CCP6114 Programming Fundamentals
Trimester: 2430

Tutorial: T16L
Group 4

Assignment title: Light Mariadb Interpreter
Submitted to: Mr. Sean Siea Chin Chuan

No.	Student Name	Student ID
1.	Wong Kai Shen	243UC247DH
2.	Teh Shin Rou	243UC2467K
3.	Nyiam Zi Qin	243UC2466T
4.	Yen Ming Jun	243UC246NQ

Submission Dates
Assignment milestone 1: 25 Dec 2024, Wed, 5:00pm, Week 8
Assignment milestone 2: 26 Jan 2024, Sun, 5:00pm, Week 12

Acknowledgment

First and foremost, we would like to send our utmost gratitude to Multimedia University for giving us an opportunity to further my studies in my desired area.

Our supervisor, Mr Sean Siea Chin Chuan has given his all-out support and motivation on our thesis subject.

To our families who have been encouraging us and supportive of our chosen path.

Especially to Multimedia University(MMU), which has been our greatest university, Thank you

Finally, to all the people who have helped us not only with our thesis but bringing the best memories throughout this semester.

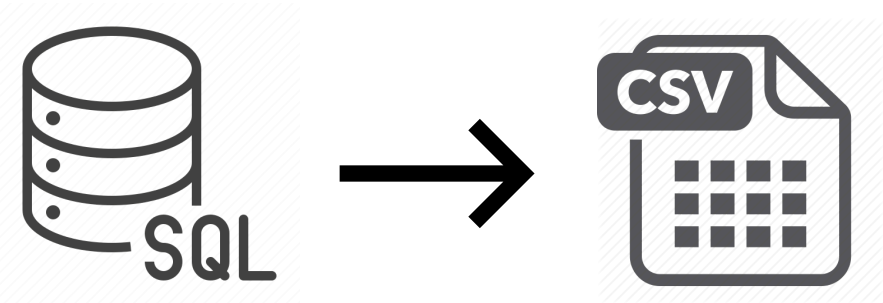
Thanks for everything.

Table of Contents

1.0 Literature Review

1.1 Introduce the Basic Idea of the Program

The ***Light MariaDB Interpreter*** assignment aims to provide a simplified simulation of a database management system using C++. This project involves building a program capable of processing and executing SQL-like commands to manage a lightweight database. The primary functionalities include creating databases and tables, inserting and updating records, and retrieving data through query commands. Through this assignment, we explore key programming concepts such as file handling, class design, data structures (e.g., ***vectors***), and error handling in C++. The project emphasizes understanding database principles and applying them programmatically, offering a practical approach to solving real-world problems using programming fundamentals. This document outlines the design, implementation, testing, and functionality of the ***Light MariaDB Interpreter*****,** showcasing how it meets the assignment's requirements.



1.2 History

Databases have been an integral part of managing, storing, and retrieving data since the advent of computing. The origins of modern database systems date back to the 1960s, with IBM's development of IMS (Information Management System), which used hierarchical structures to manage data. As data storage requirements evolved, relational databases, as conceptualized by Edgar F. Codd in 1970, introduced the relational model, offering a structured way to organize and retrieve data using tables, rows, and columns.

MariaDB, a community-developed fork of MySQL, emerged in 2009 as a response to Oracle Corporation's acquisition of MySQL. It was created to ensure the continuity of MySQL as an open-source database solution. Over the years, MariaDB has gained prominence for its high performance, scalability, and robust feature set, making it a preferred choice for developers and organizations. The lightweight MariaDB Interpreter is a simplified implementation inspired by these principles, focusing on essential database operations such as creating tables, inserting records, and executing queries in a controlled environment.

1.3 Impact

Databases, including MariaDB, have revolutionized the way information is managed and accessed. Their impact can be seen across industries such as finance, healthcare, education, and e-commerce. The demand for lightweight database systems arises from scenarios requiring simplified solutions for specific applications, such as embedded systems, education, or small-scale projects.

By developing a lightweight MariaDB Interpreter, students and developers can gain an understanding of core database functionalities while exploring programming concepts like data structures and file management. This approach bridges theoretical knowledge and practical implementation, fostering skills essential for building real-world applications. Furthermore, such tools serve as an entry point to understanding complex database management systems, paving the way for further exploration and innovation in the field.

1.4 Software Prerequisites

Check out this github repository for the source code:
https://github.com/wongkaishen/CCP6114-Programming-Fundamentals-Trimester-2430

To run and compile the Light MariaDB Interpreter project, the following software and tools are required:

1. Code::Blocks IDE

The project is developed using Code::Blocks, a free, open-source Integrated Development Environment (IDE). Code::Blocks is widely used for C++ programming and provides necessary tools for efficient development, including an editor, debugger, and compiler.

- **Version**: 20.03 or later
- **Download Link**: Code::Blocks Official Website

2. C++ Compiler

Code::Blocks integrates with several C++ compilers, and it is essential to have a working compiler for building the project. The following are compatible compilers:

- **MinGW (Minimalist GNU for Windows)**: For Windows users
- **GCC**: For Linux/macOS users
- **Clang**: Compatible with macOS and Linux systems

Required Compiler Version:

- **GCC**: Version 9.0 or later
- **Clang**: Version 10.0 or later
- **MinGW**: Version 8.1 or later

Ensure that the compiler is correctly configured within Code::Blocks for seamless compiling and debugging.

2.0 Main Function:

```
int main() {
    string filein = "fileinput1.txt";// <=== Please change the file input here
    ifstream inFile(filein);
    ofstream outFile;
    string line;
    vector<string> headers;
    vector<vector<string>> data;
    bool inCreateTable = false;
    string accumulatedColumns;
    string tableName;

    if (!inFile) {
        cout << "Please check if the file exist in your directory!" << endl;
        return 1;
    }

    while (getline(inFile, line)) {
        if (line.find("CREATE ") != string::npos && line.find("TABLE") == string::npos) {
            processCreateFile(line, outFile);
        }   else if (line.find("CREATE TABLE") != string::npos || inCreateTable) {
            //Process CREATE TABLE
            processCreateTable(line, headers, accumulatedColumns, inCreateTable, tableName);
        } else if (line.find("VALUES") != string::npos) {
            //Process INSERT VALUE
            processValues(line, data);
        } else if (line.find("UPDATE") != string::npos) {
            //Process UPDATE statements
            processUPDATE(line, headers, data);
        } else if (line.find("DELETE FROM") != string::npos) {
            //Process DELETE statements
            processDelete(line, data, headers);
        } else if (line.find("DATABASES;") != string::npos) {
            processDatabases(line, filein);
        } else if (line.find("SELECT COUNT(*) FROM") != string::npos) {
        // Handle SELECT COUNT(*) FROM queries
        processCount(line, tableName, data, outFile);
        } else if(line.find("SELECT * FROM") != string::npos) {
        //Process Select From statements
        processSelectQuery(line, headers, data, outFile, tableName);
        }
    }
    // Output TABLES
    processTables(outFile, tableName);


    inFile.close();
    outFile.close();
    return 0;
}
```

2.1 File Initialization

```
string filein = "fileinput1.txt";
ifstream inFile(filein);
ofstream outFile;
```

```
string filein = "fileinput1.txt";
```

A `string` variable named `filein` is created and assigned the value `"fileinput1.txt"`.

This specifies the name of the input file that the program will attempt to open and read.

```
ifstream inFile(filein);
```

Creates an input file stream object named `inFile` and opens the file `"fileinput1.txt"` for reading.

```
ofstream outFile;
```

Declares an output file stream object named `outFile` but does not immediately associate it with a file.

2.2 Variable Setup

```
vector<string> headers;
vector<vector<string>> data;
bool inCreateTable = false;
string accumulatedColumns;
string tableName;
```

Declare key variables:

vector<string> headers: Stores table column headers.
vector<vector<string>> data: Stores rows of table data.
bool inCreateTable: Tracks if the current block is a CREATE TABLE.

string accumulatedColumns: Temporarily holds column definitions.
string tableName: Stores the current table's name.

2.3 Input File Parsing

```
while (getline(inFile, line)) {
    if (line.find("CREATE ") != string::npos && line.find("TABLE") == string::npos) {
        processCreateFile(line, outFile);
    }   else if (line.find("CREATE TABLE") != string::npos || inCreateTable) {
        //Process CREATE TABLE
        processCreateTable(line, headers, accumulatedColumns, inCreateTable, tableName);
    } else if (line.find("VALUES") != string::npos) {
        //Process INSERT VALUE
        processValues(line, data);
    } else if (line.find("UPDATE") != string::npos) {
        //Process UPDATE statements
        processUPDATE(line, headers, data);
    } else if (line.find("DELETE FROM") != string::npos) {
        //Process DELETE statements
        processDelete(line, data, headers);
    } else if (line.find("DATABASES;") != string::npos) {
        processDatabases(line, filein);
    } else if (line.find("SELECT COUNT(*) FROM") != string::npos) {
    // Handle SELECT COUNT(*) FROM queries
    processCount(line, tableName, data, outFile);
    } else if(line.find("SELECT * FROM") != string::npos) {
    //Process Select From statements
    processSelectQuery(line, headers, data, outFile, tableName);
    }
}
// Output TABLES
processTables(outFile, tableName);
```

CREATE: Calls `processCreateFile` to create a file output based on the file input.

CREATE TABLE: Calls `processCreateTable` to handle the creation of a table, extracts the table name, and collects the column headers defined in the `CREATE TABLE` statement.

VALUES: Calls `processValues` to process the values that need to be inserted into the table then It extracts and parses the values and stores them in the `data` structure.

UPDATE : Calls `processUPDATE` to updates the specified column(s) with the new values

DELETE FROM: Calls `processDELETE` to process the `DELETE` operation by checking each row in the table then removed from the table.

DATEBASES: Calls processDatabases to write the table name to the output file and terminal

SELECT * FROM: Calls `processSelectQuery` to output the headers and data rows to the output file and terminal

SELECT COUNT(*): Calls `processCount` to count the total row of data rows

2.4 File Cleanup

```
inFile.close();
outFile.close();
```

Finally, the input and output files are closed using **close()** to free up resources.

3.0 CREATE FILE Rows:

```
void processCreateFile(const string& line, ofstream& outFile) {
    size_t start = line.find("CREATE") + 7; // Find "CREATE" and move past it
    size_t end = line.find(";");           // Find the semicolon
    if (start != string::npos && end != string::npos) {
        string fileName = line.substr(start, end - start);
        fileName.erase(remove(fileName.begin(), fileName.end(), ' '), fileName.end()); // Remove spaces

        // Close the current output file and open the new one
        outFile.close();
        outFile.open(fileName);
        if (!outFile) {
            cerr << "Error: Unable to create file " << fileName << endl;
            return;
        }
    }
}
```

From Main Function :

```
if (line.find("CREATE ") != string::npos && line.find("TABLE") == string::npos) {
    processCreateFile(line, outFile);
```

This is the function key to run CREATE function. The processCreateTable function is executed in main.

3.1 Detecting CREATE Statement:

```
size_t start = line.find("CREATE") + 7; // Find "CREATE" and move past it
size_t end = line.find(";");           // Find the semicolon
if (start != string::npos && end != string::npos) {
    string fileName = line.substr(start, end - start);
    fileName.erase(remove(fileName.begin(), fileName.end(), ' '), fileName.end()); // Remove spaces
```

The function `processCreateFile` detects the presence of the `CREATE` command within the input line. It starts by finding the position of the keyword "CREATE" in the line and moves past it. It then finds the position of the semicolon (`;`) to mark the end of the command.

Key Operations:

`start = line.find("CREATE") + 7`: Finds the starting position of the file name after the `CREATE` keyword.

`end = line.find(";")`: Finds the semicolon which marks the end of the line.

3.2 Extracting and Handling File Name:

Once the positions of "CREATE" and ";" are located, the code extracts the file name from the substring between these two positions. After extracting the file name:

Extra spaces are removed by using `erase` and `remove`, ensuring that only the file name remains.

Key Operations:

`string fileName = line.substr(start, end - start)`: Extracts the file name.

`fileName.erase(remove(fileName.begin(), fileName.end(), ' '), fileName.end())`: Removes any spaces from the file name.

3.3 Opening and Closing Files:

```
// Close the current output file and open the new one
outFile.close();
outFile.open(fileName);
if (!outFile) {
    cerr << "Error: Unable to create file " << fileName << endl;
    return;
}
```

The code then closes any previously opened output file and attempts to open a new file using the extracted file name.

Key Operations:

- `outFile.close()`: Closes the previously opened file.
- `outFile.open(fileName)`: Opens the newly created file with the specified file name.

If the file cannot be opened, an error message is displayed using `cerr`.

Error Handling:

- `if (!outFile)`: Checks if the file failed to open and prints an error message: "Error: Unable to create file [fileName]."

4.0 CREATE TABLE Rows:

```
void processCreateTable(const string& line, vector<string>& headers, string& accumulatedColumns, bool& inCreateTable, string& tableName) {
      if (line.find("CREATE TABLE") != string::npos) {
          // Extract table name
          size_t start = line.find("CREATE TABLE") + 12;
          size_t end = line.find("(");
          if (start != string::npos && end != string::npos) {
              tableName = line.substr(start, end - start);
              tableName.erase(remove(tableName.begin(), tableName.end(), ' '), tableName.end()); // Remove extra spaces
              inCreateTable = true; // Indicate that we're processing the table
          }
      }

      if (inCreateTable) {
          size_t start = line.find('(');
          size_t end = line.find(')');
          if (start != string::npos) {
              // Start of column definitions
              accumulatedColumns += line.substr(start + 1);
          } else {
              // Accumulate multi-line column definitions
              accumulatedColumns += line;
          }

          if (end != string::npos) {
              // End of column definitions
              accumulatedColumns = accumulatedColumns.substr(0, accumulatedColumns.find(')')); // Remove trailing `)`
              stringstream ss(accumulatedColumns);
              string column;
              while (getline(ss, column, ',')) {
                  size_t spacePos = column.find(' ');
                  if (spacePos != string::npos) {
                      headers.push_back(column.substr(0, spacePos)); // Extract column name
                  }
              }
              inCreateTable = false; // Done processing
              accumulatedColumns.clear(); // Clear accumulated data
          }
      }
  }
```

From Main Function (Processing Each Line):

```
if (line.find("CREATE TABLE") != string::npos || inCreateTable) {
     //Process CREATE TABLE
     processCreateTable(line, headers, accumulatedColumns, inCreateTable, tableName);
```

This is the function key to run CREATE TABLE function. The processCreateTable function is executed in main.

4.1 Detecting CREATE TABLE Statement:

```
if (line.find("CREATE TABLE") != string::npos) {
     // Extract table name
     size_t start = line.find("CREATE TABLE") + 12;
     size_t end = line.find("(");
     if (start != string::npos && end != string::npos) {
          tableName = line.substr(start, end - start);
          tableName.erase(remove(tableName.begin(), tableName.end(), ' '), tableName.end()); // Remove extra spaces
          inCreateTable = true; // Indicate that we're processing the table
     }
}
```

When the line contains CREATE TABLE, the program sets **inCreateTable** to ***true*** and extracts the table name from the line. This marks the start of a CREATE TABLE block.

4.2 Extracting Column Names (Inside CREATE TABLE):

```
if (inCreateTable) {
    size_t start = line.find('(');
    size_t end = line.find(')');
    if (start != string::npos) {
        // Start of column definitions
        accumulatedColumns += line.substr(start + 1);
    } else {
        // Accumulate multi-line column definitions
        accumulatedColumns += line;
    }
```

This block is executed when **inCreateTable** is ***true***. It processes the columns inside the parentheses and extracts the column names from the SQL **CREATE TABLE** statement. These names are added to the **headers** vector.

4.3 End of CREATE TABLE Rows:

```
if (end != string::npos) {
    // End of column definitions
    accumulatedColumns = accumulatedColumns.substr(0, accumulatedColumns.find(')')); // Remove trailing `)`
    stringstream ss(accumulatedColumns);
    string column;
    while (getline(ss, column, ',')) {
        size_t spacePos = column.find(' ');
        if (spacePos != string::npos) {
            headers.push_back(column.substr(0, spacePos)); // Extract column name
        }
    }
    inCreateTable = false; // Done processing
    accumulatedColumns.clear(); // Clear accumulated data
}
```

If the line contains the closing parenthesis **‘) ’**, it means the CREATE TABLE block has ended. The columns are processed and **inCreateTable** is set to ***false*** to indicate the block is finished.

5.0 Parsing VALUES Rows (Data Insertion):

```
void processValues(const string& line, vector<vector<string>>& data) {
    if (line.find("VALUES") != string::npos) {
        size_t start = line.find('(') + 1;
        size_t end = line.find(')');
        string values = line.substr(start, end - start);
    vector<string> row;
    string value;
    bool inQuote = false;

    for (char c : values) {
        if (c == '\'') {
            inQuote = !inQuote;
        } else if (c == ',' && !inQuote) {
            if (!value.empty()) {
                row.push_back(value);
                value.clear();
            }
        } else if (inQuote || (!inQuote && (isdigit(c) || isalpha(c) || c == ' '))) {
            value += c;
        }
    }
    if (!value.empty()) {
        row.push_back(value);
    }
    data.push_back(row);
}
}
```

From MAIN Function:

```
else if (line.find("VALUES") != string::npos) {
    //Process INSERT VALUE
    processValues(line, data);
```

This is the function key to run VALUES function. The processValues function is executed in main.

5.1 Detecting VALUES Statement:

```
    else if (line.find("VALUES") != string::npos) {
        // Parse the VALUES line to extract data
        size_t start = line.find('(') + 1;
        size_t end = line.find(')');
        string values = line.substr(start, end - start);
```

When a line containing **VALUES** is found, it extracts the data inside the parentheses by finding the positions of the opening and closing parentheses and capturing the substring.

5.2 Processing Data Values:

```
            vector<string> row;
            string value;
            bool inQuote = false;

            for (char c : values) {
                if (c == '\'') {
                    inQuote = !inQuote;
                } else if (c == ',' && !inQuote) {
                    if (!value.empty()) {
                        row.push_back(value);
                        value.clear();
                    }
                } else if (inQuote || (!inQuote && (isdigit(c) || isalpha(c) || c == ' '))) {
                    value += c;
                }
            }
            if (!value.empty()) {
                row.push_back(value);
            }
            data.push_back(row);
        }
    }
```

This block processes the values within the **VALUES** line:

It handles quoted strings by toggling the ***inQuote*** flag whenever a single quote (') is encountered.It splits the values based on commas (ignoring commas inside quoted strings) and stores them in a row vector. After processing all values, the row is added to the ***data*** vector.

6.0 UPDATE Table Rows :

```
void processUPDATE(const string& line, vector<string>& headers, vector<vector<string>>& data) {
    // Parse UPDATE statement
    size_t setPos = line.find("SET");      // Find where 'SET' starts in the line
    size_t wherePos = line.find("WHERE"); // Find where 'WHERE' starts in the line

    if (setPos != string::npos && wherePos != string::npos) {
        // Extract table name
        string tableName = line.substr(line.find("UPDATE") + 6, setPos - (line.find("UPDATE") + 6));
        tableName = tableName.substr(tableName.find_first_not_of(' '), tableName.find_last_not_of(' ') - tableName.find_first_not_of(' ') + 1);

        // Extract column name and value (from SET)
        string setClause = line.substr(setPos + 4, wherePos - (setPos + 4));
        size_t equalSetPos = setClause.find("="); // Find the equal sign to separate column name and value
        size_t startQuote = setClause.find("'", equalSetPos); // Start after the first quote
        size_t endQuote = setClause.find("'", startQuote + 1); // Find the end quote
        string updateColumn = setClause.substr(0, equalSetPos);
        updateColumn = updateColumn.substr(updateColumn.find_first_not_of(' '), updateColumn.find_last_not_of(' ') - updateColumn.find_first_not_of(' ') + 1);

        // Extract the update value
        string updateValue = setClause.substr(startQuote + 1, endQuote - startQuote - 1);

        // Extract the WHERE value (from WHERE)
        string whereClause = line.substr(wherePos + 5);
        size_t equalWherePos = whereClause.find("="); // Find the equal sign to separate column name and value
        string whereColumn = whereClause.substr(0, equalWherePos);
        whereColumn = whereColumn.substr(whereColumn.find_first_not_of(' '), whereColumn.find_last_not_of(' ') - whereColumn.find_first_not_of(' ') + 1);

        string whereValue = whereClause.substr(equalWherePos + 1);
        whereValue = whereValue.substr(whereValue.find_first_not_of(' '), whereValue.find_last_not_of(' ') - whereValue.find_first_not_of(' ') + 1);

        //Remove trailing semicolon from WhereValue if it exists
        if (!whereValue.empty() && whereValue.back() == ';') {
            whereValue = whereValue.substr(0, whereValue.size() - 1);
        }

        // Find column indices for UPDATE and WHERE
        int updateIndex = -1, whereIndex = -1; // Initialize indices as not found (-1)
        for (size_t i = 0; i < headers.size(); i++) {
            if (headers[i] == updateColumn) updateIndex = i;
            if (headers[i] == whereColumn) whereIndex = i;
        }

        // Update the data in matching rows
        if (updateIndex != -1 && whereIndex != -1) { // Check if the indices are valid
            for (auto& row : data) {
                if (row[whereIndex] == whereValue) { // Compare the value in WHERE column
                    row[updateIndex] = updateValue; // Update the value if WHERE condition is matched
                }
            }
        } else {
            cerr << "Error: Column not found in the table headers.\n";
        }
    } else {
        cerr << "Error: Malformed UPDATE statement.\n";
    }
}
```

From MAIN Function:

```
} else if (line.find("UPDATE") != string::npos) {
    //Process UPDATE statements
    processUPDATE(line, headers, data);
```

This is the function key to run **UPDATE** function. The **processUpdate** function is executed in main.

6.1 Parsing UPDATE Statement:

```
void processUPDATE(const string& line, vector<string>& headers, vector<vector<string>>& data) {
    // Parse UPDATE statement
    size_t setPos = line.find("SET");     // Find where 'SET' starts in the line
    size_t wherePos = line.find("WHERE"); // Find where 'WHERE' starts in the line
```

The **processUPDATE** function is used to handle and apply the **UPDATE** statement on a dataset (**headers and data).** It also locates the keywords **SET** and **WHERE** in the SQL-like command to set where the clauses (**SET & WHERE**) starts.

6.2 Extract Table Name:

```
if (setPos != string::npos && wherePos != string::npos) {
    // Extract table name
    string tableName = line.substr(line.find("UPDATE") + 6, setPos - (line.find("UPDATE") + 6));
    tableName = tableName.substr(tableName.find_first_not_of(' '), tableName.find_last_not_of(' ') - tableName.find_first_not_of(' ') + 1);
```

The code first identifies the positions of **UPDATE** and **SET** in the SQL query. It extracts the substring between those two positions, which is the ***table name***.It also removes any leading or trailing spaces from the ***table name***. This helps ensure that only the clean table name is stored in the ***tableName*** variable.

6.3 Extract Column Name and Value (from SET):

```
// Extract column name and value (from SET)
string setClause = line.substr(setPos + 4, wherePos - (setPos + 4));
size_t equalSetPos = setClause.find("="); // Find the equal sign to separate column name and value
size_t startQuote = setClause.find("'", equalSetPos); // Start after the first quote
size_t endQuote = setClause.find("'", startQuote + 1); // Find the end quote
string updateColumn = setClause.substr(0, equalSetPos);
updateColumn = updateColumn.substr(updateColumn.find_first_not_of(' '), updateColumn.find_last_not_of(' ') - updateColumn.find_first_not_of(' ') + 1);

// Extract the update value
string updateValue = setClause.substr(startQuote + 1, endQuote - startQuote - 1);
```

This part of code extracts the ***column name*** and ***value*** from the **SET** clause. **updateColumn** contains the column name and **updateValue** contains the value to be updated later.

Eg. From input file:

```
UPDATE customer SET customer_name='Bob' WHERE customer_id=4;
```

customer_name *is the* ***updateColumn*** *and '****Bob****' is the* ***updateValue***

6.4 Extract Value (from WHERE):

```
// Extract the WHERE value (from WHERE)
string whereClause = line.substr(wherePos + 5);
size_t equalWherePos = whereClause.find("="); // Find the equal sign to separate column name and value
string whereColumn = whereClause.substr(0, equalWherePos);
whereColumn = whereColumn.substr(whereColumn.find_first_not_of(' '), whereColumn.find_last_not_of(' ') - whereColumn.find_first_not_of(' ') + 1);

string whereValue = whereClause.substr(equalWherePos + 1);
whereValue = whereValue.substr(whereValue.find_first_not_of(' '), whereValue.find_last_not_of(' ') - whereValue.find_first_not_of(' ') + 1);
```

This part of code also extracts the value and column name from **WHERE** .

Eg. From input file:

```
UPDATE customer SET customer_name='Bob' WHERE customer_id=4;
```

customer _id *is the* ***whereColumn*** *while* ***4*** *is the* ***whereValue***

6.5 Preventing Error (from WHERE):

```
// Remove trailing semicolon from whereValue if it exists
if (!whereValue.empty() && whereValue.back() == ';') {
    whereValue = whereValue.substr(0, whereValue.size() - 1);
}
```

Checks if **whereValue** ends with a semicolon (;). If it does, it removes it to prevent errors when dealing with empty strings.

6.6 Find Column Indices :

```
//Find column indices for update and where
int updateIndex= -1, whereIndex = -1; //Initialize update and where set as not found (-1)
    for (size_t i =0; i < headers.size();i++) {  //Loop through header to compare with update and where column
        if (headers[i]== updateColumn) updateIndex = i; //If match update column, then store in update index
        if (headers[i]== whereColumn) whereIndex = i; //If match where column, then store in where index
    }
```

The code loops through the headers to find the indices of **SET** column and **WHERE** column.

6.7 Update to Data Rows:

```
// Update the data in matching rows
if (updateIndex != -1 && whereIndex != -1) { // Check if the indices are valid
    for (auto& row : data) {
        if (row[whereIndex] == whereValue) { // Compare the value in WHERE column
            row[updateIndex] = updateValue; // Update the value if WHERE condition is matched
        }
    }
```

This block of code updates the value of **UPDATE** column where the **WHERE** column matches the condition if both columns are found

6.8 Error Handling

```
            }
        } else {
            cerr << "Error: Column not found in the table headers.\n";
        }
    } else {
        cerr << "Error: Malformed UPDATE statement.\n";
    }
```

6.9 Result of Update Function:

Input file:

```
UPDATE customer SET customer_name='Bobby' WHERE customer_id=4;
```

The customer name is set as '**Bobby**' and it will update at the 4th row of the customer table.

Output :

```
4,Bobby,city4,state4,country4,phone4,email4
```

The name is updated and named as '**Bobby**'.

7.0 DELETE Table Rows :

```
void processDelete(const string& line, vector<vector<string>>& data, const vector<string>& headers) {
    size_t wherePos = line.find("WHERE");
    if (wherePos == string::npos) {
        cout << "Error: WHERE clause not found" << endl;
        return;
    }

    string conditions = line.substr(wherePos + 6);  // Skip "WHERE "
    conditions = conditions.substr(0, conditions.find(';')); // Remove semicolon

    vector<pair<string, string>> deleteConditions;  // Column-value pairs

    while (!conditions.empty()) {
        size_t orPos = conditions.find(" OR ");
        string singleCondition = (orPos != string::npos) ? conditions.substr(0, orPos) : conditions;

        size_t equalsPos = singleCondition.find('=');
        if (equalsPos != string::npos) {
            string columnName = singleCondition.substr(0, equalsPos);
            string value = singleCondition.substr(equalsPos + 1);

            columnName.erase(remove(columnName.begin(), columnName.end(), ' '), columnName.end());
            value.erase(remove(value.begin(), value.end(), ' '), value.end());

            deleteConditions.push_back({columnName, value});
        }
        if (orPos != string::npos) {
            conditions = conditions.substr(orPos + 4);  // Skip " OR "
        } else {
            conditions.clear();
        }
    }
    size_t originalSize = data.size();
    auto rowIter = data.begin();
    while (rowIter != data.end()) {
        bool shouldDelete = false;

        for (const auto& [columnName, value] : deleteConditions) {
            int columnIndex = -1;
            for (int i = 0; i < headers.size(); i++) {
                if (headers[i] == columnName) {
                    columnIndex = i;
                    break;
                }
            }
            if (columnIndex == -1) {
                cout << "Column not found: " << columnName << endl;
                continue;
            }
            if ((*rowIter)[columnIndex] == value) {
                shouldDelete = true;
                break;
            }
        }
        if (shouldDelete) {
            rowIter = data.erase(rowIter);
        } else {
            ++rowIter;
        }
    }

}
```

From MAIN Function:

```
} else if (line.find("DELETE FROM") != string::npos) {
    //Process DELETE statements
    processDelete(line, data, headers);
}
```

This is the function key to run **DELETE** function. The **processDelete** function is executed in main.

7.1 Parsing DELETE Statement:

```
void processDelete(const string& line, vector<vector<string>>& data, const vector<string>& headers) {
    size_t wherePos = line.find("WHERE");
    if (wherePos == string::npos) {
        cout << "Error: WHERE clause not found" << endl;
        return;
    }
```

This snippet checks if the **WHERE** clause exists in the **DELETE** SQL statement by searching for the keyword **WHERE** in the input **line**. If it is not found (wherePos == string::npos), the function prints an error message ("Error: WHERE clause not found") and exits without processing further.

7.2 Extracting and Cleaning the WHERE Clause Conditions:

```
string conditions = line.substr(wherePos + 6);  // Skip "WHERE "
conditions = conditions.substr(0, conditions.find(';')); // Remove semicolon
```

The function surgically extracts the actual conditions by skipping past the **WHERE** keyword and remove any trailing semicolon. This transforms the raw text into a clean, processable condition string ready for detailed parsing.

7.3 Extracting Individual Conditions from the WHERE Clause:

```
vector<pair<string, string>> deleteConditions;  // Column-value pairs

while (!conditions.empty()) {
    size_t orPos = conditions.find(" OR ");
    string singleCondition = (orPos != string::npos) ? conditions.substr(0, orPos) : conditions;
```

This code creates a vector to store column-value pairs (**deleteConditions**) and iterates through the **conditions** string, extracting individual conditions separated by " OR"; if " OR" is found, it isolates the first condition before it; otherwise, it takes the entire remaining string as a single condition.

7.4 Parsing Column-Value Pairs from a Condition:

```
size_t equalsPos = singleCondition.find('=');
if (equalsPos != string::npos) {
    string columnName = singleCondition.substr(0, equalsPos);
    string value = singleCondition.substr(equalsPos + 1);
```

This code finds the position of the = symbol in the **singleCondition** string and, if found, extracts the text before it as the **columnName** and the text after it as the value.

7.5 Processing and Storing Column-Value Pairs for Deletion:

```
columnName.erase(remove(columnName.begin(), columnName.end(), ' '), columnName.end());
value.erase(remove(value.begin(), value.end(), ' '), value.end());

deleteConditions.push_back({columnName, value});
```

The code removes extra spaces from the **columnName** and **value**, then stores them as a pair in the **deleteConditions** vector to represent conditions for deleting rows.

7.6 Handling Multiple Conditions with OR:

```
if (orPos != string::npos) {
    conditions = conditions.substr(orPos + 4);  // Skip " OR "
} else {
    conditions.clear();
}
```

This code checks if there's an "**OR**" in the conditions, and if so, it removes the "**OR**" and everything before it; otherwise, it clears the conditions.

7.7 Initializing for Row Deletion

```
size_t originalSize = data.size();

auto rowIter = data.begin();
```

This code stores the original number of rows in the **data** vector (**originalSize)** and initializes an iterator (**rowIter**) to traverse the rows of the data vector for potential deletion.

7.8 Matching Row Conditions for Deletion:

```
for (const auto& [columnName, value] : deleteConditions) {
    int columnIndex = -1;
    for (int i = 0; i < headers.size(); i++) {
        if (headers[i] == columnName) {
            columnIndex = i;
            break;
        }
    }

    if (columnIndex == -1) {
        cout << "Column not found: " << columnName << endl;
        continue;
    }

    if ((*rowIter)[columnIndex] == value) {
        shouldDelete = true;
        break;
    }
}
```

This code iterates through **deleteConditions**, finds the index of the specified **columnName** in the **headers**, and checks if the value in the matching column of the current row equals the specified **value**; if so, it sets **shouldDelete** to **true**.

7.9 Deleting or Skipping Rows Based on Condition:

```
if (shouldDelete) {
    rowIter = data.erase(rowIter);
} else {
    ++rowIter;
}
```

This code checks the **shouldDelete** flag: if **true**, it deletes the current row using **data.erase(rowIter)** and updates the iterator; otherwise, it moves the iterator to the next row.

7.10 Result of Update Function:

Input:

```
DELETE FROM customer WHERE customer_id=4;
```

Output:

```
1,kelly,kuala lumpur,selangor,country1,0123123123,kelly@test.com
2,ben,los angeles,california,united state,123455667,ben@test.com
3,son,cyberjaya,selangor,malaysia,0161241241,son@test.com
5,mike,new york,new york state,united state,1241241,mike@test.com
```

8.0 Writing Data to Output File and Terminal (CSV Format):

```
void processSelectQuery(const string& line, const vector<string>& headers, const vector<vector<string>>& data, ofstream& outFile, string& tableName) {
    if (line.find("SELECT * FROM") != string::npos) {
        // Handle SELECT * FROM queries
        for (size_t i = 0; i < headers.size(); i++) {
            cout << headers[i];
            outFile << headers[i];
            if (i < headers.size() - 1) {
                cout << ",";
                outFile << ",";
            }
        }
        cout << endl;
        outFile << endl;

        for (const auto& row : data) {
            for (size_t i = 0; i < row.size(); i++) {
                cout << row[i];
                outFile << row[i];
                if (i < row.size() - 1) {
                    cout << ",";
                    outFile << ",";
                }
            }
            cout << endl;
            outFile << endl;
        }
    }
}
```

From Main Function:

```
} else if(line.find("SELECT * FROM") != string::npos) {
//Process Select From statements
processSelectQuery(line, headers, data, outFile, tableName);
}
```

8.1 Writing CSV Header Row:

```
// Write the headers
for (size_t i = 0; i < headers.size(); i++) {
    cout << headers[i];
    outFile << headers[i];
    if (i < headers.size() - 1) {
        cout << ",";
        outFile << ",";
    }
}
cout << endl;
outFile << endl;
```

The program outputs the column headers (from **headers**) both to the terminal and to the output file in CSV format.

8.2 Writing Data Rows:

```
// Write the CSV data rows
for (const auto& row : data) {
    for (size_t i = 0; i < row.size(); i++) {
        cout << row[i];
        outFile << row[i];
        if (i < row.size() - 1) {
            cout << ",";
            outFile << ",";
        }
    }
    cout << endl;
    outFile << endl;
}
```

This block writes each row of data (from ***data***) to both the terminal and the output file in ***CSV*** format, separating each value with commas.

8.3 Result of Select Query

The function successfully outputs the CSV format of header and data in the specified table to both the console and the output file.

Input file:

```
> SELECT * FROM customer;
```

The function identifies SQL queries containing **SELECT * FROM <table_name>** by searching for the keyword. **SELECT * FROM**

If the table is named customers ,the function will print out the Table and Value in CSV format.

customer_id,customer_name,customer_city,customer_state,customer_country,customer_phone,
customer_email
1,name1,city1,state1,country1,phone1,email1
2,name2,city2,state2,country2,phone2,email2
3,name3,city3,state3,country3,phone3,email3
4,name4,city4,state4,country4,phone4,email4

9.0 SELECT COUNT FROM FUNCTION:

```
void processCount(const string &line, const string &tableName, const vector<vector<string>> &data, ofstream &outFile) {
    // Parse the SELECT COUNT(*) query
    size_t pos = line.find("FROM") + 5;
    string queryTable = line.substr(pos);
    queryTable.erase(remove(queryTable.begin(), queryTable.end(), ';'), queryTable.end());
    queryTable.erase(remove(queryTable.begin(), queryTable.end(), ' '), queryTable.end());

    // Match the table name and count rows
    if (queryTable == tableName) {
        cout << data.size() << endl;
        outFile << data.size() << endl;
    } else {
        cerr << "Error: Table '" << queryTable << "' not found." << endl;
        outFile << "Error: Table '" << queryTable << "' not found." << endl;
    }
}
```

From MAIN Function:

```
} else if (line.find("SELECT COUNT(*) FROM") != string::npos) {
// Handle SELECT COUNT(*) FROM queries
processCount(line, tableName, data, outFile);
```

This is the function key to run **COUNT** function. The **processCOUNT** function is executed in main.

9.1 Parse the SELECT COUNT (*) Query

```
// Parse the SELECT COUNT(*) query
size_t pos = line.find("FROM") + 5;
string queryTable = line.substr(pos);
queryTable.erase(remove(queryTable.begin(), queryTable.end(), ';'), queryTable.end());
queryTable.erase(remove(queryTable.begin(), queryTable.end(), ' '), queryTable.end());
```

The function takes a string input, line, which contains the SQL query. It identifies the keyword FROM within the query to locate the table name. Once located, the substring following FROM is extracted, and any unnecessary spaces or semicolons(;) are removed to obtain a clean table name.

9.2 Table Name Validation

```
    // Match the table name and count rows
    if (queryTable == tableName) {
        cout << data.size() << endl;
        outFile << data.size() << endl;
    } else {
        cerr << "Error: Table '" << queryTable << "' not found." << endl;
        outFile << "Error: Table '" << queryTable << "' not found." << endl;
    }
}
```

The function performs table name validation by comparing the extracted table name (queryTable) with the provided table name (tableName). If the table names match, the function proceeds to count the rows in the table.

Eg. From input file:

> SELECT COUNT(*) FROM customer;

9.3 Count Rows in the Table

The size of the data vector representing the table rows is obtained using the size() method. This value represents the total number of rows in the table.

Eg. From output file:

```
customer
customer_id,customer_name,customer_city,customer_state,customer_country,customer_phone,customer_email
1,name1,city1,state1,country1,phone1,email1
2,name2,city2,state2,country2,phone2,email2
3,name3,city3,state3,country3,phone3,email333
3
```

9.4 Result of SELECT COUNT

The function successfully outputs the total number of rows in the specified table to both the console and the output file.

Input file:

> SELECT COUNT(*) FROM customer;

The function identifies SQL queries containing **SELECT COUNT(*) FROM <table_name>** by searching for the keyword FROM.

If the table is named customers and contains 3 rows, the function will produce the following output:

```
customer
customer_id,customer_name,customer_city,customer_state,customer_country,customer_phone,customer_email
1,name1,city1,state1,country1,phone1,email1
2,name2,city2,state2,country2,phone2,email2
3,name3,city3,state3,country3,phone3,email333
3
```

9.5 Error Handling

```
} else {
    cerr << "Error: Table '" << queryTable << "' not found." << endl;
    outFile << "Error: Table '" << queryTable << "' not found." << endl;
}
```

10.0 Error Handling:

10.1 File Handling Error

```
if (!inFile) {
    cout << "Please check if the file exist in your directory!" << endl;
    return 1;
}
```

If the input file fileinput2.txt cannot be opened, the program prints the message "Error opening fileinput2.txt" and terminates execution.

10.2 processDelete Function

```
size_t wherePos = line.find("WHERE");
if (wherePos == string::npos) {
    cout << "Error: WHERE clause not found" << endl;
    return;
}
```

Missing WHERE Clause in Delete Statements:
If a DELETE statement does not contain a WHERE clause, the program outputs the message "Error: WHERE clause not found" and skips the statement.

```
if (columnIndex == -1) {
    cout << "Column not found: " << columnName << endl;
    continue;
}
```

Column Not Found:
If a specified column in a DELETE statement WHERE condition is not found in the table headers, the program outputs "Column not found: <column_name>" and continues processing.

10.3 processUpdate Function

```
    if (setPos != string::npos && wherePos != string::npos) {
        // Extract the SET and WHERE clauses
    } else {
        cerr << "Error: Malformed UPDATE statement.\n";
    }
```

Malformed UPDATE Statement:
If an UPDATE statement is missing the SET or WHERE clause or is formatted incorrectly, the program outputs the message "Error: Malformed UPDATE statement." and skips the statement.

```
    // Update the data in matching rows
    if (updateIndex != -1 && whereIndex != -1) { // Check if the indices are valid
        for (auto& row : data) {
            if (row[whereIndex] == whereValue) { // Compare the value in WHERE column
                row[updateIndex] = updateValue; // Update the value if WHERE condition is matched
            }
        }
    } else {
        cerr << "Error: Column not found in the table headers.\n";
```

Column not found:
If the column specified in the SET or WHERE clause of an UPDATE statement is not found in the headers, the program outputs "Error: Column not found in the table headers." and skips processing the update.

10.4 processCount Function

```
    // Match the table name and count rows
    if (queryTable == tableName) {
        cout << data.size() << endl;
        outFile << data.size() << endl;
    } else {
        cerr << "Error: Table '" << queryTable << "' not found." << endl;
        outFile << "Error: Table '" << queryTable << "' not found." << endl;
    }
```

If the table name specified in a SELECT COUNT(*) query does not match the existing table name, the program outputs "Error: Table '<table_name>' not found." in the console and writes the same message to the output file.

10.5 Data Validation Error

The program checks for valid column names during row operations (e.g., UPDATE, DELETE). An error message is displayed when a specified column is not found. The program does not explicitly validate data types, which can lead to issues if data formats are inconsistent.

11.0 DATABASES FUNCTION

```
void processDatabases(const string& line, const string& filein) {
    // Check if the line contains "DATABASES;"
    if (line.find("DATABASES;") != string::npos) {
        cout << "C:\\mariadb\\" << filein << endl; // Output the required message
    }
}
```

From MAIN Function:

```
else if (line.find("DATABASES;") != string::npos) {
  processDatabases(line, filein); // Output the required message
```

This is the function key to run **DATABASES** function. The **processDatabases** function is executed in main.

11.1 Parsing DATABASES

```
(line.find("DATABASES;") != string::npos) {
 cout << "C:\\mariadb\\" << filein << endl;
```

The program reads the file `fileinput1.txt` line by line using `getline(inFile, line)`. For each line read, it checks whether the line contains a specific SQL command `DATABASES;`.

In the main loop, the program encounters lines one at a time. If it detects the `DATABASES;` keyword, it calls the `processDatabases` function with that line and the name of the input file (`fileinput1.txt`).

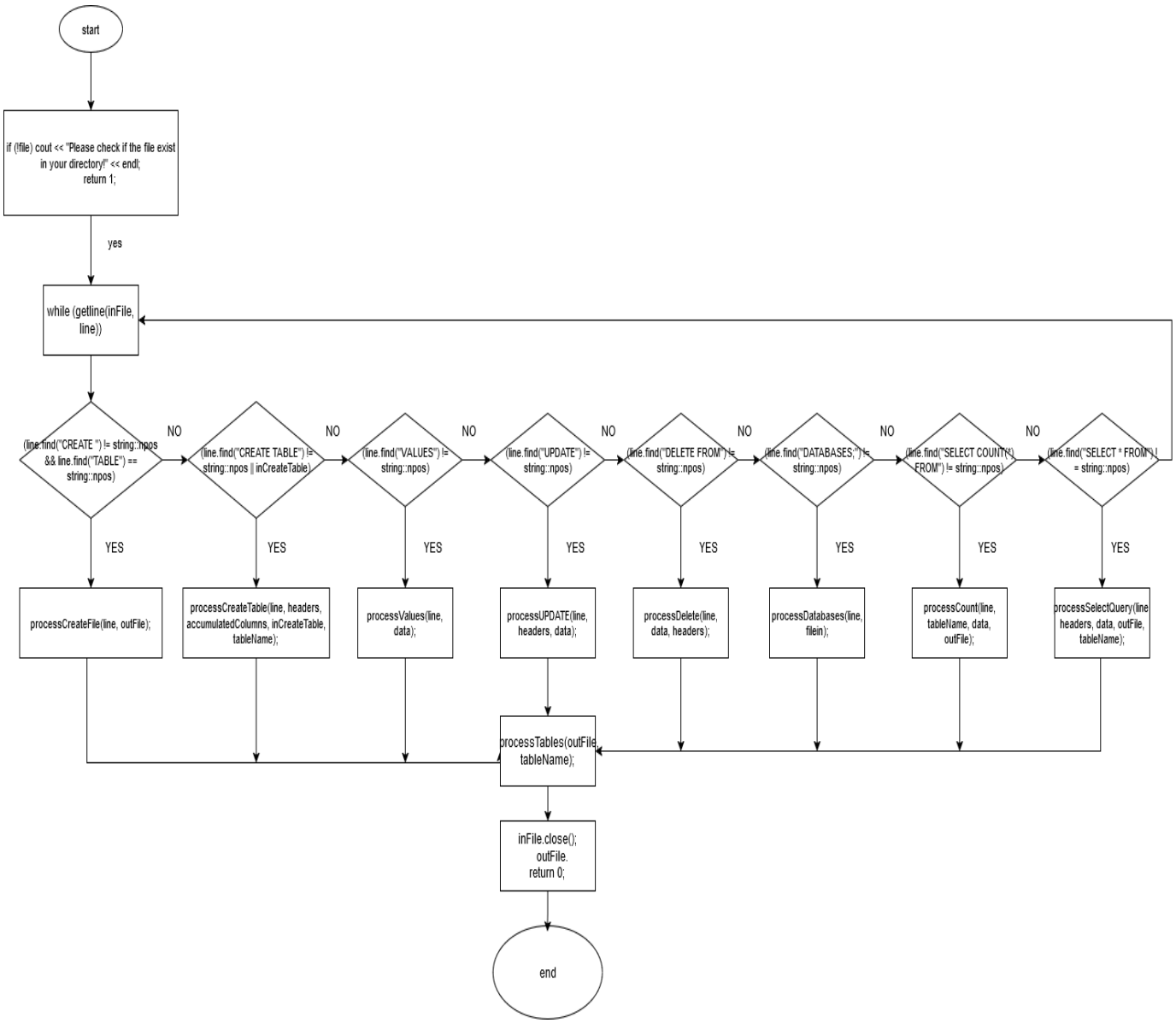
12.0 Closing Files:

```
inFile.close();
outFile.close();
```

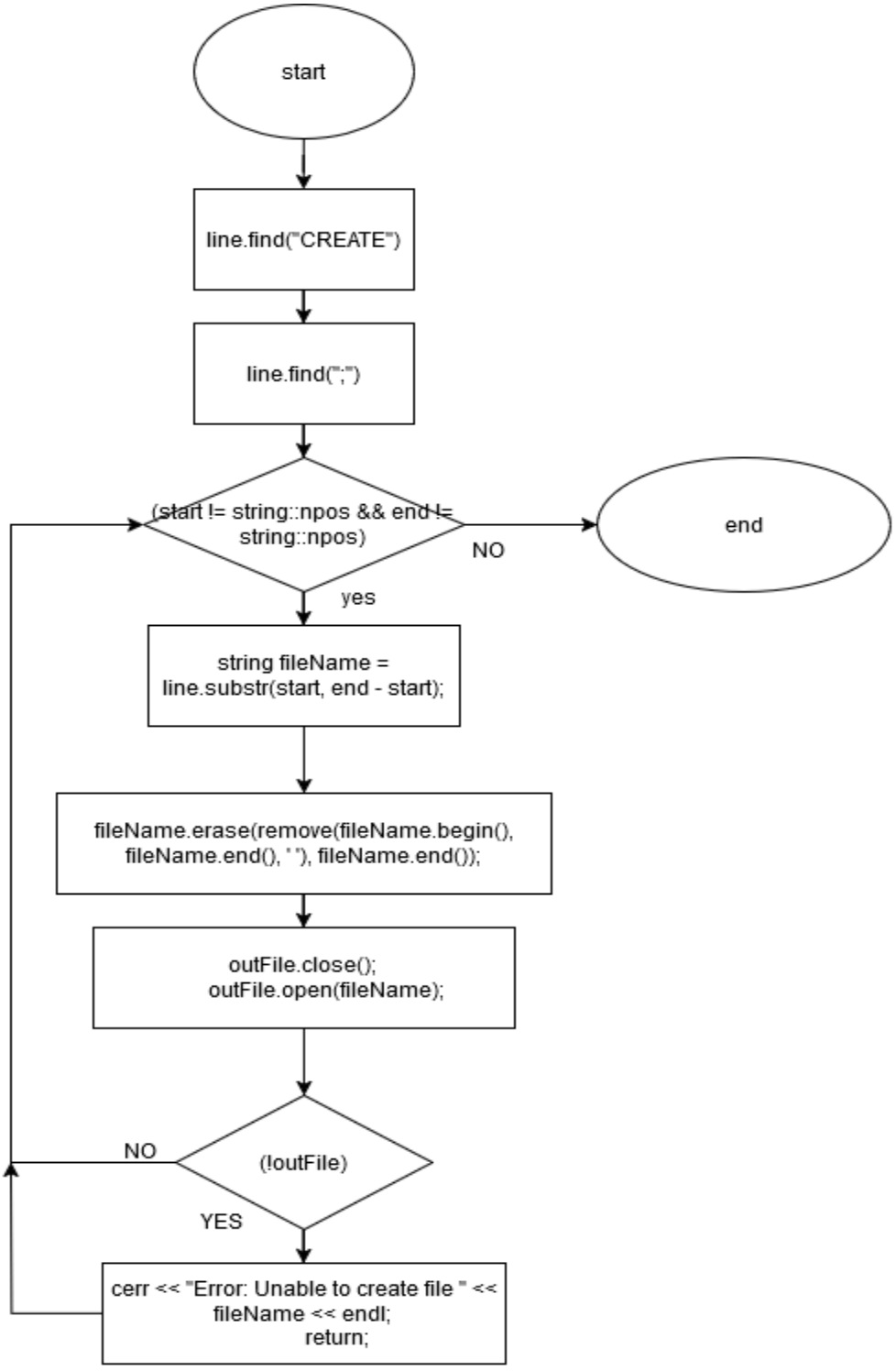
Finally, the input and output files are closed using **close()** to free up resources.

13.0 Program Flowchart:

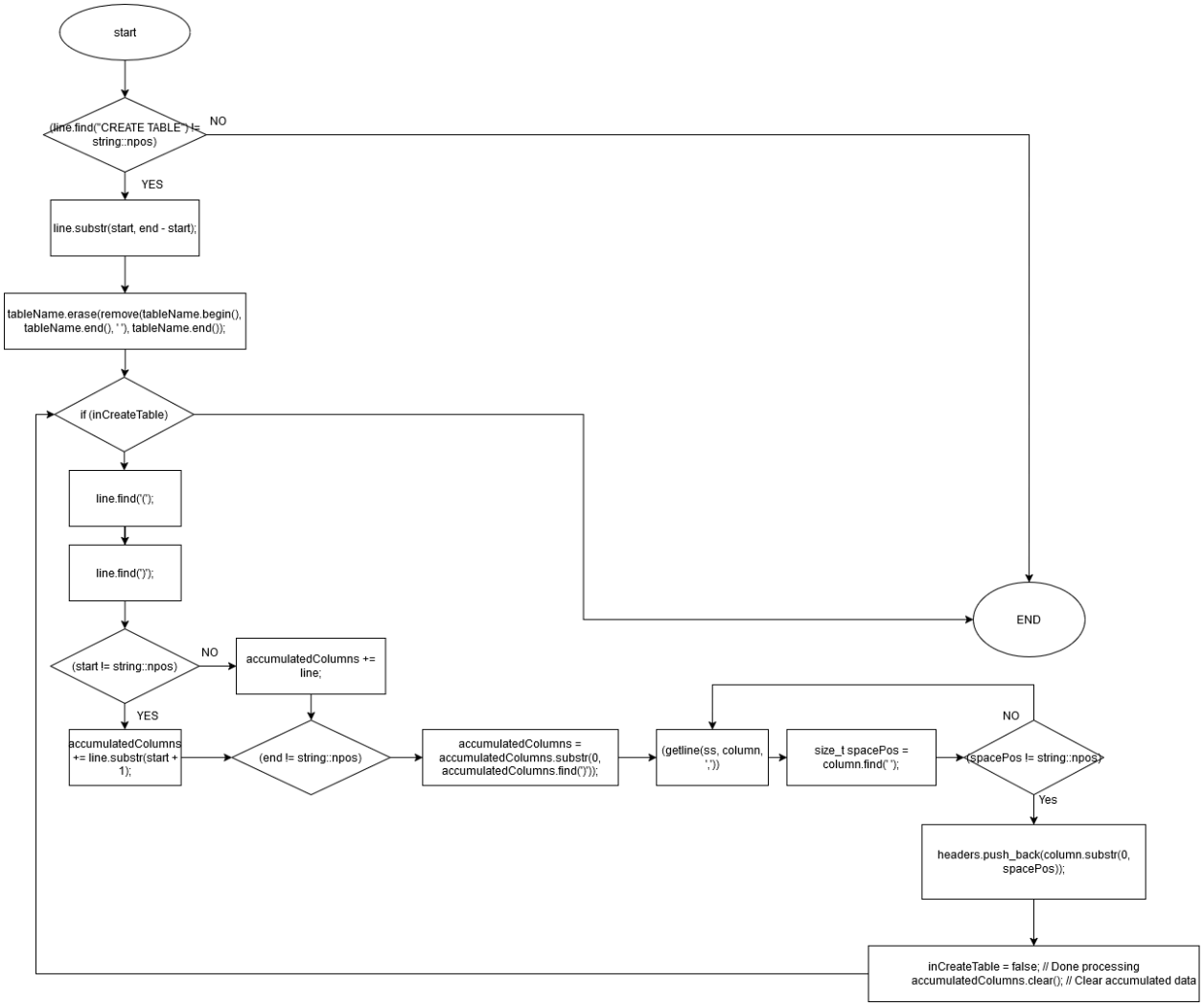
13.1 Main Flowchart



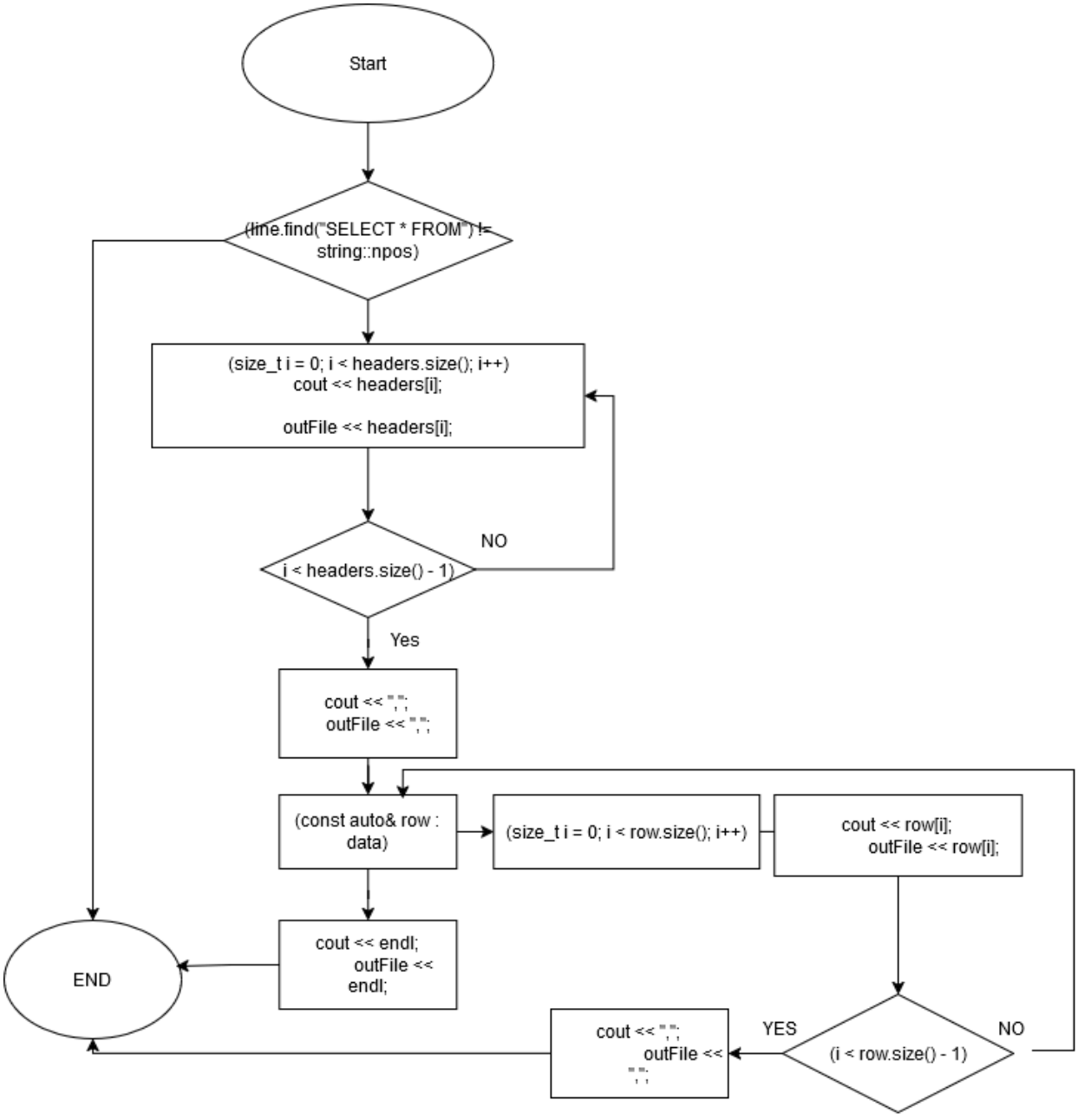
13.2 Create File Function Flowchart



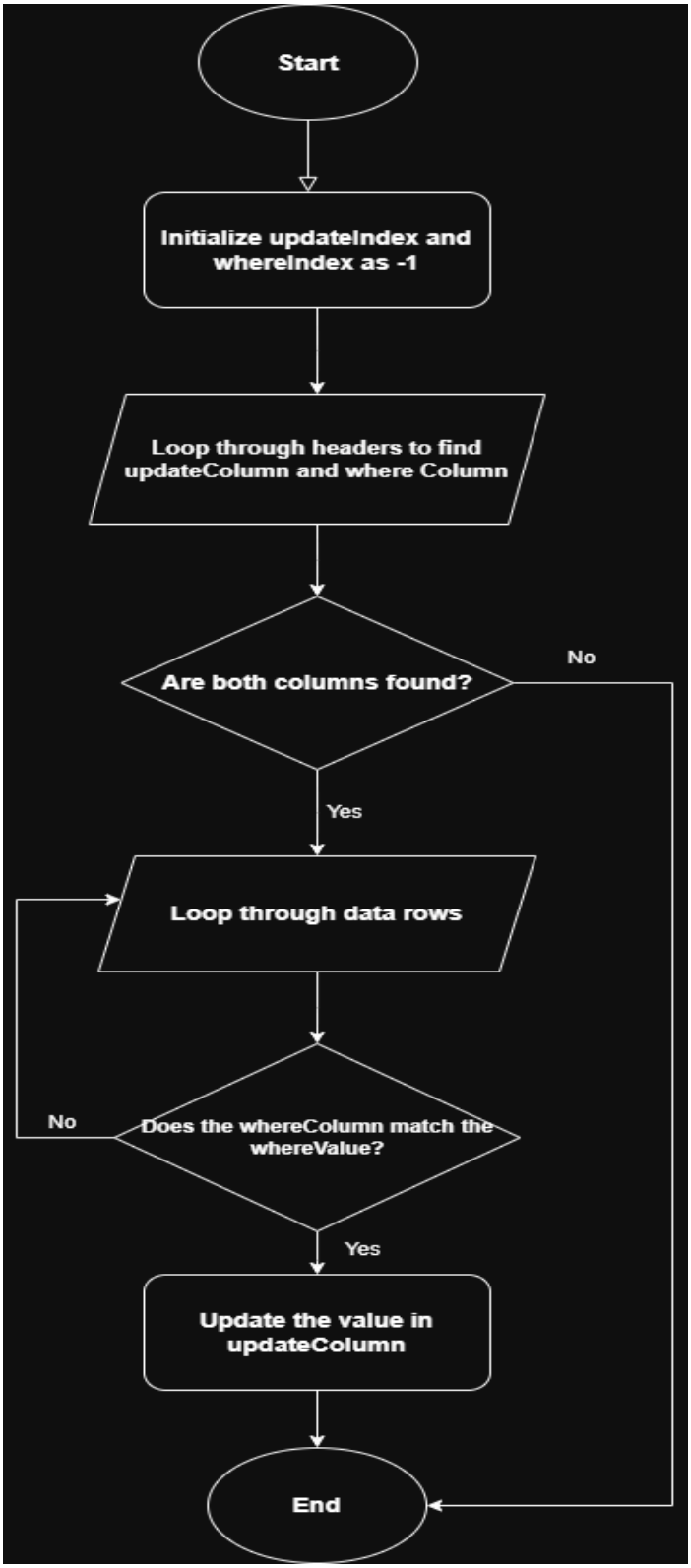
13.3 Create Table Function Flowchart



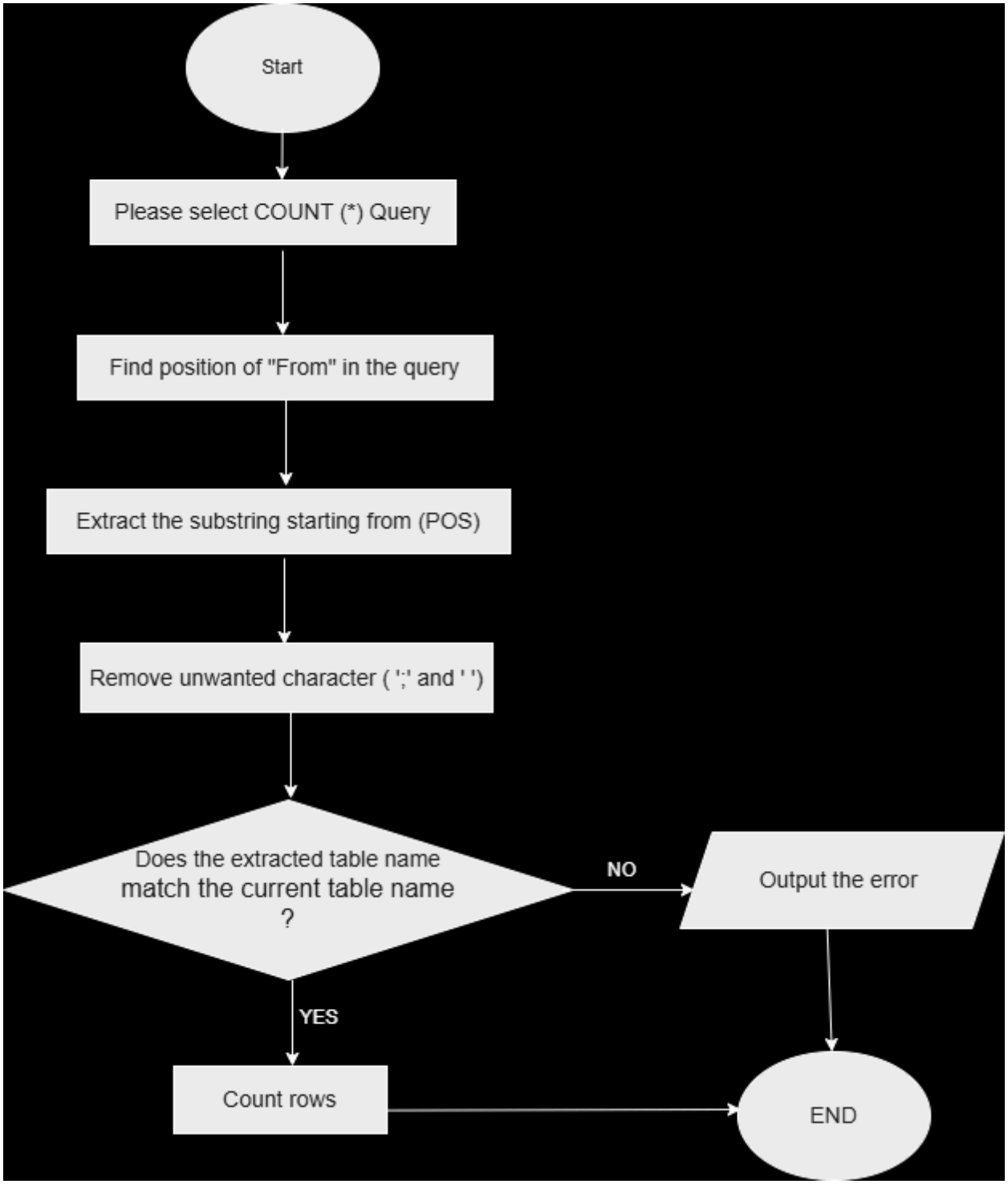
13.4 Select Query Flowchart



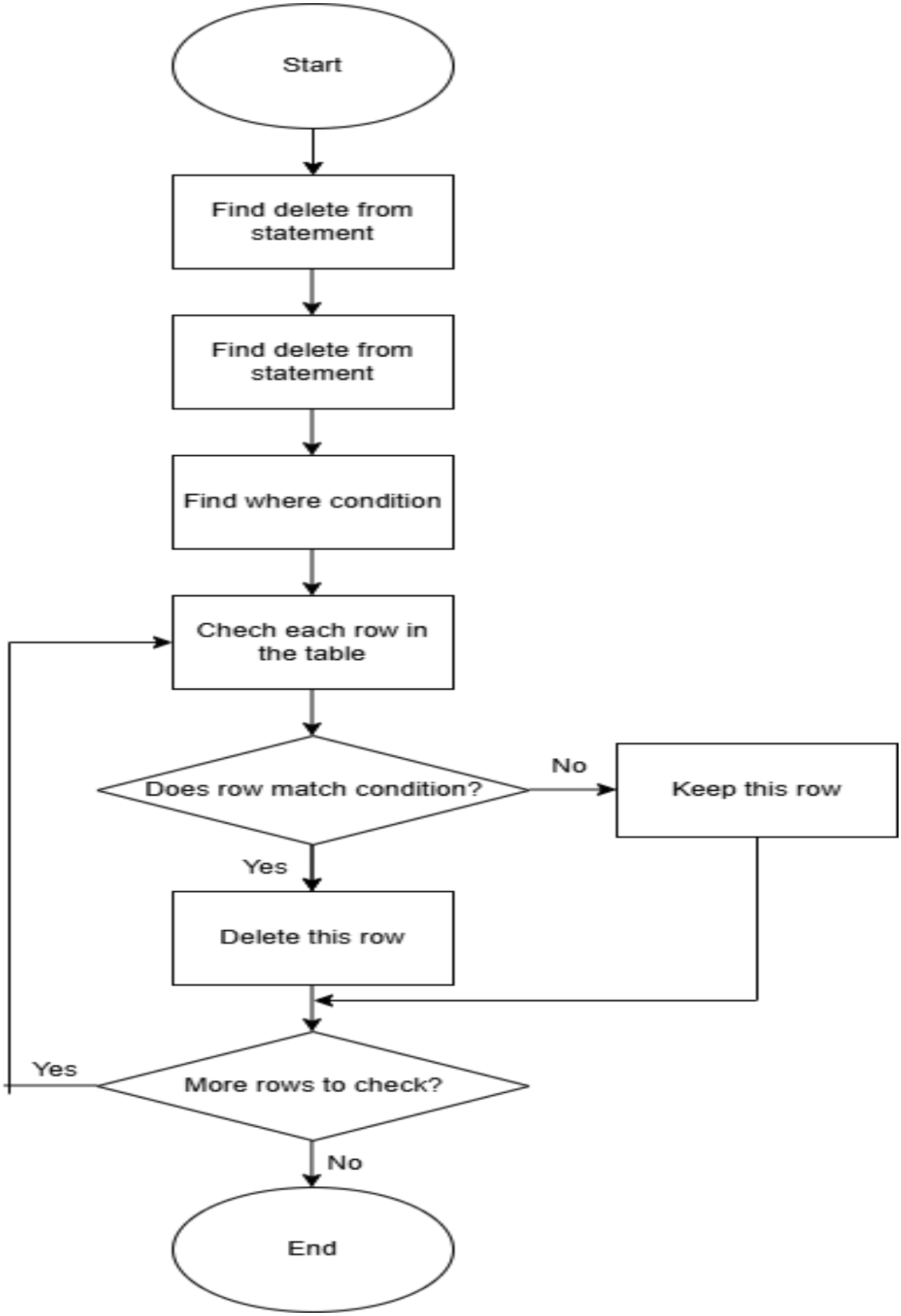
13.5 Update Function Flowchart



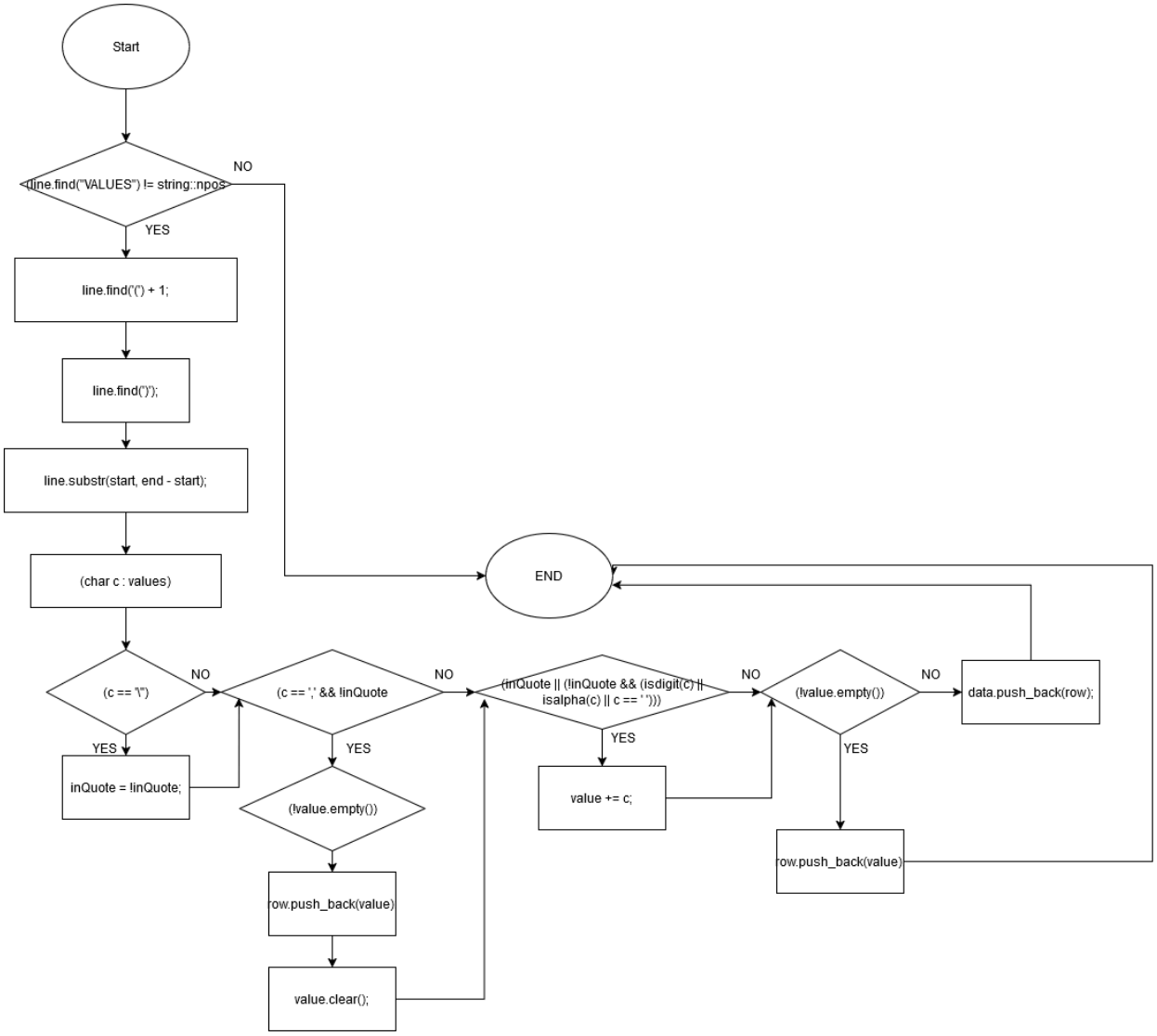
13.6 COUNT Function Flowchart



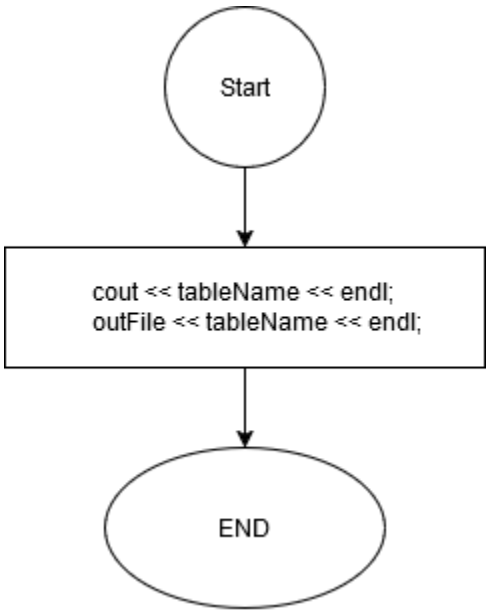
13.7 Delete Function Flowchart



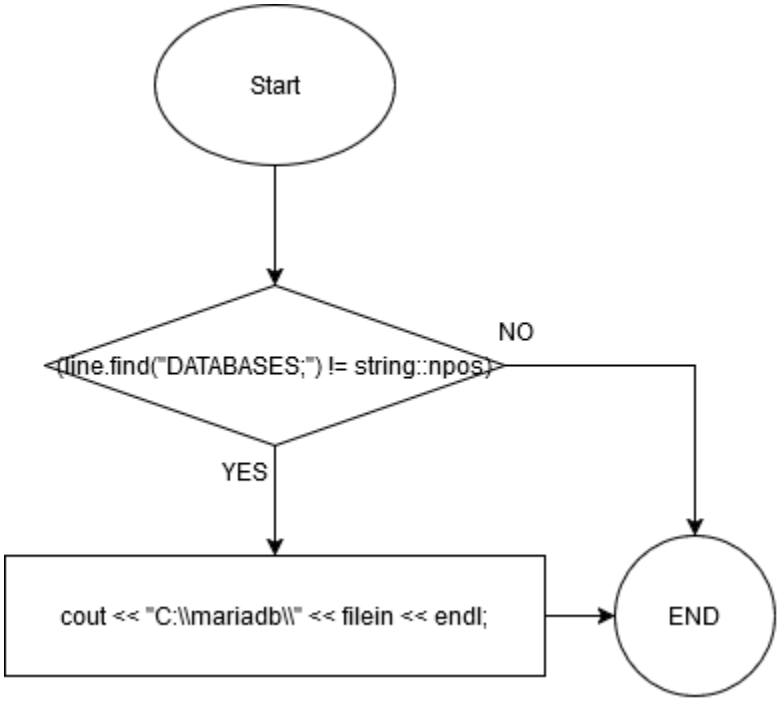
13.8 VALUES Rows Flowchart



13.9 Tables Name Flowchart



13.10 DATABASES File Flowchart:



14.0 Apendix

14.1 Task Distribution Table

Student ID	243UC247DH
Student name	Wong Kai Shen
Task percentage(%)	40%
Task descriptions	ProcessCreateFile,processTables,ProcessCreateTable, ProcessValues, ProcessSelectQuary,processDatabases, main
Total score (30m)	

Student ID	243UC2467K
Student name	Teh Shin Rou
Task percentage(%)	20%
Task descriptions	ProcessUpdate, Update flowchart
Total score (30m)	

Student ID	243UC2466T
Student name	Nyiam Zi Qin
Task percentage(%)	20%
Task descriptions	processCount, Count Flowchart
Total score (30m)	

Student ID	243UC246NQ
Student name	Yen Ming Jun
Task percentage(%)	20%
Task descriptions	processDelete, Delete Flowchart
Total score (30m)	

14.2 Assignment Mark Table

Criteria	Max	A1	A2	Mark
Q1. Create database and view database name Create table, view table name Table supports two data types i.e. INT, TEXT Insert rows to the table View table in csv mode	5	*	*	
Q2. Reading from a file, outputting to screen, writing to a file (0 if no files used or no screen outputs)	3		*	
Q3. Update table rows and view table Delete table rows and view table	4		*	
Q4. Count and output number of rows in the table	2		*	
Q5. Must use vectors or arrays, functions or classes, to store file output contents	2		*	
Q6. Inline comments, function or class comments, indentation, following proper C++ naming and styling conventions Any violation is penalized by a reduction of 1 mark.	2		*	
Q7. The program demonstrates error handlings. [0: Below Expectation, 1: Within Expectation, 2: Exceed Expectation]	2		*	
Q8. Correct structured diagrams	2		*	
Q9. Correct flowcharts or pseudocodes with explanations for all the file input statements. Any missing flowchart or pseudocode will cause you to lose 1 mark.	2		*	
Q10. Sample file inputs at least 3, their screen outputs, their file outputs with screenshots and explanations.	3		*	

Q11. User documentation done and is coherence with the all implementations. Any missing input statement will cause you to lose 1 mark.	3		*	
Total	30			

Additional Comment
