CS457 Project 4

Yifeng Qin

11/29/20

1 How does my program organize multiple databases (Project 1)

The design of my program follows the example in the Assignment Overview.
One Directory corresponds to a database.
For example the parent directory will be /your_home/cs457
If you were to create a new database, it would create a new directory inside the parent directory.
Example: /your_home/cs457/db_1 (is what it would look like after creating db_1)
Then if you want to remove this database it would remove that directory with the db name.
Example: /your_home/cs457 (is what it would look like after removing db_1)
You can create as many distinct databases within the parent directory because they will all be represented by a directory. This means you can also delete a database by deleting the corresponding directory.Look in the functional requirements section for a more detailed explanation.

2 How does my program organize multiple tables (Project 1)

One file corresponds a table in a directory.
For example the parent directory will be /your_home/cs457
To create a table you have to already created a database because you need to have a directory to place the table file in.
You also have to USE a database to create a table in that database.
When wanting to create a table we join the parent directory with database directory and create a file with the table name.
So each database can have the same distinct names. Look in the functional requirements section for a more detailed explanation.

3 How does my program store data (Project 2)

So all data that is inserted into a table is written to the corresponding file for that table. Meaning that everything is written to the file as a string including numbers. Each line is separated by a newline and each column is separated by a '|'. I created a helper function that reads all the data from the file and returns the all the data in a list format. All the data that is read from the file is still in string format. Then I take each line and make it it's own list making the data structure a list of lists. Each list represents a line and each index in that line represents a specific column. I decided against a tuple because specific operations like '==' and '!=' still compare numbers the same in string and number format. If I see a '>' or '<' operation that means I have to compare numbers, meaning that I will cast the string to a float to perform the logic operation. Look in the functional requirements section for a more detailed explanation.

4 How does my program join tables (Project 3)

To join the tables together we use a double for loop depending on the type of join. Since the the '=' operator is the same as the inner join operation. We can combine these two into one function meaning we only need one function be called if we see these two. For this inner join we must find the index of both parameters we we checking and run through every combination to check if they match. If they match we can concatenate them together and add it to the list to print to the terminal. The outer join is very similar to inner join. Since it's a left outer join for this project we need all show all the data on the left side. But we must still match all the data on the right side. So we set a flag to check if we find a match and if we don't see a match we have to append only the left side to the list to be printed. Look in the functional requirements section for a more detailed explanation.

5 Begin Transaction and Commit (Project 4)

Starts the transaction and waits for and update command or commit command. Any other command is not viable within the transaction because it does no modifications. Before anything it will lock for locked files in the current directory. If there are any it will append the tables to a list. It it will check if the table the update command wants to be modify is not locked. If it is not locked it will create a copy of that file with a lock appended to the name in the directory. Then it will call the update function that was implemented in the previous projects. If the table that wants to be updated is in the locked tables array then it will print an error and not update the table. This function will continue until it sees an commit command. If there is nothing that has been updated it will abort the transaction. If there is something modified then it will take the locked file and overwrite the original table. Then it will delete the lock table. Look in the functional requirements section for a more detailed explanation.

6 Functional Requirements

List the functional requirements that were listed in the assignment. Each functionality correlates to a function, which can be found at the end of the document.

6.1 Project 1

Database Creation - 4.2.1 create_database(self, db):
Joins the parent directory with the entered database name. Checks if the database already exists. If it exists already it will print out and error, if not it will create a directory of the database name.

Deletion - 4.2.2 drop_database(self, db) :
Joins the parent directory with the entered database name. Checks if the database already exists. If it does not exist it will print out and error, else it will delete a directory of the database name.

Table Creation - 4.2.4 create_table(self, tbl, inp) :
Gets the path to that table and will check if that already exists. Then if it does not exist, it will change the directory and create a file of that table name in that directory. If there are inputs to within the command it will write those variables to that file.

Deletion - 4.2.7 drop_table(self, tbl):
Joins the parent directory, database directory, and the name of the table, and if it exists delete that path. If it does not exist it will error and print the error message.

Update - 4.2.6 alter_table(self, tbl, inp):
Joins the parent directory, database directory, and the name of the table, and if it exists open the file and

append the update values to file. If the table does not exist it will print and error message.

Query - 4.2.5 select_all(self, table):
Joins the parent directory, database directory, and the name of the table, and if it exits it will open the file and read all the contents into a list. Then it will print out all the contents to the terminal. If it does not exits it will print and error out to the terminal.

6.2 Project 2

Insert - 5.2.8 insert_table(self, table, new_data):
Checks if there the table is a valid path, then it will read all the current data from that table and store it in a list of lists. We take the new data that needs to be inserted and place them in the respective columns. We append that new data to the new current data. Then then call a helper function to write the new data back to the file.

Delete - 5.2.12 delete_items(self, table, data):
Checks if the table is a valid path. Reads all the data from the current table and stores it in a list of list. Checks the logic operation to determine how to compare the values that need to be deleted. Then calls a helper function to delete the data from the current list of lists. This helper function will find the index of the variable we are comparing. Then we use the eval function to turn the string operation into a logic operation and perform it on the index of the variable we are comparing. If the variable we are comparing satisfies the condition we will delete that line from the current data. When all lines have been compared, it will call the insert helper function to write the new data back to the file.

Modify: - 5.2.10 update_table(self, table, data)
Checks if the table is valid. Then it will read all the data from the current table and store it in a list of lists. It will then pass all that information the update helper function to update the current data. This helper function will first find the index of the variable we want to compare and the variable we want to change. Uses the eval function to perform a logic operation based on the operation in the command. It will compare the index of the variable compared and if it satisfies the condition it will change the value at the index of the desired variable. Then it will call the insert helper function to write all the data back the file.

Query - 5.2.14 select_specific(self, var, table, where):
Checks if the table is valid. Then it will read all the data from the current table and store it in a list of lists. It find all the missing variables that are not needed to be shown. It will take all those unnecessary indexes and delete the columns from the print. Then it will call the delete helper function to get rid of rows that don't satisfy the where clause. This will return the final result to be printed to the terminal.

6.3 Project 3

Select - 4.2.5 select_all(self, table):
This function will determine how many different tables it has to show and determine if it needs to join the tables together. It will take the input command and then slice it to get the table names and if there is a join. We store the table names and then create a dictionary where the key is the name of the table and the value represents the data in that table in list form. Then it calls a helper function to determine if it needs to join the tables and calls the corresponding function to join it.

Inner Join - 6.2.20 inner_join(rows, res, d, tbls_alias, cols_id):
Will take the data from the two tables and run a double for loop to compare all the combinations. If will first look at the variables in each table and determine which columns need to be coma pared. Then it will

look through all the combinations and if the columns match based on the logic operation it will concatenate them together and append it to the list to be printed. Because the inner join and the equals operator have the same implementation we just call this same function for both.

Outer Join - 6.2.21 outer_join(rows, res, d, tbls_alias, cols_id):
Will take the data from the two tables and run a double for loop to compare all the combinations. If will first look at the variables in each table and determine which columns need to be coma pared. Then it will look through all the combinations and if the columns match based on the logic operation it will concatenate them together and append it to the list to be printed. For this left outer join we have a flag variable in the first for loop to check if there is a match. If there is no match then we still need to include the data in the left side so we must append the data without a match from the right side.

6.4 Project 4

Transaction - 7.2.22 transaction(self):
The transaction will first get any locked files by looking into the current directory and finding all the flies that contain locked in their name. It will store it in a list to represent all the tables that are currently locked. This will prevent another process from trying to access that file. It will then wait for a update or commit to run the respective commands. If the command is update it will check if the table that it wants to modify in is in the locked table list then it will not do anything and print and abort. If the table is not locked then it will create a copy of the original table file and name it with a lock appended to the end. Now it will update the locked table instead of the original so other process can look at the original without see new updates. Once everything is verified it will call the update command implemented in earlier assignments with all the new data. Another command the function looks for is the commit command. If it sees the commit command then it will call the commit function if there has been more then one successful update. All other inputs will result in printing an error and wanting a valid command.

Commit - 7.2.23 commit(path, lock_path):
Once there is a commit we want to move all the data in the locked file to the original table. So we just overwrite the original table with the locked table, then delete the locked table.

7 Functions

This section contains all the code that runs the functionality of the program. Broken down into python files and functions.

7.1 main.py

Main driver that takes in the input from the file and calls functions in the run_script.py file to execute those commands.

7.1.1 get_input()

```
1     def get_input():
2         command = input("Enter Command: ")
3         if '--' in command:  # removes comments at end of line
4             command = command[:command.index('--')-1]
5 
6         return command
7 
```

7.1.2 run_commands_inline()

```
    """
    Calls the helper function get_input to get command from terminal
    :return: None
    """
    def run_commands_inline():

        command = get_input()
        if '.exit' in command:
            print("All Done.")
            return
        else:
            command = re.sub(r"[\n\t]*", "", command)  # removes random special characters like tabs
            l = command.split(' ')  # splits the string command into a list based on spaces
            command = command.upper()  # converts the command to all uppercase so it can cover case sensitivity
            size = len(l)  # gets length to handle missing spaces
            if 'CREATE DATABASE' in command:
                if size == 3:  # checks if all arguments are present
                    script.create_database(l[2][:-1].upper())  # only gets the database name and removes the ';' from the back
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'DROP DATABASE' in command:
                if size == 3:  # checks if all arguments are present
                    script.drop_database(l[2][:-1].upper())   # only gets the database name and removes the ';' from the back
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'DROP TABLE' in command:
                if size == 3:  # checks if all arguments are present
                    script.drop_table(l[2][:-1].upper())   # only gets the database name and removes the ';' from the back
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'USE' in command:
                if size == 2:  # checks if all arguments are present
                    script.use_database(l[1][:-1].upper())   # only gets the database name and removes the ';' from the back
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'CREATE TABLE' in command:
                command = " ".join(l)
                idx = command.index('(')
                var = command[idx:]
                temp = command[:idx].split(' ')
                if size >= 3:  # checks the the minimum amount of arguments are present
                    script.create_table(temp[-1], var[1:-2])  # passes in the name of and the sliced variables to input
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'SELECT * FROM' in command:
                if size == 4:
                    script.select_all_no_condition(l[3].upper()[:-1])
                elif size > 4:  # checks if all arguments are present
                    script.select_all(l[3].upper(), l[3:])  # only gets the table name and removes the ';' from the back
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'ALTER TABLE' in command:
```

```
                if size >= 4:  # checks if all arguments are present
                    command = " ".join(l[4:])  # gets all the variables after the table name and converts it into a string
                    command = command[:-1]  # removes the ';' from the back of string
                    script.alter_table(l[2].upper(), command)  # passes in the name of table and sting of variables
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'INSERT' in command:
                command = " ".join(l)
                idx = command.index('(')
                var = command[idx:]
                temp = command[:idx].split(' ')
                if size >= 4:  # checks if all arguments are present
                    script.insert_table(temp[-2].upper(), var[1:-2])
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'UPDATE' in command:
                if size >= 8:  # checks if the minimum amount of variables are present
                        script.update_table(l[1].upper(), l[2:])
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'DELETE' in command:
                if size >= 7:  # checks if the minimum amount of variables are present
                    script.delete_items(l[2].upper(), l[3:])
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'SELECT' in command:
                if size >= 7:  # checks if the minimum amount of variables are present
                    from_idx = l.index('from')
                    script.select_specific(l[1:from_idx], l[from_idx+1].upper(), l[from_idx+2:])
                else:
                    print('Syntax Error:', command)  # if size does not match there has to be a syntax error with cmd
            elif 'BEGIN TRANSACTION' in command:
                script.transaction()
            elif 'COMMIT' in command:
                print('Transaction abort.')
            elif '.EXIT' in command:
                print('All Done')
                return
            else:  # if the command is not recognised it's and unknown command or there is something wring with the syntax
                print('Syntax Error | Unknown Command')
                print(command)
            run_commands_inline()


```

7.2 run_script.py

Contains all the functions that will execute the commands from the script.

7.2.1 create_database(self, db)

```
        """
        Joins the parent directory with the entered database name. Checks if the database already exists. If it exists
        already it will print out and error, if not it will create a directory of the database name.
        :param db: string that contains the name of the database
```

```
        :return: None
        """
        path = os.path.join(self.parentDir, db)  # joins cwd and db name
        if os.path.exists(path):  # check if path exists
            output = '!Failed to create database ' + db + ' because it already exists'
            print(output)
        else:
            os.mkdir(path)  # creates directory of path
            output = 'Database ' + db + ' created.'
            print(output)

```

7.2.2 drop_database(self, db)

```
        """
        Joins the parent directory with the entered database. Checks if the database already exists, and will either
        error out or delete that database.
        :param db: string that contains the name of the database
        :return: None
        """
        path = os.path.join(self.parentDir, db)  # check if path exists
        if os.path.exists(path):   # check if path exists
            cmd = 'rm ' + '-rf ' + path  # concatenate command to input
            os.system(cmd)  # runs the command
            output = 'Database ' + db + ' deleted.'
            print(output)
        else:
            output = '!Failed to delete ' + db + ' because it already exists.'
            print(output)

```

7.2.3 use_database(self, db)

```
    """
    Joins the parent directory with the entered database. Checks if the database already exists, and will either
    error out change the working directory to the database.
    :param db: string that contains the name of the database
    :return: None
    """
    path = os.path.join(self.parentDir, db)  # joins cwd and db name
    if os.path.exists(path):  # check if path exists
        os.chdir(path)  # changes cwd to this path
        self.dbDir = path
        output = 'Using database ' + db + '.'
        print(output)
    else:
        print('Cannot Use Database | Does Not Exist')

```

7.2.4 create_table(self, tbl, inp)

```
    """
    Gets the path to that table and will check if that already exists. Then if it does not exist, it will change the
    directory and call a helper function to append data to the table.
    :param tbl:
    :param inp: Contains all the data that will be entered into the table
    :return: None
    """
    path = os.path.join(self.dbDir, tbl)  # joins cwd and db name
    if os.path.exists(path):   # check if path exists
```

```
        output = '!Failed to create table ' + tbl + ' because it already exists.'
        print(output)
    else:
        os.mknod(path)  # creates file system of path
        out = inp.split(',')
        out = "|".join(out)
        f = open(path, "a")  # opens file
        f.write(out)  # write to file
        f.close()  # close file
        output = 'Table ' + tbl + ' created.'
        print(output)

```

7.2.5 select_all(self, table)

```
        """
        Checks if the table exists and then reads all the data from the file and prints it
    out.
        :param table: String that contains name of the table
        :param inp:
        :return: Check tables first * Notes
        """
        tbls = []
        tbls_alias = []
        cmd = " ".join(inp)
        d = collections.defaultdict(list)

        if 'inner join' in cmd:
            where_idx = inp.index('on')
            new = " ".join(inp[:where_idx])
            new = new.replace(' inner join', ',')
            type = 'inner join'
        elif 'left outer' in cmd:
            where_idx = inp.index('on')
            new = " ".join(inp[:where_idx])
            new = new.replace(' left outer', ',')
            type = 'left outer'
        else:
            where_idx = inp.index('where')
            new = " ".join(inp[:where_idx])
            type = 'equal'
        new = new.split(',')

        for i, t in enumerate(new):
            temp = t.split(' ')  # splits the table names to name and alias
            tbls.append(temp[-2].upper())
            tbls_alias.append(temp[-1])  # stores alias in list
            path = os.path.join(self.dbDir, tbls[i])  # joins cwd and tbl name
            if os.path.exists(path):
                d[tbls_alias[i]] = self.read_all(path)  # stores table data into a
    dictionary key = alias, value = data
            else:
                print('!Failed to query table ' + table + ' because it does not exist.')
                return
        logic = inp[where_idx + 1:]  # contains the rest of the logic comparisons
        res = self.join_helper(d, logic, tbls_alias, type)  # calls helper to combine tables
    , returns result
        for line in res:
            print(line)

```

7.2.6 alter_table(self, tbl, inp)

```
        """
        Will check if the table exists, if it doesn't exist it will print out an error.
```

```
        If it exists it will then append the extra values to the file.
        :param tbl: String containing name of table
        :param inp: string that need to be inputted
        :return: None
        """
        path = os.path.join(self.dbDir, tbl)  # joins cwd and db name
        if os.path.exists(path):  # check if path exists
            out = inp.split(',')  # takes the string a separates all the values by comma's
    and storing it into a list
            out = "|".join(out)  # joins the list back together into a string with a '|' at
    value
            f = open(path, "a")  # opens file
            f.write('| ' + out)  # adds '|' to separate existing values and then writes the
    output string
            f.close()  # close file
            output = 'Table ' + tbl + ' modified.'
            print(output)
        else:
            output = '!Failed to alter table ' + tbl + ' because it does not exist'
            print(output)

```

7.2.7 drop_table(self, tbl)

```
        """
        Checks if the table exists and if that table exists it will delete that path. If it
    does not exist
        it will error and print the error message
        :param tbl: string that contains the name of the table
        :return: None
        """
        path = os.path.join(self.dbDir, tbl)  # check if path exists
        if os.path.exists(path):  # check if path exists
            cmd = 'rm ' + '-rf ' + path  # concatenate command to run
            os.system(cmd)  # runs the command
            output = 'Table ' + tbl + ' deleted.'
            print(output)
        else:
            output = '!Failed to delete ' + tbl + ' because it does not exists.'
            print(output)

```

7.2.8 insert_table(self, table, new_data)

```
        """
        Takes in the name of the table and inserts the new data to that data.
        :param table: string with name of the tale
        :param new_data: contains the new data that needs to be inserted
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            res = []
            data = self.read_all(path)  # reads all the current data from the file and
    stores it in a list
            for line in data:  # reads the lines and splits the data
                res.append(line.split('|'))

            new_data = "".join(new_data)[7:-2]  # gets rid of the excess data
            new_data = new_data.split(',')  # splits the data on commas
            res.append(new_data)  # adds the new list to the rest of the data
            self.insert_helper(path, res)  # calls the helper function to print the data to
    the file
            output = '1 New Record Inserted'
            print(output)
```

```
        else:
            output = '!Failed to insert into table ' + table + ' because it does not exist.'
            print(output)

```

7.2.9 insert_helper(self, path, inp)

```
        """
        Helper that writes the new list of values to the file
        :param path: Path that leads to the table file
        :param inp: the new data that needs to be written to the file
        :return: None
        """
        f = open(path, "a")  # opens file
        f.truncate(0)
        for line in inp:
            out = "|".join(line)
            out += '\n'  # adds the new line
            f.write(out)  # write to file
        f.close()  # close file

```

7.2.10 update_table(self, table, data)

```
        """
        Updates specific value if conditions match. Finds the index of the desired variable. Then it changes the
        variable that needs to be changed based on the index of the variable that needs to change.
        :param table: string with name of the table
        :param data: contains the data to update the table
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            var = data[1]  # gets the specific values that are in the list
            var2 = data[5]
            operation = data[2]
            changeto = data[3]
            change = data[7][:-1]
            new, count = self.update_helper(path, var, var2, operation, change, changeto)  # calls helper func. to get/
            # new data data and count of how many objects were updated
            self.insert_helper(path, new)  # calls helper function to print new data to file
            if count > 1:  # print plural of singular
                output = str(count) + ' Records Modified'
            else:
                output = str(count) + ' Records Modified'
        else:
            output = '!Failed to update table ' + table + ' because it does not exist.'
        print(output)

```

7.2.11 update_helper(self, path, var, var2, operation, change, changeto)

```
        """
        Helper function that finds all the indexes of the values that need to be change. it will compare those values
        using the operators specified and then update the specific value that matches up the the desired conditions.
        :param path: string that contains the path of the file
        :param var: contains the variable that needs to match
        :param var2: contains the variable that needs to change
```

```
        :param operation: the operation that needs to compare the values to satisfy the conditions.
        :param change: The variable that needs to be changed
        :param changeto: The variable that needs to be changed too
        :return:
        """
        data = self.read_all(path)  # reads in all the data from the current file
        res = [[data[0]]]  # creates the final list to be printed to file
        curr = self.read_all_list(data)  # turns all the lines into lists making it a list of lists
        count = 0  # initiates the count for the number of changes
        where_idx = self.find_idx(curr[0], var2)  # finds the index that needs to found
        set_idx = self.find_idx(curr[0], var)  # finds the index that equates to being changed
        if operation == '=':
            operation = '=='

        for line in curr[1:]:
            if operation == '==':
                if eval('line[where_idx]' + operation + 'change'):  # if the variable satisfies the condition to change
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
                else:
                    res.append(line)
            elif operation == '!=':  # checks if operation is not equal because of differences in float and strings
                if eval('line[where_idx]' + operation + 'change'):  # if the variable satisfies the condition to change
                    res.append(line)
                else:
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
            else:
                if eval('float(line[where_idx])' + operation + 'float(change)'):  # <, > can only compare numbers
                    line[set_idx] = changeto  # changed the desired variable that needs to be changed
                    res.append(line)
                    count += 1
                else:
                    res.append(line)

        return res, count

```

7.2.12 delete_items(self, table, data)

```
        """
        Takes in the table and deletes the desired rows that satisfy the conditions.
        :param table: string with name of table
        :param data: contains the data and conditions needed to delete a row
        :return: None
        """
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            where = data[1]
            if data[2] == '=':  # turns it into and '==' to use as an operator
                operation = data[2] + data[2]
            else:
                operation = data[2]
            var = data[3][:-1]
```

```
            new, count = self.delete_helper(path, where, operation, var)  # calls helper to get new data and count
            self.insert_helper(path, new)  # calls helper to print new data to file
            if count >= 1:  # determines if singular or plural print
                output = str(count) + " Records Deleted"
            else:
                output = str(count) + " Record Deleted"
        else:
            output = '!Failed to delete items in table ' + table + ' because it does not exist.'
        print(output)

```

7.2.13 delete_helper(self, path, where, operation, var)

```
        """
        Helper to delete where it find the index of the variable that needs to compared. Then it will compare it with
        the operation specified to determine if that row needs to be deleted.
        :param path: string that has the path of the table
        :param where: the variable that needs to satisfy the condition
        :param operation: the logic operation
        :param var: the variable that needs to match the where to delete the row
        :return: the new list that needs to be printed and the count of the items deleted.
        """
        data = self.read_all(path)  # reads all the data that is stored in the current file
        res = [[data[0]]]  # initiates the new data with variable names
        curr = self.read_all_list(data)  # turns all the lines into a list
        where_idx = self.find_idx(curr[0], where)  # finds the index where the variable that needs to check is
        count = 0
        for line in curr[1:]:
            if operation == '==':  # checks if operation is equal because of differences in float and strings
                if eval('line[where_idx]' + operation + 'var'):  # eval turns string into a logic comparison
                    count += 1
                    continue
                else:
                    res.append(line)
            elif operation == '!=':  # checks if operation is not equal because of differences in float and strings
                if eval('line[where_idx]' + operation + 'var'):
                    res.append(line)
                else:
                    count += 1
            else:  # checks if operation is equal because of differences in float and strings
                if eval('float(line[where_idx])' + operation + 'float(var)'):  # greater or less than operations for num
                    count += 1
                    continue
                else:
                    res.append(line)
        return res, count

```

7.2.14 select_specific(self, var, table, where)

```
        """
        Selects a specific row depending on the where condition and the variables it wants to see.
        First deletes the the values that depend on the where clause. Then goes through and deletes the columns
        that are not needed to be seen.
```

```
        :param var: contains the variable for the where clause
        :param table: contains the name of the table in string
        :param where: contains the string of the conditions that needs to be satisfied
        :return:
        """
        obj = where[1]
        op = where[2]
        w = where[3][:-1]
        path = os.path.join(self.dbDir, table)
        if os.path.exists(path):  # check if path exists
            data = self.read_all(path)
            missing = []s
            variables = self.get_curr_var(data[0])
            for i in range(len(var)):
                var[i] = var[i].replace(',', '')  # gets rid of all commas
            for v in variables:
                if v not in var:  # checks if the variable needs to be seen
                    missing.append(v)
            res = self.select_specific_helper(path, missing, data, obj, op, w)  # calls
    helper to delete rows and cols
            for line in res:
                print("|".join(line))
        else:
            output = '!Failed to select items in table ' + table + ' because it does not
    exist.'
            print(output)

```

7.2.15 select_specific_helper(self, path, missing, data, obj, op, w)

```
        """
        It will call the delete helper to get rid of rows that don't have the specific value
    .
        Finds the missing values that are not needed to be displayed. It will delete those
    columns that are not
        needed.
        :param path: String with Path to the table
        :param missing: Contains the list of values you want
        :param data: list that has the data from the file
        :param obj: the object that wants to be found
        :param op: the logic operator
        :param w: the values that needs to be compared with the operator
        :return: return the list that needs to be inputted into the file
        """
        res = []
        data, count = self.delete_helper(path, obj, op, w)  # delete specific rows that
    satisfy condition
        for line in data:
            new = "| ".join(line)
            res.append(new.split('| '))
        for miss in missing:
            where_idx = self.find_idx(res[0], miss)
            for line in res:
                del line[where_idx]  # deletes the column from all the lines
        return res

```

7.2.16 read_all(self, path)

```
        """
        Checks if the table exists and then reads all the data from the file and prints it
    out.
        :param path: String that contains path of the table
        :return: None
        """
```

```
        self.data = []
        if os.path.exists(path):  # check if path exists
            with open(path) as file_in:  # starts reading from file
                for line in file_in:
                    self.data.append(line.rstrip())
            return self.data
        else:
            output = '!Failed to query table because it does not exist.'
            print(output)


```

7.2.17 read_all_list(self, data)

```
        """
        Turns a string into a list
        :param data: Data with strings in list
        :return: return a list with the strings separated into lists
        """
        curr = []
        for line in data:
            curr.append(line.split('|'))
        return curr

```

7.2.18 get_curr_var(self, data)

```
        """
        Splits the input data and removes commas in list leaving only the variable names
        :param data: List of data from the file
        :return: A new list that contains only the variable names
        """
        res = []
        new = data.split('| ')  # splits into list by '|'
        for var in new:
            s = var.split(' ')  # splits into a list generated by spaces
            s[0] = s[0].replace(',', '')  # removes commas
            res.append(s[0])
        return res

```

7.2.19 find_idx(self, inp, var)

```
        for i in range(len(inp)):
            if var in inp[i]:
                return i
        return -1

```

7.2.20 inner_join(rows, res, d, tbls_alias, cols_id)

```
        """
        Will perform the inner join by concatinating all the values that match the logic
    operation
        """
        for i in range(1, rows):  # compare all the rows and if the variables being compared
     match join them together
            for j in range(1, rows):
                d1 = d[tbls_alias[0]][i]
                d2 = d[tbls_alias[1]][j]
                if d1.split('|')[cols_id[0]] == d2.split('|')[cols_id[1]]:  # compare
    variables
```

```
                    res.append(d1 + '|' + d2)
        return res

```

7.2.21 outer_join(rows, res, d, tbls_alias, cols_id)

```
        """
        Will perform the inner join by concatenating all the values that match the logic operation
        """
        for i in range(1, rows):  # compare all the rows and if the variables being compared match join them together
            found = False  # determines if they found a match
            for j in range(1, rows):
                d1 = d[tbls_alias[0]][i]  # gets the data in the first table
                d2 = d[tbls_alias[1]][j]  # gets the data in the second table
                if d1.split('|')[cols_id[0]] == d2.split('|')[cols_id[1]]:  # compare variables
                    res.append(d1 + '|' + d2)
                    found = True
            if not found:
                res.append(d[tbls_alias[0]][i] + '|' * len(cols_id))
        return res

```

7.2.22 transaction(self)

```
        """
        Starts the transaction and waits for and update command or commit command. Any other command is not viable within the transaction because it does no modifications. Before anything it will lock for locked files in the current directory. If there are any it will append the tables to a list. It it will check if the table the update command wants to be modify is not locked. If it is not locked it will create a copy of that file with a lock appended to the name in the directory. Then it will call the update function that was implemented in the previous projects. If the table that wants to be updated is in the locked tables array then it will print an error and not update the table. This function will continue until it sees an commit command. If there is nothing that has been updated it will abort the transaction. If there is something modified then it will take the locked file and overwrite the original table. Then it will delete the lock table.
        """
        print('Transaction starts.')
        self.get_locked()
        path = None
        lock_path = None
        count = 0
        self.lock = True
        while self.lock:  # keeps running if more then one command
            command = self.get_input()
            command = re.sub(r"[\n\t]*", "", command)  # removes random special characters like tabs
            l = command.split(' ')  # splits the string command into a list based on spaces
            command = command.upper()  # converts the command to all uppercase so it can cover case sensitivity
            size = len(l)  # gets length to handle missing spaces

            if 'UPDATE' in command:
                if size >= 8:  # checks if the minimum amount of variables are present
                    tbl = l[1]
                    if tbl.upper() in self.lockedTables:  # checks if table is locked
                        print('Error: Table ' + l[1] + ' is locked!')
                        return
                    count += 1
                    path = os.path.join(self.dbDir, tbl.upper())  # gets path to the table
```

```
                    lock_path = os.path.join(self.dbDir, tbl.upper() + '_LOCK')  # creates a
 table that is locked
                    copyfile(path, lock_path)  # copy over the data from original table to
locked
                    self.update_table(tbl.upper() + '_LOCK', l[2:])  # update the locked
table
                else:
                    print('Syntax Error:', command)  # if size does not match there has to
be a syntax error with cmd
            elif 'COMMIT' in command:
                if count == 0:
                    print('Transaction abort.')
                    return
                self.commit(path, lock_path)
                self.lock = False
            else:
                print('Syntax Error: Not a Update Command | ', command)
        return

```

7.2.23 commit(path, lock_path)

```
        """
        When commit command seen, will copy over data from locked table to original table
        :param path: string with the path of the original table
        :param lock_path: string with the path of the locked table
        :return:
        """
        copyfile(lock_path, path)  # copies locked file data over to original table
        os.remove(lock_path)  # deletes the locked path
        print("Transaction committed.")

```

7.2.24 get_locked(self

```
    def get_locked(self):
        arr = os.listdir(self.dbDir)
        for tbl in arr:
            if 'LOCK' in tbl:
                self.lockedTables.append(tbl[:-5])

```