EECS 281: Project 3 - "SillyQL"

Due Tuesday, November 15 at 11:59pm

Overview

A relational database is the most common means of data storage and retrieval in the modern age. A relational database model stores data into one or more tables, where each table has columns and rows. The columns represent a value that can be attributed to an entry (such as `color`, `name`, or `ID`) and the rows represent individual entries or records (such as `Paoletti`, `my_cat`, or `BBB_1695`). You may find it helpful to think about *rows as objects and columns as descriptors for those objects*. For instance, the table pictured below has each row corresponding to a car (a data type), and the columns group a car's vendors, model, etc. such that this information can be easily retrieved. Rows in relational databases are also called records or tuples, but in this project we will use the terminology "row" for consistency. **Rows in Project 3 are not guaranteed to be unique.**

Cars_Marketplace			
Vendor	**Model**	**Miles**	**Price**
Chevrolet	Corvette	67226	25965.00
Chevrolet	Malibu	97874	7273.00
Chevrolet	Malibu	15600	27441.00
Chevrolet	Corvette	27982	50209.99
Tesla	S	8437	63901.00
Chevrolet	Malibu	52477	46700.95
Tesla	X	43801	73429.00

However, a database can do much more than simply store information. You must be able to retrieve specific information in an efficient manner. For relational databases, the structured query language (SQL) is a defined method of retrieving information programmatically. It looks like real English, where a valid command for the above table could be:

```
1  SELECT Model from Cars marketplace where Price < 30000
```

which would return a list of models:

```
1  [Corvette, Corvette, Malibu, Malibu, Malibu]
```

Note that the version of SQL used in this project is simplified and modified somewhat from a typical query language.

For this project, you will write a program to emulate a basic relational database with an interface based on a subset of a standard query language. Your executable will be called `silly`. You will gain experience writing code that makes use of multiple interacting data structures. The design portion of this project is significant; spend time thinking about what data structures you will use and how they will interact.

Learning Goals

- Selecting appropriate data structures for a given problem. Now that you know how to use various abstract data types, it's important that you are able to evaluate which will be the optimal for a set of tasks.
- Evaluating the runtime and storage tradeoffs for storing and accessing data contained in multiple data structures.
- Designing a range of algorithms to handle different situations.
- Evaluating different representations of the same data.

Command Line Arguments

`silly` should accept the following (both optional) command line arguments:

- `-h, --help`

This causes the program to print a helpful message about how to use the program and then immediately `exit(0)`.

- `-q, --quiet`

This causes the program to run in quiet mode. In quiet mode, your program will run very similarly to normal, except that it will print only numerical summaries of the rows affected by the command, not any of the actual data. Quiet mode exists so that we may stress test your program without overloading the autograder with too much output. This flag only affects the `JOIN` and `PRINT` commands, and specific instructions for quiet mode output is given for these commands. Otherwise, if there is no mention of quiet mode with respect to a given piece of output, you may assume it is always printed. You can implement this feature last; with a well-built project, adding this functionality should be very simple.

You may find it simpler to check for these flags directly rather than use getopt.

Database Shell

Your program will create a shell that allows users to interact with a single database that is created in memory and destroyed when the program is quit by the user. A shell is a program that presents a prompt ("% ") to allow a user to enter commands and also view displayed output while interacting with the system. Since the commands to the shell come from standard input

(`cin`) and display to standard output (`cout`), the program will be useable both directly from the keyboard locally and via redirected input (`<`) and output (`>`) locally and on the autograder.

Each command is specified by a keyword followed by a *required* space character ( ) and then the command-specific arguments where applicable. Like other database languages, some commands use additional required words that help make them more readable. These words can be both guaranteed to appear and completely ignored. For example, the command to delete rows from a table uses the keyword `DELETE` and adds `FROM` , for improved readability/pronuciation. So to delete the rows from `table1` where the entries in column `name` equal `John` , the correct command would be:

```
DELETE FROM table1 WHERE name = John
```

Spec Example contains an example program illustrating the use of all the commands.

Database Commands

These commands affect the structure of the database and the running of the command shell. They include the creation and removal of tables, but do not deal with any actual data (items of type `string` , `double` , `int` or `bool`).

CREATE

Add a new table to the database:

CREATE Syntax

```
CREATE <tablename> <N> <coltype1> <coltype2> ... <coltypeN> <colname1> <colname2> .
```

Creates a new table with `N` columns (where `N` > 0). Each column contains data of type `<coltype>` and is accessed with the name `<colname>` . Table names and column names are guaranteed to be space-free. No two columns in the same table will have the same name (you do not need to check). Valid data types for `coltype` are `{double, int, bool, string}` . This table is initially empty.

CREATE Output

Print the following on a single line followed by a newline:

```
New table <tablename> with column(s) <colname1> <colname2> ... <colnameN> created
```

CREATE: Possible errors

1. A table named `<tablename>` already exists in the database

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
1  CREATE 281class 3 string string bool emotion person Y/N
```

The output should be:

```
1  New table 281class with column(s) emotion person Y/N created
```

QUIT

Exit the program and delete all data in the database.

QUIT Syntax

```
1  QUIT
```

Cleans up all internal data (i.e. no memory leaks) and exits the program. Note that the program must exit with a `return 0` from `main()`.

QUIT Output

Print a goodbye message, followed by a newline.

```
1  Thanks for being silly!
```

QUIT: Possible errors

None, except for lacking a `QUIT` command. Every interactive session or redirected input file should end with a `QUIT` command.

(comment)

Used to add comments to command files, produces no actions or output.

Syntax

```
1  # Any text on a line that begins with # is ignored
```

Discard any lines beginning with #. This command does not produce any output nor can it generate any errors.

#: Possible errors

None.

REMOVE

Remove existing table from the database, deleting all data in the table and its definition.

REMOVE Syntax

```
REMOVE <tablename>
```

Removes the table specified by `<tablename>` and all associated data from the database, including any created index.

REMOVE Output

Print a confirmation of table deletion, followed by a newline, as follows:

```
Table <tablename> deleted
```

REMOVE: Possible errors

1. `<tablename>` is not the name of a table in the database

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
REMOVE pets
REMOVE 281class
```

The output should be:

```
Table pets deleted
Table 281class deleted
```

Table Commands

These commands work on one or more tables and are responsible for interacting with actual data, including insertion, deletion, printing, and indexing.

INSERT

Insert new rows at the end (append) of the specified table.

INSERT Syntax

```
INSERT INTO <tablename> <N> ROWS
<value11> <value12> ... <value1M>
<value21> <value22> ... <value2M>
...
<valueN1> <valueN2> ... <valueNM>
```

Inserts `<N>` new rows into the table specified by `tablename`. The first line of the command specifies the `tablename` and the number of rows, `N > 0`, to be inserted. The next `N` lines specify the values to be inserted into the table. Each line contains `M` values, where `M` is guaranteed to be equal to the number of columns in the table. The first value, `<value11>`,

should be inserted into the first column of the table in the first inserted row, the second value, `<value12>`, into the second column of the table in the first inserted row, and so on. Additionally, the types of the values are guaranteed to be the same as the types of the columns they are inserted into. For example, if the second column of the table contains integers, `<value12>` is guaranteed to be an `int`. Further, `string` items are guaranteed to be a single string of whitespace delimited characters (i.e. `foo bar` is invalid, but `foo_bar` is acceptable).

INSERT Output

Print the message shown below, followed by a newline, where `<N>` is the number of rows inserted, `<startN>` is the index of the first row added in the table, and `<endN>` is the index of the last row added to the table, 0 based. So, if there were `K` rows in the table prior to insertion, `<startN> = K`, and `<endN> = K + N - 1`.

```
Added <N> rows to <tablename> from position <startN> to <endN>
```

INSERT: Possible errors

1. `<tablename>` is not the name of a table in the database

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
INSERT INTO 281class 8 ROWS
happy Darden true
stressed students false
busy office_hours true
stressed students true
stressed Paoletti true
happy Darden true
happy Sith true
victorious Sith true
```

The output should be:

```
Added 8 rows to 281class from position 0 to 7
```

PRINT

Print values from selected columns, in the given order, from specified rows.

PRINT Syntax

```
PRINT FROM <tablename> <N> <print_colname1> <print_colname2> ... <print_colnameN> {
```

Directs the program to print the columns specified by `<print_colname1>`, `<print_colname2>`, ... `<print_colnameN>` from some/all rows in `<tablename>`. If there is no condition, the command is

of the form `PRINT ... ALL`, and the matching columns from all rows of the table are printed strictly in insertion order. If there is a condition, the command is of the form `PRINT ... WHERE <colname> <OP> <value>`, and only rows whose values in the selected `<colname>` pass the condition are printed. The rules for the conditional portion are the same as for the `DELETE` command. The number and order of the column names given in the command **do not** have to match the number or order of columns in the specified table.

The table must be searched as follows to ensure compatibility with the autograder:

1. If no index exists or there is a `hash` index on the **conditional** column, the results should be printed in order of insertion into the table.
2. If a `bst` index exists on the **conditional** column, the results should be printed in the order in the BST (least item to greatest item for `std::map<>` constructed with the default `std::less<>` operator), with ties broken by order of insertion into the table.

PRINT Output

Print the requested data followed by a summary. Printing the data is accomplished by printing a single line with the names of the selected columns, followed by a single line from each specified row, where the values of each of the selected columns are separated by a single space. Every line should be followed by a newline.

```
<print_colname1> <print_colname2> ... <print_colnameN>
<value1row1> <value2row1> ... <valueNrow1>
...
<value1rowM> <value2rowM> ... <valueNrowM>
```

To print the summary, on a single line, print the number of rows `<M>` printed, and the `<tablename>` from which the rows were printed, followed by a newline.

```
Printed <M> matching rows from <tablename>
```

In quiet mode, do not print the data, print **only the summary.**

```
Printed <M> matching rows from <tablename>
```

PRINT: Possible errors

1. `<tablename>` is not the name of a table in the database
2. `<colname>` is not the name of a column in the table specified by `<tablename>`
3. One (or more) of the `<print_colname>` s are not the name of a column in the table specified by `<tablename>` (only print the name of the first such column encountered)

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
PRINT FROM 281class 2 person emotion WHERE Y/N = true
```

The output should be:

```
person emotion
office_hours busy
students stressed
Paoletti stressed
Sith happy
Sith victorious
Printed 5 matching rows from 281class
```

Or in quiet mode:

```
Printed 5 matching rows from 281class
```

DELETE

Delete selected rows from the specified table.

DELETE Syntax

```
DELETE FROM <tablename> WHERE <colname> <OP> <value>
```

Deletes all rows from the table specified by `<tablename>` where the value of the entry in `<colname>` satisfies the operation `<OP>` with the given value `<value>`. You can assume that `<value>` will always be of the same type as `<colname>`. For example, to delete all rows from `table1` where the entries in column `name` equal `John`, the command would be:

```
DELETE FROM table1 WHERE name = John
```

Or, to delete all rows from `tableSmall` where the entries in column `size` are greater than 15, the command would be:

```
DELETE FROM tableSmall WHERE size > 15
```

For simplicity, `<OP>` is strictly limited to the set `{<, >, =}`.

DELETE Output

Print a summary of the number of rows deleted from the table as shown below, followed by a newline:

```
Deleted <N> rows from <tablename>
```

DELETE: Possible errors

1. `<tablename>` is not the name of a table in the database
2. `<colname>` is not the name of a column in the table specified by `<tablename>`

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
1  DELETE FROM 281class WHERE person = Darden
```

The output should be:

```
1  Deleted 2 rows from 281class
```

The search is case sensitive, which makes it easier to code. If the selection clause was `WHERE person = darden`, no rows would be deleted.

JOIN

Join two tables where values in selected columns match, and print results.

Syntax:

```
1  JOIN <tablename1> AND <tablename2> WHERE <colname1> = <colname2> AND PRINT <N> <pri
```

Directs the program to print the data in `<N>` columns, selected by `<print_colname1>`, `<print_colname2>`, ... `<print_colnameN>`. The `<print_colname>`s will be the names of columns in either the first table `<tablename1>` or the second table `<tablename2>`, as chosen by the `<1/2>` argument directly following each `<print_colname>`.

The `JOIN` command is unique in that it accesses data from multiple tables. The rules for the conditional portion are the same as for the `DELETE` command except that for **the `JOIN` command, `<OP>` will be strictly limited to `{=}` for simplicity.** The number and order of the column names given in the command **do not** have to match the number or order of columns in the specified tables.

The `JOIN` must be accomplished as follows in order to insure compatibility with the autograder:

1. Iterate through the first table `<tablename1>` from beginning to end.
2. For each row's respective `<colname1>` value in `<tablename1>`, find matching `<colname2>` values in `<tablename2>`, if any exist.
3. For each match found, print the column values in the matching rows in the order specified by the `JOIN` command.
4. The matching rows from the second table must be selected in the order of insertion into that table.
5. If no rows in the second table match a row in the first table, that row is ignored from the join.

JOIN Output

Print the requested data followed by a summary. Printing the data is accomplished by printing a single line with the names of the selected columns, followed by a single line from each joined row, where the values of each of the selected columns are separated by a single space. Every line should be followed by a newline.

```
<print_colname1> <print_colname2> ... <print_colnameN>
<value1rowA> <value2rowA> ... <valueNrowA>
...
<value1rowM> <value2rowM> ... <valueNrowM>
```

To print the summary, on a single line, print the number of rows `<N>` printed, and both `<tablename1>` and `<tablename2>` which were joined, followed by a newline.

```
Printed <N> rows from joining <tablename1> to <tablename2>
```

In quiet mode, do not print the data, print **only the summary.**

```
Printed <N> rows from joining <tablename1> to <tablename2>
```

JOIN: Possible errors

1. `<tablenameX>` is not the name of a table in the database
2. One (or more) of the `<colname>` s or `<print_colname>` s are not the name of a column in the table specified by `<tablenameX>` (only print the name of the first such column encountered)

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
CREATE pets 3 string bool bool Name likes_cats? likes_dogs?
INSERT INTO pets 2 ROWS
Sith true true
Paoletti true false
JOIN pets AND 281class WHERE Name = person AND PRINT 3 Name 1 emotion 2 likes_dogs?
```

The join specific output should be (Note: the `CREATE` and `INSERT` commands will generate their own output, but for simplicity, they are not included in this example):

```
Name emotion likes_dogs?
Sith happy true
Sith victorious true
Paoletti stressed false
Printed 3 rows from joining pets to 281class
```

Or in quiet mode:

```
Printed 3 rows from joining pets to 281class
```

Note that the `JOIN` is case sensitive and creates a table only for the purposes of printing and does not retain that table beyond the command nor add it to the database.

GENERATE

Create a search index on the selected column in the specified table.

GENERATE Syntax

```
GENERATE FOR <tablename> <indextype> INDEX ON <colname>
```

Directs the program to create an index of the type `<indextype>` on the column `<colname>` in the table `<tablename>`, where `<indextype>` is strictly limited to the set `{hash, bst}`, denoting a hash table index and a binary search tree index respectively. Given the `<indextype> hash` on column `<colname>`, the program should create a hash table that allows a row in the table to be found rapidly given a particular value in the column `<colname>`. Given the `<indextype> bst` on column `<colname>`, the program should create a binary search tree that allows rows in the table to be found rapidly given a particular value in the column `<colname>`. **Only one user-generated Index may exist per table, at any one time.** If a valid index is requested on a table that already has one, discard the old index before building the new one.

> ⚠ If an invalid index request is made, do not discard any existing index.

When `bst` is the specified index type, you should make use of a `std::map<>`; when `hash` is the specified index type, you should utilize a `std::unordered_map<>`. It is acceptable for both types to *exist* at the same time, but only one should be in use or contain data at any given time.

An index is a tool used in databases to speed up future commands. A `hash` index creates a hash table, which associates values in a selected column with a collection of rows in the table for which the index was created. Similarly, a `bst` index creates a binary search tree which associates values in a selected column with a collection of rows. Both are useful for different types of commands. In order to get the correct output in all cases, you must use an index when it is appropriate. **Further, you must remember to update your indices upon edits to the table**. `bst` indices are ordered by `operator<` for the type in the specified column.

GENERATE Output

Print this message, followed by a newline.

```
Created <indextype> index for table <tablename> on column <colname>
```

GENERATE: Possible errors

1. `<tablename>` is not the name of a table in the database
2. `<colname>` is not the name of a column in the table specified by `<tablename>`

See Errors and Error Messages for acceptable output formats.

Given the following as input:

```
GENERATE FOR 281class hash INDEX ON emotion
```

The output should be:

```
Created hash index for table 281class on column emotion
```

In this example, the user should now have created an index that uses a hash table to map from specific emotions of type string to rows in the table `281class`, allowing the program to identify all rows where `emotion = happy` quickly, among other things.

Error Checking

Except for the errors specifically noted above and below, we will not test your error handling. However, we recommend that you implement a robust error handling and reporting mechanism to make your own testing and debugging easier. Normally, you would print error messages to the standard error stream (stderr via `cerr`), rather than `cout`. **For this project, however, you must print the specified error messages to stdout via `cout` so that they may be tested in conjunction with the rest of the project. DO NOT `exit()` when one of these errors occurs, display the error and keep running.**

As stated in the commands described in Database Shell, there are a few specific errors that your code must check for and print the appropriate output. They are as follows:

Errors and Error Messages

Error 1: A table named `<tablename>` already exists in the database

Output:

```
Error during <COMMAND>: Cannot create already existing table <tablename>
```

Error 2: `<tablename>` is not the name of a table in the database

Output:

```
Error during <COMMAND>: <tablename> does not name a table in the database
```

Error 3: `<colname>` is not the name of a column in the table specified by `<tablename>`

Output:

```
Error during <COMMAND>: <colname> does not name a column in <tablename>
```

Error 4: Unrecognized first letter of command (i.e. not one of **C**REATE, **P**RINT, **R**EMOVE, **#**, etc.)

Output:

```
1  Error: unrecognized command
```

For all input errors, you should print the matching response, with the braced variables replaced with the items from the offending command and followed by a newline, clear the rest of the command input, and re-prompt the user ("% ") as usual. **When any of these errors occur, you must clear the rest of the input line to get rid of unread text. But DO NOT TERMINATE THE PROGRAM.** Other than the errors noted above, commands will be well formed.

For the first three errors, `<COMMAND>` should be replaced with the **single word** name of the command that encountered the error. For example, if an `INSERT` command specified a table name that doesn't exist, you should output something like this:

```
1  Error during INSERT: junk does not name a table in the database
```

Using the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file:

  ```
  // Project Identifier: C0F4DFE8B340D81183C208F70F9D2D797908754D
  ```

- The Makefile must also have this identifier (in the first TODO block).
- You have deleted all .o files and your executable(s). Your Makefile should include a rule or rules that cause make clean to accomplish this.
- Your makefile is called Makefile. To confirm that your Makefile is behaving appropriately, check that "make -R -r" builds your code without compiler errors and generates an executable file called silly. (Note that the command line options -R and -r disable automatic build rules, which will not work on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option -O3 (This is the letter "O," not the number "0"). This is extremely important for getting all of the performance points, as -O3 can speed up code by an order of magnitude.
- Your test files have names of the form test-n.txt, and no other project file names begin with "test." Up to 15 test files may be submitted.
- The total size of your program and test files does not exceed 2MB.
- You don't have any unneeded files in your submit directory.
- Your code compiles and runs correctly using version 6.2.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of gcc, the course staff will not support anything other than gcc 6.2.0 running on Linux. To compile with

g++ version 6.2.0 on CAEN you must put the following at the top of your Makefile (or use our provided Makefile):

```
1  PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
2  LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
3  LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Creating a Submission

Turn in the following files:

- All of your `.h` / `.hpp` and `.cpp` files for the project
- Your Makefile
- Your test files

You must prepare a compressed tar archive (`.tar.gz` file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:

```
1  % tar -czvf submit.tar.gz *.cpp *.h *.hpp Makefile test-*.txt
```

This will prepare a suitable file in your working directory. Alternatively, the 281 Makefile has useful targets `fullsubmit` and `partialsubmit` that will do this for you. Use the command `make help` to find out what else it can do!

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu or https://g281-2.eecs.umich.edu. Note that when the autograders are turned on and accepting submissions, there will be an announcement on Piazza. The auto graders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to two times per calendar day with autograder feedback (more in Spring semester). For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade.** If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.

We strongly recommend that you use some form of revision control (ie: SVN, Git, etc) and that you "commit" your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and Canvas regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to note if the autograder shows that one of your own test files exposes a bug in your solution (at the bottom).

Checkpoint Test Cases

This project has three test cases named `cpX-in.txt` that represent a checkpoint (see Starter and Sample Files). The checkpoint involves implementing base functionality for several commands, and you should have it completed within 10 days of receiving the project specification (less in the Spring; about halfway between receiving this document and the final due date).

The checkpoint test cases will only be testing a subset of the functionality of your code. The functionality we will be testing are listed as follows:

- # (comment)
  - Used in all 3 checkpoint test cases
- CREATE
  - Used in all 3 checkpoint test cases
- INSERT
  - Used in checkpoint test cases 2 and 3
- PRINT ... ALL
  - Used in checkpoint test cases 2 and 3
- REMOVE
  - Used in all 3 checkpoint test cases
- QUIT
  - Used in all 3 checkpoint test cases

For example, checkpoint 1 has a comment, a few `CREATE` and `REMOVE` commands, and a `QUIT`. None of the commands produce errors.

Autograder Test Case Legend

All test cases on the autograder test the `QUIT` command, because all valid input files must end with it. Many include comment(s) so that we know what the test case is doing. Most of the test case names on the autograder are fairly self-explanatory, but here's a guide:

- CP*
  - Tests the commands listed in the checkpoint section above
- CREATE*
  - Primarily tests the CREATE command, but also PRINT and REMOVE
- DELETE*
  - Primarily DELETE; also needs CREATE, INSERT; sometimes REMOVE and GENERATE
- GENERATE*
  - Primarily GENERATE, needs CREATE, INSERT, PRINT; sometimes REMOVE, DELETE
- INSERT*
  - Primarily tests INSERT, but also needs CREATE, PRINT, and REMOVE
- JOIN*
  - Primarily JOIN, but also needs CREATE, and INSERT; some test REMOVE and GENERATE
- Long*

    - Tests all commands; always run in quiet mode
- Medium*
    - Tests all commands; always run in quiet mode
- PRINT*
    - Primarily tests PRINT, but also needs CREATE, INSERT, and DELETE
- REMOVE*
    - Primarily tests REMOVE, but also needs CREATE, INSERT, and sometimes GENERATE
- SPEC
    - The specification test case (see Spec Example)
- Short*
    - Tests all commands, "short" only with respect to Medium* and Long*; always runs in quiet mode

Test Files and Autograder Bugs

A major part of this project is to prepare a suite of test files that will help expose defects in your program. Each test file should be a text file containing a series of commands to run. Your test files will be run against several buggy project solutions. If your test file causes the correct program and an incorrect program to produce different output, the test file is said to expose that bug.

Test files should contain valid SillyQL commands, and should be named `test-n.txt`, where $0 < n \leq 15$. Make sure that every test file ends with a `QUIT` command.

Your test files may contain no more than 35 commands in any one file, including `QUIT` as the last command. These commands can insert up to a maximum of 100 rows of data into any table. Tables cannot have more than 10 columns each. You may submit up to 15 test files (though it is possible to get full credit with fewer test files). Note that the tests the autograder runs on your solution are NOT limited to 35 lines in a file; your solution should not impose any size limits (as long as sufficient system memory is available).

Libraries and Restrictions

The use of the C/C++ standard libraries is highly encouraged for this project, especially functions in the `<algorithm>` header and container data structures. The smart pointer facilities, and thread/atomics libraries are prohibited. As always, the use of libraries not included in the C/C++ standard libraries is forbidden.

Grading

- 80 points – Your grade will be derived from correctness and performance (runtime). Details will be determined by the autograder.

- 10 points – Test file coverage (effectiveness at exposing buggy solutions).
- 10 points – No memory leaks. Make sure to run your code under valgrind before each submit. (This is also a good idea because it will let you know if you have undefined behavior, which will cause your code to crash on the autograder.)

When you start submitting test files to the autograder, it will tell you (in the section called "Scoring student test files") how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!

CREDITS

Current version by: Cris Noujaim, Shahab Tajik, Ankit Shah, and David Paoletti

Originally composed by: David Snider and Genevieve Flaspohler; Special thanks to Waleed Khan