Carli DeCapito, Sanya Gupta, Eugene Nelson
CS 457: Database Management Systems
Programming Assignment 3 - Documentation
April 15, 2018

Functionality

This programming assignment required the group to implement additional functionalities similar to that of a SQLite Database Management System. This assignment primarily focused on two new functionalities related to table queries: inner joins and left outer joins. This project required the group to parse query requests specifically for these types of joins and output data to the user accordingly. Another implementation added to this project was correction of the file redirection. The previous problem in PA2 was that the group was using the file converted as a UNIX file, where the line endings were '\n'. However, the file provided was a DOS file, where the line endings were '\r\n'. The group had to consider this difference and alter the file read in accordingly.

Organization

The program uses both files systems and abstract data structures to organize and keep track of the existing databases and tables. *To read further on the database and table organization described in PA1 and PA2, continue reading after the "Special Circumstances" section.*

File System Organization and Use

The program stores all file information on the computer disk. All databases and their folders are stored in a directory called "Database System". For each database created, a directory with an equivalent name is created in the "Database System" directory. For each table created, a file with an equivalent name is created in the appropriate database directory. For each database and table that calls the "drop" function, the appropriate file or directory is removed from the disk. Each table file contains its corresponding columns names and datatypes at the top, followed by the actual values below. These attributes are written to the appropriate table file and can be deleted, modified, or queried. The table functionality also retains the previous project features, such as altered and dropped. By using the disk to store database and table information, this ensures that after the program ends, the database information is still available to the user and be used when for the next SQLite program.

Table Features

Tuple Organization

Tuples, or records, are stored inside the table files. The list of tuples follow the first line of attribute names and types. One single tuple takes up one line in a file. Each element in a tuple is separated by a tab in order to parse through for update, delete, and query purposes. For example, when adding values (1, 'Gizmo', 14.99) to the table Product(pid int, name varchar(20), price float), the file will be formatted like the following:

pid int<\t>name varchar(20)<\t>price float<\n>
1<\t>'Gizmo'<\t>14.99

The <\t> symbol resembles the white space tab in a file, while the <\n> represents the file pointer to the next line. This format can also be seen after running the test code and looking through the DBMS table files.

Inner Join

The inner join feature takes a query of attributes from two different tables, and compares a value of that first table to the second table's value. If the two values are the same then the program outputs the tuples of each table. An example inner join may look like one of two queries. The first can be seen below.

SELECT *
FROM <table1> <table1_var>, <table2> <table2_var>
WHERE <table1_var>.<attribute1> = <table2_var>.<attribute2>;

Another example of an inner join may look like the following.

SELECT *
FROM <table1> <table1_var> INNER JOIN <table2> <table2_var>
ON <table1_var>.<attribute1> = <table2_var>.<attribute2>;

The pseudocode for this can be seen below.

```
for each tuple i in table1:
    joinFound = false
    vector< int > table2JoinsIndexes
    for each tuple j in table2:
        if element of tuple in table1 == element of tuple in table2
            joinFound = true
            Push j onto vector of tuple indexes
    If join found
        for size of table2JoinsIndexes:
```

output table1 tuple
output table2 tuple that matches

The purpose of the vector table2JoinsIndexes is that it keeps track of all the indexes in Table2's tuples that match with the element of table1 tuples.

Outer Join

The outer join feature takes a query of attributes from two different tables, and compares a value of that first table to the second table's value. If the two values are the same then the program outputs the tuples of each table. An example inner join may look like the following.

SELECT *
FROM <table1> <table1_var> LEFT INNER JOIN <table2> <table2_var>
ON <table1_var>.<attribute1> = <table2_var>.<attribute2>;

The pseudocode for this can be seen below.

```
for each tuple i in table1:
    joinFound = false
    vector< int > table2JoinsIndexes
    for each tuple j in table2:
        if element of tuple in table1 == element of tuple in table2
            joinFound = true
            Push j onto vector of tuple indexes
    If join found
        for size of table2JoinsIndexes:
            output table1 tuple
            output table2 tuple that matches
    Else
        Output table 1 tuple
        For each attribute in table2:
            Output empty tuples
```

The purpose of the vector table2JoinsIndexes is that it keeps track of all the indexes in Table2's tuples that match with the element of table1 tuples.

How to Run and Compile

The user must first navigate to where the program is stored on the computer through terminal. The SQLite test file should also be located in the same directory as the makefile and program files. A makefile is provided to compile this program so all the user has to do is type:

```
make
./main < (test file name)
```

The program should now run and execute based on the commands stored in the file that is being fed in.

Special Circumstances

To ensure that the program works as expected, the following circumstances must be met. Each SQLite instruction should end with a semi-colon, except the .EXIT command. The SQLite program must contain a .EXIT to tell the program to stop running. Otherwise, the program will infinite loop until terminated manually. The spacing also matters. Although the program accounts for most spacing differences from the provided SQLite file, the SQLite file tested should still follow the spacing convention displayed in the provided SQLite test file.

PA1 Documentation

Class Organization and Use

The program stores all the file and directory names into a vector of databases, where each database contains a vector of tables. For example, if database db_1 contains tables tbl_1 and tbl_2, and database db_2 contains table tbl_1, the physical representation would look like the drawing in Fig. 1. The vector of databases was only designed to keep track of which files and directories exist. No further information besides the directory and filenames is store in these abstract data types.

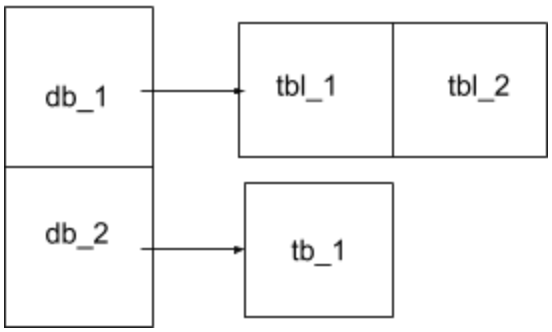

Fig. 1: A physical representation of the layout of the database and table information storage. Each database is an element of a vector, and each table is an element of a vector. (Note: The arrows are not pointers. They simply indicate vectors within vectors.)

Since at the end of every program, the data in these abstract data types are removed from memory, there is not way to ensure that when ".EXIT" is called the database information is saved saved for future use. On the startup of every program run through, the disk is checked to see if any database or table information exists within it. If it does, that data is stored into these vectors and can allow for further modification and manipulation of the existing databases and tables (directories and files).

Database Organization

Create

The Database Create functionality is responsible for creating a new database directory on the disk and for push_back of the database onto the database system vector. For example, to create a database, the user will type the following into the terminal:

CREATE DATABASE <database name>;

Before creating the new database, the program first must make sure that there are no existing databases with the same exact name. For example, if the following input was provided by a user, it would result in an error:

CREATE DATABASE db_1;
CREATE DATABASE db_1;

The terminal layout will result in an error because db_1 already exists after the first input. If there is an existing database with the same name, then an error indication will occur. Otherwise a new directory will be created with the appropriate name and a new database will be pushed on the database system vector.

Drop

The Database Drop functionality is responsible for removing an existing database directory on the disk and for removing the database from the database system vector. For example, to remove a database, the user will type the following into the terminal:

DROP DATABASE <database name>;

Before removing the database, the program first must make sure that there is an existing database with the same exact name. For example, if the following input was provided by a user, it would result in an error:

```
CREATE DATABASE db_1;
DROP DATABASE db_1;
DROP DATABASE db_1;
```

The terminal layout will result in an error after the third input because db_1 does not exist after the second input. If there is not an existing database with the same name, then an error indication will occur. Otherwise the appropriate database will be deleted and removed from the database system vector.

Use

The Database Use functionality is responsible for ensuring that all following table manipulation is from that given database until another database use is specified. This functionality ensures that the database provided already exists and sets the current database variable as the provided input. For the user to specify which database he/she would like to use, the following command is appropriate:

```
USE <database name>;
```

Table Organization

Create

Table Create function creates a file that contains the attribute(s) name and attribute type(s) that is read in from the command line. For example, the table create function is called once a user types the following within the terminal program:

```
CREATE TABLE <Table Name> (<attribute name 1> <attribute type 1>, <attribute name 2> <attribute type 2>);
```

The program check if the table already exists within the database, if so the program will give an error; otherwise the program will create the table and parse in and store the attributes in the disk. If a user specifies a table with the same name for the attributes more than once, then the table will be deleted and an error will be given. For example, the following input will cause the program to give an error because a1 attribute has been declared more than once, regardless of the type:

```
CREATE TABLE tbl_1 (a1 int, a1 float);
```

As a result, the table will fail to be created, and will result in an error display in the command line. Likewise, if the user does not input the attribute without parentheses while creating the table, then an error will be displayed. However, if the user creates the table with the command below:

CREATE TABLE tbl_2();

an empty table without any columns will be created.

Drop

Table Drop function will delete an existing table that belongs to the database in use. The program first checks if the given table exists in the current database. If so, then it will go ahead and delete the table from the disk by deleting the file that stores that table's contents in the database directory. This is done through the command displayed before:

DROP TABLE <table name>;

However, if the user tries to delete a table that does not exist in the directory, the program will notify the user through an error.

PA2 Documentation

Insert

The insert feature inserts a new record into the table and stores that record into the corresponding table file using the output stream. For example, to insert values n into a table, the instruction will look like the following:

INSERT INTO <table name> values(value0, value1, ..., valuen);

The program parses the instruction and checks that the table exists. It then sends the records to the tableInsert() function in Table.h, where the records are inserted into the table file. Before writing to the file, the attributes are extracted from the file, and rewritten when outputting the raw data for each attribute.

Delete

The delete feature deletes the tuples that are associated with the corresponding "where" condition. For example, to remove tuples from a table, the instruction will look like the following:

DELETE FROM <table name> WHERE <attribute name> <operator> <value>;

Given the functionality of the program, the operator in the delete instruction can be any of the following: =, !=, <, <=, >, or >=. The program will first check to see that the table name provided exists. If it does, then it will parse the where condition. It will store the attribute name, operator, and value to be used to compare against the table records. The attribute name will be compared with the table attributes, and will return the index associated with that given attribute. This will be used to compare when looping through the record data. It will check for each attribute name condition, if the condition is not met then that record will be pushed onto a vector. Otherwise, if the condition is met, the record will not be pushed onto the vector. At the end of the tableDelete() function, the table attributes will be outputted to the file as well as the new vector containing the records that did not meet the where condition.

Modify

The modify feature takes an existing tuple, and modifies the tuple based on the user information. For example, to update tuples in a table, the instruction will look like the following:

UPDATE <table name> SET <attribute name> = <value> WHERE <attribute name> <operator> <value>;

Given the functionality of the program, the operator in the update instruction can be any of the following: =, !=, <, <=, >, or >=. The program parses the word “update” to understand that a modification is to occur. The program then checks to make sure that the table name exists in the Database. If it does then it parses the WHERE and SET conditions appropriately. The WHERE and SET conditions are separated into attribute names, operators, and values to be set or compared. The program reads in the attributes from the file, and finds the index of the WHERE and SET condition’s attributes names in the attribute vector. The program then loops through the records to check that if the WHERE condition is met, then it sets the appropriate SET condition attribute value. This data is then outputted back into the table file.

Query

The query feature allows the user to select the columns from the table that should be displayed based on a given condition. To read in all the columns, an example query may look like:

SELECT * FROM <table name> <optional where condition>;

The additional functionality provided by the team allows so that the user can query all column attributes given a condition.

The program is also able to query subsets. An example subset query may look like the following:

SELECT <attribute name>,<attribute name>...<attribute name> FROM <table name> WHERE <attribute name> <operator> <value>;

Given the functionality of the program, the operator in the update instruction can be any of the following: =, !=, <, <=, >, or >=. The program will read in the table data file, search for all the attributes that match the columns selected, and then go through those attributes that match the where condition that is specified by the user. The resulting tuples are then outputted to the terminal.

Alter

Although the alter functionality is not part of the test file functionality, the team thought it would be appropriate to change its functionality in the case that records exist in the table. If altering the table after records have been inserted, the records will hold a 'null' value for the additional attributes. This functionality was not tested with the SQLite test file, but will work in the case that the alter function is used after inserting records to a table.