CompSci 316 Spring 2017: Homework #2

100 points (8.75% of course grade) + 10 points extra credit
Assigned: Monday, February 6
Due: Friday, February 17 (Problem 6 and Extra credit problem X1 due on Thursday, 02/23)

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, 3, 5, and 6, 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. The Gradiance assignments will be released right after the corresponding lectures.

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 2 and 4 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 (12 points)

Complete the Gradiance homework titled "Homework 2.1 (SQL Querying)."

Problem 2 (36 points)

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

Drinker(*name*, *address*)
Bar(*name*, *address*)
Beer(*name*, *brewer*)
Frequents(*drinker*, *bar*, *times_a_week*)
Likes(*drinker*, *beer*)
Serves(*bar*, *beer*, *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 `2-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 2-queries.sql &> 2-answers.txt
```

to verify that everything works and to generate the final answers. Submit the files 2-queries.sql and 2-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 2-answers.txt for submission from the given, unmodified database.

(a) Find names of beers that *Dan* likes.
(b) Find names and addresses of bars that serve some beer for more than $3.00 each. Don't return duplicates.
(c) Find names of bars serving some beer of price > $2.50 that *Eve* does not like. Don't return duplicates.
(d) Find names of pairs of drinkers who frequent some bar the same number of times a week. (Just list the drinker names, not the bar. 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 frequent *Talk of the Town* pub and like all beers served there.
(f) For each drinker, find the beers that they like and that are served by at least two bars they frequent, along with the number of such bars for each beer.
(g) Find names of all drinkers who frequent *only* those bars that serve only beers they like.
(h) Find names of all drinkers who frequent *only* those bars that serves some beers they like.
(i) For each beer, find the total number of drinkers who like it, and the average price of this beer served in all the bars (do not include the bars that do not serve this beer in calculating the average). Sort the output first by the number of drinkers who like the beer (in descending order), and then in ascending order of the average price. You need to list every beer, even if it is not liked by any drinker (show 0 as the total number of drinkers in this case) or is not served by any bar (show NULL as average price in this case).

Problem 3 (12 points)

Complete the Gradiance homework titled "Homework 2.3 (SQL Constraints and NULLs)."

Problem 4 (30 points)

Recall Problem 4 of Homework #1. Here is a relational design for a simplified problem (we have omitted the "Artists", "Groups", "Likes", and a few attributes for simplicity).

Artwork (<u>id</u>, title, is_for_sale, quoted_price, is_photo, is_available, no_copies_sold, camera)

- Is_for_sale, is_photo, and is_available take "f" (false/no) or "t" (true/yes) value

PhotoCopy (<u>copy_id</u>, photo_id)
Customer (<u>id</u>, name, total_spent, date_of_last_purchase)

UniqueArtPurchasedBy(unique_art_id, customer_id, pdate, price)
PhotographsPurchasedBy(photo_id, copy_id, customer_id, pdate, price)

- Assume all dates are strings of the form 20170105 for Jan 05, 2017 etc.

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 (see the note in red below):

```
cp /opt/dbcourse/assignments/hw2/4-create-template.sql .
```

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

NOTE: The 4-create-template.sql file is available at
http://www.cs.duke.edu/courses/spring17/compsci316/assignments/4-create-template.sql

It is also available in the VM if you run sync.sh.

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

```
dropdb galleries; createdb galleries; psql galleries -af 4-create.sql
```

The file 4-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. For any part below where you need subqueries to enforce a constraint, you can use a trigger.
- 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 if a unique art or a photograph is sold, then it is for sale.
(c) Enforce that all quoted prices are between 500 and 500,000 (inclusive).
(d) Enforce that if the artwork is a photograph, then is_available is NULL, and if it is a unique art, then no_copies and camera are NULL.
(e) Enforce that if a customer purchases a photograph/unique art on a date D, then his/her "date_of_last_purchase" is >= D.
(f) Enforce that "no_copies_sold" of a photograph matches with the total number of copies sold for that photograph (from PhotographsPurchasedBy relation).
(g) Write an INSERT statement that fails because a PhotoCopy refers to a non-existing photograph.
(h) Write an INSERT statement that fails because of violating (b).

(i) Write an UPDATE statement that fails because of violating (c).
(j) Write two INSERT statements that fail because of violating constraints in (d) on UniqueArt and Photograph respectively.
(k) Write an UPDATE statements for Photograph that fail because of violating (e).
(l) Enforce using TRIGGER that when a customer purchases a new unique art or photograph, his/her last day of purchase is updated. Further, if it is a photograph, the number of copies sold is updated too.
(m) Define a view that lists, for each photograph, its total number of copies sold (num_copies), the average price (as avg_price, which is NULL if no copies are sold yet), and the standard deviation of the prices (as stdev, which is also NULL if no copies are sold yet). Recall that the standard deviation of x1, x2, …, xn is $\sqrt{(1/n)(\sum_{i=1 \text{ to } n}(xi - x_{avg})^2)}$, where x_{avg} is the mean of x1, …, xn.

When you are all done, run

```
dropdb galleries; createdb galleries; psql galleries -af 4-create.sql &> 4-out.txt
```

to verify that everything works and to generate the final answers. Submit the files 4-create.sql and 4-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 5 (6 points)

Complete the Gradiance homework titled "Homework 2.5 (SQL Triggers, Views)."

Problem 6 (4 points)

(due on Thursday, 02/23)

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

Extra Credit Problem X1 (10 points)

(due on Thursday, 02/23)

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.