University of Toronto
CSC343, Fall 2023

Assignment 3

Assignment Due: Wednesday ~~27~~ 29 November, at 3:00 pm sharp!
You must declare your partnership (or that you are working solo) on MarkUs before the above due date, even if you have an extension.

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

A recording company owns studios across North America They want to integrate some of the critical data from their recording software with some of their other business data into one database, and you've been hired to design a proof-of-concept version of the database. Below is the information that they want to be able to store. There is much more to be added later, but this is not your responsibility.

- A studio has a name and address, and can be booked for recording sessions. (We'll sometimes call them just "sessions" for short.) Each session is associated with one studio
- Even if a studio can accommodate multiple recording sessions at the same time, they mustn't start at the same time.
- Each session has at least one recording engineer, and at most 3. We also need to keep track of the date and the start and end time for the session, and what the fee was for the session. (The fee can't simply be computed from the length of the session; sometimes deals are made where there is a flat fee, or a discount for booking a series of sessions. So we just record the session fee.)

- A session could start on one date and end on another, for instance if it starts in the evening and goes past midnight.
- We will use the term "band" to mean any group of people who sometimes play together. A band has a name of course, and at least one person in it. There is no upper limit.
- One or more people can play on a recording session. Or one or more bands can play on a session. Or both. Whatever the case, there must be at least one person playing on a session.
- We will use the term "play" to mean either sing or plan an instrument, or any kind of performance that can be recorded.
- Everyone in the database has a name, an email address, and a phone number. People who are sound engineers may have one or more certifications (these will be recorded as strings) from various accreditation organizations.
- You can treat a phone number as a single string. It would be more realistic to define a type that would ensure every phone number has a valid structure, but this is not necessary.
- Each studio has one manager at any one time, but that manager can change over time. We record the history of a studio's managers by recording their start dates. You can assume that there are no gaps in management; a manager is in their role until the day when the next manager begins.
- Some people manage multiple studios.
- A recording session normally results in several (perhaps many) segments of sound recording. Everyone who was booked to play at that session is considered to have played on every segment recorded during that session. Occasionally, no segments are recorded in the entire session. (For example, if a band comes in, practices a bit, has a disagreement, and leaves in a huff.) Even if no segments are recorded, everyone is still considered to have played at the session.
- A recording segment has a unique ID, a length (an integer number of seconds) and a format it was recorded in (a string). A recording segment is the product of exactly 1 recording session, but it may be used on multiple tracks—or on none. And a track usually includes multiple segments, for instance one segment could be a bass line and another a rhythm guitar.
- A track has a unique ID and a name, and appears on at least one album
- An album has a unique ID, a name, and a release date. It must contain at least two tracks.

Several features above are not realistic, but they simplify your assignment. If we have not constrained something, assume it is unconstrained. For example, if I said houses have windows but didn't constrain it further, you should be prepared that a house may have no windows, 1 window, or many windows.

Task 1: 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. Make it a single attribute that is an integer (you can use type SERIAL if you like). We are about to learn that single-attribute integer keys are ideal. One reason is that it makes search by the primary key, which we expect to do a lot or we wouldn't have defined something as the primary key, faster. Every time a comparison is needed during the search, it is just a comparison of two ints. Computers are really fast at comparing ints! Having a single integer as primary key also makes joining on the primary key much faster, and this is something that will happen *a lot*.
  - 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 recording cascade;
create schema recording;
set search_path to recording;
```

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

Is it really okay to invent IDs?

Don't be reluctant to invent IDs—it can sometimes really simplify things. Think back to the old, familiar university database. In the Offering table, we made up the oID column so that we wouldn't have to use the combination of {dept, cnum, term, instructor} to identify an offering when we needed to refer to a single offering, for instance, in the Took table. This simplified the Took table, and also made joins between Offering and Took much easier!

Sometimes, introducing an invented ID will take away the opportunity to enforce a constraint. For example, suppose we wanted to enforce that no student can take the same course (dept-cnum combination) twice. With relational algebra, we can write arbitrarily complex constraints. But in SQL, we have more limited things we can express in a table definition. If the Took table only has oID, we can't enforce the constraint. So we might consider including dept and cnum into Took. We'd still need oID so we could do a join to find out the term and instructor. What are the pros and cons of this design?

- It allows us to enforce the new constraint: we could say that sid, dept, cnum is unique in Took.
- But it makes the design much more complicated, and it introduces redundancy. If Offering already says that offering 14239 is csc108, we don't need to repeat that in the Took table every time there is a row about someone's grade in offering 14239! This also opens up the opportunity for update and deletion anomalies, which is very bad.

On balance, I would not choose this design. I'd sacrifice constraint enforcement in order to avoid the negative consequences that go with it in this case. And in a real situation, we wouldn't need to rely on the DDL to do all constraint enforcement; we could write the Python methods that update the Took table so that they enforce the constraint!

Task 2: 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?
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 and document them here.

Task 3: Make an instance and write 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 `recording-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 studio, report the ID and name of its current manager, and the number of albums it has contributed to. A studio contributed to an album iff at least one recording segment recorded there is part of that album.
2. For each person in the database, report their ID and the number of recording sessions they have played at. Include everyone, even if they are a manager or engineer, and even if they never played at any sessions.
3. Find the recording session that produced the greatest total number of seconds of recording segments. Report the ID and name of everyone who played at that session, whether as part of a band or in a solo capacity.
4. Find the album that required the greatest number of recording sessions. Report the album ID and name, the number of recording sessions, and the number of different people who played on the album. If a person played both as part of a band and in a solo capacity, count them only once.

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 both the queries being run and 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 both the queries being run and the output of your queries, or you will get zero for the queries part of the assignment**.

How Part 1 will be marked

Part 1 will be graded for how well you document your design choices, and 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 above 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 DDL file 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 $ABCDEFG$ and with functional dependencies S:

$$S = \{\ AC \to D, \quad BG \to E, \quad D \to CFG, \quad DG \to B, \quad G \to F\ \}$$

   (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 $LMNOPQRS$ and functional dependencies B.

$$B = \{\ N \to M, \quad NO \to LR, \quad NQR \to MP, \quad P \to R, \quad Q \to NO\ \}$$

   (a) Compute a minimal basis for B. In your final answer, put the LHS and RHS of each individual FD into alphabetical order. This means stating an FD as $BC \to AFD$, not as $CB \to FAD$. Also, list the FDs in alphabetical order ascending according to the left-hand side, then by the right-hand side. This means, $CB \to FAD$ comes before $CBG \to E$ which comes before $CBG \to H$.

   (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.