Project 4: Registrar

Due: 2 December 2019

We've just made our way through registration season, and it serves as the inspiration for the fourth project in this course: maintaining a searchable and updatable database of course offerings and registrations. The universe here is somewhat simplified, but broadly similar to what the registrar at Longwood (and any other college) needs to maintain.

Objectives

In the course of this project, the successful student will:

- use a published API to access a library,
- design a relational data schema with multiple tables, and
- write complex queries in a standard database query language (SQL).

The data

The central data for the system will be the courses themselves. As at Longwood, the courses will have titles as well as course numbers and sections—so, CMSC 201-1 "Computer organization" would be a perfectly valid course. As a bit of a simplification, we'll say that all courses are either Monday-Wednesday-Friday or Tuesday-Thursday, and each course meets during exactly one numbered period (1–8 on MWF and 1–5 on TT). Courses have a maximum number of enrolled students (e.g. 20 or 30), and when that many students are in the course, the course is full.

Each course is taught by exactly one professor, whose name you can assume to be unique. (Each professor may teach multiple classes, of course.)

Students will, in general, sign up for more than one course. All their courses will meet at distinct times.

The requests

Students need to be able to register for classes. In addition, they need to be able to get useful listings of courses, as well as (after registration) a useful list of the courses they actually signed up for.

Professors need to be able to get a listing of courses they're teaching, and for individual courses, they need to be able to get a listing of all the students in the course.

The registrar herself need to create the classes in the first place; and needs to be able to (for courses) change the course title and the enrollment cap, and (for students and professors) change names. (Note that if a student's name changes they should still be enrolled in the same classes, and if a professor's name changes they should still be teaching the same classes.) The registrar also needs to be able to create summary reports such as student transcripts.

See the "rubric" section at the end of this handout to get more specific details on some of these requests.

Prep work

Your initial work on this project involves getting your programming language of choice to successfully talk to a database file through the SQLite API. This is in some ways more complicated than previous prep work (although more amenable to following tutorials, and see the back pages of this handout for additional help), but it's absolutely prerequisite to getting anything else in the project done, so I've front-loaded it.

For the prep work, you should have:

- A `.db` file in SQLite format that contains sample data: at least one table, with at least two columns and at least three rows,
- A program that, when compiled and run with the `.db` file as its command-line argument, accesses that file using the SQLite API for that language, runs "`SELECT * FROM`" the table in the file, and prints the results; and then reads a single word from standard input and runs a "`SELECT * FROM ... WHERE ...`" query based on that word, and prints the results.

- A readme file that says how to compile and run that program, even if this is just repeating back instructions I've given you at some point.

The database file can be built by hand through running the `sqlite3` interactive command-line program. The program will have the following broad structure:

```
Import/include necessary libraries or headers
Open connection to given database file
Prepare SQL statement object representing SELECT query
Bind parameter in the prepared statement to the value read from input
Execute the query statement
While there are rows left in the result,
    access and print the columns in the current row
Clean up the statement object
Close the connection
```

See the back pages of this handout for some info on how to get started with these in a few common languages.

This work is due next Wednesday **at 8pm**. You'll have a chance to ask questions about it in class on Wednesday to clear up any last-minute issues, but you really don't want to wait until then to start it. When you're ready to hand it in, use the `handin` script as described at the end of this document.

Design work

Once you've got the shell of the program running (or perhaps while you are working on that, but the design work depends on course content we won't cover until next week), you can start thinking about the query and schema design.

1. Devise a short list of courses and other data that can serve to fuel test cases. Each type of data should in general have just two or three instances, to facilitate easy typing and editing and quick queries.
2. Reflect on your data and write out your database schema: for each table, be sure to name each column *and* give it an SQL-appropriate

data type. As you edit your schema, you may feel the need to update your examples from the previous part, and vice versa—this is fine! But when you're done, the schema should be consistent with the sample data.

3. Identify the relationships between the different tables in your schema—how does information in one connect to information in another? For each such relationship, designate it as "one to one", "many to one", or "many to many". (If you're reading ahead and want to draw this point as an E-R diagram, that's cool, but prose analysis is ok too.)

4. Reread the section "The requests" from the beginning of the handout and look through the rubric at the end, and write out an interaction scenario between a student and your program—from the time they sit down, what does the program ask them? What do they ask the program? How should the program respond to their request? What changes occur as a result? Use your data example as a starting point and be as specific as possible.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of the database system described here, TOGETHER WITH convincing proof that it is correct. When run, it should be provided with one command line argument—the name of the database file containing the registrar data.

Your front end, written in the high-level language of your choice (e.g. C++, Java, etc), will use a standard database API to interact with a database in SQLite format, and print out results. Some statements will be commands that cause an update to the system; others will be SELECT queries, whose output may be in table form but should still reflect the *user's* understanding of the data.

That means:

- DO use database queries to do things like provide the user with a valid list of options to select from;
- DO NOT just print out a bunch of tables and assume the user can "just figure it out";

- DO NOT just make a bunch of `SELECT *` queries and read them into maps and vectors and combine and aggregate information in the front end; and
- ESPECIALLY DO NOT overly subject your users to weird artifacts of how things are stored internally, such as cryptic ID numbers or the fact that some data is spread over multiple tables.

It's ok if the output isn't especially formatted, but the response to a query should be a single table with all the requested info and not a lot of cruft. You should in general be displaying the whole result of the query to the user—if there *is* too much cruft, trim it out by refining the SQL query itself.

To make this project doable for students who haven't had experience with writing GUIs, we'll do the interaction via a text-based menu, giving it a bit of a 1980s feel. In a "real" version of this, there would be a web or GUI front end that would handle the user interaction and display results—but that's outside the scope of this course.

Your system should be able to handle all the requests indicated in the opening section of the handout, as well as requests to do basic maintenance of the system (e.g. adding and removing courses and sections from the system). It doesn't need to worry about authentication or security, or preventing one student from seeing another's listings, or anything like that.

Implementation notes

Spend time early in the project thinking about what data you'll need to represent, even for the parts that you don't plan to implement until later. It's not impossible to change your data schemas later, but it's easiest if you get the structure of the data (mostly) right first.

Work on the queries and updates one at a time; and when you're working on one, start by constructing the SQL by hand to make sure you can find an SQL statement that works, before trying to make the front end program build and execute it.

Though SQL is generally case-insensitive, use the all-caps forms of at least the keywords when you're generating the SQL statements. This sometimes makes it easier to tell the difference between the part of your code that is in the general-purpose language and the part that's in SQL.

Rubric

Note that I am not able to spend a ton of time with your program (although I will read it at least enough to verify your use of the SQLite APIs, I definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to). Having complete and correct documentation is an easy 25 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (15 points), design work (10 points), and documentation (25 points), there remain 100 points in the rubric, which will be awarded according to the table below. Under each score, I show (for your convenience) the total cumulative points if you get that item plus *all* the previous points, and the letter grade this corresponds to. It is arranged roughly in the order I suggest you attempt them, with the earlier ones being easier or enlightening with respect to the later ones, but you can in general get points for the later ones (if they work) without getting the earlier ones.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

Score	Description
10 (60/D−)	As in prep work, `.db` file includes at least one table with at least two columns and three rows; executable program can access the database via API and print full contents of a table.
10 (70/D)	`.db` file has a clean schema that supports all (or almost all) required requests for registrar project, and includes a useful amount of sample data
Basic queries:	
5 (80/D+)	Executable program can list all the courses on offer (this term)
10 (85/C−)	Parameters in SQL statements are bound to values at run-time, using appropriate library calls, with at least two parameter types (not necessarily in the same query).
5 (90/C)	Executable program can list all the courses from a particular department (this term)

5 (95/C)	Executable program can list all the courses taught by a particular professor (this term)
Basic updates:	
5 (100/C+)	Executable can interact with the user (registrar) to add courses to the databse
5 (105/B−)	Executable can interact with the user (registrar) to change course titles and capacities, and student and professor names
Complex queries and updates:	
5 (110/B−)	Implemented at least one query involving joining data from two tables, and none show duplicate columns or spurious data
5 (115/B)	List all the courses (this term) that meet during a given range of times (e.g. MWF during 1st through 4th period)
5 (120/B+)	Register a student for class(es) (for this term)
5 (125/B+)	Show a student the course schedule that they signed up for (this term)
5 (130/A−)	Give a professor course lists showing all the students enrolled in one of their courses
5 (135/A)	List all the courses (this term) that are not yet full
5 (140/A)	Give a student a transcript that shows all the courses they've taken at this school, over all terms
Design	
5 (145/A+)	Implemented at least one query and at least one update, and all library calls have return value error-checked (C++) or use appropriate exception handling (Java, Python)
5 (150/A+)	Implemented at least three of the "complex queries" and all have clean front end that reflects user's mental model

When you add each one, make sure you are not compromising the user interface (i.e. doing something that confuses the user because it reflects the internal structure and not the user's mental model)—five of the points are dedicated specifically to this.

The last three of the "complex queries" are each tricky for somewhat different reasons. You may wish to work on them in parallel; if you get stuck on one, try a different one.

You *don't* have to do combinations of the above query types, even though some of them might make sense (e.g. to let the user ask specifically for CMSC courses that don't conflict with their existing schedule, or all the courses in a time range that aren't full). If you want to, feel free! But it's not in the rubric.

To guarantee that you get credit for everything you implemented, be sure your documentation says what you did; and tell me how to interact with the `.db` file(s) that you include in your handin to verify that each piece is implemented.

Handing in

For both the prep work and the final version, hand it in as `proj4` using the handin script. The final version is due at 8pm on the due date.

Getting started with SQLite libraries

In C++:

The sqlite3 library and its header `sqlite3.h` is already installed on torvalds and the lab machines. Go to `https://www.sqlite.org/cintro.html` , which documents the interface, and bookmark the URL.

It is a very C-style interface, and when it describes an "OUT" parameter, this normally means that you will create a variable before calling the function, then give the address of that variable to the function (so that it can fill in the value). For instance, to first open the database, you might write

```
int result;
sqlite3* db;
result = sqlite3_open_v2(argv[1], &db, SQLITE_OPEN_READWRITE, nullptr);
```

The actual return value of each function is a result code; in this case if it is not `SQLITE_OK` it means there was an error opening the file, and you should exit the program gracefully. You should *always* assign the result of the function call to a variable, and you should *always* check the value of that variable to know if the command succeeded.

Use the `v2` versions of the SQLite functions where they are available.

Look over the documentation and compare to the pseudocode on p. 3 of this handout; try to see which functions correspond to which lines of pseudocode (for instance, the line above corresponds to “open connection to given database file”). Write code to call them, getting each to compile before tackling the next one.

The sample code that you’ll find online that uses `sqlite3_exec` is really really not what you want to be looking at—among other things, it doesn’t let you prepare statements and bind values into the parameters.

Note that the type of several parameters (notably in `sqlite3_bind_text`) is not a true C++ `string`, but a `const char *`, so if you’re using `string`s you’ll need to use `.c_str()` to get them into the right type for use in binding.

Note further that the return type of `sqlite3_column_text` is `const unsigned char *`, so you *really* can’t assign it to a `string`: first, it could return `nullptr` (if the value in that column of the current row is NULL), and converting that value to a `string` throws a runtime exception. Second, the builtin `string` type is built to contain `char`, not `unsigned char`. The good news is, `printf` and `cout` are both happy to work with C-strings of type `const unsigned char *`. So, since you mostly shouldn’t be doing a lot of processing of these result strings anyway, you can mostly just use variables of the type `const unsigned char *` to hold them until you print them out. (If you feel you need to go back and forth between `string` and this other type, see me and I can help you make that work.)

When you compile your `.cpp`, you will need to explicitly link it against the sqlite3 library:

```
compile myfile.cpp -lsqlite3
```

In Java:

The library for you is an implementation of a Java-standard JDBC API specified in the `java.sql.*` package. Its documentation can be found right in the standard Java API docs (which are installed, among other places, at `http://cs.longwood.edu/java/docs/api/`).

The two particularly weird things you’ll have to do to start are: at the top of `main`, make a call to

```
Class.forName("org.sqlite.JDBC");
```

This forces the class to be initialised at runtime. Second, to establish the connection, you'll call

```
DriverManager.getConnection("jdbc:sqlite:" + args[0]);
```

which returns a `Connection` object. (The filename is prefixed with indications of its storage format and intended interface.) From there on out, you'll follow the API's use of `PreparedStatement`, `ResultSet`, and so on.

Look over the documentation and compare to the pseudocode on p. 3 of this handout; try to see which methods correspond to which lines of pseudocode (for instance, the line above corresponds to "open connection to given database file"). Write code to call them, getting each to compile before tackling the next one.

When you compile your `.java` file, you don't need to do anything special, but when you *run* it you need to make sure your `CLASSPATH` includes the SQLite library. You can do that by editing your `.bashrc` file, or else by running your program as

```
java -cp '.:/usr/share/java/sqlite-jdbc-3.8.7.jar' MyProgram
```

In Python:

The SQLite library is included with python, but you do still need to `import` it at the top of your code. From there, you can follow the API documentation at `https://docs.python.org/3.7/library/sqlite3.html` .

The non-"shortcut" functions for the Python-SQLite interface involve an additional level of indirection from the code relative to the pseudocode on p. 3: from the database connection (a `Connection` object), you create a `Cursor` object, which then is what performs the actual queries and updates. On the other hand, there's no separate "prepare statement" and "bind parameter" steps; the cursor can execute an SQL statement directly.

In those three or any other:

There are squillions of pages on the internet about interfacing SQLite3 with every programming language out there. Use them! What I've written above is nowhere near a complete tutorial, just enough to get you started and have some idea of what to search for.