Homework 3: Design Improvements

Björn Thór Jónsson

Fall 2021

1 SQL DDL

Write SQL DDL commands to create the WASP database from Homework 2. Your script must implement the official solution ER diagram posted on LearnIT, must run in PostgreSQL as a whole, and must use the methodology given in the textbook and lectures. The relations must include all relevant primary key, candidate key, foreign key and NOT NULL constraints. Constraints that cannot be enforced with these standard constraints can be omitted (but if you are having fun, feel free to implement queries to detect violations, or triggers to prevent them). Note, however, that your SQL DDL script should also take into account requirements of Homework 2 that could not be implemented in ER but can be implemented in the DDL (e.g., additional keys). Select reasonable types for attributes.

2 Practical Normalization

In the homework quiz on LearnIT, you will find a script to create and populate five independent relations, each of which has seven columns and a primary key, with a few thousand rows. For this project, *use only the Rentals relation!*

Each relation models a potential real-life database design situation, with some design flaws that must be addressed. In short, each of the relations has embedded a set of functional dependencies, which are not stated explicitly. You must a) find these dependencies and b) use them to guide decomposition of the relations to 3NF/BCNF normal forms, using the methodology covered in class.

2.1 Steps

More specifically, for the Rental relation, take the following steps:

(a) Find all the important FDs in the relations, given the constraints and simplifying assumptions that are listed in detail below.

Note: We strongly recommend to use your favourite programming language to generate an SQL script with all the FD detections queries, using the SQL query template covered in slide 46 of the lecture. The program could take as input a list of column names, and output all the required SQL queries into a text file. You can then run the text file using `psql`. *Figure 1 shows example Python code to get you started.*

(b) Decompose the relation until each sub-relation is in 3NF/BCNF, while preserving all non-redundant FDs. Write down the resulting schema description in a simple Relation(columns) format.

(c) Write the detailed SQL commands to create the resulting tables (with primary keys and foreign keys) and populate them, by extracting the relevant data from the original relations.

Note: In this homework, use the "homework" version of commands to create the new relations (see slide 9), so that you can compare the decomposition result with the original relation. Specifically, you should not run any ALTER TABLE commands.

(d) Select the correct normal form for the decomposed schema.

You can find a video from last year explaining the process (there are small differences, but in the main the video is relevant) at: `https://app.vidgrid.com/view/hP1KPli1OvRm`

2.2 Simplifying Assumptions = Reality

In this project, assume that the following simplifying assumptions hold for each of the relations (this is the "reality" that you check them against):

- The relations must each be considered in isolation. The columns have short names that hint at their meaning, but you should not count on implicit FDs derived from the expected meaning of the columns. In short, the column names may trick you!

```
----------------------------------------------------------------------
def PrintSQL(Att1,  Att2)
    print ("Here make the SQL query to check %s --> %s" % (Att1, Att2));
    print ("Use the query in Slide 43 of the Normalisation lecture");

R = ['A', 'B', 'C']
for i in range(len(R)):
    for j in range(len(R)):
        if (i != j):
            PrintSQL(R[i], R[j])
----------------------------------------------------------------------
```

Figure 1: Python skeleton for code to check all FDs for a relation $R(A, B, C)$ with three attributes in step (a).

- Assume that all functional dependencies in each relation (aside from primary key dependencies and dependencies derived from that) can be found by checking only FDs with one column on each side, such as $A \rightarrow B$.

  Note: As discussed in lectures, you can use SQL queries to detect potential FDs. Using the assumptions above, it is indeed relatively simple to create a program to output queries for all possible checks for FDs, which you can then run with `psql`*, thus automating the detection process.*

- If you find a functional dependency that holds for the instance given, you can assume it will hold for all instances of the relation.

  Exception: When an ID column and a corresponding string column both determine each other, consider that only the ID column determines the string column, not vice versa. For example, if both $CID \rightarrow CN$ and $CN \rightarrow CID$ are found to potentially hold, then consider that only $CID \rightarrow CN$ is valid.

- The only dependencies you need to consider for decomposition are a) the dependencies that can be extracted from the data based on the assumptions above, and b) the given key constraints. As covered in the lecture, you can ignore trivial, unavoidable and redundant functional dependencies.

3 Index Selection

Consider the following relation with information on parts (*stock* is the quantity in stock):

Part (id, descr, price, stock, ...)

Assume that this is a luxury store where prices are very high and stocks are very low. You are given a set of already existing *unclustered* B+-tree indexes for the Part relation. For each query, the index (or indexes) that a good query optimiser is most likely to use to process the query, or select "no index" if a full table scan would yield better performance than any of the available indexes.

3.1 Available Indexes

The available indexes are:

(a) Part(id)

(b) Part(stock)

(c) Part(price)

(d) Part(stock, price)

(e) Part(stock, price, id)

(f) Part(stock, price, descr)

(g) No index

3.2 Queries

The queries are:

Query 1:
```
select id
from Part
where stock > (select max(price) from Part);
```

Query 2:
```
select id, descr
from Part;
```

Query 3:
```
select stock
from Part
where price = 23;
```

Query 4:
```
select id, descr, price
from Part
where stock > 35;
```

4 SQL

In this exercise you will work with the fictional database of countries, cities and languages used for Homework 2. Please refer to that homework for the script and description of the database.

Answer each of the following questions using a single SQL query on the examination database. Enter the result of each query into the quiz on LearnIT. As before, queries should adhere to the detailed guidelines given in Homework 1.

(a) In the database there are 3,264 cities which have a population that is less than 1% of the population of their country. How many cities in the database have a population that is more than 50% of the population of their country?

(b) There are two countries for which the percentages of languages spoken add up to more than 100%. For how many countries in the `countries` table do the percentages add up to less than 100%?

(c) In France, the largest city is Paris with 2,125,246 inhabitants, while the smallest city is Montreuil with 90,674; the size ratio between the two is about 23.4. What is the ID of the city that has the highest size ratio relative to the smallest city from the same country?

Note: This query returns an ID of a city, not a count.

Hint: This is an example of the pattern to find entries with highest/lowest values: `... WHERE <attribute> = (SELECT MAX(<attribute> ...)`*. To use this pattern, you should first define a view to give all ratios between cities within the same country (using a self-join, see Week 3), and then use this view as your relation.*

(d) Let us define the 'urban population' of a country as the population that lives in one of the country's cities, according to the database. For example, the urban population of the Netherlands is 5,180,049. Write a query to find the code of the country with more than 1 million inhabitants that has the highest *ratio* of urban population.

Note: The return value of this query is the three character country code for a single country, not a number. Also, note that this query is intended to be quite hard!

Hint: I solved it by writing a view to sum up all cities in countries, and then another view to find the urban population ratio in all countries. With these views defined, it is actually just another example of the MAX pattern above.