B561 Advanced Database Concepts
Assignment 4
Fall 2021

Dirk Van Gucht

This assignment relies on the lectures

- Functions and expressions in SQL;
- Aggregate functions and partitioning; and
- Triggers.

To turn in your assignment, you will need to upload to Canvas a single file with name `assignment4.sql` which contains the necessary SQL statements that solve the problems in this assignment. The `assignment4.sql` file must be so that the AI's can run it in their PostgreSQL environment. You should use the `Assignment-Script-2021-Fall-assignment4.sql` file to construct the `assignment4.sql` file. (Note that the data to be used for this assignment is included in this file.) In addition, you will need to upload a separate `assignment4.txt` file that contains the results of running your queries.

You will also see several problems that are listed as practice problems. You should not include your solutions for such problems in the materials you submit for this assignments. Only solutions for problems 1 through 10 need to be submitted.

Database schema and instances

For the problems in this assignment we will use the following database schema:[1]

$\texttt{Person(}\underline{\texttt{pid}}\texttt{, pname, city)}$
$\texttt{Company(}\underline{\texttt{cname}}\texttt{,headquarter)}$
$\texttt{Skill(}\underline{\texttt{skill}}\texttt{)}$
$\texttt{worksFor(}\underline{\texttt{pid}}\texttt{, cname, salary)}$
$\texttt{companyLocation(}\underline{\texttt{cname, city}}\texttt{)}$
$\texttt{personSkill(}\underline{\texttt{pid, skill}}\texttt{)}$
$\texttt{hasManager(}\underline{\texttt{eid, mid}}\texttt{)}$
$\texttt{Knows(}\underline{\texttt{pid1, pid2}}\texttt{)}$

In this database we maintain a set of persons (`Person`), a set of companies (`Company`), and a set of (job) skills (`Skill`). The `pname` attribute in `Person` is the name of the person. The `city` attribute in `Person` specifies the city in which the person lives. The `cname` attribute in `Company` is the name of the company. The `headquarter` attribute in `Company` is the name of the city wherein the company has its headquarter. The `skill` attribute in `Skill` is the name of a (job) skill.

A person can work for at most one company. This information is maintained in the `worksFor` relation. (We permit that a person does not work for any company.) The `salary` attribute in `worksFor` specifies the salary made by the person.

The `city` attribute in `companyLocation` indicates a city in which the company is located. (Companies may be located in multiple cities.)

A person can have multiple job skills. This information is maintained in the `personSkill` relation. A job skill can be the job skill of multiple persons. (A person may not have any job skills, and a job skill may have no persons with that skill.)

A pair (e, m) in `hasManager` indicates that person e has person m as one of his or her managers. We permit that an employee has multiple managers and that a manager may manage multiple employees. (It is possible that an employee has no manager and that an employee is

[1]The primary key, which may consist of one or more attributes, of each of these relations is underlined.

not a manager.) We further require that an employee and his or her managers must work for the same company.

The relation `Knows` maintains a set of pairs (p_1, p_2) where p_1 and p_2 are pids of persons. The pair (p_1, p_2) indicates that the person with pid p_1 knows the person with pid p_2. We do not assume that the relation `Knows` is symmetric: it is possible that (p_1, p_2) is in the relation but that (p_2, p_1) is not.

The domain for the attributes `pid`, `pid1`, `pid2`, `salary`, `eid`, and `mid` is `integer`. The domain for all other attributes is `text`.

We assume the following foreign key constraints:

- `pid` is a foreign key in `worksFor` referencing the primary key `pid` in `Person`;
- `cname` is a foreign key in `worksFor` referencing the primary key `cname` in `Company`;
- `cname` is a foreign key in `companyLocation` referencing the primary key `cname` in `Company`;
- `pid` is a foreign key in `personSkill` referencing the primary key `pid` in `Person`;
- `skill` is a foreign key in `personSkill` referencing the primary key `skill` in `Skill`;
- `eid` is a foreign key in `hasManager` referencing the primary key `pid` in `Person`;
- `mid` is a foreign key in `hasManager` referencing the primary key `pid` in `Person`;
- `pid1` is a foreign key in `Knows` referencing the primary key `pid` in `Person`; and
- `pid2` is a foreign key in `Knows` referencing the primary key `pid` in `Person`

Pure SQL and RA SQL

In this assignemt, we distinguish between Pure SQL and RA SQL. Below we list the **only** features that are allowed in Pure SQL and in RA SQL.

In particular notice that

- `join`, `NATURAL join`, and `CROSS join` are **not** allowed in Pure SQL.
- The predicates [`not`] `IN`, `SOME`, `ALL`, [`not`] `exists` are **not** allowed in RA SQL.

The only features allowed in Pure SQL

`select ... from ... where`
`WITH ...`
`union, intersect, except` operations
`exists` and `not exists` predicates
`IN` and `not IN` predicates
`ALL` and `SOME` predicates
`VIEW`s that can only use the above RA SQL features

The only features allowed in RA SQL

`select ... from ... where`
`WITH ...`
`union, intersect, except` operations
`join ... ON ...`, `natural join`, and `CROSS join` operations
`VIEW`s that can only use the above RA SQL features
commas in the `from` clause are **not** allowed

Full SQL

all the features of Pure SQL and RA SQL
user-defined functions
aggregate functions
`group ... by ...`
`having ...`

1 Solving queries using aggregate functions

Formulate the following queries in SQL. You should use aggregate functions to solve these queries. You can use views, temporary views, parameterized views, and user-defined functions.

1. Find each pair (c, n) where c is the cname of a company that pays an average salary between 50000 and 55000 and where n is the number of employees who work for company c.

2. Find the pid and name of each person who lacks at least 4 job skils and who knows at least 4 persons.

3. Find the pid and name of each person who has fewer than 2 of the combined set of job skills of persons who work for Google. By combined set of jobskills we mean the set

$$\{s \mid s \text{ is a jobskill of an employee of Google}\}.$$

4. Find the cname of each company that employs at least 3 persons and that pays the lowest average salary among such companies.

5. Find each pair (c_1, c_2) of different company cnames such that, among all companies, company c_2 pays the closest average salary compared to the average salary paid by company c_1.

6. Without using set predicates, find each pid of a person who does not know each person who (1) works for Apple and (2) who makes less than 55000.

7. Without using set predicates, find each pairs (s_1, s_2) of skills such that the set of persons with skill s_1 is the same as the set of persons with skill s_2.

8. Find each pairs (s_1, s_2, n) of different skills s_1 an s_2 and such that (1) the number of persons with skill s_1 is the same as the number of persons with skill s_2 and (2) where n is the number of such persons associated with s_1 and s_2.

9. (a) Using the GROUP BY counting method, define a function

```
create or replace function numberOfSkills(c text)
  returns table (pid integer, salary int, numberOfSkills bigint) as
  $$
  ...
  $$ language sql;
```

that returns for a company identified by its cname, each triple (p, s, n) where (1) p is the pid of a person who is employed by that company, (2) s is the salary of p, and (3) n is the number of job skills of p. (Note that a person may not have any job skills.)

(b) Test your function for Problem 9a for the companies Apple, Amazon, and ACM.

(c) Write the same function `numberOfSkills` as in Problem 9c but this time without using the `GROUP BY` clause.

(d) Test your function for Problem 9c for the companies Apple, Amazon, and ACM.

(e) Using the function `numberOfSkills` but without using set predicates, write the following query: "*Find each pair (c, p) where c is the name of a company and where p is the pid of a person who (1) works for company c, (2) makes more than 50000 and (3) has the most job skills among all the employees who work for company c.*"

2 Introduction to Dynamic SQL

The examples in this section introduce you to **Dynamic SQL**. Dynamic SQL is a powerful generalization of SQL: it permits writing programs that generate queries, and furthermore, these queries can subsequently be evaluated in the programs that generated them.

Intuitively, in a dynamic program, a string is constructed that represents a SQL query. When an `execute` statement in the dynamic program is then applied to that string, the corresponding query is evaluated.

We will consider a simple example illustrating Dynamic SQL. (For more information about Dynamic SQL, we refer to the PostgreSQL manual under the topic of Dynamic SQL.)

Example 1 *Assume that there are 3 unary relations*

$$\begin{array}{c}\texttt{P(p : boolean)}\\ \texttt{Q(q : boolean)}\\ \texttt{R(r : boolean)}\end{array}$$

You should think of `P`*,* `Q`*, and* `R` *as boolean variables that may be* `true` *or* `false`*. This situation is set up in SQL as follows:*

```
create table P (p boolean);
create table Q (q boolean);
create table R (r boolean);

insert into P values (true), (false);
insert into Q values (true), (false);
insert into R values (true), (false);
```

So we have the following situation:

P	Q	R
t	*t*	*t*
f	*f*	*f*

Next, we consider the set of <u>boolean propositions</u> *(propositions, for short) over these variables* `P`*,* `Q`*, and* `R`*. An inductive definition of these propositions is as follows:*[2]

[2]We have used parantheses around the boolean connectives '`or`', '`and`', and '`not`' to remove issues related to the order of precedence for these connectives. We could have ommitted such parantheses but then we need to impose a precedence order. Typically, this is such that '`not`' has higher precendence than '`or`' and '`and`', and '`and`' has a higher precedence than '`or`'.

- `P`, `Q`, *and* `R` *are propositions.*
- *If E and F are propositions then* (E `or` F) *is a proposition.*
- *If E and F are propositions then* (E `and` F) *is a proposition.*
- *If E is a propositon then* (`not` E) *is a proposition.*
- *If E is a propositon then* (E) *is a proposition.*

Here are some examples of propositions over `P`, `Q`, *and* `R`.

```
              P
              Q
              R
           (not P)
          (not (P))
           (P or R)
          (P and P)
    (P and (not (R and Q)))
```

We will now consider the <u>*truth table*</u> *of a proposition. (For the concept of truth table of a propositon, we refer to the document* `Propositional_Logic` *in the course module* <u>*Notes on Discrete Mathematics*</u>*.) For example, the truth table for the proposition* (`P or R`) *is a follows:*[3]

`p`	`q`	`r`	`truthValue`
t	*t*	*t*	*t*
t	*t*	*f*	*t*
t	*f*	*t*	*t*
t	*f*	*f*	*t*
f	*t*	*t*	*t*
f	*t*	*f*	*f*
f	*f*	*t*	*t*
f	*f*	*f*	*f*

and that for `(P and (not (R and Q)))` *is*

[3]Notice that we have 3 propositional variables in play, i.e., `P`, `Q`, and `R`, and therefore, each truth table will have $2^3 = 8$ truth assignments. This is even the case when the proposition does not mention some of these variables.

p	q	r	truthValue
t	t	t	f
t	t	f	t
t	f	t	t
t	f	f	t
f	t	t	f
f	t	f	f
f	f	t	f
f	f	f	f

It is possible to generate the truth table of a proposition in SQL. For example, for the proposition `'(P and (not (R and Q)))'` *we can write the SQL query (let's call it Q_1)*

```
select p, q, r, (P and (not (R and Q)))
from   P, Q, R;
```

The problem with this approach is that to determine the truth table for another proposition such as `'(not (Q and (not P)))'`*, we would need to write a different query (let's call it Q_2)*

```
select p, q, r, (P and (not (R and Q)))
from   P, Q, R;
```

even though the blue parts in Q_1 and Q_2 are same.

A more general approach to generate the truth table of an arbitrary proposition is to use **Dynamic SQL**. *Here is the user-defined function Dynamic SQL that can facilitate this:*

```
create or replace function truthTable(proposition text)
   returns table(p boolean, q boolean, r boolean, truthValue boolean) as
$$
begin
   return query
      execute 'select p, q, r, ' || proposition ||
             ' from P, Q, R;';
end;
$$ language plpgsql;
```

To generate the truth table for the proposition

“`P and (not (R and Q))`”

we would call the `truthTable` *function with the string representation*

```
'(P and (not (R and Q)))'
```

of this proposition as follows:

```
select * from truthTable('(P and (not (R and Q)))');
```

The critical component of the DynamicSQL code for the function `truthTable` *is the statement*

```
  return query
     execute 'select p, q, r, ' || proposition ||
           ' from P, Q, R;';
```

What happens in this statement is that we build a string that represents an SQL query by concatenating the string

```
'select p, q, r, '
```

with the string that is passed into the `truthTable` *function's* `proposition` *parameter (in this case the string*

```
'(P and (not (R and Q)))')
```

and, finally, with the string

```
' from P, Q, R;'.
```

This concatenation of strings is accomplished using the string concatenation function `||`*. As such we get the string*

```
'select p, q, r, (P and (not (R and Q))) from P, Q, R;'
```

This is a string that represents a valid SQL query which we can then evaluate with the `execute` *operation. The rest of the code, i.e., the* `return query` *statement, then returns the result of this evaluation, i.e., the truth table of the proposition* `(P and (not (R and Q)))`*.*

Now that we have the `truthTable` function, we will consider in the next example the problem of verifying if two Propositional Logic propositions are logically equivalent.

Example 2 *Consider two Propositional Logic propositions E and F. We say that E and F are logically equivalent if their respective truth tables are the same.*

We will write a boolean user-defined SQL function

```
logicallyEquivalent(E text, F text) returns boolean
```

which takes as arguments two strings that represent the propositions E and F respectively and that returns 'true' if E and F are logically equivalent, and 'false' otherwise.

The code for this function relies on the following insight. Let T_E and T_F be the truth table of E and F, respectively. Then, by definition E and F are logically equivalent, i.e., $T_E = T_F$. But the later set condition is equivalent with the condition

$$(T_E - T_F) = \emptyset \text{ and } (T_F - T_E) = \emptyset.$$

Here is the function:

```
create or replace function
   logicallyEquivalent(E text, F text)
   returns boolean as
$$
select not exists (select truthTable(E) except (select truthTable(F))) and
       not exists (select truthTable(F) except (select truthTable(E)));
$$ language sql;
```

3 Operations on polynomials and matrices using SQL aggregate functions

In the problems in this section, you will practice working with aggregate functions.

A useful other aspect of solving these problems is that you will learn how relations can be used to represent polynomials and matrices and how SQL can be used to define operations on such objects.

10. Let $P(x)$ and $Q(x)$ be 2 polynomials with integer coefficients.

    Let `P(coefficient, degree)` and `Q(coefficient, degree)` be two binary relations representing $P(x)$ and $Q(x)$, respectively. E.g., if $P(x) = 2x^2 - 5x + 5$ and $Q(x) = 4x^4 + 3x^3 + x^2 - x$ then their representations in the relations **P** and **Q** are as follows:

    P

coefficient	degree
2	2
−5	1
5	0

    Q

coefficient	degree
4	4
3	3
1	2
−1	1
0	0

    (a) Write a dynamic SQL function

    ```
    create or replace function multiplyPolynomials(polynomal1 text, polynomial p2 text)
           returns table(coefficient integer, degree integer) as
    $$
    ...
    $$ language plpgsql;
    ```

    that computes a binary relation representing the multiplication of $P(x)$ and $Q(x)$, i.e., the polynomial $P(x) * Q(x)$.

    For example, consider $P(x) = 2x^2 - 5x + 5$ and $Q(x) = 4x^4+3x^3+x^2-x$. Then $P(x)*Q(x) = (8)x^6+(6-20)x^5+(2-15+20)x^4+(-2-5+15)x^3+(5+5)x^2+(-5)x = 8x^6-14x^5+7x^4+8x^3+10x^2-5x$. So, for these polynomials, your function when called with the name of the relation representing the polynomials should return the relation

coefficient	degree
8	6
-14	5
7	4
8	3
10	2
-5	1
0	0

(b) Your solution should work for arbitrary polynomials $P(x)$ and $Q(x)$. Show how your function can be used to compute

i. the polynomial $P * Q$;

ii. the polynomial $P * P$; and

iii. the polynomial $P * (Q * P)$.

11. (**Practice problem – not graded**)

Let $P(x)$ and $Q(x)$ be 2 polynomials with integer coefficients. We can consider their *composition* $P(Q(x))$.

For example, consider $P(x) = 2x^3 - 5x + 5$ and $Q(x) = 4x^4 + 3x^3$, then

$$\begin{aligned} P(Q(x)) &= 2(Q(x))^3 - 5Q(x) + 5 \\ &= 2(4x^4 + 3x^3)^3 - 5(4x^4 + 3x^3) + 5 \\ &= 2(4x^4 + 3x^3)(4x^4 + 3x^3)(4x^4 + 3x^3) - 5(4x^4 + 3x^3) + 5 \\ &= 128x^{12} + 288x^{11} + 216x^{10} + 54x^9 - 20x^4 - 15x^3 + 5 \end{aligned}$$

and

$$\begin{aligned} Q(P(x)) &= 4(P(x))^4 + 3(Q(x))^3 \\ &= 4(2x^3 - 5x + 5)^4 + 3(2x^3 - 5x + 5) \\ &= 64x^{12} - 640x^{10} + 664x^8 + 2400x^8 - 4980x^7 - 1420x^6 + \\ &\qquad 12450x^5 - 10400x^4 - 5925x^3 + 16125x^2 - 11125x + 2875 \end{aligned}$$

(a) Write a dynamic SQL function

```
create or replace function compositionPolynomials(polynomial1 text, polynomial2 text)
   returns table(coefficient integer, degree integer) as
$$
...
$$ language plpgsql;
```

that computes a binary relation representing the composition of $P(Q(x))$.

Your solution should work for arbitrary polynomials $P(x)$ and $Q(x)$. Show how your function can be used to compute

i. the polynomial $P(Q(x))$;
ii. the polynomial $Q(P(x))$; and
iii. the polynomial $P(P(P(x)))$.

12. (**Practice problem – not graded**)

Let M be an $n \times n$ matrix of boolean values. (We will assume that $n \geq 0$.) We will use **T** to denote 'true' and **F** to denote 'false'.

For $i, j \in [1, n]$, we will denote by $M[i, j]$ the boolean value in matrix M at row i and column j. (Notice that when $n = 0$, M has no elements.)

A boolean matrix M can be represented using a relation `M` with schema

(`rw: integer`, `colmn: integer`, `value: boolean`)

and such that for each $i, j \in [1, n]$

$$(i, j, M[i, j]) \in \mathtt{M}.$$

Notice that we use the attribute names '`rw`' and '`colmn`' since the words '`row`' and '`column`' are reserved words in PostgreSQL.

For example if M is the 3×3 boolean matrix

$$M = \begin{array}{ccc} \mathbf{T} & \mathbf{F} & \mathbf{T} \\ \mathbf{F} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} \end{array}$$

then `M` is the following relation of 9 tuples:

M

rw	colmn	value
1	1	**T**
1	2	**F**
1	3	**T**
2	1	**F**
2	2	**T**
2	3	**T**
3	1	**T**
3	2	**F**
3	3	**T**

Let M and N be two $n \times n$ boolen matrices represented by the two relations `M` and `N`.

(a) Write a dynamic SQL function

```
create or replace function booleanMatrixMultiplication (M text, N text)
     returns table (rw integer, colmn integer, value boolean) as
$$
begin
    return query execute ...;
end;
$$ language plpgsql;
```

that computes a relation over schema (`rw`, `column`, `value`) that represents the matrix $M * N$ when given the names '`M`' and '`N`' of the relations that represent M and N, respectively. Your solution should work for any $n \geq 1$.

For example if M and N are by the following 3×3 boolean matrices stored as the relations

M

rw	colmn	value
1	1	**T**
1	2	**F**
1	3	**T**
2	1	**F**
2	2	**T**
2	3	**T**
3	1	**T**
3	2	**F**
3	3	**T**

N

rw	colmn	value
1	1	**T**
1	2	**T**
1	3	**T**
2	1	**F**
2	2	**F**
2	3	**F**
3	1	**T**
3	2	**F**
3	3	**T**

then your function should produce the relational representation of $M * N$, i.e., the relation

$M * N$

rw	colmn	value
1	1	**T**
1	2	**T**
1	3	**T**
2	1	**T**
2	2	**F**
2	3	**T**
3	1	**T**
3	2	**T**
3	3	**T**

Consider the `Person` relation. Then the relation `Knows`, which is a binary relation over `Person`, can be modeled by a boolean matrix M_`Knows` by using the `Person` pids as row and column indices for this matrix, and by using the value **T** at position (i, j) if $(i, j) \in$ `Knows`, and the value **F** at position (i, j) if $(i, j) \notin$ `Knows`. We can then represent this boolean matrix with a relation `M_Knows(rw integer, colmn integer, value)` by using the following SQL statements:

```
create table M_Knows(rw integer, colmn integer, value boolean);

insert into M_Knows
     select rw.pid, colmn.pid,
                exists (select 1
                        from   Knows k
                        where  rw.pid = k.pid1 and colmn.pid = k.pid2)
     from   person rw, person colmn;
```

(b) Compute the matrix `M_Knows` $*$ `M_Knows`. What does the matrix represent?

(c) `M_Knows` $*$ (`M_Knows` $*$ `M_Knows`). What does the matrix represent?

4 Triggers (Practice problems – not graded)

To begin the problems in this section, you should first remove the entire database, including the relation schemas. You should then create the relations but without specifying the primary and foreign key constraints. You should also not yet populate the relations with data.

Solve the following problems:

13. Develop appropriate insert and delete triggers that implement the primary key and foreign key constraints that are specified for the `Person`, `Company`, and `Worksfor` relations.

    Your triggers should report appropriate error conditions.

    For this problem, implement the triggers such that foreign key constraints are maintained using the **cascading delete** semantics.

    For a reference on cascading deletes associated with foreign keys maintenance consult the PostgreSQL manual page

    `https:www.postgresql.orgdocs9.2ddl-constraints.html`

    Test your triggers using appropriate inserts and deletes.

14. Repeat Problem 13 but subject to the constraint that a person may not appear in the `worksFor` relation if he or she has fewer than 2 job skils.

15. Consider the following view definition

```
create or replace view PersonIsKnownByNumberofPersons as
  (select p1.pid, p1.name,
         (select count(1)
          from   Person p2
          where (p2.pid, p1.pid) in (select k.pid1, k.pid2
                                     from Knows k)) as NumberofKnownByPersons
   from Person p1);
```

    This view defines the set of tuples (p, n, c) where p and n are the pid and name of a person and c is the number of persons who know the person with pid p.

    You should not create this view! Rather your task is to create a relation `PersonIsKnownByNumberofPersons` that maintains a

materialized view in accordance with the above view definition under insert and delete operations on the `Person` and `Knows` relation.

Your triggers should be designed to be **incremental**. In particular, when an insert or delete occurs, you should not always have to reevaluate all the number of persons who know persons. Rather the maintenance of `PersonIsKnownByNumberofPersons` should only apply to the person information that is affected by the insert or delete.

Provide tests that show that your solution works.