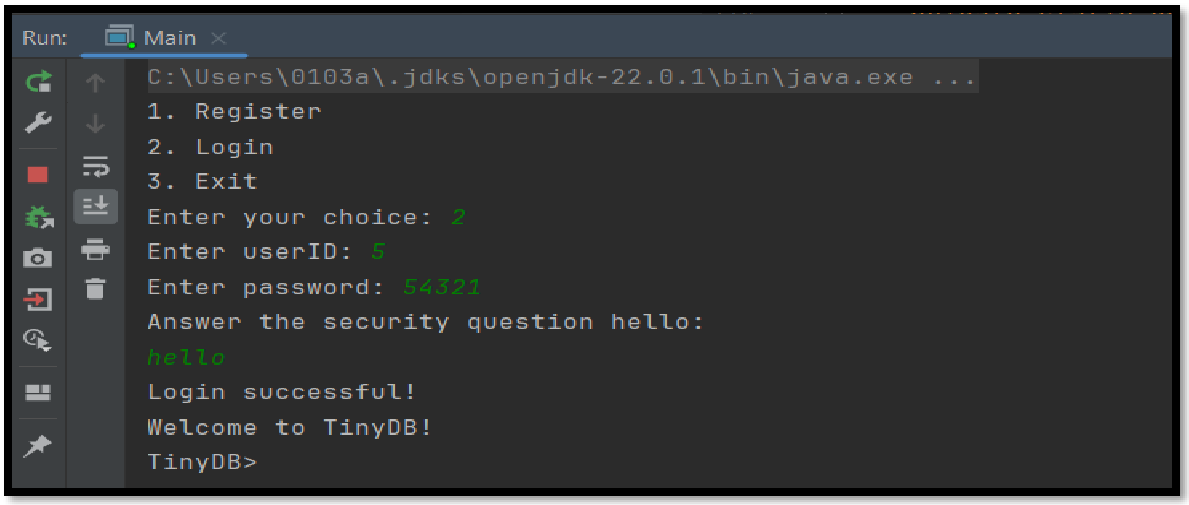
Explanation: User activity logs are important for monitoring user interactions with the system, enhancing security by tracking potentially suspicious activities, and providing insights into user behaviour. The logUserActivity method logs user activities, capturing the type of activity and the username involved. This method writes the log entry to "user_activity_log.json".

Input :

Id = 5;

Password = 54321

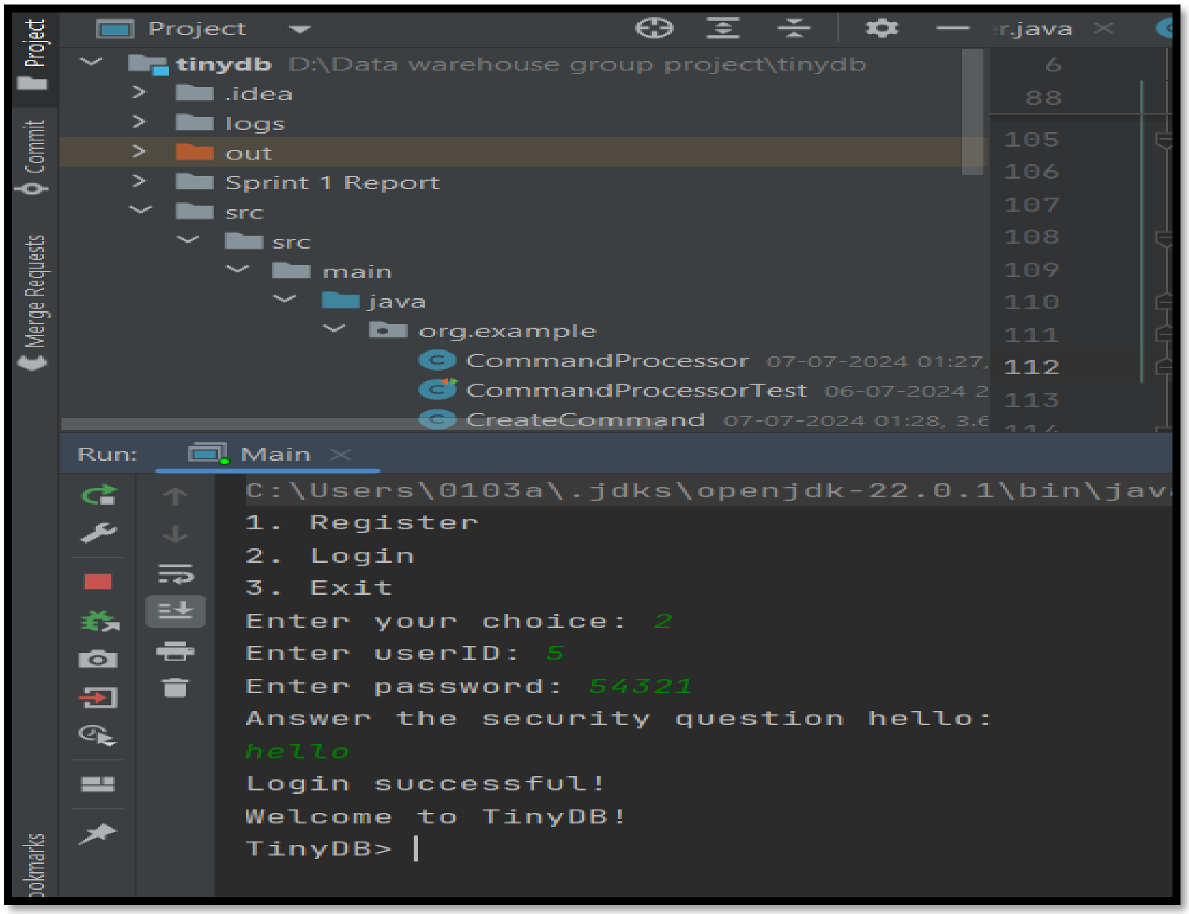
Answer of security question hello: hello



```
C:\Users\0103a\.jdks\openjdk-22.0.1\bin\java.exe ...
1. Register
2. Login
3. Exit
Enter your choice: 2
Enter userID: 5
Enter password: 54321
Answer the security question hello:
hello
Login successful!
Welcome to TinyDB!
TinyDB>
```

Figure 13: Console Input for UserActivity Logs

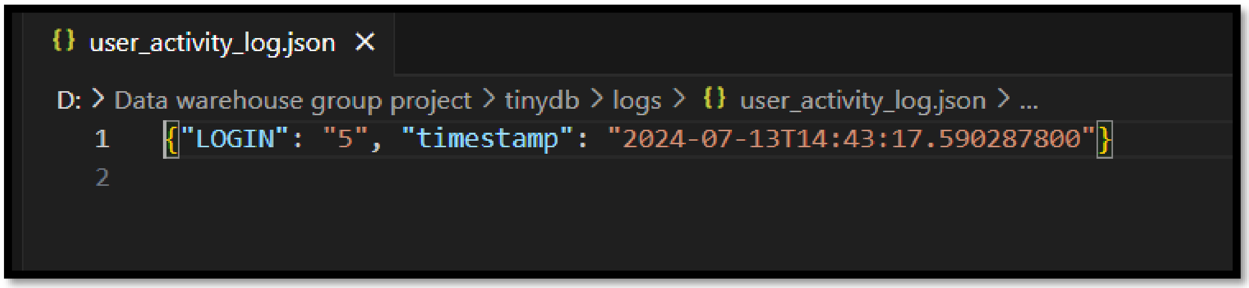
- In the project's directory structure, we can see the presence of a logs folder, which is where all log files are stored. Among these log files is user_activity_log.json, which records the user activities within the application.


Project
tinydb D:\Data warehouse group project\tinydb
.idea
logs
out
Sprint 1 Report
src
src
main
java
org.example
CommandProcessor 07-07-2024 01:27,
CommandProcessorTest 06-07-2024 2
CreateCommand 07-07-2024 01:28, 3.6
Run:
Main

```
C:\Users\0103a\.jdks\openjdk-22.0.1\bin\jav
1. Register
2. Login
3. Exit
Enter your choice: 2
Enter userID: 5
Enter password: 54321
Answer the security question hello:
hello
Login successful!
Welcome to TinyDB!
TinyDB> 
```

Figure 14 : Logs Folder creation for logs

- After command execution a file named user_activity_log.json has been created inside the logs folder. The user_activity_log.json file captures and stores each user action along with a timestamp, ensuring a detailed and precise log of user activities.


user_activity_log.json
D: > Data warehouse group project > tinydb > logs > user_activity_log.json > ...

```
{"LOGIN": "5", "timestamp": "2024-07-13T14:43:17.590287800"}
```

Figure 15 : Logs created for user activities in user_activity_log.json

Explanation of above output:

In this particular example, the log entry shows that a user with ID 5 performed a "LOGIN" activity. This entry is formatted as a JSON object with two key-value pairs: the activity type ("LOGIN") and the timestamp of the activity ("2024-07-13T14:43:17.590287800").

Event Logs:

Explanation: Logging events helps in tracking specific actions or occurrences within the system, which can be crucial for debugging and understanding the application's behaviour. The logEvent method is designed to log specific events that occur within the application. This method takes an event type and a description as parameters, creating a detailed log entry that describes what event took place. The log entry is written to "event_log.json" using the writeLog method.

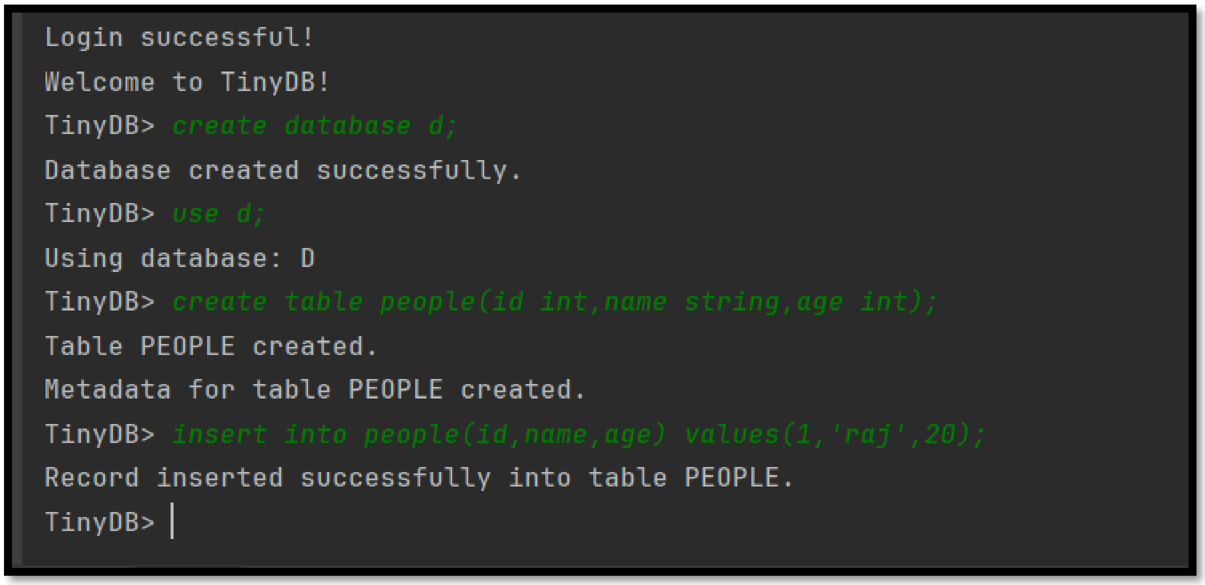
Input:

create database a;

use a;

create table people(id int,name string,age int);

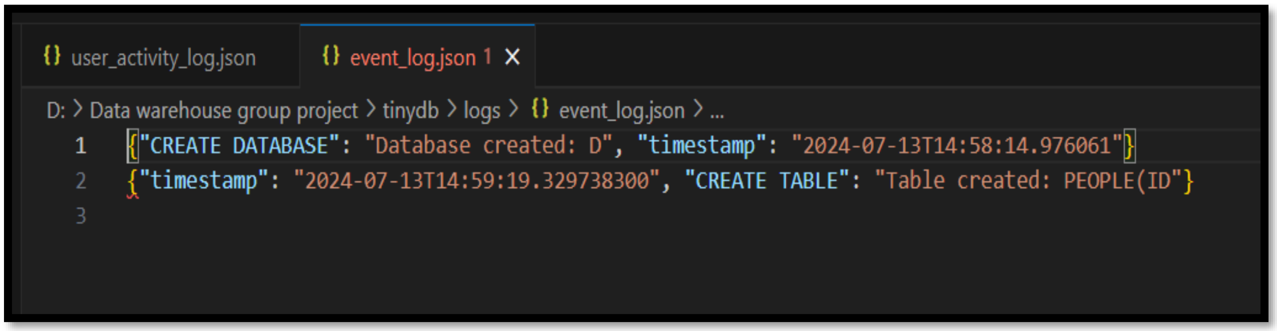
insert into people(id,name,age) values(1,'raj',20);



```
Login successful!
Welcome to TinyDB!
TinyDB> create database d;
Database created successfully.
TinyDB> use d;
Using database: D
TinyDB> create table people(id int,name string,age int);
Table PEOPLE created.
Metadata for table PEOPLE created.
TinyDB> insert into people(id,name,age) values(1,'raj',20);
Record inserted successfully into table PEOPLE.
TinyDB> |
```

Figure 16 : Console Input for Events logs

- After command execution a file named event_log.json has been created inside the logs folder, that will store all the logs for major events that occur in database.
- The application logs various actions such as creating a database, using a database, creating a table, and inserting a record into the table. These actions are logged in the event_log.json file, capturing essential information about the events.


user_activity_log.json
event_log.json 1
D: > Data warehouse group project > tinydb > logs > event_log.json > ...

```
{"CREATE DATABASE": "Database created: D", "timestamp": "2024-07-13T14:58:14.976061"}
{"timestamp": "2024-07-13T14:59:19.329738300", "CREATE TABLE": "Table created: PEOPLE(ID"}
```

Figure 17 : Events logs are written in event_log_json

__**Explanation above output:**__

Creating a Database: Creating a database is an event, hence a log entry is created for this event.

- Command: create database d;
- Log Entry: {"CREATE DATABASE": "Database created: D", "timestamp": "2024-07-13T14:58:14.976061"}

Creating a Table: Creating a table is an event, hence a log entry is created for this event

- Command: create table people(id int,name string,age int);
- Log Entry: {"CREATE TABLE": "Table created: PEOPLE(ID)", "timestamp": "2024-07-13T14:59:19.329738300"}

__**General Logs:**__

Explanation: The general logs are used to capture broad and non-specific information about the application's operation, which can be useful for tracking the overall state and behaviour of the system. The logGeneral method is designed to handle general log entries that are not specific to any particular event, query, transaction, or user activity. This method takes a message and a state as parameters and calls the writeLog method, specifying "general_log.json" as the log file and using the createLogEntry method to format the log entry.

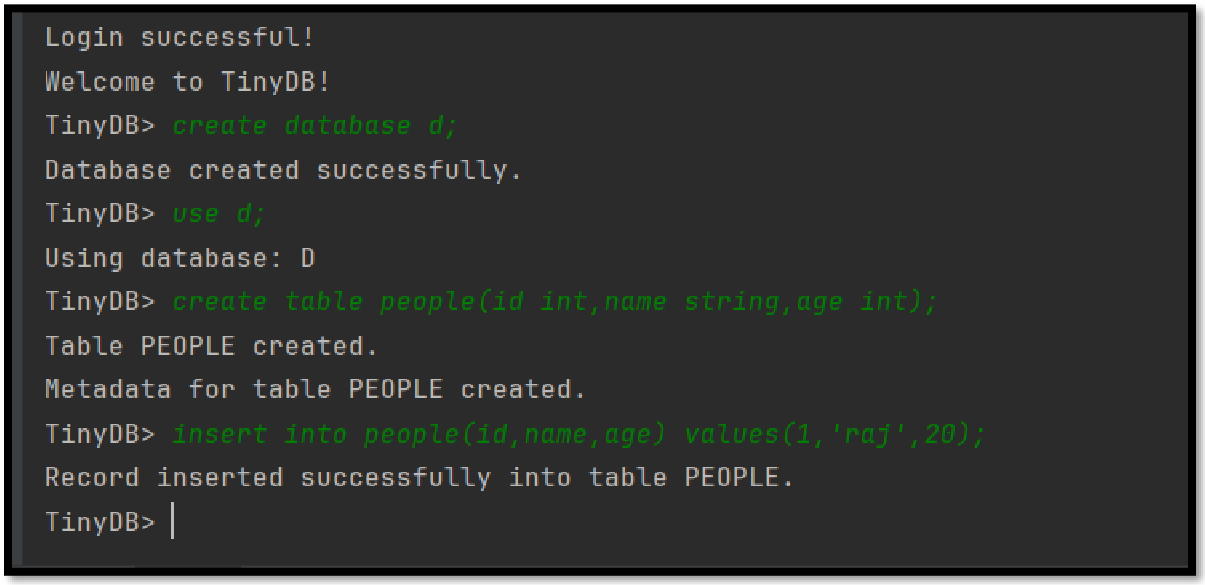
Input:

create database a;

use a;

create table people(id int,name string,age int);

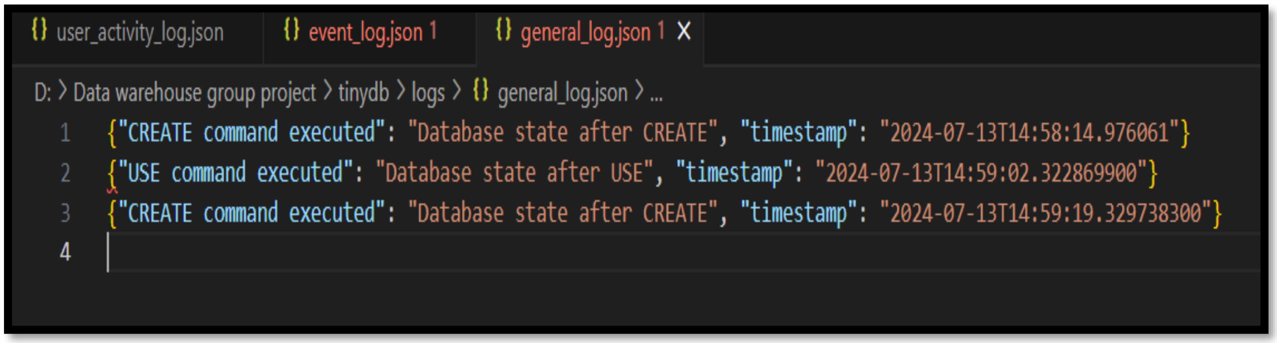
insert into people(id,name,age) values(1,'raj',20);



```
Login successful!
Welcome to TinyDB!
TinyDB> create database d;
Database created successfully.
TinyDB> use d;
Using database: D
TinyDB> create table people(id int,name string,age int);
Table PEOPLE created.
Metadata for table PEOPLE created.
TinyDB> insert into people(id,name,age) values(1,'raj',20);
Record inserted successfully into table PEOPLE.
TinyDB> |
```

Figure 18 : Console input for general logs

- After command execution a file named general_log.json has been created inside the logs folder.
- The general_log.json file captures various general actions performed within the database system, such as the execution of CREATE and USE commands, along with their corresponding states and timestamps.



{} user_activity_log.json
{} event_log.json 1
{} general_log.json 1 X
D: > Data warehouse group project > tinydb > logs > {} general_log.json > ...

```
{"CREATE command executed": "Database state after CREATE", "timestamp": "2024-07-13T14:58:14.976061"}
{"USE command executed": "Database state after USE", "timestamp": "2024-07-13T14:59:02.322869900"}
{"CREATE command executed": "Database state after CREATE", "timestamp": "2024-07-13T14:59:19.329738300"}
```

Figure 19 : General logs are written in general_log.json

Explanation of above output:

Create Command Execution:

- Log Entry: {"CREATE command executed": "Database state after CREATE", "timestamp": "2024-07-13T14:58:14.976061"}
- This log entry records the execution of a CREATE command, capturing the database state immediately after the creation of the database.

Use Command Execution:

- Log Entry: {"USE command executed": "Database state after USE", "timestamp": "2024-07-13T14:59:02.322869900"}
- This entry logs the execution of a USE command, indicating the database state after selecting a specific database for use.

Create Command Execution (Table):

- Log Entry: {"CREATE command executed": "Database state after CREATE", "timestamp": "2024-07-13T14:59:19.329738300"}
- This log entry captures another CREATE command execution, likely related to the creation of a table within the selected database. The database state and timestamp are recorded, providing a clear record of schema changes and their timing.

Query Logs:

Explanation: The query logs is particularly useful for auditing and troubleshooting purposes, as it provides a record of all database operations that were performed, allowing for easier identification of issues related to data manipulation. The logQuery method logs all SQL-like queries executed within the application. It takes a query string as a parameter and writes a log entry to "query_log.json".

Input:

create database a;

use a;

create table people(id int,name string,age int);

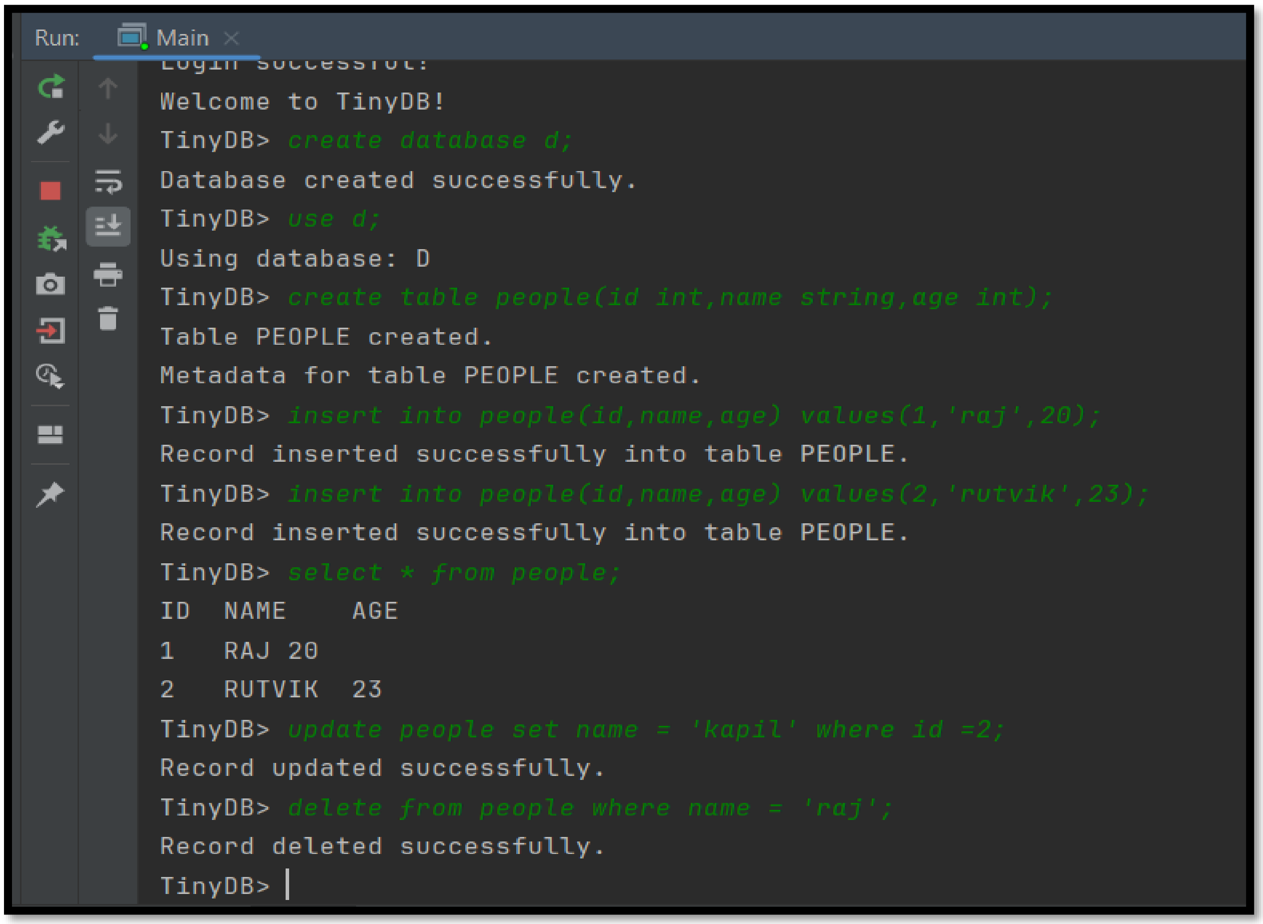
insert into people(id,name,age) values(1,'raj',20);

insert into people(id,name,age) values(2,'rutvik',23);

*select * from people;*

update people set name = 'kapil' where id =2;

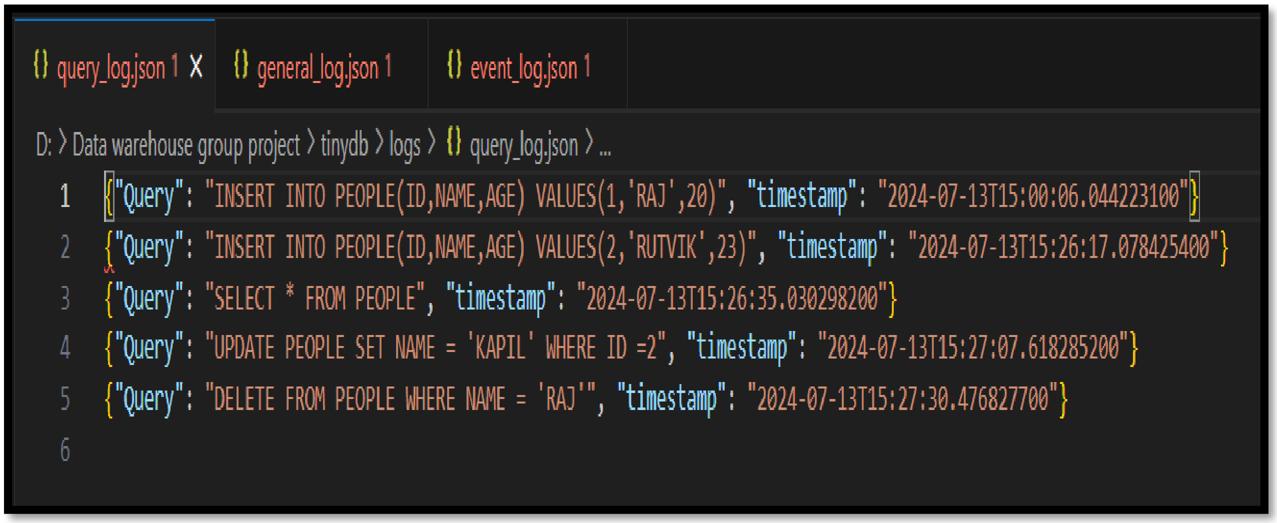
delete from people where name = 'raj';



```
Login successful!
Welcome to TinyDB!
TinyDB> create database d;
Database created successfully.
TinyDB> use d;
Using database: D
TinyDB> create table people(id int,name string,age int);
Table PEOPLE created.
Metadata for table PEOPLE created.
TinyDB> insert into people(id,name,age) values(1,'raj',20);
Record inserted successfully into table PEOPLE.
TinyDB> insert into people(id,name,age) values(2,'rutvik',23);
Record inserted successfully into table PEOPLE.
TinyDB> select * from people;
ID  NAME    AGE
1   RAJ 20
2   RUTVIK  23
TinyDB> update people set name = 'kapil' where id =2;
Record updated successfully.
TinyDB> delete from people where name = 'raj';
Record deleted successfully.
TinyDB>
```

Figure 20 : Console input Query logs

- After the successful command execution a file has been created inside the logs folder named query_log.json that stores all query logs.


{} query_log.json 1 X
{} general_log.json 1
{} event_log.json 1
D: > Data warehouse group project > tinydb > logs > {} query_log.json > ...

```
{"Query": "INSERT INTO PEOPLE(ID,NAME,AGE) VALUES(1,'RAJ',20)", "timestamp": "2024-07-13T15:00:06.044223100"}
{"Query": "INSERT INTO PEOPLE(ID,NAME,AGE) VALUES(2,'RUTVIK',23)", "timestamp": "2024-07-13T15:26:17.078425400"}
{"Query": "SELECT * FROM PEOPLE", "timestamp": "2024-07-13T15:26:35.030298200"}
{"Query": "UPDATE PEOPLE SET NAME = 'KAPIL' WHERE ID =2", "timestamp": "2024-07-13T15:27:07.618285200"}
{"Query": "DELETE FROM PEOPLE WHERE NAME = 'RAJ'", "timestamp": "2024-07-13T15:27:30.476827700"}
```

Figure 21 : Query logs are written in query_log.json

<u>**Explanation of above output:**</u>

Insert Records: Inserting record uses insert query and hence the logs for this input is created in query logs.

- Command: insert into people(id,name,age) values(1,'raj',20);
- Log Entry: {"Query": "INSERT INTO PEOPLE(ID,NAME,AGE) VALUES(1,'RAJ',20)", "timestamp": "2024-07-13T15:00:06.044223100"}
- Command: insert into people(id,name,age) values(2,'rutvik',23);
- Log Entry: {"Query": "INSERT INTO PEOPLE(ID,NAME,AGE) VALUES(2,'RUTVIK',23)", "timestamp": "2024-07-13T15:26:17.078425400"}

Select Records: Selecting record uses select query and hence the logs for this input is created in query logs.

- Command: select * from people;
- Log Entry: {"Query": "SELECT * FROM PEOPLE", "timestamp": "2024-07-13T15:26:35.030298200"}

Update Records: Updating record uses update query and hence the logs for this input is created in query logs.

- Command: update people set name = 'kapil' where id = 2;

- Log Entry: {"Query": "UPDATE PEOPLE SET NAME = 'KAPIL' WHERE ID = 2", "timestamp": "2024-07-13T15:27:07.618285200"}

Delete Records: Deleting record uses delete query and hence the logs for this input is created in query logs.

- Command: delete from people where name = 'raj';
- Log Entry: {"Query": "DELETE FROM PEOPLE WHERE NAME = 'RAJ'", "timestamp": "2024-07-13T15:27:30.476827700"}

<u>Transaction logs:</u>

Explanation: Logging transactions helps maintain an audit trail of all transactional operations, which is essential for ensuring data integrity and for performing forensic analysis in case of issues [1]. The logTransaction method logs details about transactions within the application. It takes the transaction type and transaction detail as parameters, creating a log entry that is written to "transaction_log.json".

Input:

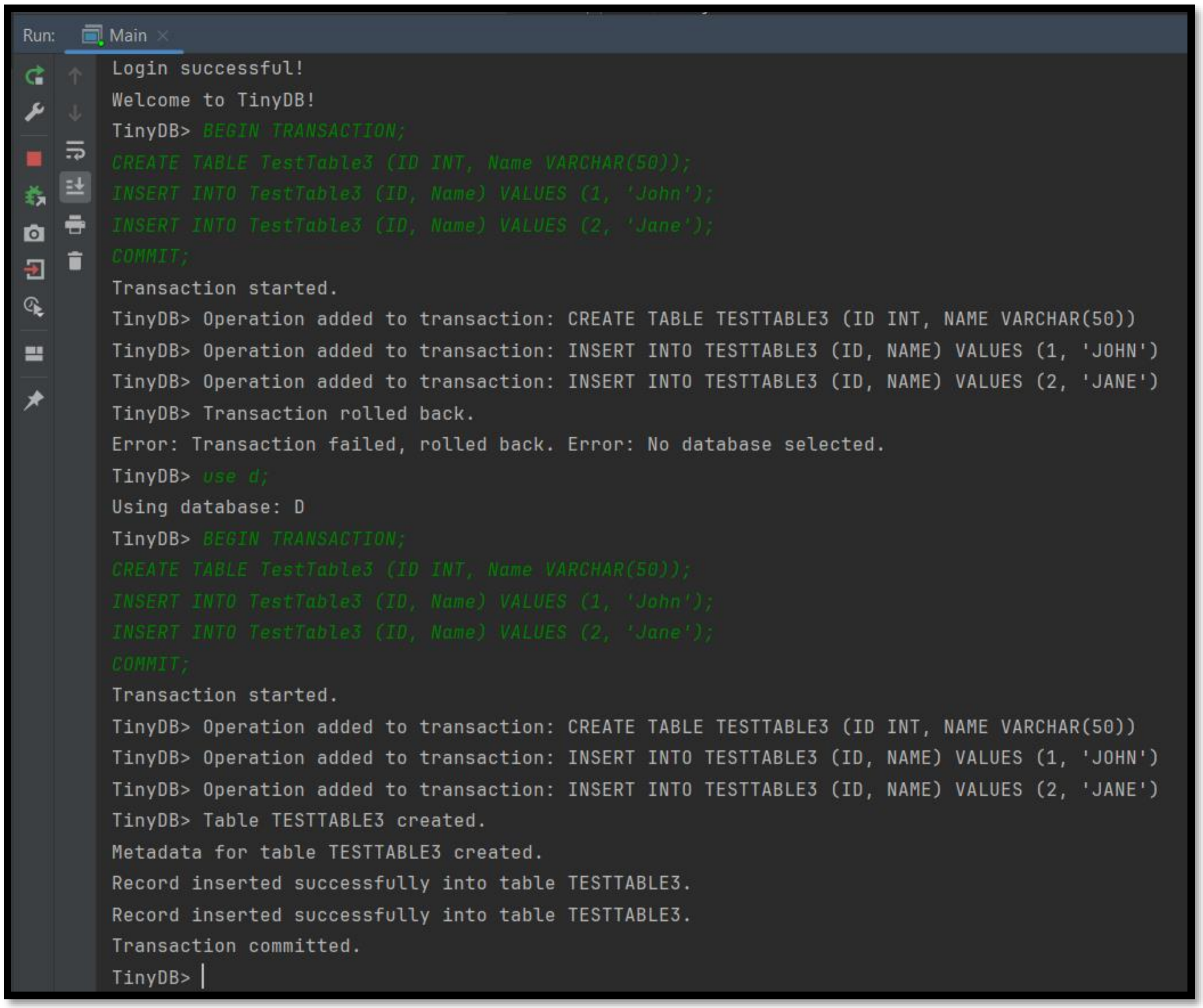
BEGIN TRANSACTION;

CREATE TABLE TestTable3 (ID INT, Name VARCHAR(50));

INSERT INTO TestTable3 (ID, Name) VALUES (1, 'John');

INSERT INTO TestTable3 (ID, Name) VALUES (2, 'Jane');

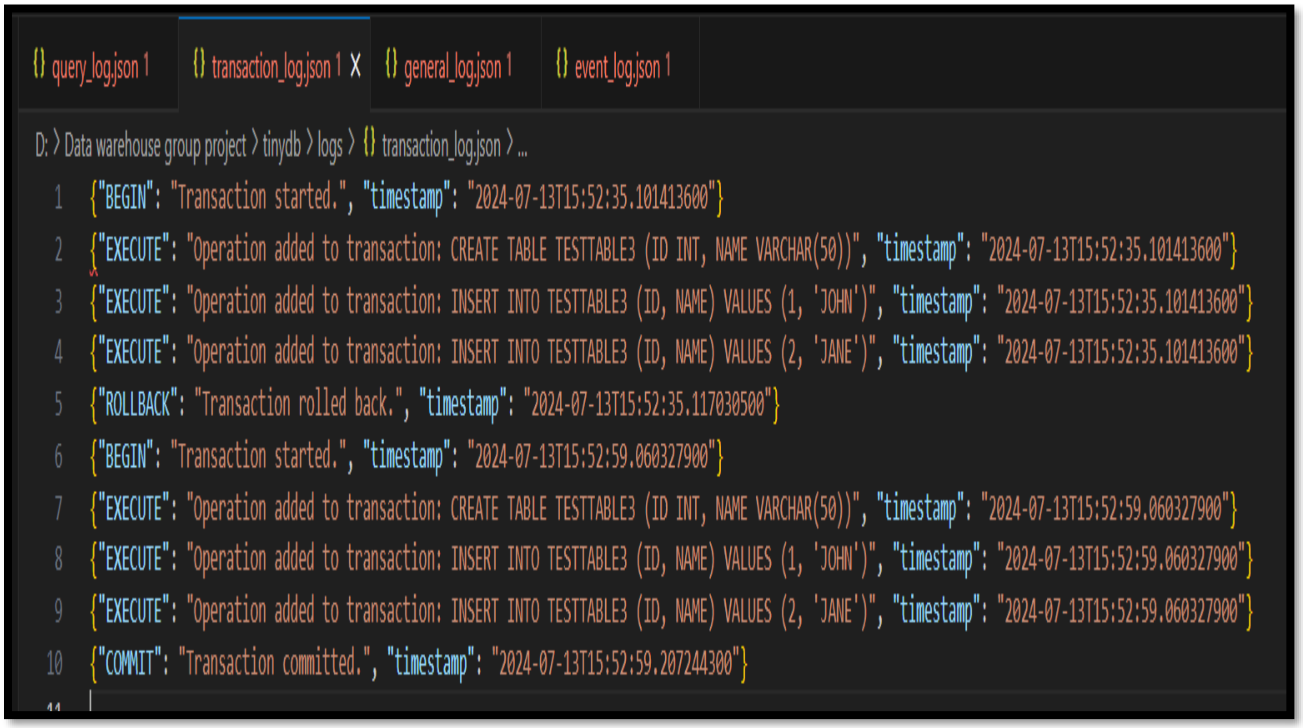
COMMIT;



```
Login successful!
Welcome to TinyDB!
TinyDB> BEGIN TRANSACTION;
CREATE TABLE TestTable3 (ID INT, Name VARCHAR(50));
INSERT INTO TestTable3 (ID, Name) VALUES (1, 'John');
INSERT INTO TestTable3 (ID, Name) VALUES (2, 'Jane');
COMMIT;
Transaction started.
TinyDB> Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))
TinyDB> Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')
TinyDB> Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')
TinyDB> Transaction rolled back.
Error: Transaction failed, rolled back. Error: No database selected.
TinyDB> use d;
Using database: D
TinyDB> BEGIN TRANSACTION;
CREATE TABLE TestTable3 (ID INT, Name VARCHAR(50));
INSERT INTO TestTable3 (ID, Name) VALUES (1, 'John');
INSERT INTO TestTable3 (ID, Name) VALUES (2, 'Jane');
COMMIT;
Transaction started.
TinyDB> Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))
TinyDB> Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')
TinyDB> Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')
TinyDB> Table TESTTABLE3 created.
Metadata for table TESTTABLE3 created.
Record inserted successfully into table TESTTABLE3.
Record inserted successfully into table TESTTABLE3.
Transaction committed.
TinyDB> |
```

Figure 22 : Console input for transaction logs

- After transaction command is executed it creates a file inside the logs folder named transaction_log.json


{} query_log.json 1
{} transaction_log.json 1 X
{} general_log.json 1
{} event_log.json 1
D: > Data warehouse group project > tinydb > logs > {} transaction_log.json > ...

```
{"BEGIN": "Transaction started.", "timestamp": "2024-07-13T15:52:35.101413600"}
{"EXECUTE": "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))", "timestamp": "2024-07-13T15:52:35.101413600"}
{"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')", "timestamp": "2024-07-13T15:52:35.101413600"}
{"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')", "timestamp": "2024-07-13T15:52:35.101413600"}
{"ROLLBACK": "Transaction rolled back.", "timestamp": "2024-07-13T15:52:35.117030500"}
{"BEGIN": "Transaction started.", "timestamp": "2024-07-13T15:52:59.060327900"}
{"EXECUTE": "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))", "timestamp": "2024-07-13T15:52:59.060327900"}
{"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')", "timestamp": "2024-07-13T15:52:59.060327900"}
{"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')", "timestamp": "2024-07-13T15:52:59.060327900"}
{"COMMIT": "Transaction committed.", "timestamp": "2024-07-13T15:52:59.207244300"}
```

Figure 23 : Transaction logs written in transaction_log.json

Explanation Of the above output:

Failed Transaction: The transaction is failed because we have not selected any database.

- **Begin Transaction**: Each BEGIN log entry records the start of a transaction along with a timestamp, providing a precise record of when the transaction was initiated. This is essential for tracking the lifecycle of transactions.
    - Command: BEGIN TRANSACTION;
    - Output: "Transaction started."
    - Log Entry: {"BEGIN": "Transaction started.", "timestamp": "2024-07-13T15:52:35.101413600"}
- **Create Table**:
    - Command: CREATE TABLE TestTable3 (ID INT, Name VARCHAR(50));
    - Output: "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))"
    - Log Entry: {"EXECUTE": "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))", "timestamp": "2024-07-13T15:52:35.101413600"}

- **Insert Records**:
  - Command: INSERT INTO TestTable3 (ID, Name) VALUES (1, 'John');
  - Output: "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')"
  - Log Entry: {"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')", "timestamp": "2024-07-13T15:52:35.101413600"}
  - Command: INSERT INTO TestTable3 (ID, Name) VALUES (2, 'Jane');
  - Output: "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')"
  - Log Entry: {"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')", "timestamp": "2024-07-13T15:52:35.101413600"}
- **Rollback**: The ROLLBACK log entry records the rollback of a transaction due to an error (e.g., no database selected). This entry includes a timestamp, providing a clear record of the failed transaction and its reason
  - Command: ROLLBACK;
  - Output: "Transaction rolled back."
  - Log Entry: {"ROLLBACK": "Transaction rolled back.", "timestamp": "2024-07-13T15:52:35.117030500"}

Successful Transaction:

- **Begin Transaction**:
  - Command: BEGIN TRANSACTION;
  - Output: "Transaction started."
  - Log Entry: {"BEGIN": "Transaction started.", "timestamp": "2024-07-13T15:52:59.060327900"}
- **Create Table**:
  - Command: CREATE TABLE TestTable3 (ID INT, Name VARCHAR(50));
  - Output: "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))"

    - Log Entry: {"EXECUTE": "Operation added to transaction: CREATE TABLE TESTTABLE3 (ID INT, NAME VARCHAR(50))", "timestamp": "2024-07-13T15:52:59.060327900"}
- **Insert Records**:
    - Command: INSERT INTO TestTable3 (ID, Name) VALUES (1, 'John');
    - Output: "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')"
    - Log Entry: {"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (1, 'JOHN')", "timestamp": "2024-07-13T15:52:59.060327900"}
    - Command: INSERT INTO TestTable3 (ID, Name) VALUES (2, 'Jane');
    - Output: "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')"
    - Log Entry: {"EXECUTE": "Operation added to transaction: INSERT INTO TESTTABLE3 (ID, NAME) VALUES (2, 'JANE')", "timestamp": "2024-07-13T15:52:59.060327900"}
- **Commit**: The COMMIT log entry captures the successful completion of a transaction, including the creation of tables and insertion of records. This entry includes a timestamp, documenting the successful execution and finalization of the transaction.
    - Command: COMMIT;
    - Output: " Transaction committed."
    - Log Entry: {"COMMIT": "Transaction committed.", "timestamp": "2024-07-13T15:52:59.207244300"}

Sprint 2 Meeting logs

Sr. No.	Date	Time	Attendees	Agenda	Meeting type
1	03-07-2024	1:35 PM ADT	Aditya, Mansi, Rutvik	Discussion on module selection to implement in Sprint 2	MS Teams - Online
2	09-07-2024	12:52 PM ADT	Aditya, Mansi, Rutvik	Discussion on individual module implementation	MS Teams-Online

Meeting links:

Meeting 1 link:

https://dalu-my.sharepoint.com/:v:/g/personal/rt691750_dal_ca/EQepACtSrZJPncktQT0DOJABEOManu7Zj7qC-akbe2B9xQ?referrer=Teams.TEAMS-ELECTRON&referrerScenario=MeetingChicletGetLink.view

Meeting 2 link:

https://dalu-my.sharepoint.com/:v:/g/personal/rt691750_dal_ca/EcrJqAhzTaNAvI9Cr65cLCABi_6oPOELF7P77kQbcdmwrw?referrer=Teams.TEAMS-ELECTRON&referrerScenario=MeetingChicletGetLink.view

References

[1] “The transaction log - SQL Server” *Microsoft Learn* [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server?view=sql-server-ver16 [Accessed: July 13, 2024]