Project #4: Database Programming

ECE 650 – Spring 2020

See Sakai site for due date

General Instructions

1. You will work individually on this assignment.
2. The code for this assignment should be developed and tested using a Linux Virtual machine that you may create at the following location:

   https://vm-manage.oit.duke.edu/vm_manage

   Select following image: Ubuntu 16.04

   Other environments, unfortunately due to complexity, will not be supported.
3. You must follow this assignment specifications carefully and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient.

Overview

In this assignment, you will build a PostgreSQL database. The relation schemas (tables) of the database will be provided, along with the database values. You will implement:

1. A C++ program to read text files containing the entries (rows) for each table, and build the database by creating each table and adding entries.
2. A C++ library (set of functions) which will provide an interface for a program to interact with the database (e.g. add rows to the tables and query for certain information).

The database relates to ACC basketball teams, and will allow queries to discover information about things like player statistics, team attributes, etc. The database will have 4 tables, specified as follows (an underlined attribute indicates the primary key for the table):

PLAYER (<u>PLAYER_ID</u>, TEAM_ID, UNIFORM_NUM, FIRST_NAME, LAST_NAME, MPG, PPG, RPG, APG, SPG, BPG)
TEAM (<u>TEAM_ID</u>, NAME, STATE_ID, COLOR_ID, WINS, LOSSES)
STATE (<u>STATE_ID</u>, NAME)
COLOR (<u>COLOR_ID</u>, NAME)

*Note the following abbreviations: MPG = minutes per game, PPG = points per game, RPG = rebounds per game, APG = assists per game, SPG = steals per game, BGP = blocks per game

Tips on Working with the Virtual Machine

When you create your virtual machine and log-in for the first time, you will notice there may be few programs installed (e.g. no gcc, emacs, vim, etc.). You can download your choice of software easily using the command: sudo apt-get install <package name>. For example:

```
sudo apt install build-essential emacs
```

In order to create a PostgreSQL database and interact with the database through a C++ API, you will need to install packages, which can be obtained via git as follows:

```
sudo apt install libpqxx-dev
```

What You Will Implement

Several skeleton files, and database table information files are provided to get you started. The following describes each file:

- **Database source text files: player.txt, team.txt, state.txt, color.txt**. These files contain table-like information for each entry and each attribute that should be inserted into the respective database table.
- **main.cpp**: The main function. Here you should implement code which will setup the database on each execution of the program. Specifically, it should drop (if needed) and add each table to the database (named ACC_BBALL), and then read information from the source text files and add rows to each table as appropriate.

- **query_funcs.h and query_funcs.cpp**: Here you will implement functions to interact with the database (add new entries and print the results of 5 different queries specified below). You may add new functions to these files if desired (you don't have to), but you should not change the definitions of the existing functions.
- **exerciser.h and exerciser.cpp**: Here you can add code in the exercise() function to test your query functions. The exercise() function is called from main() after the database is initialized.
- **Makefile**: Will compile all source files into an executable program named "test"

Your task is to do the following:

- Create a PostgreSQL database named **ACC_BBALL** (All Caps)
- Create a user for the ACC_BBALL database named "postgres" with password "passw0rd" (see the ppt charts for how to set the password of the postgres user).
- Implement support in main.cpp to connect (as user 'postgres') to the ACC_BBALL database and initialize its tables and data. Of course, usually a database will live persistently, but for the purposes of evaluating the submissions, the program should re-create the database every time. You should drop tables that may already exist (e.g. from prior runs of your test) before creating an initializing the tables in the program.
- Implement the following query functions:
    - **query1()**: show all attributes of each player with average statistics that fall between the min and max (inclusive) for each enabled statistic
    - **query2()**: show the name of each team with the indicated uniform color
    - **query3()**: show the first and last name of each player that plays for the indicated team, ordered from highest to lowest ppg (points per game)
    - **query4()**: show first name, last name, and jersey number of each player that plays in the indicated state and wears the indicated uniform color
    - **query5()**: show first name and last name of each player, and team name and number of wins for each team that has won more than the indicated number of games
    - **Important Note:** Each query function should print its output. The format of this output should be as follows. The first row of output should contain each field name (separated by a single space character. The next rows of output should contain the values returned from the query, each also separated by a single space. The field name and each row of output should appear one line after the next.
- You may test your database and query functions by adding calls to the query functions inside the exerciser() function. For the purposes of grading assignments, we will replace the exerciser.cpp file with a new one that will test a variety of query calls.

Example Output and Checking Results

The following text shows an example output. This output corresponds to **query1** which is invoked to show all players with a minimum minutes per game (MPG) of 35 and a maximum minutes per game (MPG) of 40.

```
PLAYER_ID TEAM_ID UNIFORM_NUM FIRST_NAME LAST_NAME MPG PPG RPG APG SPG BPG
153 12 3 Andrew WhiteIII 37 19 5 1 1.6 0.4
154 12 20 Tyler Lydon 36 13 9 2 1.0 1.4
30 3 5 Luke Kennard 36 20 5 3 0.8 0.4
59 5 44 Ben Lammers 35 14 9 2 1.2 3.4
87 7 5 Davon Reed 35 15 5 2 1.3 0.5
98 8 4 Dennis SmithJr. 35 18 5 6 1.9 0.4
130 10 32 Steve Vasturia 35 13 4 3 1.2 0.1
```

The Unix command 'diff' can be used to compare results. For example, if the output above is copied into a file called validation.txt, it can be compared to output for the same query from your program. If your program is in a file called output.txt, the diff command could look as follows:

```
$ diff -w output.txt validation.txt
```

Note that the -w indicates that 'diff' should not report differences in whitespace (spaces, tabs) in the output. If you want to compare two files character for character, the -w flag can be omitted.

The precision after the decimal point of the SPG, BPG should be 1, e.g. if the value is 0 or 1, output them as 0.0 or 1.0.

Make sure your SQL output header should be in the **same order and same name** as the table attribute definition of Project 4 document, (e.g. PLAYER_ID TEAM_ID UNIFORM_NUM FIRST_NAME LAST_NAME MPG PPG RPG APG SPG BPG). Since we use autograder this time, failing to print the correct header will make it trouble even if you get the correct results.

Detailed Submission Instructions

Your submission will include the following files:

1. **Source Code Files:** main.cpp, query_funcs.h, query_funcs.cpp, exerciser.h, exerciser.cpp
2. **Makefile – Even if you have not made changes to the one provided**
3. **Database source text files**: player.txt, team.txt, state.txt, color.txt

You will submit a zip file named "**proj4_netid.zip**" to **Gradescope**, e.g.:

```
    zip proj4_netid.zip Makefile *.h *.cpp *.txt
```

Extra Credit (+25%)

By implementing SQL transactions in C++, you may get a deeper understanding of SQL code and consider it as a common way to use databases. However, a typical use of databases in modern software architecture is using a higher-level framework (e.g. an Object Relational Mapping, or ORM) which obviates the need to worry about transaction details. Specifically, Object-Relational Mapping (ORM) is a technique that allows you to write transactions using an object-oriented paradigm. Many libraries have implemented this technique.

For example, some of you may have used Django to develop web applications. Django contains a default ORM layer to interact with data from relational database management systems like PostgreSQL. You may refer to *django.db.models* for details. And in addition to Django ORM, there are other ORM libraries, such as Hibernate, SQLAlchemy, Doctrine, etc. Choose one ORM library and implement the five queries shown in the previous statement. Submit your code as a zip file named “**proj4_extra_netid.zip**” to **Gradescope**. We will schedule to let you give a demo to TAs for grading.