

# **CSCI 5408**

## **DATA MANAGEMENT AND WAREHOUSING**

**Group-9**  
**Project: Tiny DB**

**SPRINT 2 Report**

### **Group Members:**

Naqib Ali(B00981531)

Shrey Nimeshkumar Patel(B00960433)

Syeda Misbah Hussain(B00982800)

**Gitlab:** [https://git.cs.dal.ca/naqib/csci\\_5408\\_s24\\_9](https://git.cs.dal.ca/naqib/csci_5408_s24_9)

## Table of Contents

1. Pseudocode .....	3
1.1 Pseudocode for Reverse Engineering (ERD): .....	3
1.2 Pseudocode for Export structure: .....	3
1.3 Pseudocode for Log Management: .....	4
2. Link to gitlab code repository .....	5
3. Test cases and evidence of testing .....	6
3.1 Testing sql dump: .....	6
3.2 Testing Reverse Engineering: .....	9
3.3 Testing Log Management: .....	12
4. Weekly meeting recordings: .....	16

# 1. Pseudocode

Below is the pseudocode for all the functionalities of Log management, ERD creation and SQL dump that have been implemented in our Tiny DB java program for part of Sprint 2:

## 1.1 Pseudocode for Reverse Engineering (ERD):

- Step 1: Define the function createERD(databaseName) which throws IOException.
- Step 2: Initialize databaseDir as a new File object named ERD where all ERDs would be stored if it doesnot exist.
- Step 3: Check if database exist or not:
- Step 4: Create a new File object erdFile in databaseDir with the name databaseName + "\_ERD.txt".
- Step 5: Open a BufferedWriter for erdFile inside a try-with-resources block.
- Step 6: Write the initial ERD headers to writer:
- Step 7: List all metadata files in allERDFiles that end with "\_metadata.txt" and store them in files.
- Step 8: Check if files is not null, For each metadataFile in files: Extract the tableName by replacing "\_metadata.txt" in the file name.
- Step 9: Write table name and column headers to writer:
- Step 10: Read all lines from metadataFile into lines.
- Step 11: For each line in lines: Format the column details
- Step 12: Call FOREIGN KEY function to extract the relationship with other tables and write it in the ERD file.
- Step 13: Check for any other constraints from the meta data file like cardinality and PRIMARY key and write in the file.
- Step 13: Print the ERD file path to the console.
- Step 14: Handle IOException by throwing a new IOException with the error message.

## 1.2 Pseudocode for Export structure:

- Step 1: Method takes database full path as input
- Step 2: Check for if there is a pre existing database dump file. If exists, it deletes the old file
- Step 3: Creates a database dump file path where the file will be stored.
- Step 4: Before looping through every table, Create database and insert database query is written to Stringbuilder.
- Step 5: Loop through the database folder and extract the tables and their corresponding metadata files.
- Step 6: Table name is extracted from the table file that has actual records.
- Step 7: Metadata file of that table is then read to extract the data types primary keys and foreign keys

Step 8: The extraction of Metadata file is used to generate the create table query for each table

Step 9: The table file is read and generates insert query for each row which is appended to stringbuilder

Step 10: The stringbuilder is written to the sql dump file.

Step 11: Step 6 to 10 is repeated for every existing table in the database.

Step 12: After the file is completed the user is notified of file creation and file path.

## 1.3 Pseudocode for Log Management:

Step 1: Define the Logger Class

1. Create a Logger class.
2. Define constants for file paths:
  - GENERAL\_LOG\_FILE for general logs.
  - EVENT\_LOG\_FILE for event logs.
  - QUERY\_LOG\_FILE for query logs.
3. Define a date-time formatter for timestamps.
4. Implement a writeToLogFile method:
  - Open the log file for appending.
  - Write the log entry.
  - Add a new line.
  - Close the log file.
5. Implement a logGeneral method:
  - Get the current timestamp.
  - Format the log entry with execution time, table count, and record count.
  - Call writeToLogFile with the general log file path and log entry.
6. Implement a logEvent method:
  - Get the current timestamp.
  - Format the log entry with the event description.
  - Call writeToLogFile with the event log file path and log entry.
7. Implement a logQuery method:
  - Get the current timestamp.
  - Format the log entry with the query.
  - Call writeToLogFile with the query log file path and log entry.

Step 2: Integrate Logging into Operations

1. In each operation (like createTable, updateTable, etc.), integrate logging:
  - For general logs:
    1. Measure the start time.
    2. Execute the operation.

3. Measure the end time.
  4. Calculate the execution time.
  5. Get the current table count and record count.
  6. Call `Logger.logGeneral` with the execution time, table count, and record count.
- For event logs:
    1. Describe the event (e.g., table created, table updated).
    2. Call `Logger.logEvent` with the event description.
  - For query logs:
    1. Log the query before execution.
    2. Call `Logger.logQuery` with the query string.

### Step 3: Implement `getTableCount` Method

1. Define the `getTableCount` method:
  - Create a `File` object for the database path.
  - If the database directory does not exist or is not a directory, return 0.
  - List all files in the database directory with a `.txt` extension.
  - Return the length of the list of files.

### Step 4: Implement `getRecordCount` Method

1. Define the `getRecordCount` method:
  - Initialize a counter for total records.
  - Create a `File` object for the database path.
  - If the database directory does not exist or is not a directory, return 0.
  - List all files in the database directory with a `.txt` extension.
  - For each file:
    1. Open the file for reading.
    2. Skip the header line.
    3. While reading lines, increment the total record counter.
    4. Close the file.
  - Return the total record count.

## 2. Link to gitlab code repository

[https://git.cs.dal.ca/naqib/csci\\_5408\\_s24\\_9](https://git.cs.dal.ca/naqib/csci_5408_s24_9)

### 3. Test cases and evidence of testing

We have rigorously tested every module of the JAVA implementation that we have done of user authentication and queries for database and tables. We have made sure that all the edge cases get covered and the Tiny DB works the same as the way a standard SQL editor like Workbench works. For this we have taken the image of the CLI based application that user can see in the console as well as the database that gets updated in form of a .txt file. Below are all the images of the testing done both of CLI and database level of the .txt file for Module 4, Module 5 and Module 6.

#### 3.1 Testing sql dump:

**Description:** Giving an out-of-range value gives an error. It is expected from the user to give proper input of the database name that are available. The user has to select the proper number input, if he doesn't then valid message is shown as mentioned below in the snippet.

```
User Menu
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Choose an option: 2
Exporting Data and Structure...
Choose option from below

1: my_database
2: test
Choice: 3
Invalid choice
```

*Figure 1: Example input validation*

**Description:** Giving a string, when number is expected. Throws an error. This is another validation that makes sure that the system receives proper input from the user as wanted and does not allow to enter anything that he wants. Entering a string can return a error as shown below in the image.

```

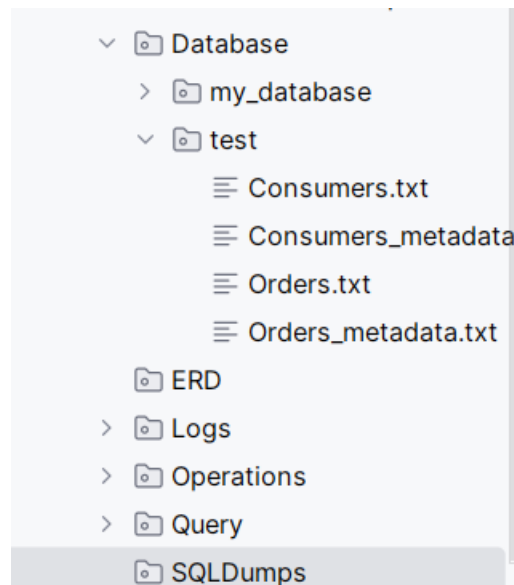
User Menu
1. Write Queries
2. Export Data and Structure
3. ERD
4. Exit
Choose an option: 2
Exporting Data and Structure...
Choose option from below

1: my_database
2: test
Choice: ads
Invalid input. Please enter a number.

```

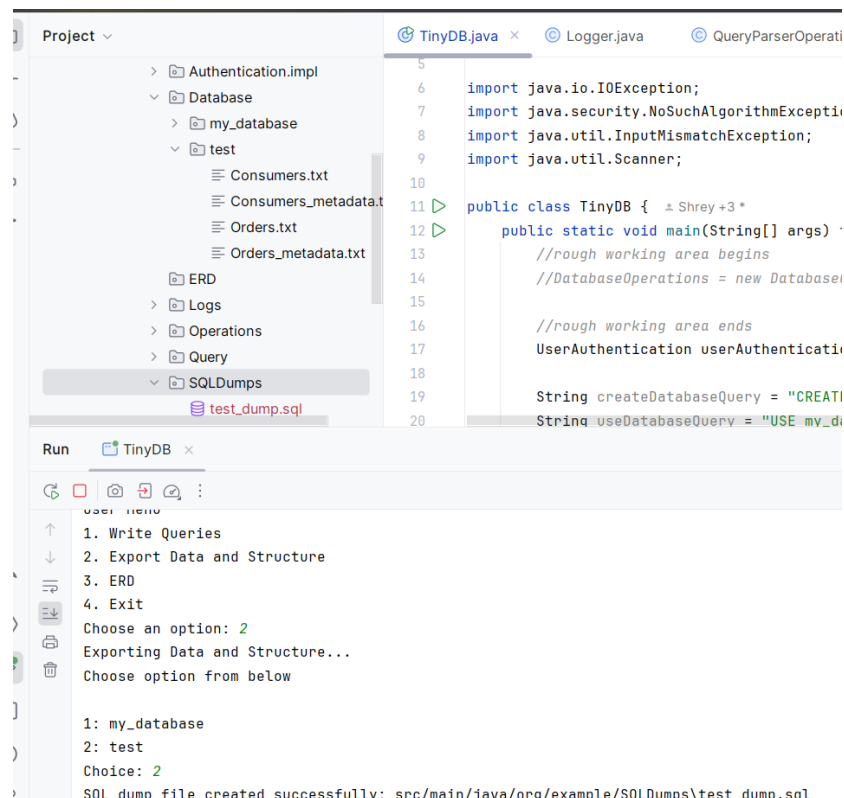
*Figure 2: Example input validation*

**Description:** We will generate export the structure for above mentioned database. The algorithm will scan through all metadata files of the tables and tables recent records and on the basis of that it will create metadata file.



*Figure 3: Test database have following tables and their metadata file*

**Description:** Under the SQLDumps folder the file is created. We will now try to see that a separate folder gets created after the user wants SQL dump, the file gets created under the folder called SQLDumps. Where all the dumps of all database will get created once the user asks for that particular data base's dump. In the below image you can see that the SQL dump named test\_dump.sql got created for the user selection.



**Figure 4: A new sql dump file is created**

**Description:** Attempting to recreate SQLDump file will first check for the existing file, delete it and create newer file as shown above. So now it will fetch the details in the tables, either it is updated, deleted or added. The new SQL dump file will be created based on the new database state replacing the old one.

```
1: my_database
2: test
Choice: 2
Old dump deleted!!
SQL dump file created successfully: src/main/java/org/example/SQLDumps/test_dump.sql
```

**Figure 5: Attempting to recreate sql dump file when file already exist.**

**Description:** After the dump file has been created, it looks like below in the image. Firstly it will show the name of database with create statement, then after using it the create table statements followed by the latest insert statements with the current state of the database. For create table it will show all the foreign key, not null and primary key constraints as well. Below is the proof of that in the form of snippet.



```

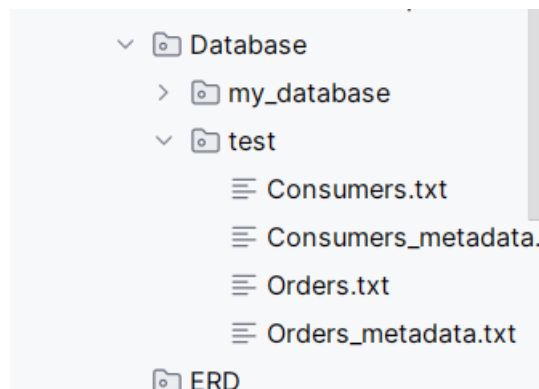
1 CREATE DATABASE test;
2 USE test;|
3
4 CREATE TABLE Consumers (
5     email varchar(255) None,
6     Name varchar(255) None,
7     ID int PRIMARY KEY
8
9 );
10
11 INSERT INTO Consumers VALUES (ID, Name, email);
12 INSERT INTO Consumers VALUES (1, jason, email@gmail);
13 INSERT INTO Consumers VALUES (2, smothj, email@gmail);
14 INSERT INTO Consumers VALUES (3, what, email@gmail);
15
16
17 CREATE TABLE Orders (
18     PersonID int FOREIGN KEY REFERENCES Persons(ID),
19     OrderNumber varchar(255) None,
20     OrderID int PRIMARY KEY
21
22 );
23
24 INSERT INTO Orders VALUES (PersonID, OrderNumber, OrderID);
25 INSERT INTO Orders VALUES (1, abc, 340);
26 INSERT INTO Orders VALUES (2, efg, 341);
27 INSERT INTO Orders VALUES (3, xyz, 342);
28 INSERT INTO Orders VALUES (4, , 343);
29 INSERT INTO Orders VALUES (5, , 344);
30

```

**Figure 6:** Following is the dump file exported in .sql format

## 3.2 Testing Reverse Engineering:

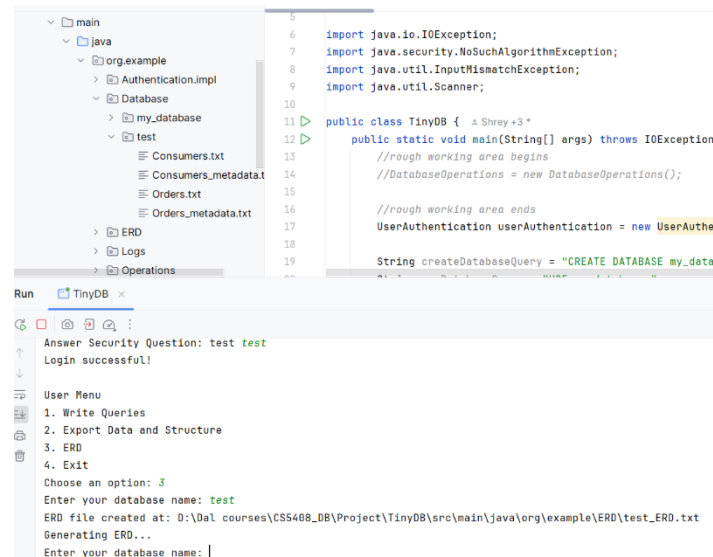
**Description:** Let's attempt to create an ERD. As you can see in the below image, the folder named ERD will be created once the user requests to get one ERD for any database. Right now there is no erd requested, so no ERD is there in the folder.



**Figure 7:** No ERD exists

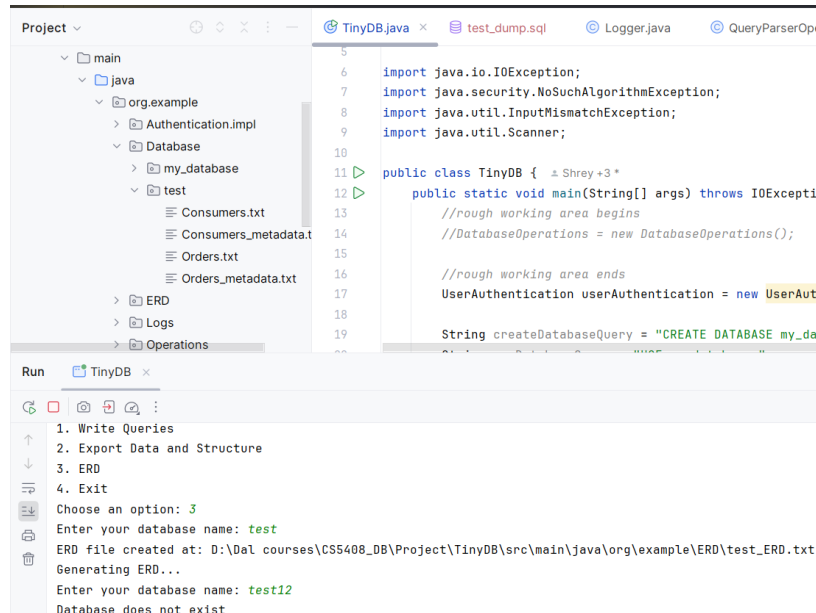
**Description:** ERD for test database is generated under ERD folder. As you can see in the below image, the user enters 2 for creating an erd, then he is asked for database name. If the database exists then the ERD will be created in the ERD folder named table\_erd.txt. This file

will have all the possible details that the database needs to have for all its tables and their relationship with each other.



**Figure 8: ERD for test database is generated**

**Description:** Validation is applied to give error if user inputs database that does not exists. ERD for test database is generated under ERD folder. As you can see in the below image, the user enters invalid database name that does not exist, then the system throws error and shows a proper message to the user that the database name does not exist and that he needs to enter the database name again.



**Figure 9: Attempt to create ERD of a database that does not exist**

**Description:** Below is the sample ERD for database test. The ERD consists of information of all the tables, their column names, their datatype, their relation with other tables and any constraints like foreign key or primary key. We are still left to implement the cardinality in this module, which we will complete by next sprint for sure, which will show relationship with other tables as well.

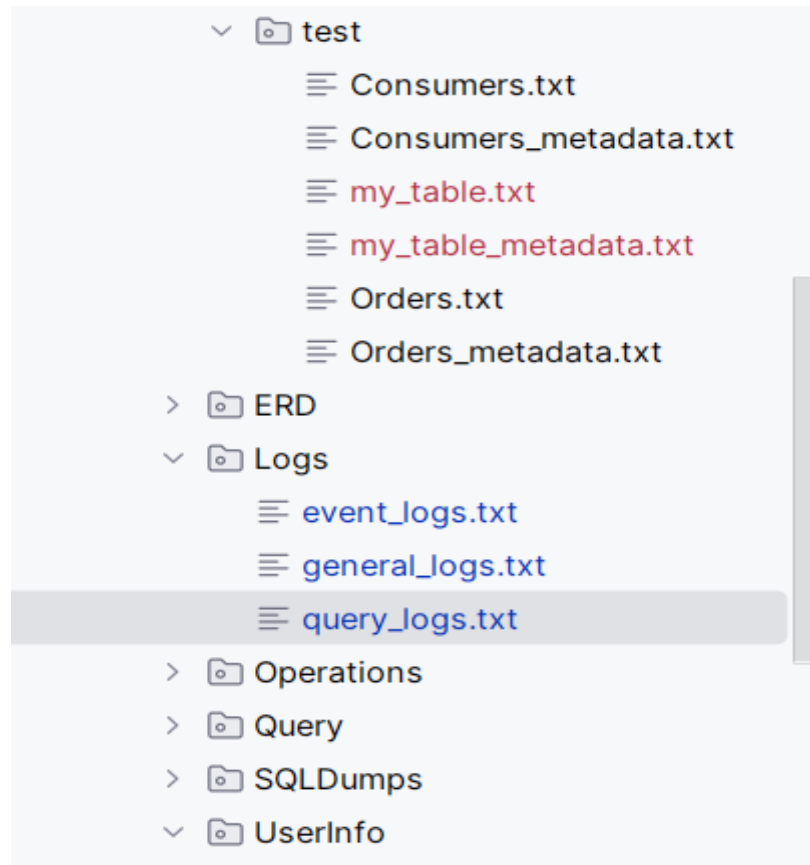
```
TinyDB.java  test_dump.sql  test_ERD.txt x  Logger.java
1 Entity-Relationship Diagram (ERD) for Database: test
2 =====
3
4 Table: Consumers
5 Columns | Datatype | Constraints | Relationship
6 -----
7 email | varchar(255) | None | NONE
8 Name | varchar(255) | None | NONE
9 ID | int | PRIMARY KEY | NONE
10 -----
11
12 Table: Orders
13 Columns | Datatype | Constraints | Relationship
14 -----
15 PersonID | int | FOREIGN KEY | Persons(ID)
16 OrderNumber | varchar(255) | None | NONE
17 OrderID | int | PRIMARY KEY | NONE
18 -----
19 ERD generation completed successfully.
```

***Figure 10: Sample ERD***

### 3.3 Testing Log Management:

**NOTE:** We are still left to complete module 3 of transaction, so we will add on the transaction logs after we complete module 3 and also the crash reports.

**Description:** Our system is designed in such a way that it captures three different types of logs in three different files for every database, so that it becomes easy to further debug the error via logs. Below is the snapshot.



*Figure 11: Log files according to type*

**Description:** Below is the snapshot of the simple use database query that the user enters. Our system captures the logs for the same.

```

Run TinyDB x
[Icons]
↑ User Menu
↓ 1. Write Queries
≡ 2. Export Data and Structure
≡ 3. ERD
≡ 4. Exit
🖨 Choose an option: 1
🗑 Enter your query: use test;
Writing Queries...
Enter your query:

```

*Figure 12: Writing a simple use Query*

**Description:** Below three are the snapshots of the event log, general logs and query logs for the operations of use database that was just executed above. We have noted the timestamp, EventDescription execution time, database state and queries that has been executed for each operation that the user does.

```

TinyDB.java test12_ERD.txt event_logs.txt x general_logs.txt Consumers.txt
1 {timestamp: "2024-07-13 21:11:55", eventDescription: "Switched to database 'test'"}
2

```

*Figure 13: Event logs for use database query*

```

TinyDB.java test12_ERD.txt event_logs.txt general_logs.txt x Consumers.txt my_table.txt
1 {timestamp: "2024-07-13 21:11:55", executionTime: "0 ms", databaseState: {tables: 3, records: 9}}
2

```

*Figure 14: General logs for use database query*

```

test12_ERD.txt event_logs.txt general_logs.txt Consumers.txt
1 {timestamp: "2024-07-13 21:11:55", query: "USE DATABASE test"}
2

```

*Figure 15: Query logs for use database query*

**Description:** Below is the execution of the create table operation from the CLI by the user. Our system captures all three types of logs for the same query execution.

```

run TinyDB x
Choose an option: 1
Enter your query: Use test;
Writing Queries...
Enter your query: CREATE TABLE logs_table (id INT, name VARCHAR(255));
Writing Queries...
[id INT, name VARCHAR(255)]
Table created with name: logs_table
Metadata file created for table: logs_table
Enter your query:
  
```

*Figure 16: Create table Query in CLI*

**Description:** Below three are the snapshots of the event log, general logs and query logs for the operations of create database that was just executed above. We have noted the timestamp, EventDescription execution time, database state and queries that has been executed for each operation that the user does.

```

TinyDB.java test12_ERD.txt event_logs.txt general_logs.txt query_logs.txt my_table.txt Orders.txt
{timestamp: "2024-07-13 21:11:55", eventDescription: "Switched to database 'test'"}
{timestamp: "2024-07-13 21:13:30", eventDescription: "Table 'logs_table' created with columns [id INT, name VARCHAR(255)]"}
  
```

*Figure 17: Event logs for Create table query*

```

TinyDB.java test12_ERD.txt event_logs.txt general_logs.txt query_logs.txt my_table.txt
1 {timestamp: "2024-07-13 21:11:55", executionTime: "0 ms", databaseState: {tables: 3, records: 9}}
2 {timestamp: "2024-07-13 21:13:30", executionTime: "8 ms", databaseState: {tables: 4, records: 9}}
3
  
```

*Figure 18: Event logs for Create table query*

```

TinyDB.java test12_ERD.txt event_logs.txt general_logs.txt query_logs.txt my_table.txt
1 {timestamp: "2024-07-13 21:11:55", query: "USE DATABASE test"}
2 {timestamp: "2024-07-13 21:13:30", query: "CREATE TABLE logs_table (id INT, name VARCHAR(255))"}
3
  
```

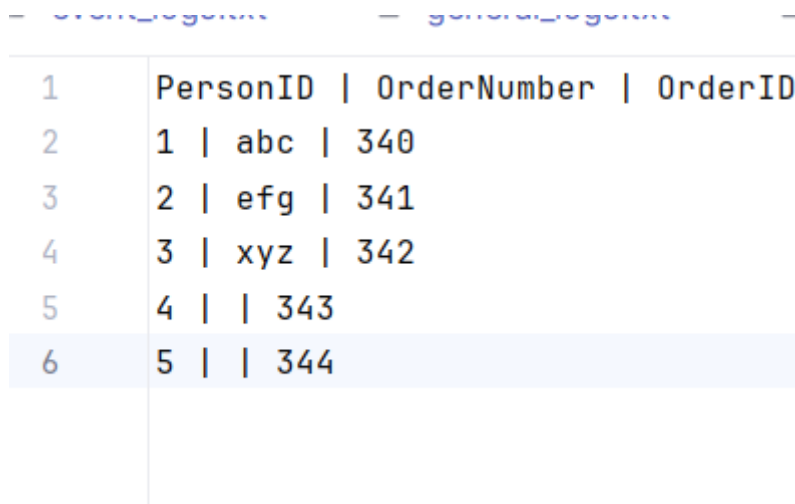
*Figure 19: Event logs for Create table query*

**Description:** Below is the snapshot of the insert query that has taken place in the CLI by the user in table logs\_table and my\_table. The logs for both the tables have been captured. The table inserts can be seen in the below two diagram.



1	ID   Name   email
2	1   jason   email@gmail
3	2   smothj   email@gmail
4	3   what   email@gmail
5	4   Alice   email@gmail
6	

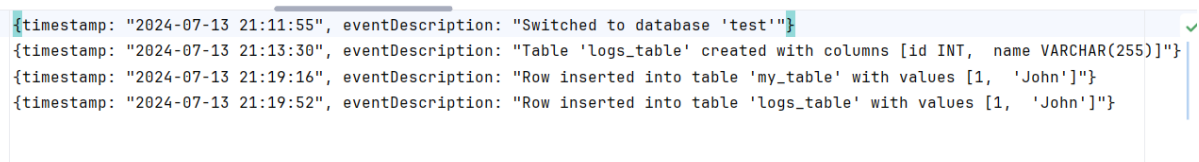
*Figure 20: Insert query for my\_table*



1	PersonID   OrderNumber   OrderID
2	1   abc   340
3	2   efg   341
4	3   xyz   342
5	4     343
6	5     344

*Figure 21: Insert query for logs\_table*

**Description:** Below three are the snapshots of the event log, general logs and query logs for the operations of create database that was just executed above. We have noted the timestamp, EventDescription execution time, database state and queries that has been executed for each operation that the user does.



```
{timestamp: "2024-07-13 21:11:55", eventDescription: "Switched to database 'test'"}
{timestamp: "2024-07-13 21:13:30", eventDescription: "Table 'logs_table' created with columns [id INT, name VARCHAR(255)]"}
{timestamp: "2024-07-13 21:19:16", eventDescription: "Row inserted into table 'my_table' with values [1, 'John']}
{timestamp: "2024-07-13 21:19:52", eventDescription: "Row inserted into table 'logs_table' with values [1, 'John']}"
```

*Figure 22: Event logs for Insert into table query*

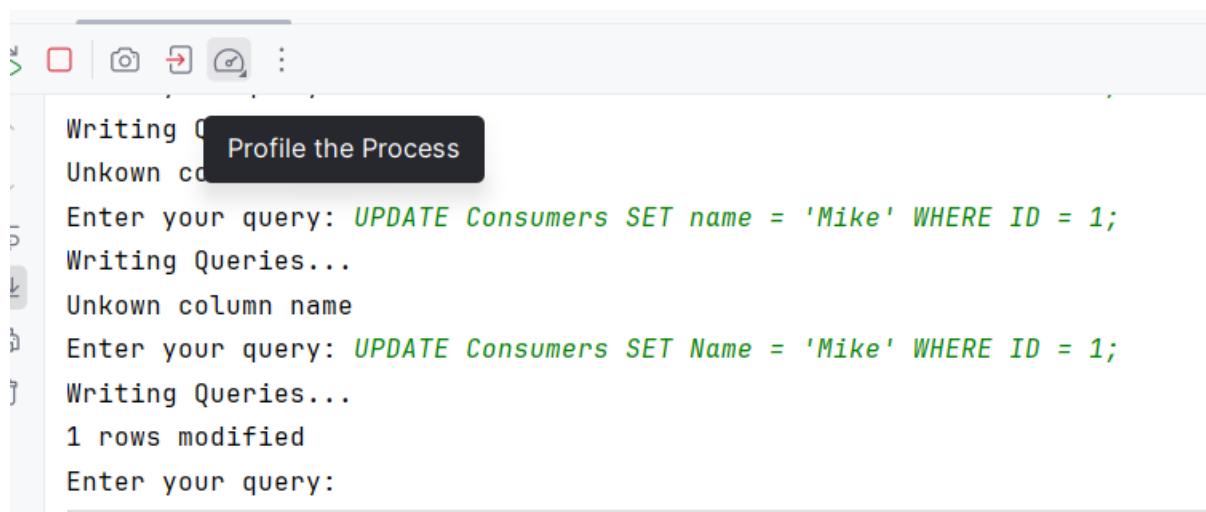
1	{timestamp: "2024-07-13 21:11:55", executionTime: "0 ms", databaseState: {tables: 3, records: 9}}
2	{timestamp: "2024-07-13 21:13:30", executionTime: "8 ms", databaseState: {tables: 4, records: 9}}
3	{timestamp: "2024-07-13 21:19:16", executionTime: "18 ms", databaseState: {tables: 4, records: 9}}
4	{timestamp: "2024-07-13 21:19:52", executionTime: "4 ms", databaseState: {tables: 4, records: 9}}
5	

**Figure 23: Event logs for Insert into table query**

1	{timestamp: "2024-07-13 21:11:55", query: "USE DATABASE test"}
2	{timestamp: "2024-07-13 21:13:30", query: "CREATE TABLE logs_table (id INT, name VARCHAR(255))"}
3	{timestamp: "2024-07-13 21:19:16", query: "INSERT INTO my_table (id, name) VALUES (1, 'John')"}
4	{timestamp: "2024-07-13 21:19:52", query: "INSERT INTO logs_table (id, name) VALUES (1, 'John')"}
5	

**Figure 24: Event logs for Insert into table query**

**Description:** Below is the snapshot of the update query that has been entered in the CLI by the user for updating the Consumers table, first wrong query was knowingly given, the logs were not executed at that time, then after correct query was given to make sure that the logs are recorded.



**Figure 25: Update Query from CLI**

{timestamp: "2024-07-13 21:11:55", eventDescription: "Switched to database 'test'"}	
{timestamp: "2024-07-13 21:13:30", eventDescription: "Table 'logs_table' created with columns [id INT, name VARCHAR(255)]"}	
{timestamp: "2024-07-13 21:19:16", eventDescription: "Row inserted into table 'my_table' with values [1, 'John']"}	
{timestamp: "2024-07-13 21:19:52", eventDescription: "Row inserted into table 'logs_table' with values [1, 'John']"}	
{timestamp: "2024-07-13 21:23:43", eventDescription: "Update query executed on table 'Consumers' with values Name = 'Mike' and condition ID = 1"}	✓

**Figure 25: Event Logs for update Query.**

## 4. Weekly meeting recordings:

[Data\\_Project\\_Record.xlsx \(sharepoint.com\)](#)



