CompSci 316 Fall 2017: Homework #2

100 points (8.75% of course grade) + 10 points extra credit
Assigned: Tuesday, September 19
Due: Tuesday, October 3

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.

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-queries.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 the names of bars serving *Corona*.
(b) Find the beers served at those bars that Ben frequents only once a week. (Just list the beers, not the bars.)
(c) Find names and addresses of drinkers who like some beer served at *Talk of the Town*.
(d) Find pairs of drinkers who live at the same address. (Just list the drinker names. Don't list (*drinkerA*, *drinkerA*). If you list (*drinkerA*, *drinkerB*) in the answer, don't list (*drinkerB*, *drinkerA*) again.)
(e) Find beers that are liked by nobody.
(f) For each beer, find the bar that serves it at the highest price. (Your output should be a list of (*beer*, *bar*) pairs. If multiple bars tie for the most expensive for a beer, list them all as different pairs.)
(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) Suppose that each time a drinker visits a bar (according to the frequency in *Frequents*), he or she always buys one glass of each beer served at this bar that he or she likes; if no such beer exists, he or she will not buy anything. Calculate, for each drinker, the amount of money he or she spends per week at the bars, and list the (*drinker*, *amount*) pairs in decreasing order of *amount*. Be sure to handle the case when somebody doesn't buy anything at any bar—you need to show an amount of 0.
Note: You might find SQL conditional expressions useful. For syntax and examples, see https://www.postgresql.org/docs/9.5/static/functions-conditional.html.

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 (slightly modified):

People (*id*, *name*, *pet*, *wand_core*); *Teacher* (*id*); *Student* (*id*, *year*, *house_name*);
House (*name*, *teacher_id*); *Deed* (*id*, *student_id*, *datetime*, *points*, *description*);
Subject (*name*); *Offering* (*subject_name*, *year*, *teacher_id*);
Grade (*student_id*, *subject_name*, *year*, *grade*); *FavoriteSubject* (*student_id*, *subject_name*).

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.)

Several things are worth noting: *Student.year*, *Grade.year*, and *Offering.year* all refer to school years; e.g., 1996 represents the 1996-1997 school year. We may use the value `NULL` for unknown information (such as Severus Snape's pet). *Deed.id* has type `SERIAL`, which means its value is automatically and serially generated to ensure uniqueness; therefore, you should omit value for *id* when inserting to *Deed* (see `5-create.sql` for examples of how to do that).

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

```
dropdb potter; createdb potter; psql potter -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`.
- In PostgreSQL, date-time values (of type `TIMESTAMP`) can be represented by string literals of format, e.g., `'2000-01-01 12:30:00'`. These values can be compared using `<`, `<=`, `=`, etc., with expected semantics.
- 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 no teach can teach two subjects or more in the same school year (a new constraint).
(c) Enforce that grades can only be "O" (outstanding), "E" (exceeds expectations), "A" (acceptable), "P" (poor), "D" (dreadful), "T" (troll), or the special value `NULL` (for grades not yet posted).
(d) Enforce that if a deed has a description that begins with "Arriving late," at least 10 House points must be deducted.
(e) Using a trigger, enforce that if a student ever receives a "D" or "T" for a subject, the student cannot take the same subject again. (Otherwise students may repeat a subject.)
(f) Using triggers, enforce that a person cannot be both student and teacher at the same time.
(g) Write an `INSERT` statement that fails because a student grade record is entered for a non-existent offering.
(h) Write an `INSERT` statement that fails for violating (b).
(i) Write an `UPDATE` statement that fails for violating (c).
(j) Write an `INSERT` statements that fail for violating (d).
(k) Write an `INSERT` statement that fails for violating (e). Then write another `UPDATE` statement that fails also for violating (e).

(l) Write an `INSERT` statement that fails for violating (f). Then write another `UPDATE` statement that fails also for violating (f).

(m) Define a view that lists, for each House, the total number of points accumulated by the House during the school year 1991-1992 (which started on September 1, 1991 and ended on June 30, 1992). Note that your view should list all Houses, even if a House didn't have any points earned or deducted during this period (in which case the total should be 0) or there were more points deducted than earned (in which case the total should be negative).

When you are all done, run

```
dropdb potter; createdb potter; psql potter -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 (10 points)

Write a program to test if a relation is in 4NF. If not, decompose it automatically until all resulting relations are in 4NF. 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
```

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.

Your program should output a sequence of decomposition steps that lead to a final schema where all tables are in 4NF. For each step, at least show the 4NF violation that you the step is fixing. 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.