CompSci 316: Intro to Databases

HW-3: ER Diagrams and SQL

Score: 100 (+ 15 extra credit)

Release Date: Thu, 01/30/2020 (Q1, Q2)

Tue, 02/04/2020 (Q3, Q4, Q5)

Due Date: Q1-Q3: Tue, 02/11/2020, 11:59 pm
Q4-Q5: Sat, 02/15/2020, 12:00 pm (= 12:00 noon)

Please check collaboration and late submission policy from the course website.

Before working on part 3,4,5, make sure you refresh your VM and database, by logging into your VM and issuing the following command:

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

1. Gradiance Questions (5 x 3 = 15 points)

(a) Solve **HW3 - Q1a (ER diagram)** on Gradiance.
(b) Solve **HW3 - Q1b (SQL Querying)** on Gradiance. (please attempt after SQL Querying is finished in class)

Multiple submissions are allowed and encouraged. Highest score before the deadline will be recorded.

2. ER Diagram (15 + 15 = 30 points)

The tennis court reservation system at a university needs your help to design a database! We need to store some basic information about the users, the tennis courts, and their relationships.

- A user is identified by the NetID. We also record the name of the user.
- We have three kinds of users: student, professor, and staff. Students additionally have a status recorded (such as freshman, sophomore, junior, senior, MS, PhD). There are departments in the university. A student can only belong to one department. A professor additionally has a record of the rank (assistant professor, associate, etc.). A professor can be affiliated with many departments. We also record the title of a staff; the staff do not have a separate department affiliation. A department is identified by its name.
- A location is identified by its address. A department may reside in multiple locations. The stadiums have unique names and can be at one location. A tennis court is identified by the name of the stadium that contains it, and its own court number.
- Reservations involve users, courts, and time slots. We keep track of reservation time slots by its date and hour. Each user can reserve at most one court given a fixed time slot (the university does not want its courts to be reserved without a proper usage!). Each court can only be reserved by at most one user given a fixed time slot (it does not want a conflict on the court either!).

(a) **Design an ER diagram for this database.** Very briefly explain the intuitive meaning of any entity and relationship sets. Do not forget to indicate keys and multiplicity of relationships, as well as ISA relationships and weak entity sets (if any), using appropriate notation.

(b) **Design a relational schema for the database.** Use a format similar to frequents(<u>drinker</u>, <u>bar</u>, times_a_week). Underline the key attributes. You can start by translating the E/R design, but may want to have some simplifications later.

Please also write any constraint that cannot be captured in your ER diagram or relational schema. For example, does your design capture all the constraints on reservations?

[https://www.draw.io is a handy tool to draw ER diagrams. You may find all necessary shapes in the tool box on the left. Look under “Entity Relation” and “UML” drawers]

If you feel that some aspects of the above description are unclear (some descriptions are intentionally vague to encourage your creative thinking), feel free to make additional, reasonable assumptions about the data. State any additional assumptions clearly in your answer. Also, keep in mind that there is no single “correct” design; if you think you are making a non-obvious design decision, please explain it briefly. **Submit your ER diagram and relational schema together as a PDF file on Gradescope.**

3. SQL Queries (5 x 6 = 30 points)

(please work on the other parts first and this should be done after SQL queries with subqueries, group by, having, aggregates are covered in class)

Description

Consider a database “beers” containing information about bars, beers, and drinkers.

drinker(name, address)
bar(name, address)
beer(name, brewer)
frequents(drinker, bar, times_a_week)
likes(drinker, beer)
serves(bar, beer, price)

Subqueries, group-by, having, aggregates are allowed.

Again, make sure you sync and setup the database in order to work on your own VM. It is necessary on your own vcm.

Log into your VM and issue the following command:

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

Questions

a. For each beer that Dan likes, find the names of bars that serve it at the highest price. Format your output as list of (beer, bar) pairs (x) [Output columns: beer, bar]
b. Find drinker pairs (drinker1, drinker2) such that the beers liked by drinker1 is a subset of beers liked by drinker2. Format your output as list of (drinker1, drinker2) pairs (x) [Output columns: drinker1, drinker2]
c. Find names of all drinkers who frequent only bars that serve only beers they like. (x) [Output columns: name]
d. 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.(x) [Output columns: beer, cnt, avg]
e. 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).(x)[Output column: name, bar]

4. SQL Constraints (5 x 5 = 25 points)

(This question modifies the database so needs to be tested on your own VM)
(Please work on the other parts first and this should be done after SQL constraints are covered in class -- you can work on Q4e when views are covered.)

Here is a relational schema about cars, users, and reviews.

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

Assume the following:

- Every seller is a user, and every dealer is a seller; a non-dealer seller is an individual seller.
- Any registered user can post reviews about automobiles. Each review is about one particular model, make, and year (but not about a particular vehicle). We also would like to record the date of each review.

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/hw3/ below with any other appropriate path):

```
mkdir -p ~/shared/hw3
cp -r /opt/dbcourse/assignments/hw3/. ~/shared/hw3/
cd ~/shared/hw3/
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 dates 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 primary key and foreign key constraints implied by the description below:
   - *To post a listing in table Automobile, the user must be either a dealer or a seller.*
   - All underlined attributes in the relation design must be primary key
   - Every seller is a user, and every dealer is a seller; a non-dealer seller is an individual seller.

b) Enforce that the color is one of “blue”, “gray”, “black”, “white”, “red”, “green”, and “other”.

c) Enforce that the date of a review must not be earlier than the first day of the year of the model.

d) Using triggers, enforce that we cannot have a dealer or seller selling an automobile and writing a review for the same make, model, and year.

e) Define a view that lists, for each make-model pair, the average number of reviews and the average number of automobiles sold per year, and the average price per sale. Here “year” is the year for the make-model, like Honda-Civic as mentioned in both the Automobile and Review tables’ year attribute. [The schema should be *CarInfo(make, model, avg_reviews, avg_sales, avg_car_price)*]

For example, for Honda Civic, if for Honda-Civic-2012 there are 3 reviews, 2 cars sold, at price 10k and 20k, for Honda-Civic-2014 there are 2 reviews, 4 cars sold, at price 15k, 15k, 6k, 6k, and for Honda-Civic-2016 there are 4 reviews and zero cars sold, then the tuple for Honda Civic in the answer should be (Honda, Civic, 3, 2, 12000). Note that the total number of years may vary for different make and model pairs and the years only appearing in at least one of the Automobile or Review tables should be used to compute the average price or number of reviews.

It is ok to have some empty/null data in this view, since for example, some cars being reviewed are not selled and thus have a null/empty data entry at avg_price.

To test your solution, 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
```

5. EXTRA CREDIT (5 x 3 = 15 points)

(This question modifies the database so needs to be tested on your own VM)
(please work on the other parts first and this should be done after SQL constraints are covered in class)

For the schema and questions given in Q4,

a) Write an INSERT statement that fails because of violating Q4(a).
b) Write an INSERT statement that fails because of violating Q4(b).
c) Write an INSERT statement that fails because of violating Q4(c).
d) Write an INSERT statement to Review that fails because of violating Q4(d).
e) Write an INSERT statement to Automobile that fails because of violating Q4(d).

How to test your answer for Q3 on a small database

Find the online SQL tester from the course webpage and try your queries there. For example, pgweb (if you don't know how to connect the database, check this link: pgweb instruction). You can also use your own VM to test Q3. **Q4 and Q5 must be worked on your own VM.**

Submission Instructions

Submit your solutions on Gradescope.
Part 3: upload one file for each question. Name the files a.txt, b.txt etc. Each file is a plain text file and only contains the SQL query as the answer. Make sure the filename matches your answer.
Part 4: upload your modified create.sql
Part 5: upload one file for each question. Name the files a.txt, b.txt etc. Each file is a plain text file and only contains the SQL query as the answer. Make sure the filename matches your answer.

Your submission will be auto-graded. **But will eventually be manually reviewed**. You may use the auto-grader output as assistance. Multiple re-submissions are allowed.

If you received any help or collaborate with others, please submit **one more file as collaboration.txt** together with your solutions and put collaborators' names and types of collaborations there. You must mention any students' names who you discussed with (and it is good to mention the course staff's help as well). If you discussed with different students different problems, please include that information too for each problem. If you are unsure what to write, remember that providing more information is better than less information.