CS127 Homework #3

Due: October 11th, 2019 11:59 P.M.

Handing In

Upload your homework to gradescope.

Important Note

Problem 5 requires some technical setup. It includes detailed instructions but we recommend starting early. Go to hours from 6-7 on Friday, Saturday, or Sunday to get help specifically for the technical setup.

Warmup #1

Views:

1. Most DBMS's support creation of views. But why are views useful? Give two reasons.

2. Materialized views are views that store the data and keep it up to date with the underlying relations. Considering between a non-materialized view and a materialized view, what are the pros/cons?

3. You are working with a database that supports inserts via views. These are your schema:

   employee(emp_id, name, dept_id, mgr_id, salary)
   department(dept_id, name)

   You have made the following views:

```
create view emp_names as
select emp_id, name
from employee;

create view dept_avg_salaries as
select dept_id, avg(salary)
from employee
group by dept_id;
```

   What happens when you try to run the following insert statements? If you think that the answer is ambiguous, then explain the different approaches that you think are valid, and you will receive full credit. You can assume for the purpose of this question that all the insertions satisfy primary and foreign key constraints.

   (a) `insert into emp_names(3780, "Lauren Zissel");`
   (b) `insert into dept_avg_salaries values (1450, 105000);`

Warmup #2

Constraints and FDs

1. The SQL standard specifies that constraint checking may involves sub-queries. It so happens that few DBMS implement this feature, but suppose that this feature were implemented. How would you add the constraint to the above schema "employee" that every manager must have at least one employee?
2. What is a lossy decomposition? Give an example. Then fix it to be a lossless decomposition.
3. What is a dependency preserving decomposition? Give an example of a decomposition that is not dependency preserving and then fix it to be dependency preserving.
4. Is BCNF decomposition dependency preserving? Give an example of BCNF decomposition using the relation (A, B, C, D) with FDs $\{AB \rightarrow C,\ C \rightarrow B,\ A \rightarrow D\}$ and explain why your result is either dependency preserving or not. Is 3NF decomposition dependency preserving?

Problem 3

When your database's schema gets more and more complex, the more important normalization becomes. In the real world, we need to have an intuitive (even though not deep technical) understanding of the implications of database normalization in our data.

We'll gain this intuitive understanding by working through the normal forms until we get to Boyce-Codd Normal Form, colloquially known as the 3.5 Normal Form. There are at least up-to the 6^{th} normal form (per Wikipedia). We'll go through the first 3 (and a half) here and that should give you a good sense of how these work in reality.

Data: We'll be normalizing the data in table 1. The schema for these data are outlined in table 2. Finally, here are few more relevant tidbits:

1. departments are given ranges of course numbers. For example, CS has courses ranging from 000-150, MATH from 151-300, and PHYS from 301-500.
2. one course is taught by one instructor but not vice versa.

c_id	**c_dept_id**	**c_dept**	**evaluations**	**inst**	**office**	**sect**	**time_slot**
61	1	CS	HW, Midterm, Final	Eddie Kohler	345	A	8am-10am
61	1	CS	HW, Midterm, Final	Eddie Kohler	345	B	1pm-3pm
165	2	MATH	Project, Final	Stratos Idreos	346	A	10am-11am
165	2	MATH	Project, Final	Stratos Idreos	346	B	7pm-8pm
265	2	MATH	HW	Andy Pavlo	346	B	1pm-2pm
455	3	PHYS	Midterm, Final	Nesime Tatbul	347	B	6pm-7pm

Table 1: A table of courses in DB University

The answers to each of these should be a drawing of your relational schema at the end of each of the normalization steps. We also mark each question we consider during grading with * so write your answers for these as well. Not answering these will lose you 1 point each. Each step is worth 10 points.

1. **First Normal Form** A schema that is in the first normal form when it meets two requirements:
   - The attributes are atomic and
   - The attribute names are unique

course
c_id
dept_id
dept
evaluations
inst
office
sect
time_slot

Table 2: Starting un-normalized schema of the table 1 ommiting the primary key indicator

Now consider the relation in table 1. This table is not in first normal form because there are two attributes that are not atomic. Which ones are they?*****

There are generally three ways of making a non-atomic attribute atomic. You can split the attribute to multiple attributes, you can split them by duplicating tuples, effectively making them part of a candidate key, or making separate relations.

Which non-atomic attribute if any makes sense to split into

(a) multiple attributes (why)?*****

(b) multiple tuples (why)?*****

(c) into a separate relation (why)?*****

There is only one set of right answers here in the context of this homework. We'll decompose the data in the following way:

(a) Time slot should have been one of the attributes you identified that is not atomic. We split this by making multiple attributes.

(b) Next, identify the other non-atomic attribute(s).

(c) For each of these attributes, create a separate relation. Give the relation an attribute called *id* in addition to other attributes you have added to this relation.

(d) By separating these attributes to a different table and removing them from our first table, we have lost information. Write out a question you can no longer answer by doing this.*****

(e) You'll need a *link table* to solve this loss of information. Go ahead and create one and call it *link_table*

As a result of this normalization, you should have three relations conforming to first normal form! Draw your current relational schema below. Make sure you identify the primary key and foreign key relationships in the relations. We'll continue to refer to the reduced course table as *course* in the next normalization steps.

2. **Second Normal Form** A schema is in the second normal form when it meets two requirements:

   - The schema is in first normal form (DONE! WOOT)
   - There are no partial dependencies

   In plain English, every attribute in a relation depends on a *whole candidate key* of the relation, not just some subset of the attributes in the candidate key. The *course* relation is not in second normal form. Why not?***** Let's change that. Note that there is only one sensible option when going through this process for the *course* relation.

   - Identify the only candidate key in the new *course* relation.*****

- Identify the attributes that depend on the entire candidate key,* and those that depend on a subset of this candidate key.*

  You've identified a partial dependency: if a relation R has a candidate key of the set of attributes A. A partial dependency is a functional dependency $\alpha \rightarrow \beta$ where $\alpha \subset A$ and β is not a candidate key.

- We can normalize this by splitting the relation to two relations - one for those attributes that are partially dependent on the identified candidate key, and another for those that are dependent on the whole key.

Draw your resulting schema. There should be **4 relations** in your new schema: the *courses* relation with a reduced primary key, two from the result of your first normal form, and one from the result of your second normal form (with the compound primary key).

3. **Third Normal Form** To satisfy third normal form, your schema should:

   - The schema is in second normal form (DONE! WOOT)
   - There are no transitive dependencies

   A transitive dependency is defined as follows: Let A, B, and C designate three distinct (but not necessarily disjoint) sets of attributes of a relation. If $A \rightarrow B$ and $B \rightarrow C$ holds and $B \rightarrow A$ does *not* hold, then $A \rightarrow C$ is a transitive dependency. This dependency exists by virtue of which of Armstrong's axioms?* Why is this undesirable (hint: think about redundancy and extraneous attributes)? *

   In plain English, this means that there are no non-key attributes that maybe determined by another non-key attribute.

   If you have successfully reached this point, there should be one relation where *c_id* is the primary key. Let us focus on that relation and get it to third normal form:

   - In the relation, identify one pair of non-key attributes where each attribute is not independent of the other, and are still dependent on the primary key. There should only be one set of non-key attributes that satisfy this. If you are still unclear, reread the description of the data before you started! You have identified a **transitive dependency**. Explain how these attributes meet the definition for transitive dependency (e.g. what dependencies hold and what dependencies doesn't hold?).*
   - Separate these attributes to another relation. Make sure to reference this relation in the *course* relation by keeping/adding an ID attribute. (Hint: you'll probably be keeping instead of adding an ID attribute).

   Draw your resulting relational schema!

4. **BCNF: Third-and-a-Half Normal Form** Finally, Boyce-Codd Normal Form is a short step up from third normal form. For a schema to be in BCNF, it needs to be:

   - In third normal form
   - For all its dependencies $\alpha \rightarrow \beta$, one of the following holds true:
     - $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e., β is a subset of α), or
     - α is a superkey in the relation R

   In plain English, *each attribute represents a fact about the key, the whole key, and nothing but the key.* Explain in English why this conclusion holds if at least one of the two conditions on functional dependencies hold (e.g. If $\alpha \rightarrow \beta$ is trivial, then what? Otherwise if α is a superkey then what?)*.

   The *course* relation still has a set of attributes that do not satisfy BCNF. Which one and why?*. All you need to do to achieve BCNF is to separate out these attributes into a relation!

5. While extremely rare and frequently highly contrived, **BCNF might lead to loss of a dependency**. We'll take a look at how this happens. You were told the following new information:

   - Section letters go from advanced to beginner with most advanced as section A
   - DB University is now hiring TAs for course-sections however,
   - for accounting purposes, a TA can only teach one type of section (e.g. TAs 2, 3, and 4 teach only the B section, and TA 1 (and potentially others) teach only the A sections), and
   - each course-section hires only one TA
   - You were given a list of TAs and the course-sections they teach as seen in table 3

c_id	**sect**	**ta**
61	A	TA1
61	B	TA2
165	A	TA3
165	B	TA4
265	B	TA4
455	B	TA5

Table 3: the new relation with the derived attribute: the part of day

Answer the following:

(a) Identify the functional dependencies and candidate keys this relation.

(b) Which functional dependency ensures that a course-section can only have one TA? Which ensures that each TA only teaches one section type?

(c) Is this in 3NF and why(not) (hint: consider the potential candidate keys in the relation)?

(d) This relation is not in BCNF. Why (e.g. is there a functional dependency that does not satisfy one of the two conditions for BCNF)?

You decide that to try to decompose this to be in BCNF. You create the two relations in tables 4 and 5. You can answer what parts of day a class occurs, and what sections occur during the part of day while still in BCNF!

(a) Explain why each these relations are in BCNF

(b) But you've lost a dependency! What dependency was lost?

(c) Which one of the new information is now no longer "easy" to guarantee?

c_id	**ta**
61	TA1
61	TA2
165	TA3
165	TA4
265	TA4
455	TA5

Table 4: the new relation decomposed from table 3

ta	sect
TA1	A
TA3	A
TA2	B
TA4	B
TA5	B

Table 5: the new relation decomposed from table 3

Problem 4 Wherein we deal with software and code

It's time to get a-coding and use what you just did in a real database management system!
We'll be using the following software:

1. **Docker**: a container software solution that ensures everything works in a cross-platform manner.
2. **Postgres RDBMS**: the progenitor of many DBMSs. Check out it's earlier iteration: Ingres!
3. **pgweb**: a web UI to manage the databases in Postgres

Technical setup is required as with everything. This process is not complicated but it maybe new to some people. If this is all new to you, you'll be learning a few skills that are used in industry. In particular, some command line/terminal skills and some familiarity with how Docker works - the industry standard container solution.
Tech hours will be available during the first few days of the homework. We'll update our calendar so to reflect this. Feel free to come by if you need help with anything!

Instructions for setup

A bunch of this is going to be terminal based commands so if you aren't familiar with the terminal or need help, come to Tech Hours!

- **Install docker** by following one of the following:
  1. Mac or Windows: Install docker desktop by following this link: `https://dockr.ly/2tEQgR4`.
  2. Linux: we'll assume you know what you are doing so here's a guide for that (Ubuntu 18.04): `https://do.co/2QwjOtN`
  3. Homebrew (OSX alternative): Also available though I haven't used it

  You might need to create an account at hub.docker.com using docker desktop.
- **Setup the postgres and pgweb** containers by entering the following commands in your terminal (ignoring the `$` sign):

```
$ docker pull postgres               # pull postgres container from docker hub
$ docker pull sosedoff/pgweb         # pull pgweb container from docker hub
$ docker volume create pg_coursedb   # setup persistent volume named pg_coursedb
```

- **Run Postgres** by entering the following command in your terminal:

```
$ docker run -p 5432:5432 -d \
    --name postgres_container \
    -e POSTGRES_PASSWORD=12345 \
    -e POSTGRES_USER=user \
    -e POSTGRES_DB=coursedb \
    -v pg_coursedb:/var/lib/postgres/data \
    postgres
```

In English, it tells docker to setup a container connected to port 5432 (`-p 5432:5432`) naming it "postgres" (`--name 5432:5432`) with a set of environment variables (`-e POSTGRES_PASSWORD=postgres -e POSTGRES_USER=postgres -e POSTGRES_DB=coursesdb`) connected to the volume pgdataata (`-v pgdata:...`) using the image `postgres`. The database you just setup is called `coursedb`.

To confirm you set up docker correctly, you can query and view your database using postgres' terminal application:

```
$ docker exec -it postgres_container psql -Uuser coursedb
```

If this connects, you should see something that looks like this:

```
$ docker exec -it postgres_container psql -Uuser coursedb
psql (11.5 (Debian 11.5-1.pgdg90+1))
Type "help" for help.
coursedb=#
```

You can type `exit` or `\q` to quit.

- To **run pgweb** we need to identify the IP address docker gave to postgres. We can find this by typing:

```
$ docker network inspect bridge | grep "postgres" -A 4
```

The part before the — (a pipe in the command line) inspects docker's default network named "bridge". The portion after the — searches for "postgres" and prints out the line containing this and 4 lines after that line. Your output should look something like this:

```
"Name": "postgres_container",
"EndpointID": "ae1b82f...",
"MacAddress": "02:42:ac:11:00:02",
"IPv4Address": "172.17.0.2/16",
"IPv6Address": ""
```

The IP address will be the entry for `"IPv4Address": 172.17.0.2` (ignoring `/16`). It will likely be this address but you should double check yours anyways. Replace `ipv4add` with this number in the following command:

```
$ docker run -p 8081:8081 \
    --name pgweb \
    -e DATABASE_URL=postgres://user:12345@ipv4add:5432/coursedb?sslmode=disable \
    sosedoff/pgweb
```

If this all worked, you'll see the following:

```
Pgweb v0.11.3 (git: c830...)
Connecting to server...
Connected to PostgreSQL 11.5
Checking database objects...
Starting server...
To view database open http://0.0.0.0:8081/ in browser
```

In your fave browser, visit `localhost:8081` and see the beauty that is pgweb.

- **Some useful docker commands**

```
$ docker stop postgres  # stops a running container named postgres
$ docker start postgres # starts a stopped container named postgres
$ docker ps -a          # lists of all running/stopped container
```

  There are a bunch of other resources online so don't hesitate to look them up. In the real world, half the time you are going to need it!

Coding time

All the work you've done so far will now coalesce in some code. If you fully understood problem 3 until **3.4**, you'll understand what we're doing here. All we're going to do in this homework is to write some DDL to create the schema relevant to the work you did in problem 3. Pay particular attention to the constraints and types we tell you to write.

Use the `pgweb` application to try out your sql! Here are the relations you will create tables for. We'll mention a few constraints but it's up to you to select things like primary keys and foreign keys as well as translating our description to PostgreSQL.

As you'll see, we'll start by creating the the simpler "fact" tables first before moving toward the connected tables. This makes it easier to reason about the referential constraints. Keep the following things in mind as rules when writing the schema below:

- An attribute that is explicitly a *something_id* should be an auto-incrementing integer. Look up the `SERIAL` data type in Postgres. These also are not-nullable if they serve as primary keys in the relation.
- When an instructor leaves the university, the course they are instructing may end up being null. We can't force an instructor to stay but we should be able to find those courses that don't have an instructor. Keep this in mind when defining constraints regarding the instructors.
- When a course is removed from the course directory, everything related to the course is deleted because none of the data are valid after the course is no longer offered.
- Unless explicitly instructed (as for example the instructor or course points made above), all constraints for referential integrity should prevent deletions when references exist.
- Preferred Postgres types for this homework:
  - **Strings**: for simplicity, use the `text` type unless we explicitly specify the length of the string
  - **Integers**: for simplicity, use the `integer` type unless we explicitly specify the size of the integer
- Use the data in table 1 to determine the data types to use in your DDL

Your answers for each point below is worth 7 points each. Write these into a text file called `ddl_dump.sql` and submit the file to Gradescope for Problem 4. Again, you can test your code using the `pgweb` GUI or using `psql`. The text in `ddl_dump.sql` should look (roughly) like this:

```
-- contents of ddl\_dump.sql

CREATE TABLE a (
    aid INT NOT NULL,
    bid INT NOT NULL,
 );
```

```
CREATE TABLE b (
    bid INT NOT NULL,
    aid INT NOT NULL,
);

-- you can write comments by starting the line with a double dash
CREATE TABLE r (
    aid INT NOT NULL REFERENCES a (aid),
    bid INT NOT NULL REFERENCES b (bid),
);
```

- Create a table called *department* with attributes *dept_id* and *dept*, along with the right constraints, types, and keys (PKs). This is the set of departments and a department can't have a null name.
- Create a table called *instructor* with attributes *inst_id*, *inst*, and *office*, along with the right constraints, types, and keys (PKs and FKs). All instructors must have a name (obviously), and (maybe not as obviously), an office.
- Create a table called *evaluation* with the following attributes: *evaluation_id*, and *evaluation*. If there is an evaluation, then such an evaluation can't be null.
- Create a table called *course* with the following attributes: *c_id*, *dept_id*, and *inst_id*. Keep in mind the constraints we pointed out above. Also, keep in mind which attribute(s) are PK and FK.
- Create a table called *course_eval* with the attributes *c_id* and *evaluation_id*. Courses in this relation must have an evaluation but not all courses need to have an evaluation.
- Create a table called *course_time* with the following attributes: *c_id*, *sect*, *start*, and *end*. The attribute *sect* is a single character field, and the attributes *start* and *end* use the `time` (as opposed to `timestamp`) data type (A good resource: https://bit.ly/2nY79Xi).

  One additional wrinkle: because all courses need a section and course time slot, you need to enforce total participation (remember that in your ER diagrams?) between *course(c_id)* and *course_time(c_id)*. For a big hint on how to do this in Postgres, see: https://bit.ly/2o2FBzU. It's important to note that different DBMSs use different techniques to make this work.
- Populate your table with the information from table 1 (use SQL's `insert`). Make sure that all the information in that table is captured by your database. One note: the *c_dept_id* attribute in table 1 does not have to be exactly the same as the department IDs your database generates so long as it maintains the same relationship (1 to 1).