EECS 281 Fall 2018
Project 3: SillyQL

->Read Appendix C

-> Complete in helper functions

Due Tuesday, November 20, 2018 at 11:59 pm

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 to the right 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**	**Price**	**Mileage**
Chevrolet	Corvette	17226	25965.0
Chevrolet	Corvette	34229	46429.0
Chevrolet	Corvette	27982	50209.0
Chevrolet	Corvette	51825	72998.0
Chevrolet	Corvette	52845	34364.0
Chevrolet	Malibu	37874	37273.0
Chevrolet	Malibu	15600	71441.0
Chevrolet	Malibu	52447	46700.0
Chevrolet	Malibu	27129	36254.0
Chevrolet	Malibu	28846	77162.0
Chevrolet	Malibu	46165	60590.0

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 “`SELECT Model from Cars marketplace where Price < 30000`”, which would return a list of models [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 plenty of time thinking about what data structures you will use and how they will interact.

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

Table of Contents

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.

Project Specification

1. Program 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.

2. User Commands

On startup, your program should prepare to accept commands from the user. These commands may or may not take the form of redirected input. Your program should print "`% `" as a prompt to the user before reading each command. Commands are specified by a command keyword followed by a *required* space character and then the command-specific arguments where applicable. For example, to delete the rows from `table1` where the entries in column `name` equal "`John`", one would input: `DELETE FROM table1 WHERE name = John`. Appendix A contains an example program illustrating the use of these commands.

Table Manipulation Commands

In the following commands, output examples are given. These examples are cumulative throughout the table manipulation command part of the spec (i.e. the table created in the example output for the CREATE command is the same table that DELETE is performed on during the delete command, and so on, so you can follow the state of the table throughout the spec).

CREATE - add a new table to the database

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

Creates a new table with `<N>` columns. 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. Valid data types for `coltype` are {`double, int, bool, string`}. This table is initially empty.

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

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

Possible errors:

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

Given the following as input:

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

The output should be:

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

INSERT INTO - insert new data into table

Syntax:

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

Inserts `<N>` new rows (where `<N>` is greater than 0) into the table specified by `<tablename>`. The number of values in each line after the first, or M in the example, 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, <value2> 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).

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>
```

Possible errors:

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

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
```

---

3 DELETE FROM - delete specific data from table

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 { <, > , = }.

Output (with example): Print the number of rows deleted from the table as shown below, followed by a newline:

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

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>`

Given the following as input:

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

The output should be:

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

The search is case sensitive (which makes it easier to code): if we had deleted `WHERE person = darden`, no rows would have been removed.

---

4 GENERATE INDEX - create a search index on the specified column

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 an 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**. If an index is requested on a table that already has one, discard the old index before building the new one.

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 (at most) should be in use at any given time (i.e. contain data).

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 specified 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 specified 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.

Output: Print the message shown below, followed by a newline

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

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>`

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
```

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

PRINT - print specified rows

Syntax: `PRINT FROM <tablename> <N> <print_colname1> <print_colname2> ... <print_colnameN> [WHERE <colname> <OP> <value> | ALL ]`

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 (i.e. statement is of the form `PRINT ... ALL`), the matching columns from all rows of the table are printed. If there is a condition (i.e. statement is of the form `PRINT ... WHERE <colname> <OP> <value>`), only rows, whose `<colname>` value pass the condition, are printed. The rules for the conditional portion are the same as for the `DELETE FROM` statement. It is not guaranteed that the columns in the command are listed in the same order as they exist in the table, nor is it guaranteed that all columns will be listed.

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.

Output : Print the names of the specified columns, followed by the values of each of the specified columns in each row, separated by space. Every line should be followed by a newline.

```
<print_colname1> <print_colname2> ... <print_colnameN>

<value1rowA> <value2rowA> ... <valueNrowA>

...

<value1rowM> <value2rowM> ... <valueNrowM>
```

Once all the data has been printed, print the following, followed by a newline, where `<N>` is the number of rows printed:

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

In quiet mode, do not print the <print_colname1>s or any of the values. Print **only** the following:

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

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)

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
```

---

6 JOIN - join two tables and print result

Syntax: `JOIN <tablename1> AND <tablename2> WHERE <colname1> = <colname2> AND PRINT <N> <print_colname1> <1|2> <print_colname2> <1|2> ... <print_colnameN> <1|2>`

Directs the program to print the the data in <N> columns, specified 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 specified by the <1/2> argument directly following each <print_colnameN>.

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 FROM statement except that for the JOIN command, **<OP> will be strictly limited to {=} for simplicity.** It is not guaranteed that the columns are listed in the same order as they exist in the table, nor is it guaranteed that all columns will be listed.

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.

Output : Print the names of the specified columns, followed by the values of each of the specified columns in each row, separated by space. Every line should be followed by a newline.

```
<print_colname1> <print_colname2> ... <print_colnameN>

<value1rowA> <value2rowA> ... <valueNrowA>

...

<value1rowM> <value2rowM> ... <valueNrowM>
```

Once all the data has been printed, print the following, followed by a newline, where N is the number of rows printed

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

In quiet mode, do not print the <print_colname1>s or any of the values. Print **only** the following:

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

Possible errors:

1. <tablename> 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 <tablename> (only print the name of the first such column encountered)

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? 1
```

The join specific output should be (Note: the CREATE and INSERT INTO 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 does not create a new table.

REMOVE - remove existing table from the database

Syntax:

```
REMOVE <tablename>
```

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

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

```
Table <tablename> deleted
```

Possible errors: `<tablename>` is not the name of a table in the database

Given the following as input:

```
REMOVE pets
REMOVE 281class
```

The output should be:

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

QUIT - exit the program

Syntax:

```
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()`.

Output: Print a goodbye message, followed by a newline.

```
Thanks for being silly!
```

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

- comment / no operation (useful for adding comments to command files)

Syntax:

```
# 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.

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 Table Manipulation Commands section, there are a few specific errors that your code must check for and print the appropriate output. They are as follows:

Error 1: A table named `<tablename>` already exists in the database
Output: `Error: Cannot create already existing table <tablename>`

Error 2: `<tablename>` is not the name of a table in the database
Output: `Error: <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: <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: `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 reprompt the user (`"% "`) as usual. For most commands, clearing the rest of the input line should suffice. If there is an error in the insert command, however, the program should clear as many lines as there are rows to be inserted so that the command is fully flushed. **Do not terminate the program**. Other than the errors noted above, commands will be well formed. For simplicity, this also holds true when clearing away an erroneous command.

Testing and Debugging

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 table manipulation 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-table-commands.txt`, where $0 < n \leq 15$. Make sure that every test file ends in a QUIT command.

Your test files may contain no more than 35 lines in any one file. 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).

Submission to 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:

- 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-table-commands.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. Note: In order to compile with g++ version 6.2.0 on CAEN you **must** put the following at the top of your Makefile:

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

Turn in the following files:

- All of your .h 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:

```
% dos2unix *; tar cvzf submit.tar.gz *.cpp *.h Makefile test*.txt
```

to prepare a suitable file in your working directory.

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 three times per calendar day with autograder feedback. 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 CTools 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).

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.)

There are 23 bugs to find with your test files; you will start earning points with the 11th bug found, and will receive 1 point per bug through 20 bugs for full points.

Checkpoint

This project has a few test cases named CP* that represent a checkpoint. The checkpoint involves implementing base functionality for several commands, and you should have it completed within 10 days of receiving the project specification (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 checkpoints
- CREATE Used in all 3 checkpoints
- INSERT INTO Used in checkpoints 2 and 3
- PRINT FROM ... ALL Used in checkpoints 2 and 3
- REMOVE Used in all 3 checkpoints
- QUIT Used in all 3 checkpoints

For example, checkpoint 1 has a comment, a few create and remove commands, and quits. None of the commands produce errors.

Test Cases

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:

Appendix: The test case from Appendix A (see below)
CP*: Checkpoint cases (see above)
CREATE*: Primarily tests the CREATE command, but also PRINT and REMOVE
INSERT*: Primarily tests INSERT, but also needs CREATE, PRINT, and REMOVE
DELETE*: Primarily DELETE; also needs CREATE, INSERT; sometimes REMOVE and GENERATE
GENERATE*: Primarily GENERATE, needs CREATE, INSERT, PRINT; sometimes REMOVE, DELETE
JOIN*: Primarily JOIN, but also needs CREATE, and INSERT; some test REMOVE and GENERATE
PRINT*: Primarily tests PRINT, but also needs CREATE, INSERT, and DELETE
REMOVE*: Primarily tests REMOVE, but also needs CREATE, INSERT, and sometimes GENERATE

Short*: Tests all commands, "short" only with respect to Medium* and Long*; always run in quiet mode
Medium*: Tests all commands; always run in quiet mode
Long*: Tests all commands; always run in quiet mode

CP*: Tests the commands listed in the checkpoint section above.

Appendix A: Example from the Spec

Given the following as input:

```
#Awesome Spec Example!
CREATE 281class 3 string string bool emotion person Y/N
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
DELETE FROM 281class WHERE person = Darden
GENERATE FOR 281class hash INDEX ON emotion
PRINT FROM 281class 2 person emotion WHERE Y/N = true
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? 1
REMOVE pets
REMOVE 281class
QUIT
```

The output should be: (prompt characters not included)

```
New table 281class with column(s) emotion person Y/N created
Added 8 rows to 281class from position 0 to 7
Deleted 2 rows from 281class
Created hash index for table 281class on column emotion
person emotion
office_hours busy
students stressed
Paoletti stressed
Sith happy
Sith victorious
Printed 5 matching rows from 281class
New table pets with column(s) Name likes_cats? likes_dogs? created
Added 2 rows to pets from position 0 to 1
Name emotion likes_dogs?
Sith happy true
Sith victorious true
Paoletti stressed false
Printed 3 rows from joining pets to 281class
Table pets deleted
Table 281class deleted
Thanks for being silly!
```

Appendix B: Variant Types and You

One of the most difficult parts of implementing an efficient database is handling the problem of heterogenous rows, or the fact that rows in the tables contain multiple types. If you wanted to store your rows as `vector<T>`, what would T be? int? string? bool? The answer, in reality, is that you must be able to store any of the valid types in that row. Unfortunately the current C++ standard does not give us a good way of implementing these types of heterogeneous containers when the types of each column aren't known at compile time (check out `std::tuple<>` for when you do!).

To remedy this, the course staff has created a special class for you to store in your tables, aptly named `TableEntry`. This `TableEntry` is an implementation of what is known as a variant type, as it can be a variety of different types. While we could modify the `TableEntry` class to contain arbitrary types, for simplicity we have limited the types that can be represented to the set of those that are valid to appear in this project. Most of how to use the `TableEntry` should be intuitive from the comments in the header file. In addition to that, see the below example for further instructions.

The one major downside to our implementation of `TableEntry` is that it doesn't play nicely when you try to compare two `TableEntry` that contain different types, or try to compare a `TableEntry` with a type other than the one it represents internally. To combat that we've placed assertions in key locations such that if you do not compile with `-DNDEBUG` (as make debug with the provided Makefile does not), these assertions will fire rather than let you have undefined behavior. If you're getting weird results with this type, try running under debug conditions and see if an assertion fires. We attached a comment to the assertions so that you can see *what* the issue is. In order to see *where* the issue stems from, you must use a debugger and read the stack trace to find the line of your code that calls offending method. We suggest against delving into the implementation of this class, as it's pretty hairy.

An example:

```
#include "TableEntry.h"
#include <unordered_map>
#include <map>
#include <vector>
#include <iostream>
using namespace std;
int main() {
    unordered_map<TableEntry, int> u_m; // They can be used in unordered_maps!
    map<TableEntry, int> m;             // They can be used in maps!
    vector<TableEntry> v;               // They can be used in other containers!
    TableEntry tt1{7}, tt2{8};
    int i = 3;
    if(tt1 < tt2 || tt2 > i) // They can be compared w/ each other and their "type"!
        cout << tt1 << endl; // They can be printed!

   return 0;
} // main()
```

Appendix C: Project Tips, Tricks, and Things to Avoid

Starting the project

Like the others, this project can be written in stages. Start with the QUIT command, it's easy and all valid input files must end with it. The # comment is easy. Then work on CREATE, INSERT, and PRINT … ALL; when those three are coded, you can test that your project is starting to work. Don't worry about indices at the beginning! When you start working on GENERATE and indices, your existing code is still perfectly usable. You'll just have to add to the existing code, things like "if a useful index exists, do it this way; else do it the old way", and the "old way" code doesn't have to change!

Reading and Printing Bools

For most of the types that can be represented in a table, just reading in and printing as usual will do the trick. That cannot be said for `bool`, however. Reading and writing `bool` as 1s and 0s isn't the most enjoyable experience. Luckily there's a fix for that in the form of `std::boolalpha`. To read in and print `bool`s as "true" and "false", you just need to use this once at the beginning of your `main()` and it will persist for the remainder of the program. All you'll need to do is add the following lines of code to your main file:

```
using std::cin;         // put under the other usings
using std::cout;        // put under the other usings
using std::boolalpha;   // put under the other usings
...
int main(...) {
    ios_base::sync_with_stdio(false);   // you should already have this
    cin >> boolalpha;  // add these two lines
    cout << boolalpha;
    ...
    // the rest of your program
```

Switches and Enums

There are many situations in this project where you will find yourself choosing from a finite set of options. The types that are allowed to be in a table is one case; the conditional operators (<, >, =) are another. Based on your time in 101/183, you'll probably be drawn to the traditional conditionals of the form

```
if … { } else if … { } else { }
```

There are many cases, however, when what you really want to be using is the `switch` statement. For reasons that this appendix won't go into, compilers may be able to optimize switch statements to be significantly faster than a traditional if/else block. If you think that what you're doing makes sense as a switch, then you should probably be using one. That being said, don't contort yourself into using one if it doesn't make sense. In those situations, it's not unlikely that the compiler will just turn it into an if/else anyway. Bear in mind that you can only switch on expressions that can be interpreted as an integer (e.g., ints, chars, and enums).

These finite set of options should lead you to more than just switches. A very useful construct for this project is the `enum class`. These provide the same functionality as `enum` with the scoping and type

safety of classes. A helpful use of `enum` in this project is to represent the different types that can be represented in a table. The following definition may prove useful; it is already coded for you at the top of `TableEntry.h`:

```
enum class EntryType { String, Double, Int, Bool };
```

You can use this `enum` to keep track of what types are stored in each column of a table; for example by creating a vector of this type:

```
vector<EntryType> types;

... types.push_back(EntryType::Bool);
```

You can then later use these stored types in, for example, a switch! You might also want to remember the names of the columns. The column names and types are called metadata: data about data. They come from the user and need to be saved, but they also describe the other data in your program.

Lists Are Not Your Friend

You will probably at least once while working on the project consider using a `std::list<>` for something. There's a specific application within this project for which using `std::list<>` actually kind of makes sense. Unfortunately it only *kind* of makes sense. One use case for lists is if you need to remove data from arbitrary locations within the list given that you already know the locations of that element. They're terrible for almost everything else. They're absolutely abysmal for iterating over due to their data being not contiguous in memory. This project requires a lot of iterating. Enough said.

Making Joins More Efficient

Doing a join where the second table has an index on the column being joined on is easy and efficient. Unfortunately, when this is not the case, joins can be quite slow and inefficient - given two tables, of sizes `n` and `m`, the complexity of the join is O(n*m), which is quite awful. But can you do better? You can't use sorting to make this faster, because that would be problematic with regards to the ordering of output. What other tools might you use to solve this problem? This is a problem to discuss with your classmates.

`unordered_map<>` is Your Friend Until It Isn't

The `std::unordered_map<>` is a quite useful tool. Many of you are probably learning about hash tables for the first time and thinking "wow, I should use these for everything." We've all been there. Some of us are still there. The thing with `unordered_map<>` is that you're trading space for time. You get average case *O*(1) lookup but at the cost of significant memory overhead, possibly even twice as much as you need. The `std::map<>` is better in some ways and worse in others. Its memory overhead is constant per element, similar to that of a linked list, but its lookup complexities don't quite match.

An important decision in this project is when to make use of these shiny new tools. In one case, with the generation of indices, we've created a specific use case for these types. There will likely be other places in the project where you will need to use these types as well. Sadly, it's not actually the case that `unordered_map<>` is always the best choice in every situation. In many cases a good old vector will do. Spend some time considering what is actually the best container for each individual situation.

Basic Templates

One of the most useful C++ language constructs for this project is probably the template. Templates are one of C++'s main tools for generic programming, or writing code that works the same across many different types. You may remember them from Project 2, in which they allowed us to store any type in a Priority Queue. Let's say I want to create a method that inserts a new item into a table and so define it as follows:

```
void insert_into_table(??? item);
```

But what is the type of `item`? We don't want to restrict it to int, bool, double, or even string, since for all of these types the internal code should be the same - stick it in a `TableEntry` and insert it into the table. The solution, as you may have guessed, is templates. Use the following definition instead!

```
template <typename T>
void insert_into_table(T item);
```

or

```
template <typename T>
void insert_into_table(const T& item);
```

This tells the compiler that `insert_into_table` should accept any type and act the same way. This can greatly reduce code duplication in your project. There are more advanced techniques that can be used to tell the compiler to restrict the types allowed to be used with a templated method, but that's beyond the scope of this course.

(unordered_)multimaps, and (unordered_)multisets

Throughout the course of this project, you will probably at least once find yourself considering one of the `multi`- containers. Do not use a `multi` container.

The `multi`- variants of `map` and `set` are comparable to using a non-`multi` version with a `vector` as the mapped-to type. We recommend using a `map`-plus-`vector` implementation instead mostly to simplify the interface that you'll be dealing with. For example, if you were considering writing `multimap<Key, Value>`, instead write `map<Key, vector<Value>>`.

The `unordered_multimap`s and `unordered_multiset`s are even worse. The main problem with using them is a lack of control over the ordering of items with the same key. There are many other performance-related issues with the implementations of these containers, but some of them are based on concepts not covered in this course, so take our word for it and just don't use them. Instead stick to variations of the `map`-plus-`vector` approach described earlier (i.e. `unordered_map<Key, vector<Value>>`).

How **NOT** to Use `TableEntry`

As you build your solution you will notice that there are a few key features of `TableEntry` that don't exist or were explicitly disabled. The two most important instances of this are the assignment operator and default constructor.

The assignment operator is disabled as there's no reason you ought to be assigning `TableEntry` objects. Once created, a `TableEntry` is essentially a constant piece of data. This indirectly disables a few other things, such as calling erase on a vector of `TableEntry` objects (i.e. `vector<TableEntry> v; /* put in data */ v.erase(...);`).
You also can't pass a `TableEntry` by value; this is intentional. Copying a `TableEntry` is inefficient.

The default constructor is disabled as a default-constructed `TableEntry` doesn't make much sense. A `TableEntry` exists to represent a cell in a Table whose type isn't known at compile time. What would the internal type of a default-constructed `TableEntry` even be? As with the assignment operator, this indirectly disables a few things, such as calling `resize` on a vector of `TableEntry` objects. That's not actually a bad thing, however, since you're probably better off using `reserve` on the vector anyway.

If you find yourself needing either of these, you might want to reconsider your design rather than trying to contort your solution to get around them. **DO NOT MODIFY** the `TableEntry` files; they will be overwritten when you submit to the autograder.

What are vectors

Vectors, while very useful, are not magic. Many people visualize a vector like this:

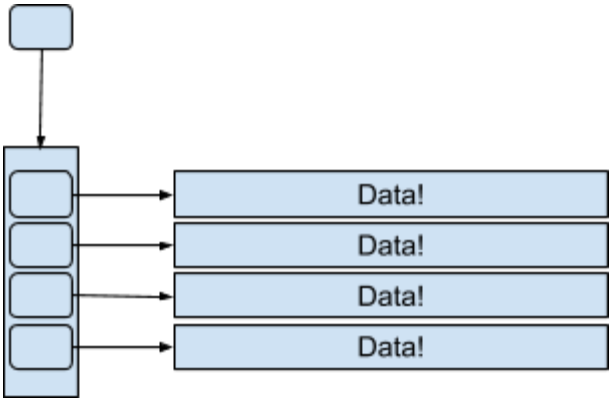
Or basically an array that magically is always big enough to hold the required amount of data. But what does a vector actually look like? Well the code for a vector (without functions) looks something like this:

```
template<typename T> class vector {
private:
    size_t size; // how much data I hold right now
    size_t size_of_alloction; // how much data I can hold before automatically
resizing
    T* data; //internal storage of data
};
```

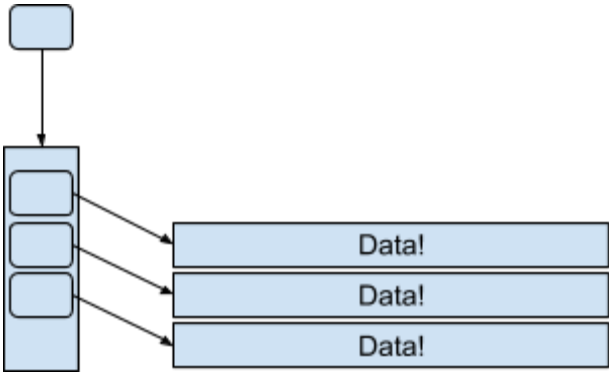
Knowing this, we really should be visualizing vectors like this:

Where the small square is the actual vector class object, and Data is the actual data that the vector "contains", or, in reality, points to.

Now let's consider what a vector of vectors looks like:



Let's say I declared this as `vector<vector<type>> v;` and then inserted my four inner vectors. Now let's think about what happens when I want to erase the first inner vector using `v.erase(v.begin())`. You might be thinking, "there's no need to move the actual data around in memory, we can just move the little squares!" Well you'd be right, and luckily the people who write compilers and the STL noticed this also! So, after calling delete, the only things that get shifted are the little squares:



Pretty efficient! You shouldn't be afraid that erasing an inner vector in a vector of vectors moves around all (or any) of the data. Regardless of what happens to the outer vector, even if it gets resized and moved around, the data for the inner vectors (big rectangles) won't get moved, just the little squares.

You can think of the data items that vector points to as simply dynamic arrays that vectors manage for you. Normally, you don't want to touch or look at the data array directly, since as the vector grows, that particular array could get deallocated and reallocated somewhere else. In specific scenarios, however, using it can actually be beneficial. For that reason, vectors actually have a member function that returns a pointer to the internal data, `vector::data()`. Do with this information what you will. Remember that it's only safe to use the pointer returned by `vector::data()` if the data array will never change size, as when vectors grow, they move stuff around in memory, invalidating pointers.

Reading and dealing with input

The input for this project is very well-formed: at any given time, you know exactly what you'll need to read next. **DON'T** read valid input using `getline()` and then break it into pieces, just read using `>>`. Remember though, once you encounter an error you will want to use `getline()` to eliminate the invalid portions of input. Also, read what you need; don't always read a `string` and then convert into what you need (an example is coming up next).

When you're reading user input, use a `do...while` loop (if you don't know what it is, look it up). This makes it very easy to prompt the user and read their input at the top of the loop. Don't prompt the user with `cout << "% "` inside of every command processing function, do it once inside the top of the loop.

Learn the difference between `.push_back()` and `.emplace_back()`. Using `.push_back()` takes what you give it, and adds a copy of it into the vector. The `.emplace_back()` member function calls a constructor to convert from what it's given to what needs to be stored into the vector. For example, if you have a vector of `TableEntry` objects, and `.emplace_back()` an `int`, it will automatically call the `TableEntry` constructor that accepts an `int`, and add a `TableEntry` containing an `int` to the

end of the vector! For example, when you're processing the INSERT INTO command, and the next column that you need to process contains an `int`, read directly into an `int` variable using `>>`, then `.emplace_back()` that `int` into the vector (which should be the correct row of your 2D table of data).