CompSci 316 Fall 2021: Homework 2

100 points (6.25% of course grade) + 10 points extra credit
Assigned: Thursday, September 9, 2021
Due: Tuesday, September 28, 2021

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 in 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 the names of all beers served at James Joyce Pub.
(b) Find the names and addresses of bars that serve Budweiser at a price higher than $2.20.
(c) Find the names of drinkers who like Corona and frequent James Joyce Pub at least twice a week.
(d) Among the drinkers who frequent The Edge, find the names of those who do not like Erdinger.
(e) Find the name of each drinker who likes at least two beers.
(f) Find the name of each drinker who frequents zero or one bar.
(g) For each bar that Ben frequents, find the most frequent visitor(s) to that bar. Format your output as a list of (bar, drinker) pairs. Note it is possible to have ties, i.e., multiple drinkers who frequent one such bar with the same highest times_a_week, in which case all of them should be included in the output.
(h) Find the name of each bar such that every beer it serves is liked by some drinker who frequents this bar.
(i) For each bar, find the drinkers who like every beer served at this bar. Format your output as a list of (bar, drinker) pairs.
(j) For each bar, find the total number of times (total_times) it is frequented over all drinkers, as well as the price range for the beers it serves (in the form of min_price and max_price). The output should be a list of (bar, total_times, min_price, max_price) tuples. Sort the output by total_times, such that the most popular bar is listed first. In the case of ties, sort by bar name (ascending). If a bar is not frequented by anybody, total_times should be NULL. If a bar serves no beer at all, both min_price and max_price should be NULLs.

   • Note that for the purpose for ORDER BY, PostgreSQL by default considers NULL to be greater than non-NULL values. To override this behavior, you can specify NULLS FIRST or NULLS LAST following the ASC or DESC keyword, e.g., DESC NULLS LAST.
(k) Find, for each beer, its lowest serving price and the bar(s) serving it at this price. The output should be list of (beer, price, bar) triples. If some beer B is not served anywhere, you should still output (B, NULL, NULL).

Problem 2 (45 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);
Subject (name);
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, inside your VM shell, copy the template files to a subdirectory in your workspace and check that everything is in order (you may substitute ~/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 in that 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 load.sql for examples of how to do that).

The file create.sql contains SQL statements to create the database schema. It is 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 find SQL’s CASE value expressions useful:
  https://www.postgresql.org/docs/current/functions-conditional.html
- 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. For additional help, see:
  http://www.postgresql.org/docs/current/datatype-datetime.html
  http://www.postgresql.org/docs/current/functions-datetime.html
- 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.
- 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. To complete this problem, you will need to consult the documentation at http://www.postgresql.org/docs/current/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. Comments in that file contain additional instructions on where your edits should go.

(a) Enforce key and foreign key constraints implied by the description in Homework #1.
(b) Enforce that no teacher can teach two subjects or more in the same school year (a new constraint).
(c) Enforce that the 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 are 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, the student may repeat a subject.)
(f) Using triggers, enforce that a person cannot be both student and teacher at the same time.
(g) 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).

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

```bash
  dropdb potter; createdb potter; psql potter -af create.sql
  psql potter -af load.sql
```

Your next job is to write a series of SQL modification statements to test the constraints you implemented,
starting with the initial data provided in load.sql (do not modify this file). You can use “psql potter”
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 student grade record is entered for a non-existent
offering.
(i) Write an INSERT statement that fails for violating (b).
(j) Write an INSERT statement that fails for violating (c).
(k) Write an INSERT statement that fails for violating (d).
(l) Write an INSERT statement that fails for violating (e).
(m) Write an UPDATE statement that fails for violating (e).
(n) Write an INSERT statement that fails for violating (f).
(o) Write an UPDATE statement that fails for violating (f).

Submit your create.sql as well as h.sql through o.sql electronically.

**Extra Credit Problem X1 (10 points)**

Write a program to implement the BCNF decomposition procedure. Your program should read from the
standard input the following specification (for example):

```plaintext
{A, B, C, D, E}
A -> C
C -> B, D
```

The first line declares the list of attributes in the relation of interest. The attribute names are strings separated
by commas; they are alphanumeric and unique. Next, there may be any number of lines specifying the given
functional dependencies. The left- and right-hand sides of the dependency (separated by ->) must specify
valid attributes declared by the first line, separated by commas.

If the relation is not in BCNF, your program should output a sequence of steps showing how to decompose
it into relations that are eventually in BCNF. Each decomposition step, on a line by itself, should show the
relation being decomposed, the BCNF violation used for decomposition, and then the decomposed relations.
Finally, the last line of the output should be the final list of decomposed relations, all in BCNF. For the
example above, your output might look as follows:
Decompose: \{A,B,C,D,E\} by \(A \rightarrow C\) into \{A, C\} and \{A, B, D, E\}
Decompose: \{A,B,D,E\} by \(A \rightarrow B, D\) into \{A, B, D\} and \{A, E\}
BCNF: \{A, C\}, \{A, B, D\}, \{A, E\}

Note that there may be multiple valid decompositions. Here is another possibility:

Decompose: \{A,B,C,D,E\} by \(A \rightarrow C, B, D\) into \{A, B, C, D\} and \{A, E\}
Decompose: \{A,B,C,D\} by \(C \rightarrow B, D\) into \{B, C, D\} and \{A, C\}
BCNF: \{A, C\}, \{B, C, D\}, \{A, E\}

Your program only needs to output one valid decomposition. If the original relation is already in BCNF, your
program should output a single line declaring that it is in BCNF. Make sure you follow the output format
(and keywords such as “Decompose:”, “by”, “BCNF:”, etc.) above.

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.

**Hint:** A critical part of the BCNF decomposition is to infer what functional dependencies hold on one of the
“narrower” relations resulting from the decomposition. For example, \{A, B, D, E\} is not in BCNF because of
violation \(A \rightarrow B, D\); this functional dependency was not one of the original ones given, but follows from them.