Tournament Results: Getting Started

PDF Download

In Project 2, you will be writing a Python module that uses the PostgreSQL database to keep track of players and matches in a game tournament.

Project 2 was designed to teach you how to create and use databases through the use of database schemas and how to manipulate the data inside the database. This project has two parts: defining the database schema (SQL table definitions) in tournament.sql, and writing code that will use it to track a Swiss tournament in tournament.py.

Getting Started

- You will complete this project within the Vagrant virtual machine we've provided and configured for you. If you would like to review that before moving on refer to the course materials for help with installing Vagrant and Virtual Box, and previously recorded office hours where we'll show you how to use Vagrant.
- **First, fork the fullstack-nanodegree-vm repository so that you have a version of your own within your Github account.**
- Next clone **your** fullstack-nanodegree-vm repo to your local machine.
- Now, let's explore the starter code for this project provided within the VM: `cd` into `/vagrant/tournament` where you will see there are 3 files you have to work with on this project:
    - `tournament.sql`
    - `tournament.py`
    - `tournament_test.py`

Understand the purpose of each file

- **tournament.sql** - this file is used to set up your database schema (the table representation of your data structure).
- **tournament.py** - this file is used to provide access to your database via a library of functions which can add, delete or query data in your database to another python program (a client program). Remember that when you define a function, it does not execute, it simply means the function is defined to run a specific set of instructions when called.
- **tournament_test.py** - this is a client program which will use your functions written in the tournament.py module. We've written this client program to test your implementation of functions in tournament.py

Using the Vagrant Virtual Machine

- The Vagrant VM has PostgreSQL installed and configured, as well as the psql command line interface (CLI), so that you don't have to install or configure them on your local machine.
- To use the Vagrant virtual machine, navigate to the full-stack-nanodegree-vm/tournament directory in the terminal, then use the command `vagrant up` (powers on the virtual machine) followed by `vagrant ssh` (logs into the virtual machine).
- Remember, once you have executed the `vagrant ssh` command, you will want to `cd /vagrant` to change directory to the synced folders in order to work on your project, once your cd /vagrant, if you type `ls` on the command line, you'll see your `tournament` folder.
- The Vagrant VM provided in the fullstack repo already has PostgreSQL server installed, as well as the psql command line interface (CLI), so you'll need to have your VM on and be logged into it to run your database configuration file (`tournament.sql`), and test your Python file with `tournament_test.py`.

Using the psql command line interface

- The very first time we start working on this project, no database will exist - so first, we'll need to create the SQL database for our tournament project. From psql, we can do this on the command line directly using a create statement or by importing `tournament.sql` (which then executes whatever commands are in the .sql script).
- `tournament.sql` is where we'll create our database schema and views; we also have the option of creating the database and tables in this file.
- With psql, you can run any SQL query on the tables of the currently connected database.
- When using psql, remember to end SQL statements with a semicolon, which is not always required from Python.
- To build and access the database we run `psql` followed by `\i tournament.sql`

Command	Description	Usage	Action
`psql`	`launches the psql command line interface`	`psql tournament`	`launches and connects to tournament database`
`\c`	`connect`	`\c tournament`	`connects to the tournament database, drops connection to previous database`
`\i`	`import`	`\i tournament.sql`	`executes the sql commands within the sql file from psql`

`\?`	`help`	`\?`	`get help with psql commands`
`\q`	`quit`	`\q`	`quit the psql command line interface`
`\d`	`describe`	`\d matches`	`describes the table structure`
`\dt`	`list tables`	`\dt`	`list tables in current database`

There are many more psql commands you will find useful! Here are two psql cheat sheets for your reference:
http://www.postgresonline.com/downloads/special_feature/postgresql83_psql_cheatsheet.pdf
http://www.petefreitag.com/cheatsheets/postgresql/

Using the tournament.sql file

The tournament.sql file should be used for setting up your schema and database prior to a client making use of the database for reporting and managing tournament players and matches. This file will only be ran once by a client setting up a new tournament database, however we will probably run this file many times as we work on this project.

An sql file can contain any sql commands, however we only need to create the database, tables, and views in the tournament.sql file, because we will be running queries from our Python file. The purpose of this file is to set up our data structure: the tables and views. Because we are using psql, you can also utilize psql commands in this file, for example towards the beginning of the file I would include a "\c tournament" command to connect to the tournament database.

Our recommendation is to test your sql commands on the psql command line before placing them in your sql file. Once you've determined how you want to structure your database and tables, write those commands into your SQL file.

> *A successful development process for this project will mean you are likely cycling between psql command line for experimenting with SQL commands and queries, and writing these commands programmatically into either tournament.sql or tournament.py.*

First off, we need to create the database and connect to it. To do so, we use the commands:

```
vagrant@trusty32: vagrant => CREATE DATABASE tournament;
vagrant@trusty32: vagrant => \c tournament;
vagrant@trusty32: tournament =>
```

Once we have created a database, we can add the tables that we will be working with. A very basic tournament would need at least a Players and Matches table. We can create them with the command:

```
CREATE TABLE [table name](....);
```

As a reminder, you may include these commands in your tournament.sql file, and it will execute when you run

```
vagrant@trusty32: psql => \i tournament.sql
```

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$ psql
vagrant=> \i tournament.sql
You are now connected to database "tournament" as user "vagrant".
tournament=>
```

Because we are likely to revise the structure of our tables by adding or removing columns, and restructure our database by adding tables and views, if we rely on the sql file, we will be running it multiple times, and as a result will likely see this error at some point:

```
vagrant@trusty32: psql => ERROR: createdb: database "tournament" already
exists.
```

If you look up the SQL command, DROP IF EXISTS, you might decide you want to include this in your SQL file, to address the error above and allow you to use your SQL file repeatedly as you revise your schema.

You'll need to determine what columns are necessary, and their data types, and include these in the create table command (which can be done from the psql command line to get started, and later added to the .sql file once you are more decided on your table structure).

tournament.py and tournament_test.py

Rely on the unit tests in `tournament_test.py` as you write your code. Writing your tournament module should be done in conjunction with testing using the `tournament_test.py` file; If you implement the functions in the order they appear, the test suite can help you incrementally test your code as you write each function. Each function has a corresponding test function and you can comment out the corresponding tests for function you have yet to implement.

Recall the sequence of SQL query commands needed to manipulate database records:

```
conn = connect()
c = conn.cursor()
c.execute("your query;")
conn.commit()
conn.close()
```

The various functions in `tournament.py` and their corresponding test functions in `tournament_test.py` are:

`tournament.py` function	**`tournament_test.py` test function**
`connect` Meant to connect to the database. Already set up for you.	
`deleteMatches` Remove all the matches records from the database.	`testDeleteMatches`
`deletePlayers` Remove all the player records from the database.	`testDelete`
`countPlayers` Returns the number of players currently registered	`testCount`
`registerPlayer --` Adds a player to the tournament database.	`testRegister, testRegisterCountDelete`
`playerStandings --` Returns a list of the players and their win records, sorted by wins. You can use the player standings table created in your .sql file for reference.	`testStandingsBeforeMatches`
`reportMatch` This is to simply populate the matches table and record the winner and loser as (winner,loser) in the insert statement.	`testReportMatches`
`swissPairings` Returns a list of pairs of players for the next round of a match. Here all we are doing is the pairing of alternate players from the player standings table, zipping them up and appending them to a list with values: (id1, name1, id2, name2)	`testPairings`

Running your project!

Once you have your .sql and .py files set up, it's a good idea to test them out against the testing file provided to you (tournament_test.py). To run the series of tests defined in this test suite, run the program from the command line `$ python tournament_test.py`

You should be able to see the following output once all your tests have passed:

```
vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$ python
tournament_test.py
1. Old matches can be deleted.
2. Player records can be deleted.
3. After deleting, countPlayers() returns zero.
4. After registering a player, countPlayers() returns 1.
5. Players can be registered and deleted.
6. Newly registered players appear in the standings with no matches.
7. After a match, players have updated standings.
8. After one match, players with one win are paired.
Success!  All tests pass!
vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$
```

If your tests don't pass that's okay! The test suite will print explanations of what didn't work. You can read the tests themselves to see exactly how they work.

To Submit
Once you have finished your project, go to this link here. If you have a Github account (which we recommend), connect with Github to get started. If you do not have a Github account, follow the instructions here for Mac OS X 10.0 or later, here for Windows 7, 8, or 8.1, or here for anything else. These links will help you create a Github account to submit your project.

If you run into any trouble, send us an e-mail at fullstack-project@udacity.com, and we will be more than happy to help you.

Example of a 16 Player Swiss Tournament:

First round pairing is by random draw. For example, with 16 players they would be matched into 8 random pairs for the first round. For now, assume all games have a winner, and there are no draws.

After the first round, there will be a group of 8 players with a score of 1 (win), and a group of 8 players with a score of 0 (loss). For the 2nd round, players in each scoring group will be paired against each other – 1's versus 1's and 0's versus 0's.

After round 2, there will be three scoring groups:
4 players who have won both games and have 2 points
8 players who have won a game and lost a game and have 1 point
4 players who have lost both games and have no points.

Again, for round 3, players are paired with players in their scoring group. After the third round, the typical scoring groups will be:
2 players who have won 3 games (3 points)
6 players with 2 wins (2 points)
6 players with 1 win (1 point)
2 players with no wins (0 points)

For the fourth (and in this case final) round, the process repeats, and players are matched with others in their scoring group. Note that there are only 2 players who have won all of their games so far – they will be matched against each other for the "championship" game. After the final round, we'll have something that looks like this:
1 player with 4 points – the winner!
4 players with 3 points – tied for second place
6 players with 2 points
4 players with 1 point
1 player with 0 points

The Swiss system produces a clear winner in just a few rounds, no-one is eliminated and almost everyone wins at least one game, but there are many ties to deal with.