Vignette: Dealing with CRUD

This is a vignette consisting of several exercises. Each exercise is a separate homework submission, and is marked with the number of assignment points it is worth. To start each exercise, use the `python new-homework.py [language] dealing-with-CRUD` *command, and submit each exercise as a separate pull request. You can do as many or as few exercises from a vignette as you wish.*

Background

Data storage is said to be *persistent* if the data outlives the process that created it. So any data stored in the computer's memory by a running program is *not* persistent because when the program ends the memory is reclaimed and the data is lost. Data stored to a file by a program, however, is persistent in general because the file remains in the file system when the program is done. *Databases* are persistent stores for data that offer various additional guarantees that we will discuss in the near future.

There are four fundamental operations we can perform on any persistent store of data: we can **C**reate a new data entry, **R**ead the data within an existing entry, **U**pdate (i.e., change) the data stored in an existing entry, and **D**elete an entry. The operations are often referred collectively by the acronym CRUD.

In this vignette, we will practice using the CRUD operations in the context of a relational database and SQL. These operations map directly onto four SQL commands:

Operation	**Command**
Create	`INSERT`
Read	`SELECT`
Update	`UPDATE`
Delete	`DELETE`

These four commands, especially `SELECT`, can be used in a wide variety of ways. You will likely want to refer to the documentation on these commands. The PostgreSQL online documentation (link) is extensive and quite good; it is also easy to go to it by Googling something like "postgresql select," for the `SELECT` command. The examples are particularly helpful for separating out all the optional bells and whistles associated with these commands. You can also type `\h select` at the psql prompt for a short usage summary of select (or any other command). Make sure you can access the documentation before starting this assignment.

Task

In the exercises below, you will work with several tables. In the `Data/dealing-with-CRUD` directory/folder, you will find three files `README`, `player-stats.sql`, and `nba-teams.txt`. The first is a brief text file describing the data, and the second is a set of SQL commands for constructing the table `player_stats`. The last is a text file with a table describing the teams in the National Basketball Association (NBA).

If you're using `sculptor.stat.cmu.edu` to access Postgres, this data is available in `/home/areinhar`, e.g. `/home/areinhar/player-stats.sql`.

You will need to run the SQL file to construct some tables. At the command line prompt on the machine where you run Postgres, type

```
psql < player-stats.sql
```

to populate the table for you, or from the psql prompt, you can enter

```
\i player-stats.sql
```

You will either need to be in the same directory as the `player-stats.sql` file or give the path to that file in addition to the name. When you run either command, you should see

```
CREATE TABLE
CREATE TABLE
CREATE TABLE
COPY 476
COPY 476
COPY 476
```

printed in the terminal window. Once you have the data loaded, start psql and type `\d` at the psql prompt. You should see something like this:

```
                   List of relations
 Schema |           Name            |   Type   |  Owner
--------+---------------------------+----------+----------
 public | more_player_stats         | table    | genovese
 public | more_player_stats_id_seq  | sequence | genovese
 public | player_bios               | table    | genovese
 public | player_bios_id_seq        | sequence | genovese
 public | player_stats              | table    | genovese
 public | player_stats_id_seq       | sequence | genovese
(6 rows)
```

If so, you are ready to proceed with the exercises. Note that you can look at the description of each of these tables, e.g., `\d player_stats` to see how that table is configured. See the `Data/dealing-with-CRUD/README` file for details and attribution of these tables.

Requirements

☐ Review the Postgres Documentation on the four main CRUD commands.

☐ Load the data into your database.

☐ Examine the table listing to make sure that the data is available.

☐ Do one or more of the exercises below. When doing exercise #N, you should make a text file `commandsN.txt` (replacing N with the number N in the filename) listing the commands you used and describing any relevant observations. **Submit this file in your pull request.**

Exercises

Exercise #1. Refactoring the Data. (2 points) The file `player-stats.sql` builds three tables from several data sources, but they are not particularly well-designed. In this exercise, we will combine these data with insertions, updates, and selections to create a few more thematic tables. The file `nba-teams.txt` gives tabular data describing teams, which we will also use here.

(Again, on `sculptor.stat.cmu.edu`, these files are available in `/home/areinhar/`.)

In this and several of the following exercises, we will create three tables that combine the data in the existing tables:

1. `teams`
2. `players`
3. `stats2015_16`

First we will create the `teams` table, The file `nba-teams.txt` has all the information you need but is not exactly in an importable form. The *id* column should be of type `char(3)` and should be set as the primary key for the table. (We are *not* using an auto-incrementing integer in this case.) The other fields should either be of type `text`, or if you are feeling ambitious `enum` types, which represent values in small fixed sets. (See the documentation for the `create type` command and for `enum`'s in the Postgres online manual.)

Create the `teams` table with a `CREATE TABLE` command and then populate it. You can do this by hand, but there are much more efficient and interesting ways, including:

1. using your editor,
2. using shell commands, or
3. writing a program (e.g., in R or Python).

In each case, you can either produce some Postgres-interpretable format or—even more fun—produce the SQL insert commands to build the table. Try something like this if you can. Submit the code or queries you used to create the table.

Next: Let's add a column listing each player's position. In NBA basketball there are three main positions (Center, Forward, and Guard), but it is more common in recent years to use five positions: Center, Power Forward, Small Forward, Point Guard, and Shooting Guard. The `more_player_stats` table encodes the player's position through the value of

$$prl = per - 67va/(gp \cdot minutes),$$

where the variables are column names in the table. This should take one of four values: 11.5 (power forwards), 11.0 (point guards), 10.6 (centers), 10.5 (shooting guards and small forwards). We cannot distinguish between shooting guards and small forwards with this information, but it is also true that some players switch between these two positions. So it's a start.

First, use SELECT compute these statistics. Because of discretization error in the statistics, you will need to round the results to 1 digit (see the round(.,1) function in Postgres) to get the correct values. Create a temporary table with a numeric column *prl* and a string column *position*, and store these values in the former. (It might also be convenient to have a foreign key referencing more_player_stats, but that's optional.) Also, there appear to be a few errors in the data that lead to odd values of computed PRL. Use UPDATE commands to adjust these odd values. If the computed values are close to one of the designated values (e.g., 10.4 or 11.6), then you can use the closest designated value, but otherwise you can set this field to NULL. Use SELECT to look at the rows where *position* is NULL.

Second, the player_bios table contains the players' heights as a string like '6-10' to mean 6 feet and 10 inches. Use select to display a table of player's heights in inches (as an integer) using this string as input. Two helpful facts: i) the function *split_part* can split a string on a chosen delimiter and return a selected part, and ii) you can cast a value to a different type (if it is convertible) by appending ::type to the expression. For example, a string followed by ::integer converts the string to an integer if possible.

Third, use ALTER TABLE and UPDATE commands and the select from the previous paragraphs to do two things: i) add the *position* column to player_bios and ii) change the player_bios table so that *height* is an integer measured in inches.

Fourth, use a select statement to determine the average player height by position, ignoring those you set to be null.

SQL hints:

- In a where clause you can use the IN and NOT IN operators to determine whether one value is included in a specified set. For instance, a condition to see if *prl* is one of the designated values looks like:

```
where prl not in (10.5,10.6,11.0,11.5).
```

- An aliased computed column in a select is not visible in the WHERE clause (because the clause is processed first in PostgreSQL), which is inconvenient. One way to get around this is to define a local, temporary table using a WITH clause before the SELECT. The WITH clause allows you to name the results of a query for use within the following SELECT. It looks like this:

```
with pvals as (
     select firstname, lastname, gp * minutes as total_minutes,
            per, 67*va/(gp * minutes) as denom,
            round(per - 67*va/(gp * minutes),1) as prl
     from more_player_stats
) select * from pvals where prl not in (10.5,10.6,11.0,11.5)
    order by per;
```

Exercise #2. Building the Player Table. (1 point) The players table will contain (relatively) constant information about each player, but will exclude statistics (and team affiliation) that can change markedly between seasons. Here we will create that table, using the results of the previous exercise.

Create a table *players* with the following attributes:

- A serial primary key *id*
- *firstname* and *lastname* (text).
- *position*, a string as defined in the previous exercise.
- *age*, *height* (in inches), *weight* (in pounds), as text.
- *college*, *country*, *draft_year*, *draft_round*, *draft_number*, as text.

Based on the previous exercise, we can get this data from `player_bios`. Now populate the `players` table by using `INSERT` with a `SELECT` as the data source.

Next, we will make the table of player statistics for the 2015-16 NBA season. For this, the *player* and *team* columns will be foreign keys into the `players` and `teams` tables, respectively. We define a foreign key using a `CREATE TABLE` command by putting `REFERENCES <table>` at the end of the column specification, where `<table>` is replaced by the corresponding table name.

Because we will need to combine data from several tables to do this task, we will need to use a *join*. Here's an example that shows how to use joins and aliases (and the `WITH` operator) to check that the last names of the players match by id:

```
with lasts as (
     select more_player_stats.lastname as a,
            player_stats.lastname as b,
            player_bios.lastname as c
     from more_player_stats
     join player_stats on more_player_stats.id = player_stats.id
     join player_bios on player_bios.id = more_player_stats.id
) select lasts.a, lasts.b, lasts.c from lasts where a != b or a != c or b != c;
```

(The `WITH` makes it possible to use aliases for the column names in the `WHERE` clause; you will not need that for the task in this exercise.)

Create the table `stats2015_16` table, including *player* and *team* (with value from from `player_stats`), the columns (gp, minutes, fgm, fga, fgpct, tpm, tpa, tppct, ftm, fta, ftpct, oreb, dreb, reb, ast, tov, stl, blk, pf, dd2, td3, pts, plusminus) from `player_stats` and the columns (tspct, astr, tovr, usg, orr, drr, rebr, per, va, ewa) from `more_player_stats`. Don't forget a primary key for this table.

This requires a slightly tedious joined query as the input for an `INSERT` command. Use simple aliases like *p* for `players`, *s* for `player_stats`, and *m* for `more_player_stats` to make things a bit easier. You can also use some text from the file `player-stats.sql` to avoid typing some of the column definitions. (Your editor can also be your friend here.)

Exercise #3. A Few Queries. (2 points) Now that we've refactored the data, let's look at what it tells us. Using the `stats2015_16` table, answer the following questions. Submit the SQL queries you used and **and** their output (submit only the first 5–10 rows if the output is large).

1. List the names (and average points per game) of the top 25 scorers, in descending order of average points per game, restricting your attention to those players who averaged at least eight minutes per game and who played at least 10 games. (The `LIMIT` modifier to `SELECT` is handy here.)

2. Rank the teams according to average points per game over all players on the team who played at least 24 minutes per game on average.

3. List the five players (last name, first name) with the highest minutes per game but a negative "estimated wins added" to the team (column *ewa*).

4. List all players by position and in descending order of average points per game and compare show the average points per game of players with the same position in each row. Do the same while restricting to players who played at least 24 minutes per game on average. (This requires an `OVER` clause using a `PARTITION BY` window function.)

5. List players who had, over the season, field-goal percentage at least 50, free-throw percentage at least 90, *and* three-point percentage at least 45.

6. The turnover ratio column is defined in the `README` file as Turnover Ratio = (Turnover $\times$ 100) divided by [(FGA + (FTA $\times$ 0.44) + Assists + Turnovers]. Craft a select using `stats2015_16` to compute this directly and show that the computed values are essentially the same as those stored in the column *tovr*.