University of Toronto
CSC343, Fall 2022

Assignment 3

Due: Wednesday, November 30th at 4:00 pm sharp!

IMPORTANT: A summary of key clarifications for this assignment will be provided in an FAQ on Piazza. Check it regularly for updates. Both the FAQ and any Quercus announcements are required reading.

Learning Goals

By the end of this assignment you should be able to:

- identify tradeoffs that must be made when designing a schema in the relational model, and make reasonable choices,
- express a schema in SQL's Data Definition Language,
- identify limitations to the constraints that can be expressed using the DDL,
- appreciate scenarios where the rigidity of the relational model may force an awkward design,
- formally reason about functional dependencies, and
- apply functional dependency theory to database design.

Part 1: Informal Relational Design

In class, we are in the middle of learning about functional dependencies and how they are used to design relational schemas in a principled fashion. After that, we will learn how to use Entity-Relationship diagrams to model a domain and come up with a draft schema which can be normalized according to those principles. By the end of term you will be ready to put all of this together, but in the meanwhile, it is instructive to go through the process of designing a schema informally.

The domain

Suppose a new company is trying to break into the concert ticket business and you are designing the database back-end for their app. Below is the information that they want to be able to store for their first proof-of-concept version of the database. There is much more to be added later, such as credit card information, but this is not your responsibility.

- Concerts are booked into venues. A venue has a name (not unique), city, and street address.
- Every venue also has an owner, which could be a person or an organization or company. For now we are just storing the owner name as a string, and a single phone number associated with the owner. No two owners have the same phone number.
- Some people/organizations own multiple venues but all venues have a single owner.
- Every venue has at least 10 seats and there is no upper limit. Each seat has an identifier, such as "B37" (but it could be any string).
- Seats in a venue are organized into sections. This is the same organization for every concert in that venue. Every seat belongs to exactly one section. Each section has a name, such as "floor level 1" that is unique to that venue, but another venue might use the same section name.

- Seat names do not repeat within the same section in a venue. But two different sections may have seats with the same name.
- Every concert has a name, such as "Mariah Carey - Merry Christmas to all example", a date and time, and is in exactly one venue. All seats in the venue are available for every concert, i.e., we won't account for venues that have different configurations where some seats are closed off for an event.
- Concert names are not unique (for instance, it may be on for several nights). But a venue can only have one concert at a given time. (We won't worry about concert durations or end times.)
- Some seats are accessible to people with mobility issues
- The price of a ticket depends the concert and the section in which the seat is located in the venue.
- Users of the app have a unique username, and that's all we'll store about them for now. A user can purchase one or more tickets to any concert. When we record this, we also record the date and time of purchase.

Several features above are not realistic, for instance that every seat in a venue is available for every concert, but they simplify your assignment.

Define a schema

Your first task is to construct a relational schema for our domain, expressed in DDL. Write your schema in a file called `schema.ddl`.

As you know, there are many possible schemas that satisfy the description above. We aren't following a formal design process for Part 1, so instead follow as many of these general principles as you can when choosing among options:

1. Avoid redundancy.
2. Avoid designing your schema in such a way that there are attributes that can be null.
3. If a constraint given above in the domain description can be expressed without assertions or triggers, it should be enforced by your schema, unless you can articulate a good reason not to do so.
4. There may be additional constraints that make sense but were not specified in the domain description. You get to decide on whether to enforce any of these in your DDL.

You may find there is tension between some of these principles. Where that occurs, prioritize in the order shown above. Use your judgment to make any other decisions. Additional requirements:

- Define appropriate constraints, i.e.,
  - Define a primary key for every table.
  - Use `UNIQUE` if appropriate to further constrain the data.
  - Define foreign keys as appropriate.
  - For each column, add a `NOT NULL` constraint unless there is a good reason not to.
- All constraints associated with a table must be defined either in the table definition itself, or immediately after it.
- To facilitate repeated importing of the schema as you correct and revise it, begin your DDL file with our standard three lines:

```
drop schema if exists ticketchema cascade;
create schema ticketchema;
set search_path to ticketchema;
```

You may invent IDs, or define additional columns if you feel this is appropriate. Use your judgment.

There may be things we didn't specify that you would like to know. In a real design scenario, you would ask your client or domain experts. For this assignment, make reasonable assumptions. Keep track of these in writing, as we will ask you to articulate them at the top of your DDL file.

Document your choices and assumptions

At the top of your DDL file, include a comment that answers these questions:

Could not: What constraints from the domain specification could not be enforced without assertions or triggers, if any?

Did not: What constraints from the domain specification could have been enforced without assertions or triggers, but were not enforced, if any? Why not?

Extra constraints: What additional constraints that we didn't mention did you enforce, if any?

Assumptions: What assumptions did you make?

Instance and queries

Once you have defined your schema, create a file called `data.sql` that inserts data into your database. the data that is described informally in file `ticket-data.txt`.

You may find it instructive to consider this data as you are working on the design. Then, write queries to do the following:

1. For each concert, report the total value of all tickets sold and the percentage of the venue that was sold.
2. For each owner, report how many venues they own.
3. For each venue, report the percentage of seats that are accessible.
4. Report the username of the person who has purchased the most tickets. If there is a tie, report them all.

We will not be autotesting your queries, so you have latitude regarding details like attribute types and output format. Make good choices to ensure that your output is easy to read and understand.

Write your queries in files called `q1.sql` through `q4.sql`. Download file `runner.txt`, which has commands to import each query one at a time. Once all your queries are working, start postgreSQL, import `runner.txt`, and cut and paste your entire interaction with the postgreSQL shell into a plain text file called `demo.txt`. We will assess the correctness of your queries based only on reading demo.txt, so it must show the results of the queries. There is no need to insert into tables (since we are not autotesting).

There will be lots of notices, like: Eg. INSERT 0 1, psql:q2.sql:16: NOTICE: view "blah" This is normal, and we are expecting to see it.

What to hand in for Part 1

Hand in plain text files `schema.ddl`, `data.sql`, and `q1.sql` through `q4.sql`, and `demo.txt`. These must be plain text files.

IMPORTANT: You must include the demo file, and it must show the output of your queries, or you will get zero for this part of the assignment.

How Part 1 will be marked

Part 1 will be graded for design quality, including: whether it can represent the data described above, appropriate enforcement of the constraints described, avoiding redundancy, avoiding unnecessary NULLs, following the priorities given for any tradeoffs that had to be made, and good use of DDL (choice of types, NOT NULL specified wherever appropriate, etc.) Your queries will be assessed for correctness, as evidenced by the `demo.txt`.

Your code will also be assessed for these qualities:

- Names: Is every name well chosen so that it will assist the reader in understanding the code quickly? This includes table, view, and column names.
- Comments:
  Does every table or view have a comment above it specifying clearly exactly what a row means? Together, the comments and the names should tell the reader everything they need to know in order to use a table or view. For views in particular, Comments should describe the data (e.g., "The student number of every student who has passed at least 4 courses.") not how to find it (e.g., "Find the student numbers by self-joining . . .").
- Formatting according to these rules:
  - An 80-character line limit is used.
  - Keywords are capitalized consistently, either always in uppercase or always in lowercase.
  - Table names begin with a capital letter and if multi-word names, use CamelCase.
  - attribute names are not capitalized.
  - Line breaks and indentation are used to assist the reader in parsing the code.

Thought questions

These questions will deepen your appreciation of issues concerning design, efficiency, and expressive power. They are for your learning, not for marks. Feel free to discuss them with each other, or with me in class or office hours.

1. Suppose we allowed you to be less strict in following the design principles listed above. Describe one compromise you would make differently.
   (a) How would the schema be different?
   (b) What would be the benefits of this new schema?
   (c) What would be lost in this new schema?
   (d) Why would you make this different compromise?
2. What aspects of the data were awkward to express in SQL? We may have time to cover a bit of JSON or XML at the end of term. If so, or if you happen to know one of these "semi-structured languages", Would any aspect of our data be more natural to express in JSON or in XML?

Part 2: Functional Dependencies, Decompositions, and Normal Forms

1. Consider a relation R with attributes $LMNOPQRS$ with functional dependencies S:

$$S = \{\ L \rightarrow NO, \quad M \rightarrow P, \quad N \rightarrow MQR, \quad O \rightarrow S\ \}$$

   (a) State which of the given FDs violate BCNF.

   (b) Employ the BCNF decomposition algorithm to obtain a lossless and redundancy-preventing decomposition of relation R into a collection of relations that are in BCNF. Make sure it is clear which relations are in the final decomposition, and don't forget to project the dependencies onto each relation in that final decomposition. Because there are choice points in the algorithm, there may be more than one correct answer. List the final relations in alphabetical order (order the attributes alphabetically within a relation, and order the relations alphabetically).

   (c) Does your schema preserve dependencies? Explain how you know that it does or does not.

   (d) Use the Chase Test to show that your schema is a lossless-join decomposition. (This us guaranteed by the BCNF algorithm, but it's a good exercise.)

2. Consider a relation A with attributes $ABCDEFGH$ and functional dependencies B.

$$B = \{\ ACD \rightarrow E, \quad B \rightarrow CD, \quad BE \rightarrow ACF, \quad D \rightarrow AB, \quad E \rightarrow AC\ \}$$

   (a) Compute a minimal basis for T. In your final answer, put the FDs into alphabetical order. Within each individual FD, this means stating an FD as $XY \rightarrow A$, not as $YX \rightarrow A$. Also, list the FDs in alphabetical order ascending according to the left-hand side, then by the right-hand side. This means, $WX \rightarrow A$ comes before $WXZ \rightarrow A$ which comes before $WXZ \rightarrow B$.

   (b) Using your minimal basis from the last subquestion, compute all keys for P.

   (c) Employ the 3NF synthesis algorithm to obtain a lossless and dependency-preserving decomposition of relation P into a collection of relations that are in 3NF. Do not "over normalize". This means that you should combine all FDs with the same left-hand side to create a single relation. If your schema includes one relation that is a subset of another, remove the smaller one.

   (d) Does your schema allow redundancy? Explain how you know that it does or does not.

Show all of your steps so that we can give part marks where appropriate. There are no marks for simply a correct answer. You must justify every shortcut that you take.

What to hand in for Part 2

Type your answers up using LaTeX or Word. Hand in your typed answers, in a single pdf file called A3.pdf.

Final Thoughts

Submission: Check that you have submitted the correct version of your files by downloading it from MarkUs; new files will not be accepted after the due date.

Marking: The marking scheme will be approximately this: Part 1 60%, and Part 2 40%.

Some parting advice: It will be tempting to divide the assignment up with your partner. Remember that both of you probably want to answer all the questions on the final exam.