

Background Research

Data Structures used:

- 1) Arrays are chosen for their simplicity and efficiency in handling linear data. Arrays provide fast access to elements using indices, which is beneficial for quick lookups and modifications. They are also memory-efficient when dealing with a fixed size of elements.
- 2) HashMap is used to provide efficient storage and retrieval of data through key-value pairs. They offer constant time complexity for insertions, deletions, and lookups, making them highly efficient for managing data where quick access and updates are required. This is particularly useful for operations like database transactions, where data needs to be accessed and modified frequently. Hash maps also handle dynamic resizing more efficiently compared to arrays, allowing the data structure to grow as needed without a significant performance impact.

Custom File Format:

Here, we use a simple plain text file to store the data and metadata for this TinyDB project. Different delimiters are used to differentiate the data from each other. For example, "@" is used to denote the table name, and "?" is used to indicate each column and its properties. Additionally, "#" is used to separate multiple data items in one line. For instance, to store a user ID and its password, we store them as userID#password in one line.

```
1      @sample_table
2      ?id#INT#0#false#false#false
3      ?name#VARCHAR#0#false#false#false
```

Figure 1: Delimiter use

Architecture Diagram

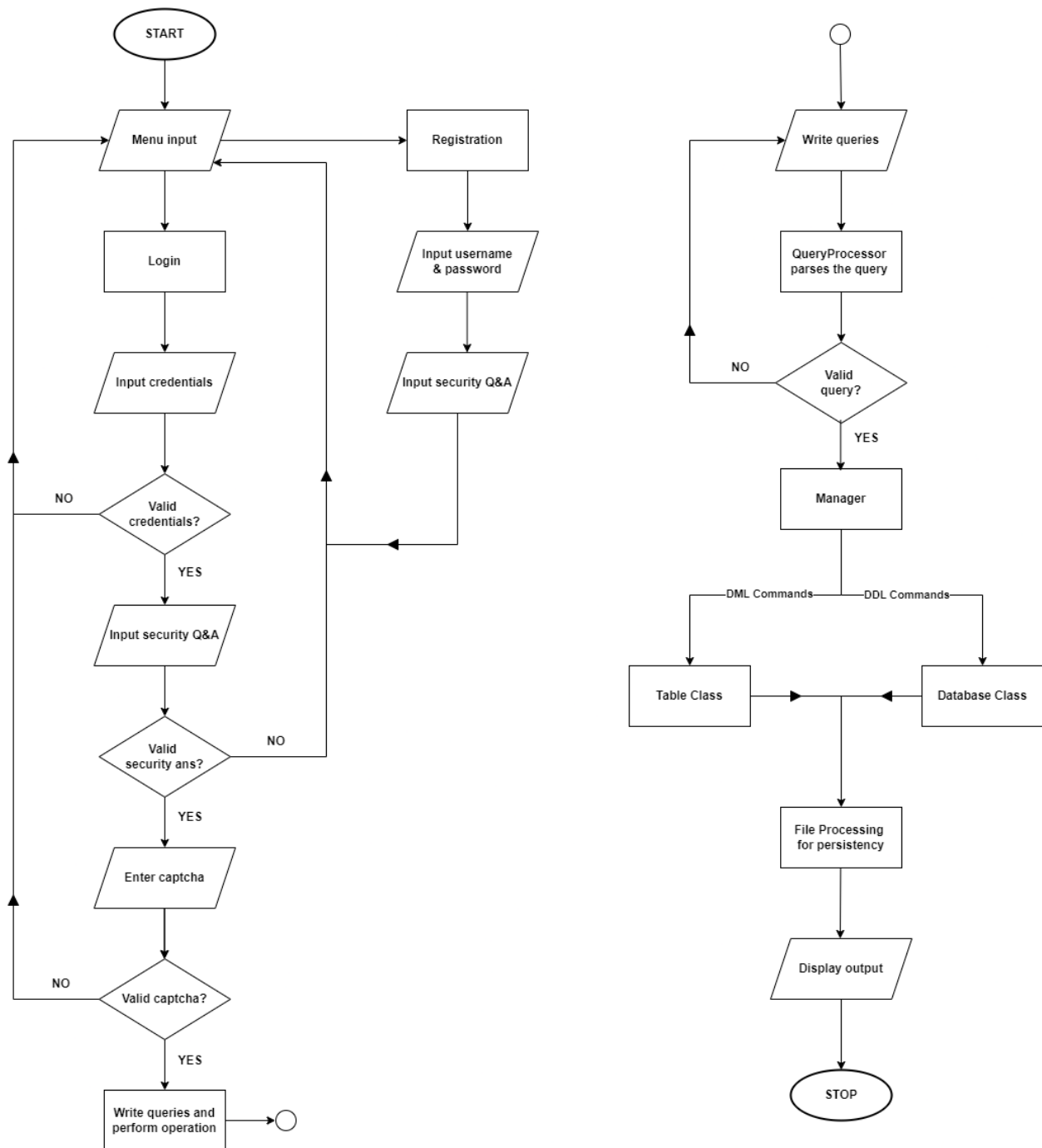


Figure 2: Application workflow

Pseudocode

1) Create Database:

```
function createDatabase(dbName):  
    if databaseExists(dbName):  
        return False  
    createDirectory(dbName)  
    return True
```

```
function databaseExists(dbName):  
    return directoryExists(dbName)
```

```
function createDirectory(dbName):  
    mkdir(dbName)
```

2) Create Table:

```
function createTable(dbName, tableName, attributes):  
    if databaseNotSelected(dbName):  
        return False  
    if tableExists(dbName, tableName):  
        return False  
    writeTableSchema(dbName, tableName, attributes)  
    return True
```

3) Use Database:

```
function processUseQuery(parts):  
    if length(parts) < 2 or upper(parts[1]) != "DATABASE":  
        print("Invalid USE DATABASE query format.")  
    if dbName not in databases:  
        return False  
    content = read("DataBase_FILE_PATH/" + dbName + "/" + dbName + ".txt")  
    for each currLine in content:  
        if startsWith(currLine, "@"):  
            add(tables, trim(substring(currLine, 1)))  
    return True
```

4) Select Query:

```
function processSelectQuery(query) {  
    parts = split(query, "\\s+FROM\\s+", 2)  
    if length(parts) != 2  
        print("Invalid SELECT query format.")  
    selectPart = trim(substring(parts[0], length("SELECT"))).trim()  
    if selectPart == ""
```

```

        columns = null
    else
        columns = split(selectPart, ",")
    fromPart = trim(parts[1])
    if contains(upper(fromPart), "WHERE") {
        tableAndConditionParts = split(fromPart, "\\s+WHERE\\s+", 2)
        tableName = trim(tableAndConditionParts[0])
        conditionParts = split(tableAndConditionParts[1], "=", 2)
        if length(conditionParts) != 2
            print("Invalid WHERE clause format.")
        conditionColumn = trim(conditionParts[0])
        conditionValue = trim(replaceAll(conditionParts[1], "^'(.*)'$", "$1"))
    else
        tableName = trim(fromPart)
        conditionColumn = null
        conditionValue = null
    print("Table Name: " + tableName)
    print("Columns: " + columns)
    print("Condition Column: " + conditionColumn)
    print("Condition Value: " + conditionValue)

```

5) Insert Query:

```

function processInsertQuery(query):
    parts = split(query.trim(), " ")
    if length(parts) < 5 or upper(parts[0]) != "INSERT" or upper(parts[1]) != "INTO":
        print("Invalid INSERT query format.")
    columnsPart = substring(query, indexOf(query, "(") + 1, indexOf(query, ")")).trim()
    valuesPart = substring(query, lastIndexOf(query, "(") + 1, lastIndexOf(query, ")")).trim()
    columns = split(columnsPart, ",")
    values = split(valuesPart, ",")

    // Clean values
    for i = 0 to length(values) - 1:
        values[i] = trim(replaceAll(values[i], "^'(.*)'$", "$1"))

    // Validate database and table existence
    if currentDatabase == null or isBlank(currentDatabase)
        return False
    if not contains(tables, tableName)
        return False

    // Validate and convert values
    for i = 0 to length(values) - 1:
        value = values[i]

```

```

attribute = attributes[i]
if not validateAndConvert(value, attribute):
    return False

// Check unique constraints
existingRecords = readRecords(tableName)
for record in existingRecords:
    for j = 0 to length(positions) - 1:
        if record[positions[j]] == uniqueValues[j]:
            print "Unique constraint violated!"
            return False

// Write to datastore
content = join(values, "#")
return write("DataStore/" + currentDatabase + "/" + tableName + ".txt", [content], False)

```

6) Update Query:

```

function processUpdateQuery(query):
    setIndex = indexOf(upper(query), "SET")
    whereIndex = indexOf(upper(query), "WHERE")
    if setIndex == -1 or whereIndex == -1 or setIndex >= whereIndex:
        print("Invalid UPDATE query format.")

    setPart = trim(substring(query, setIndex + 3, whereIndex))
    setParts = split(setPart, "=", 2)

    if length(setParts) != 2:
        print("Invalid SET clause format.")

    column = trim(setParts[0])
    value = trim(replaceAll(setParts[1], "^'(.*)'$", "$1"))

    wherePart = trim(substring(query, whereIndex + 5))
    whereParts = split(wherePart, "=", 2)

    if length(whereParts) != 2:
        print("Invalid WHERE clause format.")

    conditionColumn = trim(whereParts[0])
    conditionValue = trim(replaceAll(whereParts[1], "^'(.*)'$", "$1"))

// Validate database and table existence
if currentDatabase == null or isBlank(currentDatabase):
    return False

```

```

if not contains(tables, tableName):
    return False

// Update operation
loadTableData(tableName, currentDatabase)
for each row in rows:
    if conditionColumn is null or row[findColumnIndex(conditionColumn)] == conditionValue:
        convertValue(attributes[columnIndex], value)

        if attributes[columnIndex].isUnique():
            for each otherRow in rows:
                if otherRow != row and otherRow[columnIndex] == value:
                    return False
            row[columnIndex] = value
            updated = True
saveTableData(tableName, currentDatabase)

```

7) Delete Query:

```

function processDeleteQuery(query):
    parts = split(trim(query), "\\s+")

    if length(parts) < 4 or upper(parts[0]) != "DELETE" or upper(parts[1]) != "FROM":
        print("Invalid DELETE query format.")

    tableName = parts[2]

    conditionColumn = null
    conditionValue = null

    if contains(query, "WHERE"):
        wherePart = trim(substring(query, indexOf(query, "WHERE") + 5))
        whereParts = split(wherePart, "=")

        if length(whereParts) != 2:
            throw IllegalArgumentException("Invalid WHERE clause format.")

        conditionColumn = trim(whereParts[0])
        conditionValue = trim(whereParts[1])

// Validate database and table existence
if currentDatabase == null or isBlank(currentDatabase):
    return False
if not contains(tables, tableName):
    return False

```

```

// Delete operation
loadTableData(tableName, currentDatabase)
conditionIndex = findColumnIndex(conditionColumn)
if conditionIndex == -1:
    return False
for iterator = iterator(rows):
    row = next(iterator)
    if row[conditionIndex] == conditionValue:
        remove(iterator)
        deleted = True
if deleted:
    saveTableData(tableName, currentDatabase)
    return True
else:
    return False

```

8) Drop Query:

```

function processDropQuery(query):
    parts = split(trim(query), "\\s+")

    if length(parts) < 3 or upper(parts[0]) != "DROP" or upper(parts[1]) != "TABLE":
        return False

    tableName = parts[2]

    // Validate database and table existence
    if currentDatabase == null or isBlank(currentDatabase):
        return False
    if not contains(tables, tableName):
        return False

    // Check if table file exists
    tableFilePath = "DataBase_FILE_PATH/" + currentDatabase + "/" + tableName + ".txt"
    if not isDirectoryFileExist(tableFilePath):
        return False

    // Read database file content
    content = read("DataBase_FILE_PATH/" + currentDatabase + "/" + currentDatabase + ".txt")

    // Identify and mark lines to delete
    deleteLineNumbers = []
    for i = 0 to length(content) - 1:

```

```

if startsWith(content[i], "@" + tableName):
    add(deleteLineNumbers, i)
    while i < length(content) and startsWith(content[i], "?"):
        add(deleteLineNumbers, i)
        i = i + 1
    break

// Update database file and delete table file
if write("DataBase_FILE_PATH/" + currentDatabase + "/" + currentDatabase + ".txt", content,
True):
    deleteDirectoryOrFile(tableFilePath)
    remove(tables, tableName)
    return True
else:
    return False

```

9) Authentication

- **function registerUser(user)**
 - if user.getUserID() exists in registeredUsers
 - return false
 - encode user credentials and security questions into content list
 - write content to USER_PROFILE.txt file
 - add user to registeredUsers
 - return true if write operation successful, else false
- **function loginUser(userID, password)**
 - if userID does not exist in registeredUsers
 - return false
 - if hashed password for userID matches stored hashed password
 - if verifySecurityQuestion(user)
 - return true
 - else
 - return false
 - else
 - return false
- **function verifySecurityQuestion(user)**
 - randomly select any one of the security question from user's list
 - if user-provided answer matches stored answer for selected question
 - return true
 - else
 - return false

- **function hashPassword(password)**
generate MD5 hash of password
encode hash using Base64
return encoded hash

- **function isUserExist(userID)**
if registeredUsers is empty
 loadUsers()
iterate through registeredUsers
 if user.getUserID() equals userID
 return true
return false

- **function loadUsers()**
if isUserProfileFileLoaded is true
 return registeredUsers
clear registeredUsers
read content from USER_PROFILE.txt file
decode content into users list
set isUserProfileFileLoaded to true
return registeredUsers

Testcases

Register & Authentication:

1) Login with a user that does not exist:

```
Menu:
1. Login
2. Register
3. Exit
1
Enter username: testuser
Enter password: testpass
User Doesn't Exist!
```

Figure 3: Login with non-existent user details

- User does not exist in user's file

1	prashanth#wtoRtzsX1+TjENio10SknA==
2	@Cat?#orange
3	

Figure 4: User_Profile.txt

2) Register using a username that already exists:

```
Enter new username: prashanth
Enter password: daw
Enter security question: dwadaw
Enter answer: dwa
Do you want to add another security question? (yes/no):
no
UserID already exist!
```

Figure 5: User registration with existing user values

- Users file stays the same.

1	prashanth#wtoRtzsX1+TjENio10SknA==
2	@Cat?#orange
3	

Figure 6: User_Profile.txt

3) Register using a new username:

```
2
Enter new username: testuser
Enter password: testpass
Enter security question: testquestion
Enter answer: yes
Do you want to add another security question? (yes/no):
no
```

Figure 7: User registration with new user values

- User_Profile.txt got updated as shown below

```
1 prashanth#wtoRtzsX1+TjENio10SknA==
2 @Cat?#orange
3 testuser#F5rUXGzizy5fPECniEgRugQ==
4 @testquestion#yes
5 |
```

Figure 8: User_Profile.txt (updated)

4) Login using an existing username:

```
1
Enter username: testuser
Enter password: testpass
testquestion?
yes
Captcha: #o82q
Please enter above captcha: #o82q
Login successful!
```

Figure 9: Login with existent user details

5) Login using an existing username but with wrong password:

```
1
Enter username: testuser
Enter password: wrongpass
Incorrect Credentials!
```

Figure 10: Invalid credentials prompt in Login

6) Login with wrong security answer:

```
1
Enter username: testuser
Enter password: testpass
testquestion?
no
Wrong Security Answer!
```

Figure 11: Invalid security answer prompt in Login

7) Login using the wrong captcha combination:

```
1
Enter username: testuser
Enter password: testpass
testquestion?
yes
Captcha: ojszs
Please enter above captcha: opgm
Invalid captcha
```

Figure 12: Invalid captcha prompt in Login

Queries:

1) Creating and using a database:

1	sample
2	
3	💡

Figure 13: Metadata.txt before creating a new DB

```
Enter your SQL queries (type 'EXIT' to end):
SQL> CREATE DATABASE test;
SQL> USE DATABASE test;
```

Figure 14: Creating and using a DB

1	sample
2	test
3	

Figure 15: Metadata.txt after creating the test database

```

v DataStore
  > db
  > sample
  metadata.txt
  User_Profile.txt

```

Figure 16: DataStore directory before creating the test database

```

v DataStore
  > db
  > sample
  v test
    test.txt
  metadata.txt
  User_Profile.txt

```

Figure 17: DataStore directory after creating the test database

```

> static members of Main
v manager = {Manager@1106}
  > databases = {ArrayList@1292}
  > currentDatabase = "test"
  > database = {Database@1294}
  > DataBase_FILE_PATH = "DataStore"

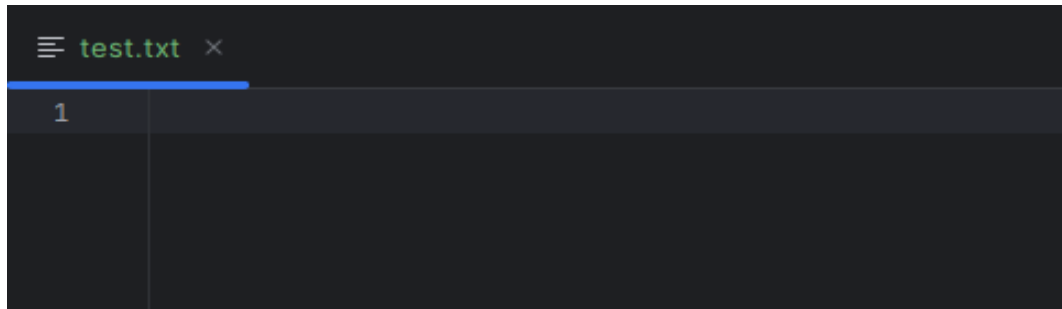
```

Figure 18: The test database is chosen after the "USE" command

2) Creating a table:

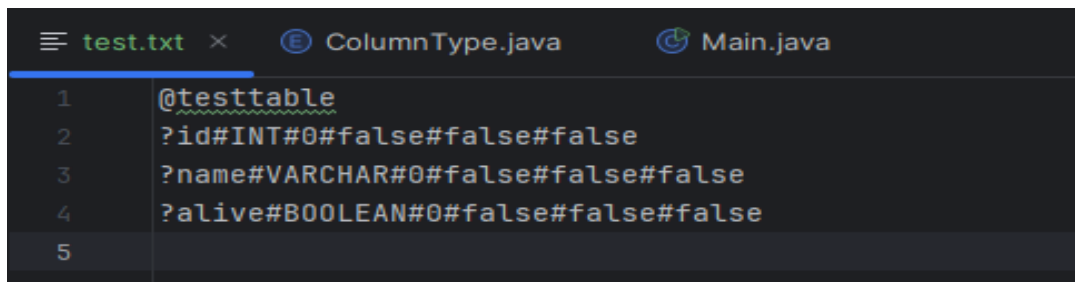
```
Enter your SQL queries (type 'EXIT' to end):
SQL> USE DATABASE test;
SQL> CREATE TABLE testtable (id INT,name VARCHAR, alive BOOLEAN);
SQL>
```

Figure 19: Creating a table



The screenshot shows a text editor window titled 'test.txt'. The file is empty, with only a line number '1' visible in the left margin.

Figure 20: test.txt database file is empty before creating a table



The screenshot shows a text editor window with 'test.txt' selected. The file contains the following text:

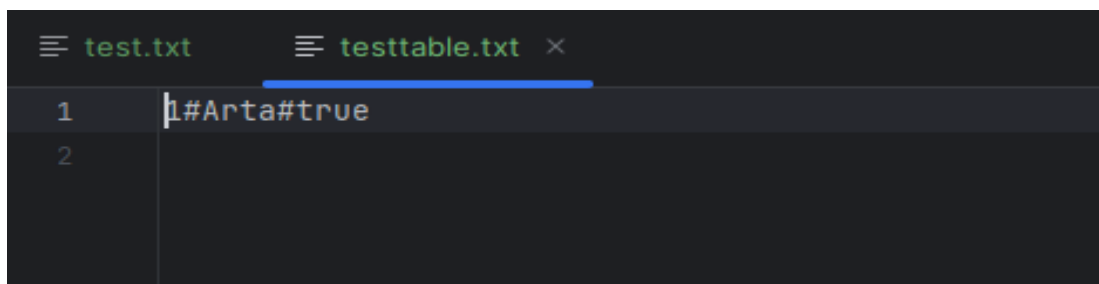
	@testtable
1	
2	?id#INT#0#false#false#false
3	?name#VARCHAR#0#false#false#false
4	?alive#BOOLEAN#0#false#false#false
5	

Figure 21: test.txt file is updated after the table creation

3) Insert into the table:

```
SQL> INSERT INTO testtable VALUES (1,Arta,true);
SQL>
```

Figure 22: Inserting values into the table



The screenshot shows a text editor window with 'testtable.txt' selected. The file contains the following text:

	1#Arta#true
1	
2	

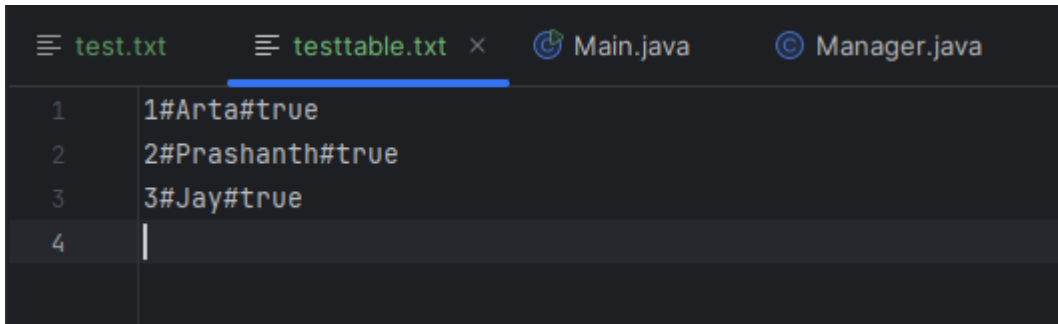
Figure 23: New record is added to the table txt file

4) Add a table with a name that already exists in the database:

```
SQL> CREATE TABLE testtable (id INT,name VARCHAR, alive BOOLEAN);
Table 'testtable' already exists in database 'test'.
```

Figure 24: Table already exists error

5) Select from a table without a WHERE clause:



1	1#Arta#true
2	2#Prashanth#true
3	3#Jay#true
4	

Figure 25: Records in table txt file

```
SQL> SELECT * FROM testtable;
Query result:
id: 1   name: Arta   alive: true
id: 2   name: Prashanth alive: true
id: 3   name: Jay    alive: true
```

Figure 26: Displaying output from SELECT query

6) Select from a table with a WHERE clause:

```
SQL> SELECT name FROM testtable WHERE id = 1;
Query result:
name: Arta
```

Figure 27: Displaying output from SELECT query (with WHERE clause)

7) Update a table with a WHERE clause:

```
SQL> UPDATE testtable SET alive = false WHERE name = Arta;
Table 'testtable' updated successfully.
SQL>
```

Figure 28: Updating a table

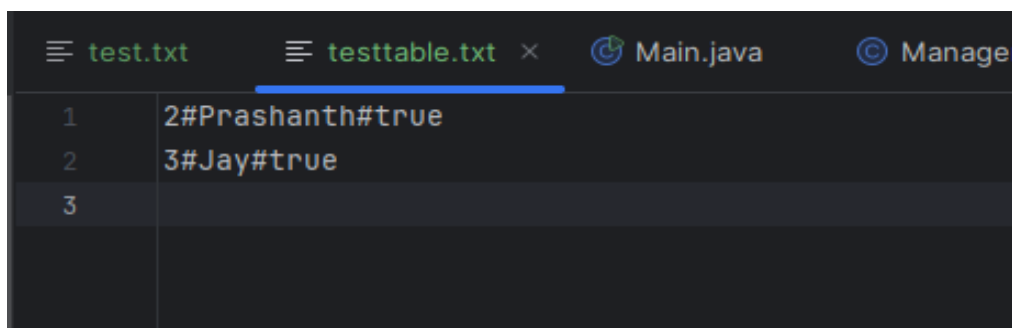
```
SQL> SELECT * FROM testtable;
Query result:
id: 1   name: Arta   alive: false
id: 2   name: Prashanth alive: true
id: 3   name: Jay    alive: true
```

Figure 29: Displaying output after updation

8) Deleting a record from the table:

```
SQL> DELETE FROM testtable WHERE name = Arta;
testtable
name
Arta
Row(s) deleted from table 'testtable'.
SQL> SELECT * FROM testtable;
Query result:
id: 2   name: Prashanth alive: true
id: 3   name: Jay    alive: true
```

Figure 30: Displaying output after deletion



	testtable.txt
1	2#Prashanth#true
2	3#Jay#true
3	

Figure 31: The record is deleted from the table.txt file after the operation

9) Dropping a table:

```
SQL> DROP TABLE testtable;
Directory deleted successfully
Table 'testtable' dropped successfully.
```

Figure 32: Success prompt after dropping a table

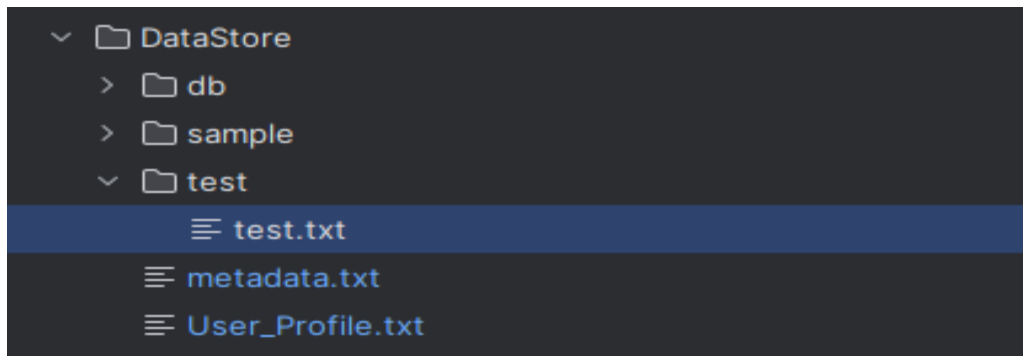


Figure 33: Table text file is removed from the Datastore directory

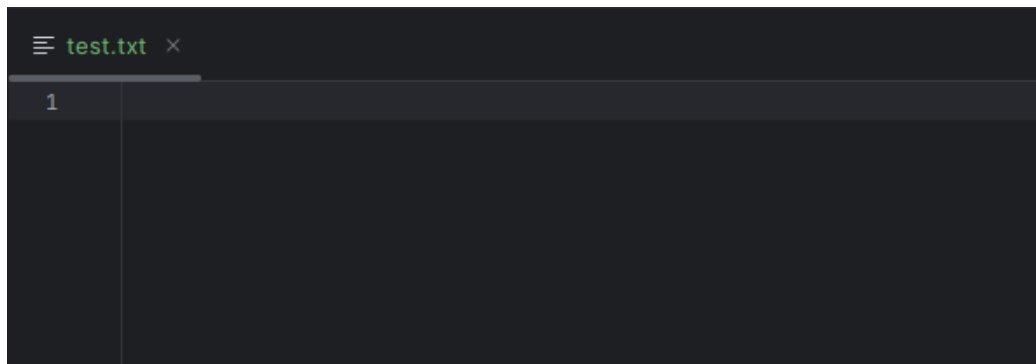


Figure 34: Table information is removed from the database .txt file

10) SELECT, UPDATE & DELETE from a table that does not exist in the database:

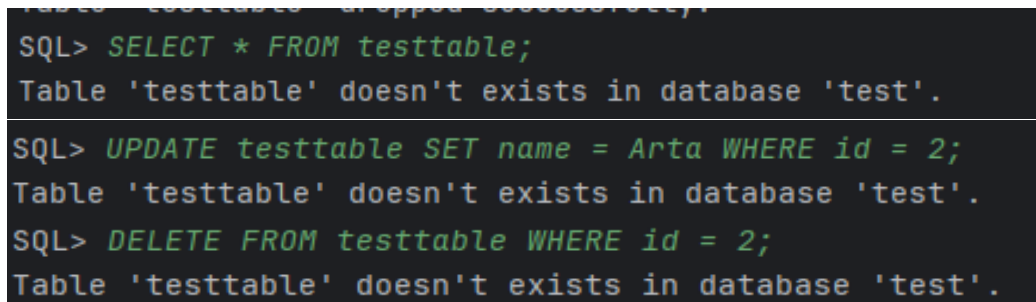


Figure 35: Doesn't exist in the database prompt for CRUD operation