CompSci 316 Fall 2016: Homework #2

100 points (8.75% of course grade) + 10 points extra credit
Assigned: Tuesday, September 20
Due: Tuesday, October 4

This homework should be done in parts as soon as relevant topics are covered in lectures. If you wait until the last minute, you might be overwhelmed.

For Problems 1, 2, 4, 6, and 7, you will need to use Gradiance. Access Gradiance via the "Gradiance" link on the course website. There is no need to turn in anything else for these problems; your scores will be tracked automatically.

For other problems, you will need to turn in the required files electronically. Please read the "Help → Submitting Non-Gradiance Work" section of the course website for instructions. When submitting your work, make sure you select the correct course and homework. Multiple submissions are okay, but please upload *all* required files in each resubmission.

Problems 3, 5, and X2 must be completed on your course VM. Before you start, make sure you refresh your VM, by logging into your VM and issuing the following command:

```
/opt/dbcourse/sync.sh
```

Problem 1 (4 points)

Complete the Gradiance homework titled "Homework 2.1 (Relational Design Theory: MVD)."

Problem 2 (12 points)

Complete the Gradiance homework titled "Homework 2.2 (SQL Querying)."

Problem 3 (36 points)

Consider again the beer drinker's database from Homework #1. Key columns are underlined.

Drinker(<u>*name*</u>, *address*)
Bar(<u>*name*</u>, *address*)
Beer(<u>*name*</u>, *brewer*)
Frequents(<u>*drinker*</u>, <u>*bar*</u>, *times_a_week*)
Likes(<u>*drinker*</u>, <u>*beer*</u>)
Serves(<u>*bar*</u>, <u>*beer*</u>, *price*)

Write the following queries in SQL. To set up the sample database called `beers` (even if you have set it up previously, you should repeat this process to refresh it), issue this command in your VM shell:

```
/opt/dbcourse/examples/db-beers/setup.sh
```

Then, type "`psql beers`" to run PostgreSQL's interpreter. For additional tips, see "Help → PostgreSQL Tips" on the course website.

When you run `psql`, as soon as you get a working solution, record the query in a plain-text file named `3-query.sql` (use "`--`" to add comments in the file to indicate which problems they correspond to). When you are done with all queries, run

```
	psql beers -af 3-queries.sql &> 3-answers.txt
```

to verify that everything works and to generate the final answers. Submit the files `3-queries.sql` and `3-answers.txt` electronically. If you cannot get a query to parse correctly or return the right answer, include your best attempt and explain it in comments, to earn possible partial credit.

Note: In order to ensure that your queries work in all cases, consider testing them on different database instances. The example instance we provide may not reveal subtle errors, e.g., failing to return a drinker who does not frequent any bar for (f). Feel free to modify the given database for testing, but make sure that you generate `3-answers.txt` for submission from the given, unmodified database.

(a) Find names of beers served at *James Joyce Pub*.
(b) Find names and addresses of bars that serve some beer for less than $2.25. Don't return duplicates.
(c) Find names of bars serving some beer *Amy* likes for no more than $2.50. Don't return duplicates.
(d) Find pairs of drinkers who like the same beer. (Just list the drinker names, not the beer. Don't list (*drinkerA*, *drinkerA*). If you list (*drinkerA*, *drinkerB*) in the answer, don't list (*drinkerB*, *drinkerA*) again.)
(e) Find names of all drinkers who like *Dixie* but frequent none of the bars serving it.
(f) For each drinker, show the bar that he or she frequents the most, along with the number of visits per week. If multiple bars tie for the most frequented by the drinker, list all of them. You need to list every drinker, even if this drinker does not frequent any bar (show `NULL` for bar and times per week in this case).
(g) Find names of all drinkers who frequent ***only*** those bars that serve some beers they like.
(h) Find names of all drinkers who frequent ***every*** bar that serves some beers they like.
(i) For each bar, find the total number of drinkers who frequent it, as well as the average price of beers it serves. Sort the output by the number of drinkers (in descending order). You need to list every bar, even if it is not frequented by anyone (show 0 as the total number of drinkers in this case) or serves no beers (show `NULL` as average price in this case).

Problem 4 (8 points)

Complete the Gradiance homework titled "Homework 2.4 (SQL Constraints)."

Problem 5 (30 points)

Recall Problem 4 of Homework #1. Here is a relational design:

Species (<u>*name*</u>, *attack*, *defense*, *stamina*, *evolves_from*)
Pokemon (<u>*id*</u>, *name*, *level*, *attack*, *defense*, *stamina*, *species*, *quick_move*, *charged_move*, *trainer_id*, *favorite*)
Move (<u>*name*</u>, *damage*, *cooldown*, *type*, *energy*)
Trainer (<u>*id*</u>, *nickname*, *exp*)

Your job is to complete and test an implementation of the above schema design for a SQL database. To get started, copy the template to your working directory:

```
cp /opt/dbcourse/assignments/hw2/5-create.sql .
```

(Don't miss the trailing dot, which represents the current directory.)

Use the following command to run the file with a fresh new database called `cars`:

```
dropdb pokemon; createdb pokemon; psql pokemon -af 5-create.sql
```

The file `5-create.sql` is actually incomplete. You need to edit it to fill in the missing parts. Use simple SQL constructs as much as possible, and only those supported by PostgreSQL. Note that:

- PostgreSQL does not allow subqueries in `CHECK`.
- PostgreSQL does not support `CREATE ASSERTION`.
- You might need some SQL math functions. For syntax and examples, see http://www.postgresql.org/docs/9.5/static/functions-math.html.
- PostgreSQL's implementation of triggers deviates slightly from the standard. In particular, you will need to define a "UDF" (user-defined function) to execute as the trigger body. For syntax and examples, see http://www.postgresql.org/docs/9.5/static/plpgsql-trigger.html.

Your job involves the following tasks (note that some of the constraints below are new from Homework #1). You may modify the `CREATE` statements in the file as you see fit, but do not introduce new columns, tables, views, or triggers unless instructed otherwise.

(a) Enforce key and foreign key constraints implied by the description in Homework #1.
(b) Enforce that trainers have unique nick names (a new constraint).
(c) Enforce that all species base attack, defense, and stamina values are greater than 0, and that all individual Pokemon attack, defense, and stamina values are between 0 and 15 (inclusive).
(d) Enforce that if a Pokemon is not owned by a trainer (i.e., *Pokemon.trainer* is `NULL`), then it cannot be favorited/un-favorited (i.e., *Pokemon.favorite* must be `NULL` as well). Furthermore, if a Pokemon is owned by a trainer, then *Pokemon.favorite* must be either "t" or "f" (*true* or *false*, respectively).
(e) Using triggers, enforce that *Pokemon.quick_move* and *Pokemon.charged_move* indeed refer to moves of the correct types, respectively.
(f) Write an `INSERT` statement that fails because a Pokemon refers to a non-existent move.
(g) Write an `INSERT` statement that fails because of violating (b).
(h) Write two `INSERT` statements that fail because of violating constraints in (c) on *Species* and *Pokemon*, respectively.
(i) Write two `UPDATE` statements that fail because of violating the two cases under (d), respectively.
(j) Write an `INSERT` statement that fails because of violating (e).
(k) Write an `UPDATE Move` statement that fails because of violating (e).
(l) Define a view that lists, for each Pokemon, its combat power (CP), defined as follows:
   - a = base attack (*Species.attack*) + individual attack (*Pokemon.attack*);
   - d = base defense (*Species.defense*) + individual defense (*Pokemon.defense*);
   - s = base stamina (*Species.stamina*) + individual stamina (*Pokemon.stamina*);
   - $m = (0.095)^2 \times$ *Pokemon.level*;
   - $\text{CP} = \max\left(10, \left\lfloor 0.1ma\sqrt{ds} \right\rfloor\right)$. Here, $\lfloor\cdot\rfloor$ is the floor function.

When you are all done, run

```
       dropdb pokemon; createdb pokemon; psql pokemon -af 5-create.sql &> 5-out.txt
```

to verify that everything works and to generate the final answers. Submit the files `5-create.sql` and `5-out.txt` electronically. If you cannot get a statement to work correctly, include your best attempt and explain it in comments, to earn possible partial credit.

Problem 6 (4 points)

Complete the Gradiance homework titled "Homework 2.6 (SQL Recursion)."

Problem 7 (6 points)

Complete the Gradiance homework titled "Homework 2.7 (SQL Triggers, Views)."

Extra Credit Problem X1 (5 points)

Write a program to implement the "chase" procedure. Your program should read from the standard input the following specification (for example):

```
A, B, C, D
fd: A, B, C: D
fd: D: A
mvd: A, B: C
chase: fd: A: C, D
```

The first line declares the list of attributes in the relation of interest. The attribute names are strings separated by commas; the names are unique.

Next, there may be any number of lines specifying the given dependencies. Each line specifies either a functional dependency (`fd:`) or a multivalued dependency (`mvd:`). The left- and right-hand sides of the dependency are separated by a colon, and both sides must specify valid attributes declared by the first line, separated by commas.

The last line of the input, starting with `chase:`, specifies the target dependency that we want to prove or disprove, in the same format as that of the given dependencies.

Your program should output either a proof of the target dependency or a counterexample showing that the target dependency does not hold. The output format is flexible but should be text that is human-readable.

You can use any programming language. Submit your code and a plain-text `x1-README.txt` file that explains how to run (and compile, if necessary) your program.

Extra Credit Problem X2 (5 points)

Continuing with Problem 5, further modify the SQL file to carry out the following tasks:

(a) Using triggers, enforce that no species can evolve (directly or indirectly) into itself.
(b) Write an `INSERT` statement that fails for violating (a).
(c) Write an `UPDATE` statement that fails for violating (a).

Create files `x1-create.sql` and `x1-out.txt` (analogously to `5-create.sql` and `5-out.txt` in Problem 5) and submit them electronically.