Computer Science and Engineering, UNR

CS 457 Programming Assignment 3

Christopher Mollise
Instructor: Dongfang Zhao
May 6, 2020

Abstract/Summary

This project is the tertiary step of a database simulator for CS457. This program allows a database user to manage the metadata of their relational data as well as being able to manipulate the data within the table. The main functionalities of this project are to simulate a relational database through the use of various linux directories representing databases and their files representing tables. Through this simulation it is possible to create and delete databases as well as add, delete, select, and alter tables within these databases.

Additionally, the program can now insert, modify, query, and delete information on the table itself.

In this final version, multiple tables can be selected in order to inner or outer join them.

Project Description

This c++ project makes use of several objects in order to keep track of information within the database efficiently. The project contains three main classes with a class for databases, tables, and columns.

The project utilizes a list of databases acting as virtual representations of physical file directories. The main program file "main.cpp " manages the database list, constructing it at the beginning of each program operation and controlling the main parser for the program. This file also manages functions such as the creation and deletion of directories in the list.

The database object controls the individual database and the tables within them. Each database object contains a list of tables and has the necessary functionality to create and delete tables from the database as well as alter or query a table within the database. This functionality is done through a secondary parser that handles functions used directly on a database.

The last iteration of the program expands on the table object allowing the ability to insert and modify information as well as query and delete information based on user input. These functions are accessed via the parser in the table object and can be adjusted based on user input. This means that functions such as delete, select, and alter will take user input through column names and conditional statements. These functions are minimal but can be expanded if need be for a future iteration of this program.

Finally, this iteration of the program implements the ability to join tables using the '=' condition. This allows two tables with equivalent information in a column to be compared. This is done via the main parser for the Select command and splitting into two functions for the inner and outer joining of tables. This program allows for an inner join only showing matches and an outer join showing matches and non-matched tuples from the first table.

Organization of Databases

Databases are organized in the simulator through a Database class as well as being physically organized as directories within the projects Database directory. If a user executes commands through the program, the directories within this Database directory will actively add, remove, and edit based on the commands given.

In the main project file, “main.cpp”, these directories are manipulated directly before splitting the parsing into parsing for individual databases. This includes actions such as the creation and deletion of databases both physically and within the program’s data structure. This is done using c++14’s experimental filesystem library which allows for queries about the existence of directories, iteration through these directories, and the creation and deletion of directories. Using this library allows for easier creation of the necessary directories and allows for the data structure created in the program to easily construct itself and stay current with the directory.

The Database is virtually implemented in the program using a Database class as implemented below:

```
/*!
 * \file Database.h
 *
 * \author Christopher Mollise
 * \date 5/6/2020
 *
 * Project 3 for CS457
 *
 * Header for Database Class
 *
 */

#ifndef DATABASE_H
#define DATABASE_H

#include <string>
#include <vector>

#include "Table.h"

/*!
 * \class Database
 *
 * \brief This class acts as a virtual implementation of a database
 *
 * This class holds the information for a database within a directory.
 * This virtual database will allow for easier data manipulation as
 * the data will be already loaded in the program.
```

```
 *
 * \author Christopher Mollise
 * \date 5/6/20
 */
class Database {
public:
	Database(std::string path); ///Default Constructor
	virtual ~Database(); ///Default Deconstructor

	std::string dataBaseParser(); ///Function to act as secondary parser

	std::string actualName; ///String containing the actual database name
	std::string dataBaseName; ///String containing the usable database name
protected:
	bool createTable(); ///Function to create table
	bool dropTable(); ///Function to drop table
	bool selectTable(); ///Function to select table
	bool innerJoin(std::string firstTable);///Function to inner join table
	bool outerJoin(std::string firstTable);///Function to outer join table
	bool alterTable(); ///Function to alter table
	bool insertTable(); ///Function to insert into table
	bool updateTable(); ///Function to update table
	bool deleteFromTable(); ///Function to delete from table

	std::vector <Table> tables; ///Vector containing Table objects
};

#endif // DATABASE_H
```

This object will handle the functionality within the database while the main function handles the creation and deletion of these databases. Within this object information about the database is stored such as its name and the list of tables within the database through a vector. The database additionally contains the functions needed to modify the database such as the create, drop, select, and alter commands. The createTable and dropTable functions work similarly to their counterparts for creating and dropping databases and make use of the filesystem library to delete the necessary files The select and alter table functions leverage the parser in the Database class but utilize functions within the Table class itself to before the necessary actions on the Column list within each Table.

Organization of Tables

Tables are organized in the simulator through a Table class as well as being physically organized as text files in individual database directories within the projects Database directory. If a user executes commands through the program, the table files in their individual database directories will add, remove, and edit based on the commands given. Additionally, the program can now deal with the information within the table such as inserting information into the table, deleting from the table based on conditions, and enhanced query functionality.

These tables are implemented virtually through a Table class as implemented below.

```
/*!
 * \file Table.h
 *
 * \author Christopher Mollise
 * \date 5/6/2020
 *
 * Project 3 for CS457
 *
 * Header for Table Class
 *
 */


#ifndef TABLE_H
#define TABLE_H

#include <string>
#include <vector>

#include "Column.h"

/*!
 * \class Table
 *
 * \brief This class acts as a virtual implementation of a table
 *
 * This class holds the information for a table within a database.
 * This virtual table will allow for easier data manipulation as
 * the data will be already loaded in the program.
 *
 * \author Christopher Mollise
 * \date 5/6/20
 */
class Table {
public:
	Table(std::string path); ///Default constructor
	virtual ~Table(); ///Default deconstructor
	void writeTable(std::string path); ///Function to write table to file
	bool alter(); ///Function to alter table columns
	void select(); ///Function to list table columns
	bool insert(); ///Function to insert into table columns
	bool insertLine(std::string input); ///Function if no spaces in insert command
	bool update(); ///Function to update table columns
	bool deleteFrom(); ///Function to delete input from tables
	void reInput(); ///Function to reload database after delete changes
	bool selectTypes(std::vector <std::string> types); ///Separate select for different
```

```
types

        std::vector<Column> columns; ///Vector of Column Objects
        std:: string lowerName; ///String containing lowercase tableName
        std:: string tableName; ///String containing tableName
        std::string tablePath; ///String containing tablePath
        size_t rowNum; ///Holds the number of rows in the column
        int rowAdjust; ///Holds an adjustment amount for reloading the file after delete
protected:
};

#endif // TABLE_H
```

The Table object handles the storage of all information within a table such as the name of the table, a string containing access to the file, and information about the size of the table. One of the main functions of this class is to manage a list of the Column class. This class contains the physical data for each column as well as its metadata. This class also has functions for the writing of the internal Table structure to its file counterpart as well as functions to handle the altering and selecting of column data.

These columns are implemented virtually through a Column class as implemented below.

```
/*!
 * \file Column.h
 *
 * \author Christopher Mollise
 * \date 5/6/2020
 *
 * Project 3 for CS457
 *
 * Header for Column Class
 *
 */

#ifndef COLUMN_H
#define COLUMN_H

#include <string>
#include <vector>

/*!
 * \class Column
 *
 * \brief This class acts as a virtual implementation of a column
 *
 * This class holds the information for a column within a table.
 * This virtual column will allow for easier data manipulation as
 * the data will be already loaded in the program.
```

```
 *
 * \author Christopher Mollise
 * \date 5/6/20
 */
class Column {
    public:
        /** Default Constructor **/
        Column(std::string newColName, std::string newColType, int newColSize);
        virtual ~Column();///Default deconstructor

        std::vector<std::string> colData; ///Vector of strings for data
        std::string colName; ///String of the column name
        std::string colType; ///String of the column type
        int colSize; ///Int of the column size
};

#endif // COLUMN_H
```

The Column object handles the majority of the data stored within the simulator. This class is inside a list in the Table object and will handle the individual column information. This includes metadata such as the name, type, and size as well as the data being stored. Currently, the class only acts as a struct would, holding information. However, this class can be expanded through further projects.

Organization of Table File

The table file is organized in an effort to make reading into and out from the file as easy as possible. The file begins with use of an indicating column and row size, this amount will allow the program to quickly read the file as the program will already know what size the table is and how much information to read. The header for the columns is separated by white space and makes use of the column size to read each column to create a column object. Tuples are implemented below the head and separated by white space. This allows the file to be quickly read into the program and makes writing to the files quicker as well. This is important for the delete functionality as the program is rewritten and then reread into the program. An example of a filled table is shown below:

```
3 4
pid int name varchar 20 price float
1 'Gizmo' 19.99
2 'PowerGizmo' 29.99
3 'SingleTouch' 149.99
4 'MultiTouch' 199.99
```

Implementation of Main Functionalities

The program begins its execution through the creation of the main list of Database objects. This list will be the base of the program, holding all information about the different databases. This list is created by calling a updateDataBase function that will use the file system library to detect directories. From this search, a new Database object is constructed using the default constructor with the pathname of the directory it belongs to. This constructor will go on to call the constructor for the Database object's list of Table objects. Using a similar method the Table objects are created with found files before using those files to construct the Column objects.
The initial updateDataBase function is shown below. A similar method is used for tables and a more in-depth constructor that will parse the given file is used to construct the Columns.

```
 /*!
 * \brief Initial configuration system for database
 *
 * This function acts as the base for updating the database objects on load.
 * When the program first executes it will call this function, constructing
 * the database based on the physical database files. This is necessary
 * to create the one to one relation between the programs simulated database
 * and the actual database. Will construct the individual databases from
 * the directory and then add the name of the database to a public variable.
 *
 * \param[out] vector<Database> &dataBases
 *             Vector of the Database class used for containing all data
 * \note Makes use of the default class constructors of Database.
 */
void updateDataBase(vector<Database> &dataBases) {
    string name; ///String containing database name

    /** Directory Reader **/
    for(auto &p: fs::directory_iterator("Databases")){
       /// Push new database to vector for each directory found
        dataBases.push_back(Database(p.path()));
       /// Get the name of the directory from the path and add to class
        name = p.path().string();
        name = name.substr(name.find('/') + 1);
        dataBases.back().dataBaseName = name;
    }
}
```

Once created the main flow of the project is conducted through the mainParser function. This function will take the first token of input and determine where in the flow it will direct the program. This may mean directing the program to a function that handles parsing an individual database function, or to a secondary parser that handles functions related to tables. Using the output from these functions, the program will determine to end or continue the processing of user commands. In the case of the USE command, the secondary parser is called. In this secondary parser, if a token is not found, it is redirected back to the main parser to find a command. If a

command is entered that is not found in the main parser, it will exit the program stating an invalid command was entered.

The mainParser function is shown below. This function is similar to the secondary parser excepts with calls to a table's CREATE, DROP, SELECT, and ALTER commands instead of the databases CREATE, DROP, and USE commands or the program EXIT command.

```
/*!
 * \brief Function for the use database command
 *
 * This function will use a database from the internal Database list for use
 * in table manipulation. When the main parser detects the USE command
 * it will direct to this function. The function will use the database if it
 * exists or give an error message if it does not. It will then call the
 * parser for the individual database for use of their parser.
 *
 * \param[out] vector<Database> &dataBases
 *            Vector of the Database class used for containing all data
 * \note Controls main flow of project with while loop controlling all parsing
 */
void mainParser(vector<Database> &dataBases) {
	string input; /// String containing input from user
	bool endInput = false; /// Boolean switch to end program
	bool returnedInput = false; /// Boolean switch to use secondary parse input

	/** Main Control Loop **/
	while (!endInput) {
	if (returnedInput) /// Check if input was passed from secondary parser
		returnedInput = false;
	else /// Receive an input token from user
		cin >> input;
	/** Exit and Ignore Commands **/
	if (input[0] == '-') /// Ignore inputs starting with - and remove line
	{
		cin.ignore();
		getline (cin, input);
	}
	else if (input == ".EXIT" || input == ".exit") /// EXIT command
	{
		 cout << "All done." << endl;
		 return;
	}

	/** Database Creation/Deletion/Use Commands **/
	else if (input == "CREATE") /// CREATE command
	{
		if (!createDataBase(dataBases)) /// Call create function for failure
		endInput = true;
	}
```

```
        else if (input == "DROP") /// DROP command
        {
                if (!dropDataBase(dataBases)) /// Call drop function for failure
                 endInput = true;
        }
        else if (input == "USE") /// USE command
        {
                 input = useDataBase(dataBases); /// Call use function for input
                 if (input == "END") /// Check if secondary parser failed
                        endInput = true;
                 else /// Tell main parser to not receive a new input for secondary
                        returnedInput = true;
        }

        /** No Acceptable Command **/
        else
                endInput = true;
        }
        /** Send Unknown Error if parser fails **/
        cout << "!Unknown Command." << endl;
}
```

The create and drop methods for both the databases and tables makes use of the file system library to determine if files exist and can be created or dropped. The creation functions make use of the default constructors to create and add a new database or table to their appropriate list. The deconstructor makes use of the vector erase function to find and delete the appropriate element in the list as well as physically delete the file. The table CREATE function adds additional functionality in that it has a secondary constructing process for creating the individual columns for a table based on the passed inputs. The table create also calls the writer for the function to rewrite the file with relevant table information.

Below is the dropDataBase function for removing databases. This function is similar to the table drop method and uses a similar process.

```
/*!
 * \brief Function for the drop database function
 *
 * This function will drop a database from the internal Database list and then
 * replicate this deletion on a file level. When the main parser detects the
 * DROP command and no database has been used it will direct to this
 * function. The function will then drop a directory based on the name and
 * drop or deny it from the database if the directory does or does not
 * exist. It will then output the appropriate error messages.
 *
 * \param[out] vector<Database> &dataBases
 *             Vector of the Database class used for containing all data
 * \return boolean true/false
```

```
 *         True or false value depending on state of function. False only sent
 *         if the parser cannot interpret the error and will send true in case
 *         of a processed create command if it is created or not.
 */
bool dropDataBase(vector<Database> &dataBases) {
       string input; ///String containing input from user
    fs::path dataPath = "Databases/"; ///Directory path for deleted database

    /** Receive and Check Input From User **/
       cin >> input;
       if (input != "DATABASE") ///Unknown command if DATABASE not specified
        return false;
    cin >> input;
    if (input.back() != ';') ///Unknown command if no ";" at end of function
        return false;

    /** Process path for deleting directory **/
    input.pop_back();
    dataPath += input;

    /** Attempt to delete directory at path **/
    if(!fs::remove_all(dataPath)) ///Delete and output error if fails
    {
        cout << "!Failed to delete "
             << input
             << " because it does not exist." << endl;
    }
    else ///Output message if success and add database to internal list
    {
        /** Locate internal database to also remove **/
        for (size_t i = 0; i < dataBases.size(); i++){
            if (dataBases[i].dataBaseName == input)
              dataBases.erase(dataBases.begin() + i);
        }

        cout << "Database "
             << input
             << " deleted." << endl;
    }
    return true; ///Return true for end of processing
}
```

Below is the createTable function for adding tables to a database. This function is similar to the database create function except that it does not include the additional parsing for adding the columns to the table and instead ending after confirmation of creating the file.

```
/*!
 * \brief Function for the create table function
```

```
 *
 * This function will create a table in the internal table list and then
 * replicate this creation on a file level. When the main parser detects the
 * CREATE command and a database has been used it will direct to this
 * function. The function will then create a file based on the name and
 * create or deny it from the list if the file does or does not
 * exist. t will then output the appropriate error messages.The function
 * will then populate the internal and external table  if the provided
 * construction information for individual columns.
 *
 * \return boolean true/false
 *      True or false value depending on state of function. False only sent
 *      if the parser cannot interpret the error and will send true in case
 *      of a processed create command if it is created or not.
 */
bool Database::createTable() {
    string input; /// String holding user input
    string tableName; /// String containing the table name
    string colNameNoSpace; /// String to hold column name if no space between table name

    bool endInput = false; /// Boolean to determine if column input is done
    bool firstInput = true; /// Boolean to adjust parser based on first input
    bool noSpace = false; /// Boolean if no space between table and column name

    string newColName; /// String containing a new columns name
    string newColType; /// String containing a new columns type
    int newColSize; /// Int containing a new columns size

    fs::path dataPath = "Databases/"; ///Path for the table file

    /** Receive and Check Input From User **/
    cin >> input;
    if (input != "TABLE" && input != "table") /// Unknown command if TABLE not specified
        return false;
    cin >> input;
    if (input.find('(') != string::npos) {
        noSpace = true;
        colNameNoSpace = input.substr(input.find('('));
        input = input.substr(0, input.find('('));
    }

    /** Process path for detecting directory **/
    dataPath += dataBaseName + '/' + input + ".txt";

    /** Attempt to detect directory at path **/
    if (fs::exists(dataPath)) /// Detect and output error if found
    {
        cout << "!Failed to create table "
             << input
             << " because it already exists." << endl;
```

```
        cin.ignore();
        getline (cin, input);
    }
    else /// Create message if success and add database to internal list
    {
        /** Set Table Name for Later Use **/
        tableName = input;

        /** Push new Table onto List **/
        tables.push_back(Table(dataPath));

        /** Main Table Creator **/
        while (!endInput) /// Run until input ends / ";" found
        {
                /** Get Input **/
                if (!noSpace)
                        cin >> newColName;
                else
                {
                        noSpace = false;
                        newColName = colNameNoSpace;
                }
                if (firstInput) /// Check for particular first input problems
                {
                        /** Check for Empty Token **/
                        if (newColName == "();") /// Handle an Empty Table
                        {
                                tables.back().writeTable(dataPath);
                                cout << "Table "
                                        << tableName
                                        << " created." << endl;
                                return true;
                        }
                        /** Remove the starting "(" **/
                        newColName.erase(0,1);
                        firstInput = false;
                }
                /** Get Column Type **/
                cin >> newColType;
                if (newColType.back() != ',' && newColType.back() != ';') /// Check for
valid token ending "," or ";"
                {
                        tables.pop_back();
                        return false;
                }
                else /// Valid type token
                {
                        /** Process Type Token **/
                        if (newColType.back() == ';') /// If token ends with ";" it is last
token
```

```
                        {
                                newColType.pop_back();
                                endInput = true;
                        }
                        newColType.pop_back();

                        /** Check if token is of type char or varchar **/
                        if (newColType.substr(0, newColType.find('(')) == "char" ||
                                newColType.substr(0, newColType.find('(')) == "varchar") {
                                        /** Process the token to get column type **/
                                        input = newColType;
                                        newColType = input.substr(0, input.find('('));
                                        /** Process token to get the column size **/
                                        input = input.substr(input.find('(') + 1);
                                        input = input.substr(0, input.find(')'));
                                        newColSize = stoi(input);
                        }
                        else if (newColType == "int" || newColType == "float") /// Check for
other types
                                newColSize = 0;
                        else /// If not valid remove table
                        {
                                tables.pop_back();
                                return false;
                        }
                        /** Add Column onto Table **/
                        tables.back().columns.push_back(Column(newColName, newColType,
newColSize));
                }
        }
        /** Add Valid Path to Table **/
        tables.back().writeTable(dataPath);
        cout << "Table "
                << tableName
                << " created." << endl;
    }
    return true; /// Return true for end of processing
}
```

Finally, the alter and select table functions use similar methods from the past for parsing and looking through the data structure to see if the appropriate tables/columns exist. One of the main differences with these functions is that the parsing for the function begins in the Database class but will invoke the Table class for the printing and edit of the columns themselves. This can be seen in the alter class in that the parsing for if a proper table is selected is done in Database before the parsing is done to add the new database information in the Table class. The alter function also calls upon the write function to rewrite the table file with new information.

Below is the implementation of the alterTable function in the Database class and the alter function in the Table class. The parser will call the alterTable function first to validate the command before invoking the alter function to add the new column.

```
/*!
 * \brief Function for the alter table function
 *
 * This function will alter a table from the internal Table list.
 * When the main parser detects the ALTER command and a database
 * has been used to direct this function. The function will
 * allow the input of a proper column or output the appropriate
 * error messages.
 *
 * \return boolean true/false
 *         True or false value depending on state of function. False only sent
 *         if the parser cannot interpret the error and will send true in case
 *         of a processed create command if it is created or not.
 */
bool Database::alterTable() {
	string input; ///String holding user input
	string tableName; ///String holding the table name

	/** Receive and Check Input From User **/
	cin >> input;
	if (input != "TABLE") ///Unknown command if TABLE not specified
		return false;
	cin >> tableName;
	cin >> input;
	if (input != "ADD") ///Unknown command if ADD not specified
		return false;

	/** Locate internal database use **/
	for (size_t i = 0; i < tables.size(); i++) {
		if (tables[i].tableName == tableName)
			return tables[i].alter();
	}

	/** Output error if location failed **/
	cout << "!Failed to alter table "
		 << tableName
		 << " because it does not exist." << endl;

	return true; ///Return true for end of processing
}
/*!
 * \brief Function for the alter table function
 *
 * This function will alter a table in the internal table list and then
 * replicate this creation on a file level. This  function will create
```

```
 * a new column including a new name, type, and size and add it to the
 * existing table. If the command fails it outputs back to the main parser
 * as usual.
 *
 * \return boolean true/false
 *         True or false value depending on state of function. False only sent
 *         if the parser cannot interpret the error and will send true in case
 *         of a processed create command if it is created or not.
 */
bool Table::alter() {
	string input; ///String containing input from file
	string newColName; ///String containing the columns name
	string newColType; ///String containing the columns type
	int newColSize; ///Int containing the column size if applicable

	/** Receive and Check Input From User **/
	cin >> newColName;
	cin >> newColType;
	if (newColType.back() != ';') ///Unknown command if no ";" at end of function
		return false;

	/** Process Input to Remove ";" **/
	newColType.pop_back();

	/** Check if token is of type char or varchar **/
	if (newColType.substr(0, newColType.find('(')) == "char" ||
		newColType.substr(0, newColType.find('(')) == "varchar") {
		/** Process the token to get column type **/
		input = newColType;
		newColType = input.substr(0, input.find('('));
		/** Process token to get the column size **/
		input = input.substr(input.find('(') + 1);
		input = input.substr(0, input.find(')'));
		newColSize = stoi(input);
	}
	else if (newColType == "int" || newColType == "float")
		newColSize = 0;
	else ///If not valid error
		return false;
	/** Add Column onto Table **/
	columns.push_back(Column(newColName, newColType, newColSize));
	/** Add Valid Path to Table **/
	writeTable(tablePath);
	cout << "Table "
		 << tableName
		 << " modified." << endl;
	return true; ///Return true for end of processing
}
```

Implementation of Table Functionalities

The main functionality for the table itself is implemented within the Table Class. Within this class are functions that allow for the selecting of particular items in a file, adding input to a file, modifying input, and deleting input from the file. These functions additionally make use of an auxiliary function that allows for the reinput of the file in the cases of deleting information from a Table.

One of the main functions implemented is the insert information function. This is a necessary part of the program that allows a user to insert new information into the file. With the implementation for the project users must input the same amount of information as there exists columns as there is currently no check for this. The information is then parsed from the command and is inserted into each column. With each input the information is parsed with the first input removing any beginning or ending information that is not necessary for the input. This information is then added to the columns, checking to make sure the columns are each receiving information. Below is the implementation:

```
*!
 * \brief Method for the insert table function
 *
 * This function will insert into a table in the internal table list and then
 * replicate this creation on a file level. This  function will create
 * a new row including relevant information and add it to the
 * existing table. If the command fails it outputs back to the main parser
 * as usual.
 *
 * \return boolean true/false
 *      True or false value depending on state of function. False only sent
 *      if the parser cannot interpret the error and will send true in case
 *      of a processed create command if it is created or not.
 */
bool Table::insert() {
    string input; ///String containing input from file
    bool firstInput = true; ///Bool checking if still on intial input

    cin >> input;
    if (input.back() == ';')
        return insertLine(input);
    /** Iterate through columns pushing information to each column **/
    for (size_t i = 0; i < columns.size(); i++) {
        if (!firstInput)
               cin >> input; ///Get initial input
        else
               firstInput = false;
        if (i == 0) ///Additional editing on first input
        {
               ///Remove the "values(" in input
```

```
		if (input.rfind("values(",0) != 0)
			return false;
		else {
			input.erase(0, 7);
			///After erasing at to column
			if (input.back() != ',')
				return false;
			else {
				input.pop_back();
				columns[i].colData.push_back(input);
			}
		}
	}
	/** Additional editing if last input **/
	else if (i == columns.size() - 1) {
		///Check for the  ';' char
		if (input.back() != ';') {
			///Remove invalid data
			for (size_t j = 0; j < i; j++)
				columns[j].colData.pop_back();
			return false;
		}
		else {
			///Remove the ; and check for ')'
			input.pop_back();
			if (input.back() != ')') {
				///Remove invalid data
				for (size_t j = 0; j < i; j++)
					columns[j].colData.pop_back();
				return false;
			}
			else ///Add correct information to column
			{
				input.pop_back();
				columns[i].colData.push_back(input);
			}
		}
	}
	/** Checking for standard ',' seperated input **/
	else {
		/** Check for ',' **/
		if (input.back() != ',') {
			///Remove invalid data
			for (size_t j = 0; j < i; j++)
				columns[j].colData.pop_back();
			return false;
		}
		else ///Add correct information to column
		{
```

```
 /
        input.pop_back();
                        columns[i].colData.push_back(input);
                }
        }
    }
    /** Increase row amount and print output **/
    rowNum++;
    writeTable(tablePath);
    cout << "1 new record inserted." << endl;
    return true; ///Return true for end of processing
}
```

Additionally, an insertLine function was added in order to deal with using an insert statement with a different syntax. For example, this function will parse the insert the same way as the insert function does, however, no spaces are necessary between each token. This function works the same as the insert function but with a different parser implementation.

The update/modify function is implemented in a similar manner by going through each column and adding information. The main difference is instead of adding information it is checking information that already exists in the program and modifying it. This works by comparing a value to look for and a value to modify and making the changes in the program. Below is the implementation

```
/*!
 * \brief Method for the update table function
 *
 * This function will update the table in the internal table list and then
 * replicate this update on a file level. This  function will edit
 * an existing row including chosen information and add it to the
 * existing table. If the command fails it outputs back to the main parser
 * as usual.
 *
 * \return boolean true/false
 *         True or false value depending on state of function. False only sent
 *         if the parser cannot interpret the error and will send true in case
 *         of a processed create command if it is created or not.
 */
bool Table::update() {
       string setName; ///String holds the name of what value will be set
       string whereName; ///String holds the name of where the value will be set
       string setValue; ///String Holds the value of the information after the change
       string whereValue; ///String Holds the value of the information to change or compare
       string input; ///String holds all input
       int recordCount = 0; ///Int holds the number of records changed
```

```
	/** Get Set Input From User **/
	cin >> input;
	if (input != "set") ///Unknown command if no "set"
		return false;
	cin >> setName;
	cin >> input;
	if (input != "=") ///Unknown command if no '='
		return false;
	cin >> setValue;

	/** Get Where Input From User **/
	cin >> input;
	if (input != "where") ///Unknown command if no "where
		return false;
	cin >> whereName;
	cin >> input;
	if (input != "=") ///Unknown command if no "="
		return false;
	cin >> whereValue;
	if (whereValue.back() != ';') ///Unknown command if no ";" at end of function
		return false;

	/** Process Input to Remove ";" **/
	whereValue.pop_back();

	/** Go through table looking for values and replacing as nessecary **/
	for (size_t i = 0; i < columns.size(); i++) ///Go through each column
	{
		if (columns[i].colName == whereName) ///Continue search if column name
matches
		{
			for (size_t j = 0; j < columns[i].colData.size(); j++) ///Go through
each row
			{
				if (columns[i].colData[j] == whereValue) ///Change if column
value matches
				{
					/** Go Through Each Column Changing Value If Matching
and Increase Count **/
					for (size_t k = 0; k < columns.size(); k++){
						if (columns[k].colName == setName) {
							columns[k].colData[j] = setValue;
							recordCount++;
						}
					}
				}
			}
		}
	}
```

```
	/** Rewrite Table **/
	writeTable(tablePath);

	/** Print Output **/
	if (recordCount == 1)
		cout << "1 record modified." << endl;
	else
		cout << recordCount << " records modified." << endl;

	return true; ///Return true for end of processing
}
```

The delete program works in a similar method to the update program with locating information based on user input. Due to the nature of the way the information is stored, it is difficult to directly remove the information. This makes it necessary to use a method of writing and reading to the table in order to simulate the delete. This means that in the delete function, values to be deleted are marked as 'DELETE'. Values marked in this way will not be written to the file. This makes it possible to print the information to the file and then reread the information into the program to simulate the new table. This is done in a re input function that is similar to the constructor for Table but without reestablishing metadata. This implementation is shown below:

```
/*!
 * \brief Method for the delete from table function
 *
 * This function will delete from the table in the internal table list and then
 * replicate this update on a file level. This  function will delete
 * an existing row based on chosen information and remove it from the
 * existing table. If the command fails it outputs back to the main parser
 * as usual.
 *
 * \return boolean true/false
 *         True or false value depending on state of function. False only sent
 *         if the parser cannot interpret the error and will send true in case
 *         of a processed create command if it is created or not.
 */
bool Table::deleteFrom() {
	string whereName; ///String holds the name of where the value will be set
	string whereValue; ///String Holds the value of the information to change or compare
	char deleteType; ///Char holds the type of information deleted
	string input; ///String holds all input
	int deleteHere = -1; ///Int holds the index of a deleted row
	int recordCount = 0; ///Holds the number of records
	bool deleted = false; ///Holds if something has been deleted

	/** Get Input **/
	cin >> input;
	if (input != "where") ///Unknown command if no "where"
		return false;
```

```
        cin >> whereName;
        cin >> deleteType;
        cin >> whereValue;
        if (whereValue.back() != ';') ///Unknown command if no ";" at end of function
                return false;

        /** Process Input to Remove ";" **/
        whereValue.pop_back();

        /** Loop through the system deleting rows when information matches **/
        do {
                deleted = false; ///Reset the deleted row status
                /** Loop through each column **/
                for (size_t i = 0; i < columns.size(); i++){
                        /** Go to column if name matches **/
                        if (columns[i].colName == whereName) {
                                /** Go through each row **/
                                for (size_t j = 0; j < columns[i].colData.size(); j++){
                                        /** Switch based on the delete type **/
                                        if (deleteType == '=') {
                                                /** Change index and status if a delete row is
found **/
                                                if (columns[i].colData[j] != "ERASE") {
                                                        if (columns[i].colData[j] == whereValue)
{
                                                                deleteHere = j;
                                                                deleted = true;
                                                                break;
                                                        }
                                                }
                                        }
                                        /** Change index and status if a delete row is found **/
                                        else if (deleteType == '>') {
                                                if (columns[i].colData[j] != "ERASE") {
                                                        if (stof(columns[i].colData[j]) >
stof(whereValue)) {
                                                                deleteHere = j;
                                                                deleted = true;
                                                                break;
                                                        }
                                                }
                                        }
                                }
                        }
                }
                /** If deleted go through row and change value to "ERASE" and increase
count**/
                if (deleted) {
                        for (size_t i = 0; i < columns.size(); i++) {
                                columns[i].colData[deleteHere] = "ERASE";
```

```
                        }
                        recordCount++;
                }
        } while (deleted); ///Loop as long as something deletable is found to get all

        /**Rewrite table and reread into system after a delete **/
        rowAdjust = recordCount;
        writeTable(tablePath);
        reInput();

        /** Output information **/
        if (recordCount == 1)
                cout << "1 record deleted." << endl;
        else
                cout << recordCount << " records deleted." << endl;
        return true; ///Return true for end of processing
}
/*!
 * \brief Reinput Method for the Table class
 *
 * This function will rewrite the Table class to the database. This
 * function works similar to the Constructor excepts it is rereading from files
 * with an existing database. The function utilizes the sizes of the
 * existing row and column lists to provide it for the next constructing
 * of the database objects.
 *
 */
void Table::reInput() {
        rowAdjust = 0; ///Resets the adjustment amount
        columns.clear(); ///Erases all columns in system

        string input; ///String containing input from file
        string newColName; ///String containing the columns name
        string newColType; ///String containing the columns type
        int newColSize; ///Int containing the column size if applicable
        size_t colNum; ///Int containing the number of columns

        /** Create ifstream based on the path parameter **/
        ifstream inputFile(tablePath);

        /** Check if the file exists and open **/
        if(inputFile.is_open()){
                /** Get column and row size from file **/
                inputFile >> colNum;
                inputFile >> rowNum;
                /** Input column settings based on number of columns **/
                for (size_t i = 0; i < colNum; i++) {
                        inputFile >> newColName;
                        inputFile >> newColType;
                                /** Input the individual size of a column if applicable **/
```

```
                        if (newColType == "char" || newColType == "varchar") {
                                inputFile >> newColSize;
                        }
                        else ///Set the column size to 0 if an int or float
                                newColSize = 0;
                    /** Construct a new column and add to list from settings **/
                                columns.push_back(Column(newColName, newColType, newColSize));
                        }
                /** Get column information based on the row number **/
                        for (size_t i = 0; i < rowNum; i++) {
                    /**Go through each column pushing info for each row **/
                                for (size_t j = 0; j < colNum; j++) {
                                        inputFile >> input;
                                        columns[j].colData.push_back(input);
                                }
                        }
                        /** Close file **/
                        inputFile.close();
                }
}
```

The select types function is the final function implemented in this iteration. This version differs from the main select function in that it can select individual columns instead of just all the information. This function is called in the same way select is with this function called when the input differs from the normal SELECT * function. This function allows users to state what columns to print and what rows to print based on information. Currently the method of selecting is hard coded but can be adjusted in future iterations. This implementation is shown below:

```
/*!
 * \brief Function for the select table function
 *
 * This function will select a table from the internal Table list.
 * When the main parser detects the SELECT command and a database
 * has been used it will direct to this function. The function will
 * then list all the column information or output the appropriate
 * error messages. This function also leads to the joining functions.
 *
 * \return boolean true/false
 *      True or false value depending on state of function. False only sent
 *      if the parser cannot interpret the error and will send true in case
 *      of a processed create command if it is created or not. This function
 *      will additionally detect if the user is selecting all or selecting a
 *      particular column and direct to the correct function.
 */
bool Database::selectTable() {
    string input; /// String holding user input
    vector <string> types; /// Boolean selected types if not all are selected
    bool typeBased = false; /// Boolean if all types are selected or not
    string tempTableName; /// String holds a temp name for joining
```

```
/** Receive and Check Input From User **/
cin >> input;
if (input != "*") /// Switch if not all types selected
{
    typeBased = true; /// Change type of function to use
    /** Receive input types **/
    if (input.back() == ',') /// Process first input to see if anymore
    {
            input.pop_back();
            types.push_back(input);
            /** Loop if more than one input checking for ',' **/
            while (true) {
                    cin >> input;
                    if (input.back() == ',')  /// Check if more inputs and process
                    {
                            input.pop_back();
                            types.push_back(input);
                    }
                    else /// Get final input and leave loop
                    {
                            types.push_back(input);
                            break;
                    }
            }
    }
    else /// No more inputs after the first input
            types.push_back(input);
}

/** Process following input **/
cin >> input;
if (input != "FROM" && input != "from") /// Unknown command if TABLE not specified
    return false;
cin >> input;
/** Switch input based on function used **/
if (!typeBased) {
    if (input.back() != ';') /// Switch if a join command instead of select all
    {
            tempTableName = input; /// Temporary storage for table one name
            /** Continue input if not a standard selection **/
            cin >> input;
            /** Switch to a regular join **/
            if (input.back() == ',')
                    return innerJoin(tempTableName);
            else /// Check if a inner or outer join
            {
                    /** Get input and check for an inner join **/
                    cin >> input;
                    if (input == "inner"){
```

```
                        cin >> input;
                        if (input == "join")
                                return innerJoin(tempTableName);
                        else /// Invalid command
                                return false;
                }
                else if (input == "left"){
                        /** Get input and check for an outer join **/
                        cin >> input;
                        if (input == "outer"){
                                cin >> input;
                                if (input == "join")
                                        return outerJoin(tempTableName);
                                else /// Invalid command
                                        return false;
                        }
                        else /// Invalid command
                                return false;
                }
                else /// Invalid command
                        return false;
        }
    }
    /** Process input for table location **/
    input.pop_back();
}

/** Change input to lower case **/
transform(input.begin(), input.end(), input.begin(), ::tolower);

/** Locate Table in Internal List **/
for (auto inputTable: tables){
    if (inputTable.lowerName == input) {
                /** Change function based on type of input **/
                if (typeBased)
                        return inputTable.selectTypes(types);
                else
                        inputTable.select();
                return true;
        }
    }

/** Output error if location failed **/
cout << "!Failed to query table "
     << input
     << " because it does not exist." << endl;

return true; ///Return true for end of processing
}
```

The select function is the main point of passage for the program to enter the inner or outer join command as the parser will detect if more than one table is attempting to be selected and divert to the inner or outer join functions accordingly.

Joining

The joining functions are accessed from the select function when the parser finds more than one table to parser. In the case of a regular selection of two tables and use of the where statement, the inner join function is used as it is the same method but a different parser format. Otherwise, the inner join and outer join functions are called when INNER JOIN or LEFT OUTER JOIN are parsed in the string.

The inner join works by first finding an index from the table list for both of the tables. Once both tables are found and the indexes recorded, the indexes for the two corresponding columns to compare are found. If any indexes are not found, the command will fail.

From these indexes it is then possible for the algorithm to go through each tuple of table 1. For each table 1 tuple, it is compared to all tuples of table 2 to find matching columns given in the join command. If a match is found, the corresponding tuples are printed side by side. This process continues as each tuple in table 1 will be compared to every tuple in table 2. Below is the code used for the inner join functionality.

```
/*!
 * \brief Function for the inner join table function
 *
 * This function will select two tables from the internal Table list.
 * When the main parser detects the INNER JOIN command and a database
 * has been used it will direct to this function. The function will
 * then list all the column information or output the appropriate
 * error messages. This function also leads to the joining functions.
 *
 * \param[in] string firstTable
 *            String corresponding name of the first table to join
 *
 * \return boolean true/false
 *     True or false value depending on state of function. False only sent
 *     if the parser cannot interpret the error and will send true in case
 *     of a processed create command if it is created or not. This function
 *     will additionally detect if the user is selecting all or selecting a
 *     particular column and direct to the correct function.
 */
bool Database::innerJoin(string firstTable) {
    string input; /// String holds input
    string secondTable; /// String holds the second table
    string firstType; /// String holds first compare type
    string secondType; /// String holds second compare type
```

```
    int firstTableIndex = -1; /// Int index of table 1
    int secondTableIndex = -1; /// Int index of table 2
    int firstTableTypeIndex = -1; /// Int index of table 1's search type
    int secondTableTypeIndex = -1;/// Int index of table 2's search type
    bool firstInput = true; /// Stores if first input to print

    /** Get second table name and process **/
    cin >> secondTable;
    cin >> input;
    cin >> input;

    /** Check for correct syntax **/
    if (input != "where" && input != "on")
        return false;
    cin >> input;

    /** Check if proper '.' format **/
    if (input.find('.') == string::npos)
        return false;

    /** Get first table column type **/
    firstType = input.substr(input.find('.') + 1);

    /** Check if proper '=' format **/
    cin >> input;
    if (input != "=")
        return false;

    /** Get second table column type **/
    cin >> input;
    if (input.find('.') == string::npos ) /// Check if proper '.' format
        return false;
    if (input.back() != ';') /// Check if ending in ';'
        return false;
    input.pop_back();
    secondType = input.substr(input.find('.') + 1);

    /** Change names to lower case **/
    transform(firstTable.begin(), firstTable.end(), firstTable.begin(), ::tolower);
    transform(secondTable.begin(), secondTable.end(), secondTable.begin(), ::tolower);

    /** Check if index exists for tables **/
    for (size_t i = 0; i < tables.size(); i++) /// Iterate through all tables
    {
        if (tables[i].lowerName == firstTable) /// Check if name matches table 1
               firstTableIndex = i;
        else if (tables[i].lowerName == secondTable) /// Check if name matches table 2
               secondTableIndex = i;
    }
```

```
    /** Check for failure **/
    if (firstTableIndex == -1 || secondTableIndex == -1)
        return false;

    /** Check if index exists for first table **/
    for (size_t i = 0; i < tables[firstTableIndex].columns.size(); i++)
        if (tables[firstTableIndex].columns[i].colName == firstType)
                firstTableTypeIndex = i;

    /** Check if index exists for first table **/
    for (size_t i = 0; i < tables[secondTableIndex].columns.size(); i++)
        if (tables[secondTableIndex].columns[i].colName == secondType)
                secondTableTypeIndex = i;

    /** Check for failure **/
    if (firstTableTypeIndex == -1 || secondTableTypeIndex == -1)
        return false;

    /** Print table 1 header **/
    for (auto inputColumn: tables[firstTableIndex].columns) {
        if (firstInput) ///Do not print dividers if first input
                firstInput = false;
        else ///Print dividers
                cout << "|";
        cout << inputColumn.colName << " " << inputColumn.colType;
        /** If char print the column size **/
        if (inputColumn.colType == "char" || inputColumn.colType == "varchar")
                cout << "(" << inputColumn.colSize << ")";
    }

    /** Print table 2 header **/
    for (auto inputColumn: tables[secondTableIndex].columns) {
        /** Print dividers and input **/
        cout << "|" << inputColumn.colName << " " << inputColumn.colType;
        /** If char print the column size **/
        if (inputColumn.colType == "char" || inputColumn.colType == "varchar")
                cout << "(" << inputColumn.colSize << ")";
    }
    firstInput = true;
    cout << endl;

    /** Print joined table **/
    for (size_t i = 0; i < tables[firstTableIndex].rowNum; i++) /// Iterate through table 1
    {
        /** Iterate through table 2 **/
        for (size_t j = 0; j < tables[secondTableIndex].rowNum; j++){
                /** Check to see if a matching tuple is found **/
                if (tables[firstTableIndex].columns[firstTableTypeIndex].colData[i] ==
                        tables[secondTableIndex].columns[secondTableTypeIndex].colData[j]) {
                        /** Loop through table 1 columns and print at tuple **/
```

```
                        for (auto outputCol: tables[firstTableIndex].columns) {
                                if (firstInput) ///Do not print dividers if first input
                                        firstInput = false;
                                else ///Print dividers
                                        cout << "|";
                                input = outputCol.colData[i];
                                /** Adjust input for character columns **/
                                if (outputCol.colType == "char" || outputCol.colType ==
"varchar") {
                                        input.erase(0,1);
                                        input.pop_back();
                                }
                                cout << input;
                        }
                        /** Loop through table 2 columns and print at tuple **/
                        for (auto outputCol: tables[secondTableIndex].columns) {
                                /** Print dividers **/
                                cout << "|";
                                input = outputCol.colData[j];
                                /** Adjust input for character columns **/
                                if (outputCol.colType == "char" || outputCol.colType ==
"varchar") {
                                        input.erase(0,1);
                                        input.pop_back();
                                }
                                cout << input;
                        }
                        firstInput = true;
                        cout << endl;
                }
        }
    }
    /** End Input **/
    return true;
}
```

In the case of an outer join, the same functionality is used. Besides differences in the parser, the only difference in the algorithm is to check if a match was found for each tuple in table 1. If no match is found, the information for that tuple is still printed but with blank information for table 2. This implementation for outer join is shown below.

```
/*!
 * \brief Function for the outer join table function
 *
 * This function will select two tables from the internal Table list.
 * When the main parser detects the OUTER JOIN command and a database
 * has been used it will direct to this function. The function will
 * then list all the column information or output the appropriate
```

```
 * error messages. This function also leads to the joining functions.
 *
 * \param[in] string firstTable
 *            String corresponding name of the first table to join
 *
 * \return boolean true/false
 *     True or false value depending on state of function. False only sent
 *     if the parser cannot interpret the error and will send true in case
 *     of a processed create command if it is created or not. This function
 *     will additionally detect if the user is selecting all or selecting a
 *     particular column and direct to the correct function.
 */
bool Database::outerJoin(string firstTable) {
    string input; /// String holds input
    string secondTable; /// String holds the second table
    string firstType; /// String holds first compare type
    string secondType; /// String holds second compare type
    int firstTableIndex = -1; /// Int index of table 1
    int secondTableIndex = -1; /// Int index of table 2
    int firstTableTypeIndex = -1; /// Int index of table 1's search type
    int secondTableTypeIndex = -1;/// Int index of table 2's search type
    bool firstInput = true; /// Boolean Stores if first input to print
    bool printedThisRow = false; /// Boolean stores

    /** Get second table name and process **/
    cin >> secondTable;
    cin >> input;
    cin >> input;

    /** Check for correct syntax **/
    if (input != "on")
        return false;
    cin >> input;

    /** Check if proper '.' format **/
    if (input.find('.') == string::npos)
        return false;

    /** Get first table column type **/
    firstType = input.substr(input.find('.') + 1);

    /** Check if proper '=' format **/
    cin >> input;
    if (input != "=")
        return false;

    /** Get second table column type **/
    cin >> input;
    if (input.find('.') == string::npos ) /// Check if proper '.' format
        return false;
```

```
    if (input.back() != ';') /// Check if ending in ';'
        return false;
    input.pop_back();
    secondType = input.substr(input.find('.') + 1);

    /** Change names to lower case **/
    transform(firstTable.begin(), firstTable.end(), firstTable.begin(), ::tolower);
    transform(secondTable.begin(), secondTable.end(), secondTable.begin(), ::tolower);

    /** Check if index exists for tables **/
    for (size_t i = 0; i < tables.size(); i++) /// Iterate through all tables
    {
        if (tables[i].lowerName == firstTable) /// Check if name matches table 1
               firstTableIndex = i;
        else if (tables[i].lowerName == secondTable) /// Check if name matches table 2
               secondTableIndex = i;
    }

    /** Check for failure **/
    if (firstTableIndex == -1 || secondTableIndex == -1)
        return false;

    /** Check if index exists for first table **/
    for (size_t i = 0; i < tables[firstTableIndex].columns.size(); i++) {
        if (tables[firstTableIndex].columns[i].colName == firstType)
               firstTableTypeIndex = i;
    }

    /** Check if index exists for first table **/
    for (size_t i = 0; i < tables[secondTableIndex].columns.size(); i++) {
        if (tables[secondTableIndex].columns[i].colName == secondType)
               secondTableTypeIndex = i;
    }

    /** Check for failure **/
    if (firstTableTypeIndex == -1 || secondTableTypeIndex == -1)
        return false;

    /** Print table 1 header **/
    for (auto inputColumn: tables[firstTableIndex].columns) {
        if (firstInput) ///Do not print dividers if first input
               firstInput = false;
        else ///Print dividers
               cout << "|";
        cout << inputColumn.colName << " " << inputColumn.colType;
        /** If char print the column size **/
        if (inputColumn.colType == "char" || inputColumn.colType == "varchar")
               cout << "(" << inputColumn.colSize << ")";
    }
```

```
    /** Print table 2 header **/
    for (auto inputColumn: tables[secondTableIndex].columns) {
        /** Print dividers and input **/
        cout << "|" << inputColumn.colName << " " << inputColumn.colType;
        /** If char print the column size **/
        if (inputColumn.colType == "char" || inputColumn.colType == "varchar")
               cout << "(" << inputColumn.colSize << ")";
    }
    firstInput = true;
    cout << endl;

    /** Print joined table **/
    for (size_t i = 0; i < tables[firstTableIndex].rowNum; i++) /// Iterate through table 1
    {
        printedThisRow = false;
        /** Iterate through table 2 **/
        for (size_t j = 0; j < tables[secondTableIndex].rowNum; j++){
               /** Check to see if a matching tuple is found **/
               if (tables[firstTableIndex].columns[firstTableTypeIndex].colData[i] ==
                      tables[secondTableIndex].columns[secondTableTypeIndex].colData[j]) {
                      printedThisRow = true;
                      /** Loop through table 1 columns and print at tuple **/
                      for (auto outputCol: tables[firstTableIndex].columns) {
                             if (firstInput) ///Do not print dividers if first input
                                    firstInput = false;
                             else ///Print dividers
                                    cout << "|";
                             input = outputCol.colData[i];
                             /** Adjust input for character columns **/
                             if (outputCol.colType == "char" || outputCol.colType ==
"varchar") {
                                    input.erase(0,1);
                                    input.pop_back();
                             }
                             cout << input;
                      }
                      /** Loop through table 2 columns and print at tuple **/
                      for (auto outputCol: tables[secondTableIndex].columns) {
                             /** Print dividers **/
                             cout << "|";
                             input = outputCol.colData[j];
                             /** Adjust input for character columns **/
                             if (outputCol.colType == "char" || outputCol.colType ==
"varchar") {
                                    input.erase(0,1);
                                    input.pop_back();
                             }
                             cout << input;
                      }
                      firstInput = true;
```

```
                        cout << endl;
                }
        }
        /** Print a blank row if no match found **/
        if (printedThisRow == false){
                /** Iterate through first table columns **/
                for (auto outputCol: tables[firstTableIndex].columns) {
                        if (firstInput) ///Do not print dividers if first input
                                firstInput = false;
                        else ///Print dividers
                                cout << "|";
                        input = outputCol.colData[i];
                        /** Adjust input for character columns **/
                        if (outputCol.colType == "char" || outputCol.colType == "varchar") {
                                input.erase(0,1);
                                input.pop_back();
                        }
                        cout << input;
                }
                for (auto outputCol: tables[secondTableIndex].columns) {
                        /** Print dividers **/
                        cout << "|";
                }
                firstInput = true;
                cout << endl;
        }
    }
    return true;
}
```

Conclusion

This project sets up the framework for future expansion of the project. By implementing the project through these separate classes, the parser can be more easily expanded for new commands and new functionality can be added.
By having both the physical version of the file system and the virtual version, it is easier to manipulate data as new commands are entered. One issue with this implementation is there is a disconnect between the program and the files while it is running. This could mean that if something else edits a file outside the program, the virtual version would no longer be in sync. This is something to be considered in future implementations.

In the past implementation of the project will read in the necessary information to manipulate columns and includes the ability to change information as necessary. This does, however, have some limitations in that there are multiple valid sql commands that will not work in this project.

This applies to the current implementation in that the joins are limited in that only joins based on if two columns in a tuple are equal work.

Installing

To install this project, download the given compressed file and open on an ECC computer. The program files are included in the main directory or in secondary include / src directories. The Debug directory contains the main executable project and the necessary make files for the project. The project can be made using the MAKE command in the debug directory.

Example of installation from :~/cmollise_pa3/CS457-2 (The unzipped projects files)

1. :~/cmollise_pa3/CS457-3$ cd Debug
2. :~/cmollise_pa3/CS457-3/Debug$ make

Testing:

3. :~/cmollise_pa3/CS457-3/Debug$ ./CS457-3 <PA3_test.sql

Testing

The program can be utilized through both traditional keyboard inputs as well as using the standard input key "<". All actions to the database are shown through messages on the command terminal. As the program runs the database is physically created in the Databases directory created in the Debug directory. Within this directory will be the directories for each individual database within the text files for each table.

To run the test go to the Debug directory to be able to launch the project. From this directory the program can be started using

```
# ./CS457-3 <enter>
```

This will allow a user to input commands as shown in the expected input. Additionally the program can be run using the standard input "<"

```
# ./CS457-3 <PA3_test.sql
```