Project 1

COP 4710, Summer 2021

Due June 4, 2021

1 Overview

In this project, you will develop code that can store arbitrary data in memory and execute queries on that data. The code to perform queries is provided; however, your code will need to provide the data to evaluate the queries.

Specifically, you will need to implement two classes, Table and Row, representing a table in the database and a row in that table, respectively. Your classes will need to be able to perform 3 primary tasks: constructing the table from a given data file, printing the data in a table, and creating a new table in response to a user's query.

2 Driver file

You have been provided with a simple driver file that parses commands from the user and calls relevant Table and Query functions. The first operation it supports is `READ`. The `READ` operation should be followed by a file name (ending in ".csv") giving the name of the CSV file to read in (see Section 3.1).

```
READ data.csv
```

The driver will call the function Table::readTableFromCSV with the name of the file, and this function should read the given CSV file and create and store a new Table object containing that data. The name of the newly created table should be the name of the CSV file without the ".csv" extension.

The second operation supported by the driver is `LIST`, which will print the names of all of the tables that have been read using the `READ` command. The syntax for this command is:

```
LIST
```

The third command is `PRINT`, which will print the all of the data in a given table. The format for this command is:

```
PRINT data
```

where `data` is the name of the table to print (in this case, the table read from data.csv).

The next and most important operation supported by the driver is `SELECT`, which will access the table with a given name, perform a query on the data it contains, and print the result to the console. The format for the query is:

```
SELECT attribs FROM table WHERE condition
```

The attributes may be specified as a comma-separated list of attribute names or as `*`, indicating all attributes should be returned. A comma-separated list of attributes could include attributes out of order or even have the same attribute multiple times, and the resulting table should have exactly these attributes in the order given. The `FROM` clause just gives the name of the table, while the condition specifies which rows in the table to return.

The condition uses syntax similar to C/C++. It recognizes operations `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&`, and `||`. Attribute names and positive integers may appear in this expression with no special syntax. String constants should appear in quotes (`"`), while Boolean constants should be represented as `TRUE` or `FALSE`.

For example, the query below would return the category name and year for all of the Oscars won by "La La Land" from the oscars table. (La La Land won 5 Oscars, all in 2016, for Cinematography, Directing, Music (Original Score), Music (Original Song), and Production Design.)

```
SELECT category, year FROM oscars WHERE entity == "La La Land" && winner
```

Note that the parser for these queries is relatively weak. It should be able to handle extraneous whitespace and handle parentheses appropriately, but it can't parse negative integers, and everything is case sensitive. The key words (`SELECT`, `FROM`, `WHERE`, `TRUE`, and `FALSE`) must be capitalized, and every query must have `SELECT`, `FROM`, and `WHERE`. (If you want to select every row from a table, you may use the condition `WHERE TRUE`.)

The parser should be able to handle malformed queries without crashing the program, but the code has not been extensively tested with bad input.

The last operation recognized by the driver is `QUIT`, which ends the program. The driver can also be exited with `Ctrl-C`.

3 The READ command

The READ command invokes the static method Table::readTableFromCSV, with a string argument containing the file that the user indicated. This function should open the given file and construct a new Table representing the data stored in that file. (The format of the CSV file is described in the following section.) The Table object should have the same name as the file that was opened, with the .csv extension removed.

The Table class contains a static data member `unordered_map<string, Table*> tables`. The purpose of this data member is to associate the name of each Table that's read in with a pointer to the corresponding Table object. So, once you've created the Table object and read the CSV file, readTableFromCSV should add an entry to Table::tables with a key equal to the table name and a value that points to the Table. In order to initialize Table::tables, you will need to include the line

```
unordered_map<string, Table*> Table::tables;
```

in your source file (table.cpp). (This line is also located in a comment at the bottom of the provided header file.)

You have a lot of freedom in defining the Table objects. Some basic member functions have been provided for you, but you'll need to decide what the constructor looks like, as well as what data members to use in order to store the rows, attributes, and data for the table. In addition to the Table class, you will also need to implement a Row class representing a single row in the table. Rows are described further in Section 6.

3.1 Input CSV file

The data file will be a comma-separated value (CSV) file, with a file extension ".csv". The first two lines of the CSV file are the table header. The first line lists the names of all of the attributes in the table (separated by commas), while the second line lists the attribute types. Valid attribute types for this project are `BOOL`, `INT`, and `STRING`, representing Boolean, integer, and string data.

Every other line in the file will contain a single row of data, with a value of the appropriate type for every attribute. All lines in the file will have the same number of columns.

4 The LIST and QUIT commands

The LIST command will call the static Table::printTableNames() function. You have been provided with an implementation for this function that prints all of the keys in Table::tables, so you don't need to code anything for this command. You don't need to implement any code for the QUIT command, either.

5 The PRINT command

The PRINT command will call the static Table::getTableByName function, which has already been implemented for you, and then print out the associated Table using the insertion operator overload (`ostream& operator<<(ostream&, const Table&)`). You will need to implement this function so that it prints out the data in the table in the same format as the CSV file. That is, you should print out all of the attribute names on the first row, separated by commas, then print out their types (domains) on the second row, then print out all of the data in the table. After printing the entire table, you should print a line of the form "`#` rows", where `#` represents the number of rows in the Table (don't print the quotation marks).

Note that the order of the attributes in the original table should be preserved; that is, the PRINT command should print the attributes of the table in exactly the same order as the original CSV file.

6 The SELECT command

In addition to reading and writing binary files, your code must interface with the provided Query class to answer queries about the data contained in a Table. Your `Table::runQuery(Query& q)` function must initialize and return a new Table with the query result. You may wish to implement a copy constructor for your Table class for use when implementing the function, though the constructors you implement are entirely up to you.

Important: `Table::runQuery` will construct a new Table object, but this Table object should *not* be added to Table::tables, *nor* should it have the same name as the Table it originated from.

(If the name of the returned Table is the same as the original Table, the provided destructor will remove the original Table from Table::tables after the query has printed.)

The attributes of the result Table should be exactly the attributes returned by the function `const vector<string>& Query::getAttributesToReturn() const`. The vector returned will contain the names of all of the attributes requested by the user, in the order requested, and the result Table should have exactly these attributes, in the order listed. The attribute list may list the attributes in a different order as the original Table and could even include the same attribute more than once. You are encouraged to write code to handle the case where the attributes requested don't match any attributes in the current Table—it's easy to make typos when writing queries—but this behavior will not be tested.

Special case: if `Query::getAttributesToReturn` returns the single string `*`, the result Table should have the same attributes as the original. You do not need to recognize `*` as valid if there is more than one attribute requested; we will not test any queries like `SELECT *, year FROM test WHERE TRUE`.

The result Table should only contain rows from the original table that cause the Query condition to evaluate to true. To evaluate the Query condition on a particular row, call `q.getCondition()->getBoolValue(row)`. This function will call `row->getValue(attributeName)` on all of the attributes that appear in the Query condition and evaluate the result of all operations. The `Table::runQuery` function will need to iterate through all of the Rows in the Table, call `q.getCondition()->getBoolValue` on each one, and adjust the output Table appropriately.

The `Row::getValue` function should take the name of the attribute to retrieve (based on the first line of the CSV file) and it should return a pointer to the value for that attribute in this Row of the Table. If the attribute is type BOOL, this pointer should be a `bool*`; if it is type INT, the pointer should be `int32_t*` (32-byte integer pointer), and if it is type STRING, the pointer should be `char*`. (If you are storing strings in string objects, you can call the `.c_str` member function of string to return the `char*` for the string.) If the argument is not a valid attribute, or if that attribute is NULL for this Row, return a null pointer.

7 List of required functions

The following lists all 5 of the functions you will be required to implement for this project, along with a brief description of what they should do:

1. `Table* Table::readTableFromCSV(const string& file)`: opens the CSV file `file` and constructs a Table object based on the data in this file (see Section 3.1 for more details). It should initialize the name of this Table to be the name of the CSV file without the ".csv" extension, and store a pointer to the newly created table in the map `tables`. Return `nullptr` if the CSV file cannot be opened.

2. `Table* Table::runQuery(Query& q) const`: executes the given Query on this Table and returns a newly-constructed Table object containing the rows and columns of this Table requested by the Query (see Section 6 for more details).

3. `datatype_t Table::getAttributeType(const string& attName)`: returns the type (`BOOL`, `INT`, or `STRING`) of the attribute with the given name. Return `UNKNOWN` if the table does not contain an attribute by that name.

4. `const void* Row::getValue(const string& attName)`: returns a pointer (`bool*`, `int32_t*`, or `char*`) to the value for the given attribute for this row of the table. Return `nullptr` if the table does not contain an attribute by that name.

5. `ostream& operator<<(ostream& out, const Table& t)`: prints the Table to the given output stream, in CSV format (described in Section 3.1). Outputs "`##` rows" on the following line, where `##` represents the number of rows in the Table

The next list includes all of the functions whose implementations have been provided to you. Your submission is required to implement these functions in order to run using the provided driver. You are allowed to modify these function if need be (e.g., deallocating memory in the Table destructor).

1. `const Table* Table::getTableByName(const string& name)`: returns a pointer to the Table with the given name from the
2. `string Table::getName() const`: returns the name of the Table
3. `void Table::printTableNames()`: prints the name of every table stored in the collection of tables (Table::tables)
4. `Table::~Table()`: destroys the Table and removes itself from Table::tables
5. `Row::Row()`: constructs a dummy Row object—no member functions will be called on this object; it doesn't need to contain any data

8 Submission

You should submit a zip archive containing `table.h` and `table.cpp`. The `table.h` and `table.cpp` files should contain the header and source (respectively) for the Table and Row classes.

9 Grading

Program compiles with the provided driver with no errors	**20 points**
Program can read in CSV files and print itself	**20 points**
Print output is fully correct	**10 points**
Program can perform queries	**30 points**
Query output is fully correct	**20 points**

Code that does not compile will receive very little credit. Programs that cannot print a table will not be tested whether they can perform a query.

10 Validating binary files

When writing binary data, you may wish to use utility programs to ensure that the data that was written exactly matches what you expect. The most basic test is to make sure the file size is correct—Windows Explorer or `ls -l` can tell you the size of a file. To validate that the contents of the file are correct, you'll want to use a hex editor or utility like `od` to read the contents of the file.

`od` is a Linux utility included in most distributions (as well as Mac and Cygwin) for reading binary files. It takes several command line arguments, in addition to the name of the file to read, to control its output. Several of these command line arguments are described below, or you can check `man od` for the full manual.

- `-j ##`: start reading the binary file from offset `##`
- `-N ##`: stop reading the file after `##` bytes
- `-t XX`: specifies how to interpret and print the binary data `od` reads (codes below)
  - `d4`: reads 4-byte integers
  - `d1, d2, d8`: reads 1-, 2-, or 8-byte integers
  - `u1, u2, u4, u8`: reads unsigned integers
  - `x1, x2, x4, x8`: prints binary data in hexadecimal
  - `c`: reads characters
  - `f4, f8`: read float or double data

Example:

`od -t d4 -j 64 -N 8 test.dat`: reads 2 four-byte integers starting at offset 64 in the file test.dat