CompSci 316 Fall 2019: Homework 2

100 points (6.25% of course grade) + 10 points extra credit
Assigned: Wednesday, September 11, 2019
Due: Monday, September 30, 2019

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.

You must 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 1, 2, and X1 should be completed on a virtual machine (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 (55 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. An autograder will run automatically on your submission and give you a report of what it finds—the `db0` test database it uses is identical to the sample one you have, while `db1` is a hidden test database. You can use the autograder report to help you debug your queries and resubmit. 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. Remember to ***(re)submit all files*** in your final submission.

(a) Find names of all bars that *Eve* frequents.
(b) Find names and addresses of drinkers who frequent *Satisfaction* more than once per week.
(c) Find names of bars serving some beer *Amy* likes for strictly less than $2.75.
(d) Find names of all drinkers who like *Corona* but not *Budweiser*.
(e) For each beer that *Eve* likes, find the names of bars that serve it at the highest price. Format your output as list of (beer, bar) pairs. If some beer liked by *Eve* is not served anywhere, do not output that beer.
(f) Find the ***cheapest and second cheapest*** drinks in town and where they are served; i.e., return the *serves* rows with the lowest and second lowest prices. If there are multiple *serves* rows with these two lowest prices, return all of them. (For the example, suppose the two lowest prices are $0.01 and $0.02, and there are 3 *serves* rows with $0.01 and 1 *serves* row with $0.02; then, the query should return a total of 4 rows.)
(g) For each drinker, find names of bars frequented by the drinker that serve none of the beers liked by the drinker. Format your output as list of (drinker, bar) 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 ***every*** bar that serves some beers they like.
(j) For each beer, find the number of drinkers who like it, as well as its average price as served by bars. Sort the output by the number of drinkers who like the beer (the most popular beer should be listed first). In the case of ties, sort by beer name (ascending). If a beer is not served anywhere, its average price should be listed as `NULL`.
(k) Find, for each drinker, the bar(s) he or she frequents the most. The output should be list of (drinker, bar) pairs. If some drinker *D* does not frequent any bar, you should still output (*D*, `NULL`).

Problem 2 (45 points)

Recall Problem 2 of Homework 1. Here is a relational design (with some simplification).

Automobile (*VIN*, *make*, *model*, *year*, *color*, *mileage*, *body_style*, *sellerID*, *price*)
RegisteredUser (*id*, *name*, *email*); *Seller* (*id*, *phone*); *Dealer* (*id*, *address*)
Review (*id*, *make*, *model*, *year*, *authorID*, *text, date*)

Note that every seller is a user, and every dealer is a seller; a non-dealer seller is an individual seller.

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/hw2-2/` below with any other appropriate path):

```
mkdir -p ~/shared/hw2-2
cp -r /opt/dbcourse/assignments/hw2-2/. ~/shared/hw2-2/
cd ~/shared/hw2-2/
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 accomplish some tasks. 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`.
- You might need some date manipulation functions. For example, to convert an integer-valued `year` value to a date object for the beginning of that year, use
  `DATE(CAST(year AS VARCHAR) || '-01-01'))`
  Here, "`||`" is the string concatenation operator in SQL, and SQL knows how to convert a string of the format "YYYY-MM-DD" into a date object. You can compare dates using `<`, `<=`, `=`, etc. For additional help, see:
  http://www.postgresql.org/docs/current/static/datatype-datetime.html
  http://www.postgresql.org/docs/current/static/functions-datetime.html
- 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 accomplish the following tasks. 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 users have unique emails.
(c) Enforce that the body style is one of "convertible," "coupe," "sedan," "suv," "van," and "other."
(d) Enforce that a user can review a particular make-model-year combination at most once.
(e) Enforce that the date of a review must not be earlier than one year before the year of the model being reviewed.
(f) Using triggers, enforce that we cannot have a user selling an automobile and writing a review for the same make, model, and year.
(g) Define a view that lists, for each make-model-year triple, the total number of reviews, the total number of automobiles for sale, and the average price.

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

```
dropdb cars; createdb cars; psql cars -af create.sql
psql cars -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 cars`" 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 (h), record your statement in a plain-text file named `h.sql` (replace "`h`" with "`i`", "`j`", and other parts as appropriate).

(h) Write an `INSERT` statement that fails because a non-seller user attempts to sell an automobile.
(i) Write an `INSERT` statement that fails because of violating (b).
(j) Write an `INSERT` statement that fails because of violating (c).
(k) Write an `INSERT` statement that fails because of violating (d).
(l) Write an `INSERT` statement that fails because of violating (e).

(m) Write an `INSERT INTO Review` statement that fails because of violating (f).
(n) Write an `INSERT INTO Automobile` statement that fails because of violating (f).

Submit your `create.sql` as well as `h.sql` through `n.sql` electronically.

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.