CIT Assignment 2 – Programming the database

This assignment concerns development of functions and procedures that extract information and do updates to the university database. As opposed to assignment 1, we will in this assignment stick to the small version of the university database from the DB-Book**,** that is, the database provided when you run the script **university_database.sql**. Since the functions will be stored in the database and some will involve updates to the data, you need to work on a fresh copy of the database and be prepared to rebuild it. Check the instructions by the end of this text.

What to do

Create the requested functions in your database and run the test queries listed below.
Collect everything (create-statements and test queries) for all questions in a single SQL script file, generate a file with output from running the script and **hand in both files.** See **details on hand-in** by the end of this text. Hand in only one submission from your group (from one of the members).

Work individually, then work in groups, then hand-in by group

You are supposed to submit one hand in per group and we strongly recommend you to work with the questions individually as well as in groups. Discuss and decide on what you consider to be the best solutions and hand these in. Feel free to provide alternatives, if you don't agree on the best among these.

Questions

Question 1)

The following SQL-query counts the number of courses the student with id='12345' has attended.

```
select count(course_id)
from takes
where id = '12345';
```

Write a function, **`course_count()`**, that takes a student id as argument and returns the number of courses the student has attended. Thus, the same as above, if called with **`'12345'`**.

Test-queries:

```
select course_count('12345');
select id,course_count(id) from student;
```

Question 2)

Create a function, **`course_count_2()`**, similar to the one from question 1 taking two parameters: the id of a student and the name of a department and returns the number of courses offered by the given department that the student has attended.

Test-queries:

```
select course_count_2('12345','Comp. Sci.');
select id,name,course_count_2(id,'Comp. Sci.') from student;
```

Question 3)

Maybe we want to avoid introducing two different functions as above, but rather prefer a unique function **`course_count()`** that can be called with either 1 or 2 parameters. How can this be done?
Explain your solution.

Question 4)

Write a function, **`department_activities()`**, that takes the name of a department and returns a list (table) of courses taught by instructors belonging to the given department including instructor name, course title, semester and year. Thus

```
SELECT department_activities('Comp. Sci.');
```

should return the list of courses taught by teachers from computer science.

Test-query:

```
SELECT department_activities('Comp. Sci.');
```

Question 5)

Write a function, **`activities()`**, that takes the name of a department or the name of a building and, depending of the type of input, returns a list of courses taught by instructors belonging to the given department respectively a list of courses taught by instructors belonging the given building (that is, belonging to a department in the given building). The returned list in both cases should include department name, instructor name, course title, semester and year.

Test-queries:

```
SELECT activities('Comp. Sci.');

SELECT activities('Watson');
```

Question 6)

Write a function, **`followed_courses_by()`**, that takes a student name as argument and returns a string, which is a comma-separated list of names of instructors that have taught courses taken by the given student.

Thus, e.g. the function call

```
SELECT followed_courses_by('Levy');
```

should return the following comma-separated list of 2 instructors:

```
followed_courses_by
---------------------
 Katz, Srinivasan
(1 row)
```

Use a cursor declared with a **cursor for** expression and loop through the query-result to assemble the string. Use a text variable (e.g. named **`result`**) and gradually extend its value using the **`||`** operator (concatenation, as in: **`result = result ||', ' || next_instructor;`**", if you have selected the instructor name into a variable **`next_instructor`**).

Test-queries:

```
select followed_courses_by('Shankar');
select name, followed_courses_by(name) from student;
```

Question 7)

Rewrite your solution to question 6 such that you use a **for** loop with a record variable to loop through a query result rather than using an explicitly declared cursor. Observe, if you define your record variable like this **`rec record;`** you can use a loop construct like **`for rec in select … loop … end loop;`**. Notice, if your select returns a single column table with column **`nn`**, **`rec`** will be the value of **`nn`** in parentheses while **`rec.nn`** will be just the value.

Test-queries:

```
select followed_courses_by('Shankar');
select name, followed_courses_by(name) from student;
```

Question 8)

The purpose with the two questions above is to try the cursor and the query result loop constructs. However, for this specific problem – concatenating strings from a list of strings (separate rows in a query result) – PostgreSQL provides a built-in function, **string_agg()**. Provide an alternative implementation of the followed_courses_by() function using this built-in function.

Test-queries:

```
select followed_courses_by('Shankar');
select name, followed_courses_by(name) from student;
```

Question 9)

Write a function, **`taught_by()`**, similar to the **`followed_courses_by()`** function above, that takes a student name as argument and returns a string, which is a comma-separated list of instructor names. The list should include instructors that have taught courses taken by the given student as well as instructors that have advised the student.

Test-queries:

```
select taught_by('Shankar');
select name, taught_by(name) from student;
```

Question 10)

This last question focus on maintaining redundant information using triggers.
Do the following:

- Alter the table student by adding an extra column "teachers" (use text as column type)
- Update all students such that the new "teachers" column will include a comma-separated list of their teachers (in the sense of question 9).
- Now create two (very similar) insert triggers[1] that ensure to keep the (redundant) value of the teachers column in the student table up to date after an insert on the takes as well as on the advisor table. (Hint: You'll need two separate triggers and two separate trigger functions, but these are pairwise very similar).
- Show that it works with the test queries below

Test-queries (in that order):

```
select id, name,teachers,followed_courses_by(name) from student;
insert into takes values ('12345', 'BIO-101', '1', 'Summer', '2017', 'A');
insert into takes values ('12345', 'HIS-351', '1', 'Spring', '2018', 'B');
insert into advisor values ('54321', '32343');
insert into advisor values ('55739', '76543');
select id, name,teachers,followed_courses_by(name) from student;
```

How to (re)build the university database

You can download and install the database on your own computer. Do the following:

1) Download the SQL script file **university_database.sql** from the CIT Moodle site.
2) Open your command line interface and change to the directory where you placed the file **university_database.sql**
3) Run the two commands

   ```
   psql -U postgres -c "create database university"
   psql -U postgres -d university -f university_database.sql
   ```

4) You may consider dropping the database if you need to start with a fresh database content while working or when finished using the database for Assignment 2. One way to drop the database is by using this command:

   ```
   psql -U postgres -c "drop database university"
   ```

[1] We consider only insertion here. So you are not required to deal with updates or deletions.

How to produce an output file and hand in your solution

When you have tested all your solutions to the questions above one by one, include all in a single SQL script file **citXX-assignment1.sql** (where x is your group number) with content like:

```
-- GROUP: citXX, MEMBERS: <name1>, <name2>, ...
-- 1.
drop function if exists course_count(char(8));
create function course_count (id char(8))
returns integer as $$
…
$$
language ...;
SELECT ….  -- test query 1
…

SELECT ….  -- test query 2
…


-- 2.
drop function if exists course_count_2(char(8),char(20));
…
```

The dropping is not necessary, but convenient because you can run your script over and over again. For question 10 you will need to do something more if you want your script to be repeatable (which is not required).
To indicate "no solution" for a question, just write "-- no solution" in place of the function definition and the test queries.
To hand in your solution do the following (**citXX** is your group name, thus one of cit01, cit02, cit03, **…**):

- Generate an output file **citXX-assignment2.txt** by executing your script, that is, by running the command:
  psql -U postgres -a -d university -f citXX-assignment2.sql > citXX-assignment2.txt
- Hand in your script file **citXX -assignment2.sql** as well as your output file **citXX-assignment2.txt** on the course Moodle page.

References

- [DSC] chapter 5.2-5.3
- [PGTUT] https://neon.com/postgresql/postgresql-plpgsql/introduction-to-postgresql-stored-procedures
- [PGMAN] 43.1 - 43.12 (lookup details)