Assignment 5
Query Translation and Optimization

For this assignment you will need the material covered in the lectures

- Lecture 13: Translating SQL queries into RA expressions
- Lecture 14: Query optimization

For this assignment, you will need to submit a single .pdf file that contains your solutions for the problems on this assignment.

1 Translating and Optimizing SQL Queries to Equivalent RA Expressions

Using the translation algorithm presented in class, translate each of the following SQL queries into equivalent RA expressions. For each query, provide its corresponding RA expression in your .pdf file. This RA expression needs to be formulated in the standard RA notation.

Then use rewrite rules to optimize each of these RA expressions into an equivalent but optimized RA expression. Include this optimized expression in you .pdf file is standard RA notation.

You are required to specify some, but not necessarily all, of the intermediate steps that you applied during the translations and optimizations. Use your own judgment to specify the most important steps.

During the optimization, take into account the primary keys and foreign key constraints that are assumed for the Person, Company, jobSkill, worksFor, Knows, and personSkill relations.

1. "Find the pid and name of each person who works for 'Google' and who knows a person who earns a lower salary.

```
select p1.pid, p1.name
from   person p1, worksfor w1
where  p1.pid = w1.pid and w1.cname = 'Google' and
       exists (select 1
               from  person p2, worksfor w2
               where  p2.pid = w2.pid and
                      (p1.pid,p2.pid) in (select k.pid1,k.pid2 from knows k) and
                      w1.salary < w2.salary);
```

   (a) Using the translation algorithm presented in the lectures, translate this SQL into and equivalent RA expression formulated in the standard RA syntax. Specify this RA expression in your .pdf file.

   (b) Using the optimization rewrite rules presented in the lectures, including those that rely on constraints, optimize this RA expression. Specify this optimized RA expression in your .pdf file.

2. Find the pid of each person who

   - has a 'Programming' skill or a 'Networks' skill.
   - does not work for 'Amazon',
   - does not know anyone who lives in 'Indianapolis',

```
select p.pid
from   person p
where  p.pid = SOME (select ps.pid
                     from   personSkill ps
                     where  ps.skill = 'Programming' or ps.skill = 'Networks') and
        p.pid <> ALL (select w.pid
                      from   worksFor w
                      where  w.cname = 'Amazon') and
        not exists (select p1.pid
                    from   person p1
                    where  p1.city = 'Indianapolis' and
                           p1.pid in (select k.pid2 from knows k where k.pid1 = p.pid));
```

   (a) Using the translation algorithm presented in the lectures, translate this SQL into and equivalent RA expression formulated in the standard RA syntax. Specify this RA expression in your .pdf file.

   (b) Using the optimization rewrite rules in the lectures, including those that rely on constraints, optimize this RA expression. Specify this optimized RA expression in your .pdf file.

3. Find each (p_1, p_2) pair where p_1 and p_2 are pids of persons and such that p_2 is among the oldest persons who are known by p_1.

```
select p1.pid, p2.pid
from   person p1, person p2
where (p1.pid, p2.pid) in (select k.pid1, k.pid2 from knows k) and
       not p2.birthyear > SOME (select p.birthyear
                                from   person p
                                where  p.pid in (select k.pid2
                                                 from   knows k
                                                 where  k.pid1 = p1.pid));
```

   (a) Using the translation algorithm presented in the lectures, translate this SQL into and equivalent RA expression formulated in the standard RA syntax. Specify this RA expression in your .pdf file.

   (b) Using the optimization rewrite rules in the lectures, including those that rely on constraints, optimize this RA expression. Specify this optimized RA expression in your .pdf file.

2 Experiments to Test the Effectiveness of Query Optimization

In the following problems, you will conduct experiments to gain insight into whether or not query optimization can be effective. In other words, can it be determined experimentally if optimizing an SQL or an RA expression improves the time (and space) complexity of query evaluation?

You will need to use the PostgreSQL system to do you experiments. Recall that in SQL you can specify RA expression in a way that mimics it faithfully.

As part of the experiment, you might notice that PostgreSQL's query optimizer does not fully exploit all the optimization that is possible as discussed in Lecture 14.

In the following problems you will need to generate artificial data of increasing size and measure the time of evaluating non-optimized and optimized queries. The size of this data can be in the ten or hundreds of thousands of tuples. This is necessary because on very small data is it is not possible to gain sufficient insights into the quality (or lack of quality) of optimization.

Consider a binary relation `R(a int, b int)`. You can think of this relation as a graph, wherein each pair (a, b) represents and edge from a to b. (We work with directed graph. In other words edges (a, b) and (b, a) represent two different edges.) It is possible that `R` contains self-loops, i.e., edges of the form (a, a). Besides the relation R we will also use a unary relation `S(b int)`.

Along with this assignment, I have provided the code of two functions

`makerandomR(m integer, n integer, l integer)`

and

`makerandomS(n int, l int)`

```
create or replace function makerandomR(m integer, n integer, l integer)
returns void as
$$
declare  i integer; j integer;
begin
    drop table if exists Ra; drop table if exists Rb;
    drop table if exists R;
    create table Ra(a int); create table Rb(b int);
    create table R(a int, b int);

    for i in 1..m loop insert into Ra values(i); end loop;
    for j in 1..n loop insert into Rb values(j); end loop;
    insert into R select * from Ra a, Rb b order by random() limit(l);
end;
$$ LANGUAGE plpgsql;
```

```
create or replace function makerandomS(n integer, l integer)
returns void as
$$
declare i integer;
begin
    drop table if exists Sb;
    drop table if exists S;
    create table Sb(b int);
    create table S(b int);

    for i in 1..n loop insert into Sb values(i); end loop;
    insert into S select * from Sb order by random() limit (l);
end;
$$ LANGUAGE plpgsql;
```

When you run

`makerandomR(m,n,l);`

for some values m, n, and k, this function will generate a random relation instance for R with l tuples that is subset of $[1, m] \times [1, n]$. For example,

`makerandomR(3,3,4);`

might generate the relation instance

R

a	b
2	1
3	3
2	3
3	1

But, when you call

`makerandomR(3,3,4)`

again, it may now generate a different random relation such as

R

a	b
1	2
2	3
3	1
1	1

Notice that when you call

```
makerandomR(1000,1000,1000000)
```

it will make the entire relation $[1, 1000] \times [1, 1000]$ consisting of one million tuples.

The function makerandomS(n, l) will generate a random set of size l that is a subset of $[1, n]$.

Now consider the following simple query Q_1:

```
select distinct r1.a
from   R r1, R r2
where  r1.b = r2.a;
```

This query can be translated and optimized to the query Q_2:

```
select distinct r1.a
from   R r1 natural join (select distinct r2.a as b from R r2) r2;
```

Image that you have created a relation R using the function makerandomR. Then when you execute in PostgreSQL the following

```
explain analyze
select distinct r1.a
from   R r1, R r2
where  r1.b = r2.a;
```

the system will return its execution plan as well as the execution time to evaluate Q_1 measured in ms.

And, when you execute in PostgeSQL the following

```
select distinct r1.a
from   R r1 natural join (select distinct r2.a as b from R r2) r2;
```

the system will return its execution plan as well as the execution time to evaluate Q_2 measured in ms.

This permits us to compare the performance of the non-optimized query Q_1 with the optimized Q_2 for various-sized relations R.

Here are some of these comparisons for various different random relations R.

makerandomR	Q_1 (in ms)	Q_2 (in ms)
(100,100,1000)	4.9	1.5
(500,500,25000)	320.9	28.2
(1000,1000,100000)	2648.3	76.1
(2000,2000,400000)	23143.4	322.0
(5000,5000,2500000)	$--$	1985.8

The "$--$" symbol indicates that I had to stop the experiment because it was taken too long. (All the experiments where done on a MacBook pro.)

Notice the significant difference between the execution times of the non-optimized query Q_1 and the optimized query Q_2. So clearly, optimization works on query Q_1.

If you look at the query plan of PostgreSQL for Q_1, you will notice that it does a double nested loop and it therefore is $O(|R|^2)$ whereas for query Q_2 it runs in $O(|R|)$. Clearly, optimization has helped significantly.[1]

4. Now consider query Q_3:

```
select distinct r1.a
from   R r1, R r2, R r3
where  r1.b = r2.a and r2.b = r3.a;
```

   (a) Translate and optimize this query and call it Q_4. Then write Q_4 as an SQL query with RA operations just as was done for query Q_2.

   (b) Compare queries Q_3 and Q_4 in a similar way as we did for Q_1 and Q_2.

   You should experiment with different sizes for `R`. Incidentally, these relations do not need to use the same m,n, and l parameters as those shown in the above table for Q_1 and Q_2.

   (c) What conclusions can you draw from the results of these experiments?

5. Now consider query Q_5 which is an implementation of the ONLY set semijoin between `R` and `S`. (See the lecture on set semijoins for more information.)

   (Incidentally, if you look at the code for `makerandomR` you will see a relation `Ra` that provides the domain of all a values. You will need to use this relation in the queries. Analogously, in the code for `makerandomS` you will see the relation `Sb` that contains the domain of all b values.)

   In SQL, Q_5 can be expressed as follows:

```
select ra.a
from   Ra ra
where  not exists (select r.b
                   from   R r
                   where  r.a = ra.a and
                          r.b not in (select s.b from S s));
```

   (a) Translate and optimize this query and call it Q_6. Then write Q_6 as an SQL query with RA operations just as was done for Q_2 above.

   (b) Compare queries Q_5 and Q_6 in a similar way as we did for Q_1 and Q_2.

[1] It is actually really surprising that the PostgreSQL system did not optimize query Q_1 any better.

You should experiment with different sizes for R and S. (Vary the size of S from smaller to larger.) Also use the same value for the parameter n in makerandomR(m, n, l) and makerandomS(n, l) so that the maximum number of b values in R and S are the same.

(c) What conclusions can you draw from the results of these experiments?

6. Now consider query Q_7 which is an implementation of the ALL set semijoin between R and S. (See the lecture on set semijoins for more information.)

In SQL, Q_7 can be expressed as follows:

```
select ra.a
from   Ra ra
where  not exists (select s.b
                   from   S s
                   where  s.b not in (select r.b
                                      from   R r
                                      where  r.a = ra.a));
```

(a) Translate and optimize this query and call it Q_8. Then write Q_8 as an SQL query with RA operations just as was done for query Q_2 above.

(b) Compare queries Q_7 and Q_8 in a similar way as we did for Q_1 and Q_2.

You should experiment with different sizes for R and S. (Vary the size of S from smaller to larger.) Also use the same value for the parameter n in makerandomR(m, n, l) and makerandomS(n, l) so that the maximum number of b values in R and S are the same.

(c) What conclusions can you draw from the results of these experiments?

(d) Furthermore, what conclusion can you draw when you compare you experiment with those for the ONLY set semijoin in problem 5?