Exam

SD202 – Databases

October 29, 2021

This is the final exam for the SD202 class, which will determine 100% of your grade for this class. The exam consists of two independent parts. **You must write your answer to each part on a *separate sheet* of paper.** You can choose to answer the questions in English or in French, as you prefer.

Write your name clearly on every sheet used for your exam answers, and number every page.

You are allowed **three A4 sheets** (i.e., six pages, one on each side) with personal notes of your choice. You may not use any other written material.

The exam is **strictly personal**: any communication or influence between students, or use of outside help, is prohibited. No electronic devices such as calculators, computers, or mobile phones, are permitted. Any violation of the rules may result in a grade of 0 and/or disciplinary action.

Part 1

Exercise 1: Functional dependencies

Question 1. Consider the following relation R having four attributes a, b, c, and d, and four tuples:

R

a	b	c	d
1	10	20	30
1	11	21	31
2	10	22	30
2	12	23	31

Which of the following FDs hold?

- $a \to b$
- $c \to ad$
- $d \to b$
- $b \to d$
- $ab \to c$
- $bd \to c$

Answer.

- $a \to b$ does not hold (first two tuples)
- $c \to ad$ holds (all values in c are unique)
- $d \to b$ does not hold (tuples 2 and 4)
- $b \to d$ holds
- $ab \to c$ holds (all values are unique)
- $bd \to c$ does not hold (tuples 1 and 3)

Question 2. Propose a table of a relation S with three attributes a, b, and c, satisfying the following FDs:

- $ab \to c$
- $c \to b$

but not the following ones:

- $a \to c$
- $b \to c$
- $c \to a$

Answer.

S

a	b	c
1	10	10
1	11	11
2	10	10
2	11	12

Exercise 2: Aggregation queries

We consider a table Student with the following attributes:

- id, a unique identifier, the primary key of the database,
- name, a string, the name of the student,
- grade, an integer, the student's grade for the year (averaged over all their courses),
- sgroup, an integer, the identifier of a student group to which the student belongs.

Question 1 Write a query in SQL returning one floating point value which is the average of the grade of all students.

Answer.

```
SELECT AVG(grade) FROM Student;
```

Question 2. Write a query in SQL returningd the list of all group ids (without duplicates) where there is a student having a grade of 10 or less.

Answer.

```
SELECT DISTINCT sgroup FROM Student WHERE grade <= 10;
```

Question 3. Write a query in SQL to list the groups and the average of the grade of the students in each group, sorted by decreasing average.

Answer.

```
SELECT sgroup, AVG(grade) AS average FROM Student
    GROUP BY sgroup ORDER BY average DESC;
```

Question 4. Write a query in SQL to compute the id, name, and grade of the student with the best grade (breaking ties at random).

Answer.

```
SELECT id, name, grade FROM Student ORDER BY grade DESC LIMIT 1;
```

Question 5. Explain in English (or French) what the following query computes.

```
SELECT S1.name, S2.name FROM Student AS S1, Student AS S2
    WHERE S1.grade = S2.grade AND S1.sgroup <> S2.sgroup;
```

Answer. It returns the names of all pairs of students that are in different groups but have the same grade.

Question 6. Explain in English (or French) what the following query computes.

```
SELECT S1.sgroup FROM Student AS S1
  WHERE NOT EXISTS (SELECT 1 FROM Student AS S2
    WHERE S2.id <> S1.id AND S1.sgroup = S2.sgroup);
```

Rewrite this query to a query that does not have a nested subquery and is equivalent (assuming that there are no NULLs in the database).

Answer. It returns the group id of all groups containing only one student, i.e., there is one student but no student in the same group which is different from that student.

This can be rewritten as:

```
SELECT sgroup FROM Student GROUP BY sgroup HAVING COUNT(*) = 1;
```

(However, the other form of the query can be used to also compute the name of the individual student in each group.)

Part 2

*Reminder: please write your answers to the exercises in Part 2 on a **separate sheet of paper**!*

Exercise 3: Schema design

We wish to design a database system for a theater. Here is a specification of the business need:

> *The theater manages people (actors and stage directors), who have a name and a phone number. The theater welcomes several productions. A production has a title, exactly one stage director, and a list of characters: each character has a name and occurs in exactly one production. Each production is played on different dates: this is called a show. Each show plays exactly one production, and has exactly one date. Then, for each show, there is a cast, specifying which actor plays which character. On every show, every character of the play will be played by exactly one actor, but an actor can play several characters. Note that the characters and the stage director are always the same for a given production; by contrast, the information of which actor plays which character may vary depending on the show.*

Question 1. Write the SQL instructions to create a database schema satisfying the business need. Make sure to define primary keys and foreign keys.

Answer.

```
CREATE TABLE Person(
  id SERIAL PRIMARY KEY,
  name VARCHAR,
  phone VARCHAR);
CREATE TABLE Production(
  id SERIAL PRIMARY KEY,
  title VARCHAR,
  director INT REFERENCES Person);
CREATE TABLE Character(
  id SERIAL PRIMARY KEY,
  name VARCHAR,
  production INT REFERENCES Production);
CREATE TABLE Show(
  id SERIAL PRIMARY KEY,
  sdate DATE,
  production INT REFERENCES Production);
CREATE TABLE Casting(
  show INT REFERENCES Show,
  character INT REFERENCES Character,
  actor INT REFERENCES Person,
  PRIMARY KEY (show, character));
```

Question 2. Write an SQL query over your schema to retrieve the name and phone number of actors who are playing today (October 29, 2021).

Answer.

```
SELECT name, phone FROM Person AS P
  INNER JOIN Casting AS C ON C.actor = P.id
  INNER JOIN Show AS S ON C.show = S.id
  WHERE sdate = '2021-10-29';
```

Question 3. Write an SQL query in your schema to retrieve the name and phone number of all stage directors who also play as an actor in some show of a production for which they are the stage director.

Answer.

```
SELECT name, phone FROM Person AS P
  INNER JOIN Casting AS C ON C.actor = P.id
  INNER JOIN Show AS S ON C.show = S.id
  INNER JOIN Production AS Pr ON S.production = Pr.id
  WHERE P.id = Pr.director;
```

Exercise 4: Query evaluation

Consider a database where we have three tables:

- A table persons describing persons and containing two attributes:
  - name of type text, the name of the person,
  - id of type integer and also the primary key, a unique identifier for that person.
- A table movies describing movies containing four attributes:
  - id of type integer and also the primary key, a unique identifier for that movie,
  - title of type text, the title of the movie,
  - year of type integer, the year the movie was released,
  - director of type integer which is a foreign key referencing the id column in the persons table.
- A table actedIn describing which person played in which movie containing two attributes:
  - idPerson of type integer and a foreign key referencing the id column in the persons table,
  - idMovie of type integer and a foreign key referencing the id column in the persons movies.

If it helps, you can make reasonable assumptions about the content of the database (e.g. all movies in the database are released between 1850 and 2050, most movies have between 5 and 50 actors, etc.).

Question 1. When creating the tables, the `PostgreSQL` database engine automatically creates an index for the primary keys. Explain in one paragraph why `PostgreSQL` creates these indexes?

Answer.

Whenever `PostgreSQL` inserts data into a table with a primary key, it needs to make sure that the primary key constraint is not violated. With the index, `PostgreSQL` can quickly check whether the constraint is violated or not. Indeed, a B-tree or a hash table can retrieve whether a key exists very quickly (with very few seeks). Without the index `PostgreSQL` would have to scan the entire table.

Furthermore primary keys are also very often used in joins and the index is very useful there as it allows some optimization (e.g. index joins).

In addition to the indexes for the two primary keys, we also suppose that there are two other indexes: one on the title column of movies and one on the director column of movies. We also suppose that all the three tables are populated with many records.

Question 2. Consider the following query:

```
SELECT p.name, m.title
FROM movies m, persons p, actedIn a
WHERE
      a.idMovie = m.id
  AND a.idPerson = p.id ;
```

Explain in English (or French) what this query computes. Explain a reasonably efficient query plan that a database engine (such as `PostgreSQL`) might consider to run this query.

Note that for this question and the two following questions, there are multiple possible answers. Your answer will be accepted as long as the plan that you propose is reasonably efficient. You must describe the query plan in English (or French) and justify briefly why the proposed plan is interesting for this query; for instance, why it is better than some more naive plan, or how many operations it performs.

Answer. This query computes the pairs (a, m) where a is the name of an actor who played in the movie titled m.

The database engine has to do a join between three tables. One way to do it is to do a sequential scan over the entire table actedIn and for each record we can use the indexes on persons and on movies to retrieve both the title of the movie and the name of the person. This method of performing a join is called an index join (and here we do two index joins).

This plan is interesting because we only perform a sequential scan over a table that has the same number of records as the output and only two index lookups per output (therefore our plan is near optimal as the output has to be produced).

Question 3. Consider the following query:

```
SELECT m1.director, m1.title, m2.title
FROM movies m1, movies m2
WHERE   m1.director = m2.director
    AND m1.id != m2.id
    AND m1.year = m2.year
```

Explain what this query computes. Explain a reasonably efficient plan for this query. Could you create an index to optimize how `PostgreSQL` evaluates this query?

Answer. This query computes the pairs (d, t_1, t_2) where d is the identifier of a director and t_1, t_2 are the titles of two movies released the same year from director d.

This query is mainly a join between the table movies and itself, joining on (director,year) and excluding pairs with the same id. This join can be realized with a hash-join or a sort-join depending on the size of the movies relation. For each tuple produced by this join we filter out pairs where the id are equal.

This query plan is interesting because the tuples filtered out at the last step are only the pairs of twice the same movie. We still have to do a scan over the whole table movie (to sort it or hash it) but there no way to really avoid this. Overall, if we use a hash-join the algorithm will run for a time that is linear in the size of movies plus the size of the output.

Another query plan that we could use is to scan the movies table and then use the index on director to retrieve movies released by the same director and finally remove the pairs where the release years do not match. This plan is less efficient because we will consider all pairs of movies directed by the

same person and a lot of directors have directed a lot of movies. Furthermore it does not prevent a full scan over the movie table.

To speed up the computation, we could create an index on the pairs (director, year). With this index PostgreSQL can do a sequential scan on the index to retrieve the movies released the same year by the same director. The index would give us the matching pairs without sorting or hashing. Furthermore we would only have to do a table lookup for pairs of movies released in the same year, which we will be faster if there are only few solutions (which might be expected considering that directors rarely release more than one movie each year).

Question 4. Consider the following query:

```
SELECT m2.title
FROM movies m1, movies m2
WHERE
      m1.title = 'Snatch'
  AND m1.director = m2.director
  AND m1.year < m2.year  ;
```

Explain what this query computes. Explain a reasonably efficient plan for this query. Could you create an index to optimize how PostgreSQL evaluates this query?

Answer. This query retrieves the titles of movies directed by the same director as the director from 'Snatch' and that were released after 'Snatch' (excluding those released the same year).

One way to evaluate this query is to perform an index-join between movies and the set of directors of movies called 'Snatch'. For this we can compute the set of directors of movies titled 'Snatch' using an index scan on titles. Then the index-join retrieves, for each director of a movie called 'Snatch', the list of movies it directed, and then checks the year criterion.

This plan is interesting because PostgreSQL can determine that only very few movies have the title 'Snatch', therefore the set of directors of a movie titled 'Snatch' is very small. Since the maximal number of movies directed by one person is also small the index-join will be fast.

If we had a b-tree index on the pair (director,year) or on the triple (title,director,year) we could directly retrieve the movies by the same director and released after 'Snatch' which may be faster than retrieving all movies and only then checking the year condition.