

CSCI 5408: Assignment 1: Problem 2: Prototype of Light-Weight DBMS

Waleed R. Alhindi (B00919848)

Overview:

The program submitted constitutes a light-weight DBMS prototype. It offers users a simple relational DBMS that supports multiple tables on one database. Users interact with the program through a command-line interface, where they must first either register or login. The registration/login function requires a user to provide their username and password in addition two-factor authentication via a security question and answer. Once successfully logged in or registered, users will be provided a MySQL command-line interface that allows them to perform create, select, update, insert, and delete operations. Furthermore, all operations executed by the user are written to a log file to simulate how a DBMS would track such actions.

To achieve these functionalities, the program uses three txt files to store data about users, tables, and logs. These files use custom delimiters such as "\$", "#", and "!".

Program Components:

The submitted program is composed of the following components:

- *Main.java* – The executable java class that will initiate the program and call, in correct sequence, other classes.
- *InputReader.java* – A class that provides a global input scanner to read user inputs from the command-line. Such functionality was placed into a separate class to reduce the number of duplicate lines of code for instantiating and closing a scanner object. Furthermore, this class provides MD5 hashing functionality, where plaintext passwords and security answers are hashed before writing them to the relevant file for an extra layer of security.
- *UserDbInterface.java* – This class is responsible for user login and registration. It acts as an interface between the user and the DB_Users.txt file, which contains user credentials (i.e., users' usernames, passwords, security questions, and security answers). Upon invoking it, this class will ask the user to either log in or register and will check the user's inputs against the contents of the DB_Users.txt file.
- *Table.java* – This class is used to encapsulate the data of a single table. Each new or existing table (i.e., that exist in the Tables.txt file) is stored as an object of this class, where information like the table's name, schema, primary key, rows, and so on is stored as an instance variable of that object. This is done to make parsing, retrieving, and updating tables easier and more efficient since method of this class are designed to be invoked during CRUD operations.

- *QueryInterface.java* – A MySQL command-line interface class that allows the user to perform create, select, update, insert, and delete operations. This class acts as an interface between the user's MySQL commands and the Tables.txt file that stores table data. Upon invoking this class, it will first read from the Tables.txt file, parse each existing table into a new Table class object, then accept users' MySQL commands, validates and executes those commands, then writes any updated data back to the Tables.txt file.
- *LogsInterface.java* – This class is invoked throughout the program's lifecycle and acts as a logger. It logs operations such as user registrations, logins, and MySQL commands into the Log.txt file to simulate a DBMS's tracking mechanisms.

As mentioned previously, this program uses several txt files to facilitate long-term storage. These files use custom delimiters to denote the start and end of data fields. The program utilizes the following files as part of its operations:

- *DB_Users.txt* – A file to store user credentials. This file is used during the registration and login process by the UserDbInterface class. Note that users' passwords and security answers are hashed using MD5.

```
@waleed,00bfc8c729f5d4d529a412b12c58ddd2,sq1,b4169f3b7f14193ebfee9110a161d23a
@w2,1d665b9b1467944c128a5575119d1cfd,sq2,3809f6746ce86180f98ffa2868d721fd
```

Fig 1. Sample DB_Users.txt Entry

- *Tables.txt* – A file to store all user tables, which includes information about tables' schemas, primary keys, foreign keys, and current rows. This file is utilized by the QueryInterface class to perform CRUD operations.

```
| t4
<
d1 int;
d2 string;
%
d1;
^
d2 t1.a1;
$
#4,"s"
```

Fig 2. Sample Table Entry in Tables.txt File

- *Log.txt* – A logs file to store tracking information about user registrations, sign ins, and MySQL commands executed. The LogsInterface class writes to this file whenever such logs are necessary. Note that each log entry is time stamped.

```
[2023.02.12.13.44.43]: User: [waleed]  
Command: [create table t1(d1 int, d2 string, primary key (d1), foreign key (d2) reference t1(a1));]  
Result: [SQL Create Syntax Error!]
```

Fig 3. Sample Log Entry in Log.txt File

Design Decisions:

Due to the light-weight nature of this DBMS prototype, some explicit design decisions needed to be made during the design and implementation of the submitted program. These design decisions and their implications are outlined as follows:

1. Upon booting up the program, the UserDbInterface class's loginMenu() method will be invoked, which presents the user with the log-in/registration menu. This menu will continue to loop until the user has successfully logged in or registered.
2. During registration, all user inputs that contain any of the following characters are rejected: “,” and “\$”. This is because these characters are used as delimiters in the DB_Users.txt file and, thus, would interfere with reading and parsing its content if such characters were allowed in a user's username, password, security question, or security answer.
3. Once successfully logged in or registered, the QueryInterface class will be invoked, which first reads and parses the Tables.txt file's contents into Table class objects. Then, will provide the user with a MySQL command-line interface that accepts CRUD operations.
4. QueryInterface's create() method accepts create commands in the following formats:
 - create table [table name] ([comma-separated attribute names and data type], primary key ([attribute]));
 - create table [table name] ([comma-separated attribute names and data type], primary key ([attribute]), foreign key ([attribute]) reference [table]([attribute]));

Note that each table must have one, and only one, primary key attribute. Because of the light-weight nature of this prototype, composite keys are not supported. Additionally, for similar reasons, a table may contain up-to one foreign key which must reference another table's primary key. In other words, a table can reference up-to one other table's prime attribute.

Furthermore, this prototype supports the following data types:

- String (i.e., “s”, “DBMS”)
- Integer (i.e., 5, 101)
- Decimal (i.e., 4.0, 6.7)
- Boolean (true, false)

5. QueryInterface's select() method accepts select commands in the following formats:

- select * from [table];
- select [comma-separated attribute names] from [table];
- select * from [table] where [attribute name] = [where value];
- select [comma-separated attribute names] from [table] where [attribute name] = [where value];

Due to the light-weight nature of the DBMS prototype, select commands support retrieval from only one table at a time. This is because implementing join functionality would require extensive effort which would go beyond the scope of a prototype. Additionally, a select command can have up-to one where clause.

6. QueryInterface's insert() method accepts insert commands in the following format:

- insert into [table] values([comma-separated values]);

Note that the insert statement must provide values for each of the table's attributes as defined by its schema. In other words, a user cannot insert only some of the row's values using [insert into [table] columns(..) values(..)], rather they must provide values for each of the columns in the order specified in the schema.

Furthermore, before inserting the values, the method will check whether each value supplied matches the corresponding attribute's datatype as defined in the table's schema. Additionally, the method also checks whether inserting the column would create a duplicate primary key, in which case the insertion is rejected.

7. QueryInterface's update() method accepts update commands in the following formats:

- update [table] set [comma-separated set clauses];
- update [table] set [comma-separated set clauses] where [attribute name] = [where value];

Note that this prototype accepts up-to one where clause during update statements.

Furthermore, the method will check that the updated value of a field matches the corresponding attribute's data type as defined in the table's schema. Any mismatches will result in the update statement being rejected. Similarly, if a row's prime attribute is being updated, the method will check whether the update will create duplicate primary keys, in which case the command will be rejected.

Finally, the method will check whether another table's foreign key references a value being updated, in which case the command will be rejected since it would violate the foreign key constraint. For example, if the row {1, 1.4, "s"} in T1 is being updated to change "s" to "x" while T2's row {true, "s", 4} has a foreign key reference to T1 row's "s" value, then this command will be rejected since T2's row would now have a foreign key reference to a value that no longer exists in T1.

8. QueryInterface's delete() method accepts delete commands in the following formats:

- delete from [table];
- delete from [table] where [attribute name] = [where value];

Note that since this is a light-weight prototype DBMS, only one where clause is accepted during delete statements. Similarly to the update() method, this method will check whether a value that is being referenced by another table's foreign key is being deleted, in which case the command will be rejected since it would violate the foreign key constraint.

9. The MySQL CLI will continue to loop and accept commands until the user enters "exit", at which point the program will terminate.
10. The prototype provides no ability for users to modify or drop tables as that was out of the scope of the assignment since the assignment instructions made no mention of such functionality.
11. The prototype consists of a single database shared between all users. This means that each user has full access to each table, and that no additional databases can be created. This is because the assignment scope, per the instructions, make no mention of such functionality.

Furthermore, the general workflow of the program's execution is outlined in the following UML sequence diagram which depicts how classes interact with one another:

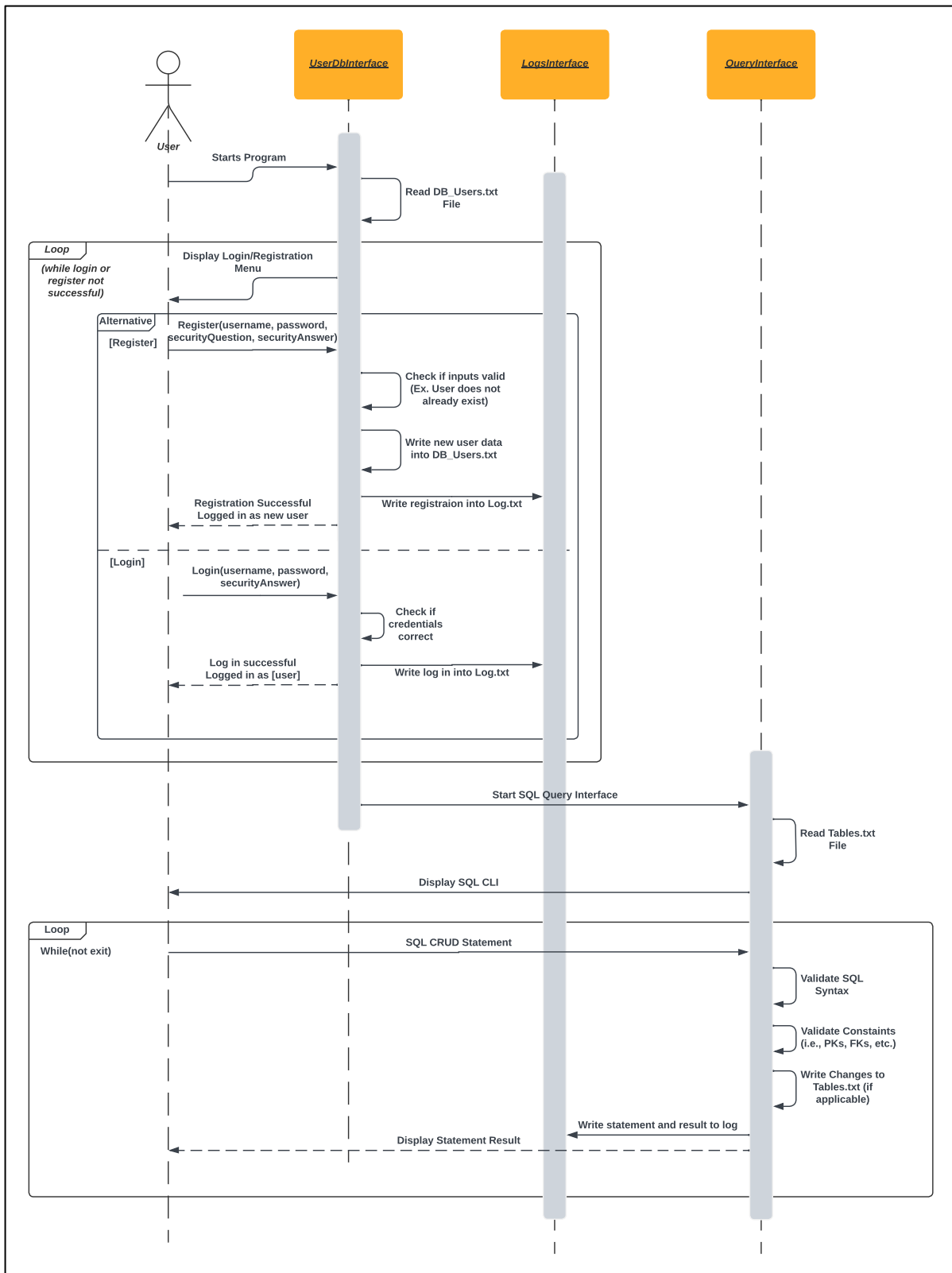


Fig 4. Submitted Prototype's UML Sequence Diagram [1]

Implementation (GitLab Repository):

The prototype's source code has been pushed to a GitLab repository which can be found at the following URLs:

https://git.cs.dal.ca/alhindi/csci5408_w23_b00919848_waleed_alhindi/-/tree/main/A1

OR

https://git.cs.dal.ca/alhindi/csci5408_w23_b00919848_waleed_alhindi/-/tree/main/ under the "A1" folder.

Additionally, this repository was shared with the following emails as instructed in the "Submission Guidelines" PowerPoint:

- saurabh.dey@dal.ca
- vs439755@dal.ca
- ar260217@dal.ca
- pr514457@dal.ca
- sh495601@dal.ca

Each of these emails were granted "reporter" access to the repository.

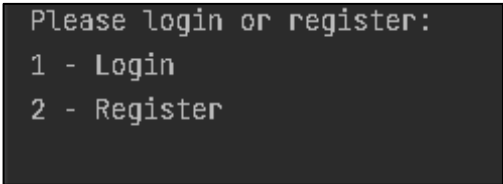
Test Cases:

To demonstrate the submitted prototype's functionality, the following demonstration test cases are outlined to verify the general workflows of the program:

1. Login

a. *Successful Login*

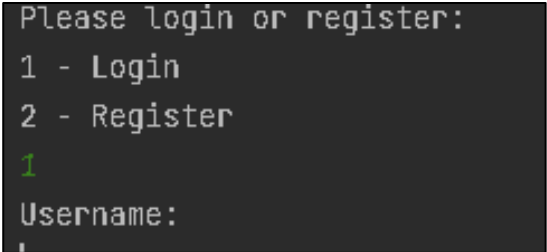
- Start the program
- Get presented with the login/registration menu



```
Please login or register:
1 - Login
2 - Register
```

Fig 5. Login/Registration Menu

- Input (1) for login



```
Please login or register:
1 - Login
2 - Register
1
Username:
.
```

Fig 6. Choose Login Option from Menu

- Provide correct (existing) credentials and granted access

```
Username:
waleed
Password:
pokemon
Answer for security question [sq1]:
sa1
Logged in as [waleed]
SQL>
```

Fig 7. Successful Login with Correct Credentials

- A log for the successful log-in is written to Log.txt file

```
2023.02.19.13.07.11]: [waleed] logged in.
```

Fig 8. Log-in Log Written in Log.txt File

b. *Unsuccessful Login*

- Start the program.
- Get presented with the login/registration menu.

```
Please login or register:
1 - Login
2 - Register
```

Fig 9. Login/Registration Menu

- Input (1) for login.

```
Please login or register:
1 - Login
2 - Register
1
Username:
```

Fig 10. Choose Login Option from Menu

- **[b.1]** Incorrect Username: Provide incorrect credentials (username does not exist). Denied access and login/registration menu loops.


```
Username:
dave
Access Denied: username does not exist!
Please login or register:
1 - Login
2 - Register
```

Fig 11. Access Denied Because of Non-Existent Username

- [b.2] Incorrect Password: Provide existing username, but incorrect password. Denied access and login/registration menu loops.

```
Username:
waleed
Password:
incorrectPass
Access Denied: incorrect password!
Please login or register:
1 - Login
2 - Register
|
```

Fig 12. Access Denied Because of Incorrect Password

- [b.3] Incorrect Security Answer: Provide existing username, correct password, but incorrect security answer. Denied access and login/registration menu loops.

```
Username:
waleed
Password:
pokemon
Answer for security question [sq1]:
incorrectAnswer
Access Denied: incorrect answer for security question!
Please login or register:
1 - Login
2 - Register
```

Fig 13. Access Denied Because of Incorrect Security Answer

2. Registration

a. *Successful Registration*

- Start the program
- Get presented with the login/registration menu
- Input (2) for registration

- Enter new, valid username, password, security question, and security answer. Registration successful and access granted

```
Username:
newUser
Password:
pass
Security Question:
question
Answer for [question]:
answer
Registration successful!
Logged in as [newUser]
```

Fig 14. Successful Registration

- Registration written to Log.txt file

```
[2023.02.19.13.34.51]: Registered user [newUser]
```

Fig 15. Registration Written to Log.txt File

b. *Unsuccessful Registration*

- Provide existing username. Registration failed, access denied, and log-in/registration menu loops

```
Username:
waLeed
Error: username already exists!
Please login or register:
1 - Login
2 - Register
```

Fig 16. Registration Failed Because of Existing Username

3. MySQL Create Command

a. *Successful Table Creation*

- Input valid MySQL create command. Table Successfully created

```
SQL>
create table T6( att1 int, att2 decimal, att3 boolean, primary key (att1));
Table [t6] was created successfully!
```

Fig 17. Valid Create Command Resulting in Table Creation

- New table data written to Tables.txt file

```
!t6
<
att2 decimal;
att1 int;
att3 boolean;
%
att1;
^
$
```

Fig 18. New Table Data Written to Tables.txt File with Custom Delimiters

- Create command logged into Log.txt file

```
[2023.02.19.13.45.36]: User: [waleed]
Command: [create table t6( att1 int, att2 decimal, att3 boolean, primary key (att1));]
Result: [Table [t6] was created successfully!]
```

Fig 19. Create Command and Outcome Logged into Log.txt File

b. *Unsuccessful Table Creation*

- **[b.1]** Table already exists: Input valid syntax create command, but table already exists. Create command rejected

```
create table T6(att1 int , att2 decimal, att3 boolean, primary key (att1));
SQL Create Error: Table [t6] already exists!
SQL>
```

Fig 20. Create Command Rejected Because Table Exists

- **[b.2]** No primary key defined: Input create command that does not define table's primary key. Create command rejected

```
create table T6(att1 int , att2 decimal, att3 boolean);
SQL Create Syntax Error!
SQL>
```

Fig 21. Create Command Rejected Because Primary Key Not Defined

- **[b.3]** Attribute data type not supplied: Input valid syntax create command, but one or more attributes do not define their data types. Create command rejected

```
create table T6(att1, att2 decimal, att3 boolean, primary key (att1));
SQL Create Syntax Error!
SQL>
```

Fig 22. Create Command Rejected Because Attribute(s) Datatype Not Defined

- **[b.4]** Unrecognized data type supplied: Input valid syntax create command, but one or more attribute data types are not recognized. Create command rejected

```
create table T8(att1 datetime, att2 decimal, att3 boolean, primary key (att1));
SQL Create Error: Attribute type [datetime] for attribute [att1] not recognized!
SQL>
```

Fig 23. Create Command Rejected Because Datatype Not Recognized

4. MySQL Insert Command

a. *Successful Row Insertion*

- Input valid insert command. Successfully inserted row into table

```
SQL>
insert into t6 values(1, 4.0 ,true );
Values inserted successfully into table [t6]!
SQL>
```

Fig 24. Successful Row Insertion into Table

- Table row data updated in Tables.txt file

```
!t6
<
att1 int;
att2 decimal;
att3 boolean;
%
att1;
^
$
#1,4.0,true
```

Fig 25. New Row Written into Tables.txt File

- Insertion logged in Log.txt file

```
[2023.02.19.14.14.48]: User: [waleed]
Command: [insert into t6 values(1, 4.0 ,true );]
Result: [Values inserted successfully into table [t6]!]
```

Fig 26. Insertion Logged into Log.txt File

b. *Unsuccessful Row Insertion*

- **[b.1]** Insert into non-existent table: Insertion rejected

```
insert into t99 values(1,2,3);
SQL Insert Error: Table [t99] does not exist!
SQL>
```

Fig 27. Insertion Rejected Because Non-Existent Table

- **[b.2]** Supplied value(s) datatype mismatch: Insertion rejected

```
insert into t6 values("s", 5.0, true);
SQL Insert Error: Table [t6] schema defines [att1] as int, but value supplied is not int!
SQL>
```

Fig 28. Insertion Rejected Because of Value Datatype Mismatch

- **[b.3]** Duplicate primary key: Insertion rejected

```
insert into t6 values(1, 5.5, false);
SQL Insert Error: Primary key value [1] already exists in table [t6]!
SQL>
```

Fig 29. Insertion Rejected Because of Duplicate PK

5. MySQL Select Command

a. *Successful Row(s) Retrieval*

- Input valid select statement. Displays retrieved row(s) data

```
SQL>
select * from t1;
a1 || a2 || a3 || a4 ||
"s" || 1.0 || 1 || true ||
"x" || 2.0 || 1 || false ||
```

Fig 30. Retrieved Rows from Select Command

```
SQL>
select a1, a3, a4 from t1 where a4=false;
a1 || a3 || a4 ||
"x" || 1 || false ||
```

Fig 31. Retrieved Rows from Select Command Containing Where Clause

b. *Unsuccessful Row(s) Retrieval*

- Input invalid select statement. Select statement rejected

```
SQL>
select & from t99;
SQL Select Syntax Error!
SQL>
select a5 from t1;
Select Error: Attribute [a5] does not exist in table [t1]!
SQL>
|
```

Fig 32. Examples of Select Statements Rejected Because of Syntax and Schema Errors

6. MySQL Update Command

a. *Successful Row(s) Update*

- Provide valid update command. Table updated successfully

```
SQL>
update t6 set att1=2 where att1=1;
Successfully updated 1 row(s) of table [t6]!
```

Fig 33. Successful Update Command

- Updated values written to Tables.txt file

```
!t6
<
att1 int;
att2 decimal;
att3 boolean;
%
att1;
^
+
#2,4.0,true
```

Fig 34. Row Values Updated in Tables.txt File

- Update is logged in Log.txt file

```
[2023.02.19.14.33.15]: User: [waleed]
Command: [update t6 set att1=2 where att1=1;]
Result: [Successfully updated 1 row(s) of table [t6]!]
```

Fig 35. Update Logged into Log.txt File

b. *Unsuccessful Row(s) Update*

```
update t6 set att6=3;
SQL Update Error: Table [t6] does not contains attribute [att6]!
SQL>
update t99 set att1=1;
SQL Update Error: Table [t99] does not exist!
SQL>
update t6 set att3="s";
SQL Update Error: Table [t6] defines attribute [att3] as boolean, but value supplied is not!
```

Fig 36. Examples of Invalid Update Statement Rejections

7. MySQL Delete Command

a. *Successful Row(s) Deletion*

- Input valid delete command. Row(s) deleted successfully

```
delete from t6 where att1=2;
Successfully deleted 1 row(s) from table [t6]!
```

Fig 37. Successful Row Deletion

- Tables.txt file updated to reflect deletion

```
!t6
<
att1 int;
att2 decimal;
att3 boolean;
%
att1;
^
$
```

Fig 38. Table.txt File Updated to Reflect Deletion

- Deletion logged into Log.txt file

```
[2023.02.19.14.38.48]: User: [waleed]
Command: [delete from t6 where att1=2;]
Result: [Successfully deleted 1 row(s) from table [t6]!]
```

Fig 39. Deletion Logged into Log.txt File

b. *Unsuccessful Row(s) Deletion*

```
SQL>
delete from t99;
SQL Delete Error: Table [t99] does not exist!
SQL>
delete from t6 where att99=false;
SQL Delete Error: Table [t6] does not contain attribute [att99]!
SQL>
delete from t1 where a1="s";
SQL Delete Error: Cannot delete row(s) because the table [t4] has a foreign key reference to [t1]'s row [{"s", 1.0, 1, true}]!
```

Fig 40. Example Delete Statement Rejections

Conclusion:

In conclusion, the submitted prototype constitutes a light-weight relational DBMS that stores data using txt files utilizing custom delimiters to parse data read from those files. The prototype supports multiple users which need to be authenticated before being granted access to the database. Once authenticated, users can access database where they can create, insert, update, delete, and select from tables. Due to the light-weight nature of this prototype, some design decisions had to be made which impose limitations on some of the prototype's DBMS functionalities (see section "Design Decisions" for more detail). With that being said, the prototype still provides a robust MySQL interface which implements rudimentary primary key and foreign key constraints, attribute datatype verification, where clauses, two-factor user authentication, and full logging capabilities. Moreover, by implementing this prototype, a deeper understanding of the inner workings of DBMS and how they store end-user data and meta-data was achieved. As such, a meaningful appreciation was gained towards the complexities of modern DBMS software, as they provide such an abstracted and user-friendly interface to such complex storage structures.

References:

- [1] "Intelligent Diagramming," *LucidChart*. [Online], Available: <https://www.lucidchart.com/> [Accessed: February 16, 2022].