CompSci 316 Fall 2018: Homework #2

100 points (8.75% of course grade) + 10 points extra credit
Assigned: Tuesday, September 18
Due: Tuesday, October 2 (Wednesday, October 3 for Problem 6; Tuesday, October 16 for Problems 5 & 7)

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, and follow the submission instructions for each problem carefully.

Problems 3, 5, and X1 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(*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.

You should check that your queries return the intended answers on our sample database. For grading, however, your answers may be tested on other databases with the same schema but different contents, so your queries need to be correct i***n general*** to receive full credits.

Unless otherwise noted, your result should contain no duplicate rows.

As soon as you get a working solution for one part of this problem, say (a), record your query in a plain-text file named `a-query.sql` (replace "`a`" with "`b`", "`c`", and other parts as appropriate). Submit all query files. 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.

(a) Find the names of beers that *James Joyce Pub* serves.
(b) Find the names of drinkers who frequent any bar serving *Corona*.
(c) Find the names and addresses of bars that serve *Corona* for no more than $3.00.
(d) Find the names of drinkers who frequent both *James Joyce Pub* and *Satisfaction*.
(e) Find the names of bars frequented by either *Ben* or *Dan*, but not both.
(f) Find all (*beer*, *bar*) pairs where *beer* is served exclusively at *bar*; no other bar serves the same beer.
(g) For each bar, find the drinker who frequents it the greatest number of times. Your output should be a list of (*bar*, *drinker*) pairs. If multiple drinkers tie for the most frequent visitor to a bar, list them all as separate pairs.
(h) Find names of all drinkers who frequent ***only*** those bars that serve some beers they like.
(i) Find names of all drinkers who frequent ***only*** those bars that serve ***only*** beers they like.
(j) Find all (*bar1*, *bar2*) pairs where the set of beer served at *bar1* is a proper subset of those served at *bar2*; i.e., *bar2* serves every beer that *bar1* serves and plus some more.
(k) 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 bar, the total amount of money it expects to get per week (over all drinkers in our database). List the (*bar*, *amount*) pairs in decreasing order of *amount*; in case of ties, sort alphabetically by *bar*. Be sure to handle the case when a bar doesn't expect to sell any beer to our drinkers—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/current/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 (slightly modified) relational design for the Pie-in-the-Sky Security Corp. (PITS):

Person (*ssn*, *name*, *address*); *Broker* (*ssn*, *phone, manager*);
Account (*aid*, *brokerssn*); *Owns* (*ssn*, *aid*);
Stock (*sym*, *price*); *Holds* (*aid*, *sym*, *amount*);
Trade (*aid*, *seq*, *type*, *timestamp*, *sym*, *shares*, *price*).

It is worth noting that *Broker.manager* can be `NULL`, because a boss would not have a manager.

Your job is to complete and test an implementation of the above schema design for a SQL database. To get started, copy the template files to a subdirectory in your workspace and check that everything is in order (you may replace `~/shared/hw2-5/` below with any other appropriate path):

```
mkdir -p ~/shared/hw2-5
cp -r /opt/dbcourse/assignments/hw2-5/. ~/shared/hw2-5/
cd ~/shared/hw2-5/
ls
```

You should see a few `.sql` files. The file `create.sql` contains SQL statements to create the database schema. It is actually incomplete. Your first job is to edit this file to enforce a number of constraints below (note that some of the constraints 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. 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 from the standard. In particular, you will need to define a "UDF" (user-defined function) to execute as the trigger body. In order to complete this problem, you will need to consult the documentation at http://www.postgresql.org/docs/current/static/plpgsql-trigger.html. Particularly useful are special variables such as `NEW`, `TG_OP`, `TG_TABLE_NAME`, as well as the `RAISE EXCEPTION` statement.

Modify `create.sql` to enforce the following constraints. Follow the comments in the file for instructions on where your edits should go.

(a) Enforce key and foreign key constraints implied by the description in Homework #1.
(b) Enforce that the type of a trade is either `buy` or `sell`.
(c) There is no room for mistakes at PITS. Since PITS records only completed trades, enforce that the *Trade* table is append-only (i.e., no `DELETE` or `UPDATE` is allowed) using a trigger. Further enforce that within each account, trades must be recorded sequentially over time: i.e., compared with old trades in the same account, a new trade must have a *seq* that is strictly larger, and a *timestamp* that is no less than the old values.
(d) Using triggers, enforce that brokers cannot own accounts, either by themselves or jointly with others.
(e) Define a view that returns all pairs of brokers (*broker1*, *broker2*) where *broker1* reports directly or indirectly to *broker2* (e.g., *broker1* reports to some other broker who in turn reports to *broker2*). This view helps implement a trigger to ensure that no broker reports (directly or indirectly) to him/herself.

To test `create.sql`, use the following commands in your VM shall to (re)create a database called `pits`:, and to populate it with some initial data:

```
dropdb pits; createdb pits; psql pits -af create.sql
psql pits -af load.sql
```

Your next job is to write a series of SQL modification statement to test the constraints you implemented, starting with the initial data provided in `load.sql` (do not modify this file). You can use "`psql pits`" to run PostgreSQL's interpreter interactively to experiment with your modification statements, but as soon as you get a working solution each part of this problem, say (f), record your statement in a plain-text file named `f.sql` (replace "`f`" with "`g`", "`h`", and other parts as appropriate).

(f) Write an `INSERT` statement on *Account* that fails because the account's *brokerssn* doesn't refer to an existing broker.
(g) Write an `UPDATE` statement on *Owns* that fails because it attempts to set *aid* to a non-existent account.
(h) Write a `DELETE` statement on *Broker* that fails because the broker being deleted is managing somebody else.
(i) Write an `INSERT` statement that fails for violating (b).
(j) Write a `DELETE` statement that fails for violating (c).
(k) Write an `INSERT` statement that fails for violating (c).
(l) Write an `UPDATE` statement on *Broker* that fails for violating (d).
(m) Write an `UPDATE` statement on *Broker* that fails for violating (e).

Submit your `create.sql` as well as `f.sql` through `m.sql` electronically.

Problem 6 (6 points)

Complete the Gradiance homework titled "Homework 2.6 (SQL Triggers, Views)."

Problem 7 (4 points)

Complete the Gradiance homework titled "Homework 2.7 (SQL Recursion)."

Extra Credit Problem X1 (10 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
A, B, C -> D
D -> A
A, B ->> C
chase: 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 (`->`) or a multivalued dependency (`->>`). The left- and right-hand sides of the dependency (separated by `->` or `->>`) 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 `README.txt` file that explains how to run (and compile, if necessary) your program.