B561 Assignment 7

Testing Effectiveness of Query Optimization and Query Planning; Object-Relational Database Programming Key-Value Databases and Graph Databases

This assignment focuses on problems related to Lectures Lectures 18 through 23:

- Lecture 18: Algorithms for RA operations
- Lecture 19: Query processing and query plans
- Lecture 20: Object-relational database programming
- Lecture 21: Key-value stores. NoSQL in MapReduce style
- Lecture 22: Key-value stores; NoSQL in Spark style
- Lecture 23: Graph databases

Other lectures that a relevant for this assignment are Lectures 8, 13, and 14:

- Lecture 8: Translating Pure SQL queries into RA expressions
- Lecture 9: Query optimization
- Lecture 12: Object-Relational databases and queries

This assignment has problems that are required to be solved. Others, identified as such, are practice problems that you should attempt since they serve as preparation for the final exam.

Turn in a single `assignment7.sql` file that contains the PostgreSQL code of the solutions for the problem that require such code. Also turn in a `assignment7.txt` file that contains all the output associated with the problems in this assignment. For all the other problems, submit a single `assignment7.pdf` file with your solutions.

Do not include solutions for the practice problems in the `assignment7.sql` and `assignment7.pdf` files.

1 Analysis of Queries Using Query Plans

The problems in this section focus on the study of certain important queries using query plans and their associated time complexities.

1. Consider the relation schema $\mathtt{R}(\mathtt{a}\ \mathtt{int}, \mathtt{b}\ \mathtt{int})$ and the '`NO`' generalized quantifier query:

$$\{(r_1.\mathsf{a}, r_2.\mathsf{a}) \mid \mathtt{R}(r_1) \wedge \mathtt{R}(r_2) \wedge \mathtt{R}(r_1.\mathsf{a}) \cap \mathtt{R}(r_2.\mathsf{a}) = \emptyset\}$$

where

$$\begin{array}{lcl} \mathtt{R}(r_1.\mathsf{a}) & = & \{r.\mathsf{b} \mid \mathtt{R}(r) \wedge r.\mathsf{a} = r_1.\mathsf{a}\} \\ \mathtt{R}(r_2.\mathsf{a}) & = & \{r.\mathsf{b} \mid \mathtt{R}(r) \wedge r.\mathsf{a} = r_2.\mathsf{a}\}. \end{array}$$

Consider Lecture 19 '*Query Processing and Query Plans*' and in particular the analysis, using query plans, for the '`SOME`' generalized quantifier. In analogy with that analysis, do an analysis for the '`NO`' generalized quantifier.[1]

Solution:

We can formulate this query in Pure SQL as

```
select distinct r1.a, r2.a
from   R r1, R r2
where  (r1.a,r2.a) not in (select distinct r3.a, r4.a
                           from   R r3, R r4
                           where  r3.b = r4.b);
```

The query plans is a follows:

```
HashAggregate
  Group Key: r1.a, r2.a
  ->  Nested Loop
        Join Filter: (NOT (hashed SubPlan 1))
        ->  Seq Scan on r r1
        ->  Seq Scan on r r2
        SubPlan 1
          ->  HashAggregate
                Group Key: r3.a, r4.a
                ->  Hash Join
                      Hash Cond: (r3.b = r4.b)
                      ->  Seq Scan on r r3
                      ->  Hash
                            ->  Seq Scan on r r4
```

$$\underset{\pi_{R_1.a,R_2.a}}{\texttt{hashProjection}}\left(R_1 \underset{(R_1.a,R_2.a)\textbf{not in}\left(\underset{\pi_{R_3.a,R_4.a}}{\texttt{hashProjection}}\left(R_3 \underset{R_3.b=R_4.b}{\overset{\texttt{hashJoin}}{\bowtie}} R_4\right)\right)}{\overset{\texttt{nestedLoopJoin}}{\bowtie}} R_2\right)$$

[1] Note that in this problem, the relations `P`, `Q`, `R` and `S`, as they appear in the Lecture 19, have been consolidated into the relation `R`.

The time complexity of this is $O(|R|^2)$.

The query can be optimized into the following RA SQL query:

```
select r1.a, r2.a
from   (select distinct a from R) r1,
       (select distinct a from R) r2
except
select distinct r1.a, r2.a
from   R r1 join R r2 on r1.b = r2.b;
```

The query plan of this RA SQL is as follows:

```
 HashSetOp Except
   ->  Append
         ->  Subquery Scan on "*SELECT* 1"
               ->  Nested Loop
                     ->  HashAggregate
                           Group Key: r.a
                           ->  Seq Scan on r
                     ->  HashAggregate
                           Group Key: r_1.a
                           ->  Seq Scan on r r_1
         ->  Subquery Scan on "*SELECT* 2"
               ->  HashAggregate
                     Group Key: r1.a, r2.a
                     ->  Hash Join
                           Hash Cond: (r1.b = r2.b)
                           ->  Seq Scan on r r1
                           ->  Hash
                                 ->  Seq Scan on r r2
```

$$\left(\underset{\pi_a}{\texttt{hashProjection}}(R_1) \times \underset{\pi_a}{\texttt{hashProjection}}(R_2)\right) \underset{-}{\texttt{hashminus}} \left(\underset{\pi_{R_3.a,R_4.a}}{\texttt{hashProjection}}\left(R_3 \underset{R_3.b = R_4.b}{\overset{\texttt{hashJoin}}{\bowtie}} R_4\right)\right)$$

The time complexity is $O(|\pi_a(R)|^2 + |R| + |R_3 \bowtie_{R_3.b=R_4.b} R_4|)$.

2. Consider the relation schema $\texttt{R}(\texttt{a}, \texttt{b})$ and the '`NOT ONLY`' generalized quantifier query:

$$\{(r_1.\texttt{a}, r_2.\texttt{a}) \mid \texttt{R}(r_1) \wedge \texttt{R}(r_2) \wedge \texttt{R}(r_1.\texttt{a}) \not\subseteq \texttt{R}(r_2.\texttt{a})\}$$

where

$$\begin{aligned} \texttt{R}(r_1.\texttt{a}) &= \{r.\texttt{b} \mid \texttt{R}(r) \wedge r.\texttt{a} = r_1.\texttt{a}\} \\ \texttt{R}(r_2.\texttt{a}) &= \{r.\texttt{b} \mid \texttt{R}(r) \wedge r.\texttt{a} = r_2.\texttt{a}\}. \end{aligned}$$

Consider Lecture 19 '*Query Processing and Query Plans*' and in particular the analysis, using query plans, for the '`SOME`' generalized quantifier. In analogy with that analysis, do an analysis for the '`NOT ONLY`' generalized quantifier.

Solution:

```
select distinct r1.a, r2.a
from   R r1, (select a from R) r2
where  (r1.b,r2.a) not in (select r3.b, r3.a from R r3);
```

The query plan for this query is as follows:

```
  HashAggregate
   Group Key: r1.a, r.a
   ->  Nested Loop
         Join Filter: (NOT (hashed SubPlan 1))
         ->  Seq Scan on r r1
         ->  Seq Scan on r
         SubPlan 1
           ->  Seq Scan on r r3
(8 rows)
```

$$\underset{\pi_{R_1.a,R.a}}{\texttt{hashProjection}} \left(R_1 \underset{(R_1.b, R.a)\textbf{not in hashed}(R_3)}{\overset{\texttt{nestedLoopJoin}}{\bowtie}} R \right)$$

The time complexity is $O(|R|^2)$.

This query can be translated and optimized into the following RA SQL query:

```
select distinct q.a, q.r2a
from   (select r1.a, r1.b, r2.a as r2a
        from   R r1 cross join (select distinct a from R) r2
        except
        select distinct r1.a, r1.b, r2.a as r2a
        from   R r1 join R r2 on (r1.b = r2.b)) q;
```

The query plan for this RA SQL query is as follows:

```
 HashAggregate
   Group Key: q.a, q.r2a
   ->  Subquery Scan on q
         ->  HashSetOp Except
               ->  Append
                     ->  Subquery Scan on "*SELECT* 1"
                           ->  Nested Loop
                                 ->  HashAggregate
```

```
                              Group Key: r.a
                              ->  Seq Scan on r
                        ->  Seq Scan on r r1
            ->  Subquery Scan on "*SELECT* 2"
                  ->  HashAggregate
                        Group Key: r1_1.a, r1_1.b, r2.a
                        ->  Hash Join
                              Hash Cond: (r1_1.b = r2.b)
                              ->  Seq Scan on r r1_1
                              ->  Hash
                                    ->  Seq Scan on r r2
```

$$\underset{\pi_{R_1.a,R.a}}{\texttt{hashProjection}}\left(\left(R_1 \times \underset{\pi_{R.a}}{\texttt{hashProjection}}(R)\right) \underset{-}{\texttt{hashminus}} \left(\underset{\pi_{R_1.a,R_1.b,R_2.b}}{\texttt{hashProjection}}\left(R_1 \overset{\texttt{hashJoin}}{\underset{R_1.b\,=\,R_2.b}{\bowtie}} R_2\right)\right)\right)$$

The time complexity is $O(|R||\pi_a(R)| + |R_1 \bowtie_{R_1.b=R_2.b} R_2|)$.

2 Experiments to Test the Effectiveness of Query Optimization

In the following problems, you will need to conduct experiments in PostgreSQL to gain insight into whether or not query optimization is 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? Additionally, can it be determined if the PostgreSQL query optimizer attains the same (i.e., better or worse) optimization as optimization by hand. (Recall that in SQL you can specify each RA expression as an RA SQL query. This implies that each of the optimization rules for RA can be applied directly to queries formulated in RA SQL.)

To solve the 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 since, on very small data, it is not possible to gain sufficient insights into the quality (or lack of quality) of optimization. You can use the data generation functions that were developed in Assignment 6. Additionally, you are advised to examine the query plans generated by PostgreSQL.

For the problems in this section, we will use three relations:[2]

```
P(a int)
R(a int, b int)
Q(b int)
```

To generate `P` or `Q`, you should use the function `SetOfIntegers` which generate a set of up to n randomly selected integers in the range $[l, u]$:

```
create or replace function SetOfIntegers(n int, l int, u int)
     returns table (x int) as
     $$
        select floor(random() * (u-l+1) + l)::int as x
        from   generate_series(1,n)
        group by (x) order by 1;
     $$ language sql;
```

To generate `R`, you should use the function `BinaryRelationOverIntegers` which generates up to n randomly selected pairs with first components in the range $[l_1, u_1]$ and second components in the range $[l_2, u_2]$:

```
create or replace function BinaryRelationOverIntegers(n int, l_1 int, u_1 int, l_2 int, u_2 int)
    returns table (x int, y int) as
    $$
        select distinct
               floor(random() * (u_1-l_1+1) + l_1)::int as x,
               floor(random() * (u_2-l_2+1) + l_2)::int as y
        from   generate_series(1,n);
```

[2] A typical case could be where `P` is `Person`, `R` is `Knows`, and `Q` is the set of persons with the Databases skill. Another case could where `P` is the set of persons who work for Amazon, `R` is `personSkill` and `Q` is the set of skills of persons who live in Bloomington. Etc.

```
    $$ language sql;
```

Example 1 *Consider the 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 generated a relation `R`*. Then, when you execute*

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

the system will return its query plan as well as the execution time to evaluate Q_1 measured in ms. And, when you execute

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

the system will return its query plan as well as the execution time to evaluate Q_2 measured in ms. This permits us to compare the non-optimized query Q_1 with the optimized query Q_2 for various differently-sized relations R. In the following table are some of these comparisons for various differently-sized random relations `R`*. In this table,* `R` *was generated with lower and upper bounds $l_1 = l_2 = 1000$ and $u_1 = u_2 = 1000$.*[3]

R	Q_1 (in ms)	Q_2 (in ms)
10^4	27.03	7.80
10^5	3176.53	58.36
10^6	69251.58	400.54

Observe 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.

Incidentally, below are the query plans for Q_1 and Q_2. Examining these query plans should reveal why Q_1 runs much slower than Q_2. (Convince yourself about the reason behind this difference.)

[3]All the experiments in this section where done on a MacMini using PostgreSQL version 13 with default work memory of 4MB.

```
             QUERY PLAN for Q1
------------------------------------
 HashAggregate
   Group Key: r1.a
   ->  Hash Join
         Hash Cond: (r1.b = r2.a)
         ->  Seq Scan on r r1
         ->  Hash
               ->  Seq Scan on r r2
```

```
                QUERY PLAN  for query Q2
------------------------------------------
 HashAggregate
   Group Key: r1.a
   ->  Hash Join
         Hash Cond: (r1.b = r2.a)
         ->  Seq Scan on r r1
         ->  Hash
               ->  HashAggregate
                     Group Key: r2.a
                     ->  Seq Scan on r r2
```

We now turn to the problems for this section.

3. Consider query Q_3

```
select distinct r1.a
from   R r1, R r2, R r3, R r4
where  r1.b = r2.a and r2.b = r3.a and r3.b = r4.a;
```

Intuitively, if we view `R` as a graph then Q_3 determines each source node of `R` from which a path of length 4 departs. I.e., each source node n_0 such that there exists a sequence of nodes (n_1, n_2, n_3, n_4) such that (n_0, n_1), (n_1, n_2), (n_2, n_3), and (n_3, n_4) are edges in `R`.

Query Plan for Q_3.

```
HashAggregate
  Group Key: r1.a
  ->  Hash Join
        Hash Cond: (r2.b = r3.a)
        ->  Hash Join
              Hash Cond: (r1.b = r2.a)
              ->  Seq Scan on r r1
              ->  Hash
                    ->  Seq Scan on r r2
        ->  Hash
              ->  Hash Join
                    Hash Cond: (r3.b = r4.a)
                    ->  Seq Scan on r r3
                    ->  Hash
                          ->  Seq Scan on r r4
```

(a) Translate and optimize Q_3 query and call it Q_4. Express Q_4 as an RA SQL query just as was done for query Q_2 in Example 1.

Compare queries Q_3 and Q_4 in a similar way as we did for Q_1 and Q_2 in Example 1.

You should experiment with different sizes for `R`. Incidentally, these relations do not need to use the same parameters as those shown in the above table for Q_1 and Q_2 in Example 1.

Solution:

Q_4 can be written in RA SQL as follows. (Note the sequence of semi-joins (i.e., IN clauses).)

```
select distinct r1.a
from   R r1
where  r1.b in
        (select distinct r2.a
         from   R r2
         where  r2.b in
                 (select distinct r3.a
                  from   R r3
                  where  r3.b in (select distinct r4.a from R r4)));
```

The query plan for Q_4 is the following:

```
HashAggregate
  Group Key: r1.a
  ->  Hash Join
        Hash Cond: (r1.b = r2.a)
        ->  Seq Scan on r r1
        ->  Hash
              ->  HashAggregate
                    Group Key: r2.a
                    ->  Hash Join
                          Hash Cond: (r2.b = r3.a)
                          ->  Seq Scan on r r2
                          ->  Hash
                                ->  HashAggregate
                                      Group Key: r3.a
                                      ->  Hash Join
                                            Hash Cond: (r3.b = r4.a)
                                            ->  Seq Scan on r r3
                                            ->  Hash
                                                  ->  HashAggregate
                                                        Group Key: r4.a
                                                        ->  Seq Scan on r r4
```

Some experimental results:

```
 r | r_n  | r_l1 | r_u1 | r_l2 | r_u2 |    q3     |  q4
---+------+------+------+------+------+-----------+-------
 R |   41 |    1 |   10 |    1 |   10 |     1.273 | 0.148
 R |  350 |    1 |   25 |    1 |   25 |   299.375 | 0.577
 R |  813 |    1 |   50 |    1 |   50 |  1085.913 |  1.38
 R | 1683 |    1 |   75 |    1 |   75 |  5987.674 |  2.48
 R | 2309 |    1 |   75 |    1 |   75 | 23743.856 |  3.37
```

(b) What conclusions do you draw from the results of these experiments regarding the effectiveness of query optimization in PostgreSQL and/or by hand?

Solution

Clearly optimization has significantly improved query evaluation. The optimization to semi joins of the original query results into a RA SQL query that has complexity $O(|R|)$. This is as opposed to the original query which has complexity of order $|R|^4$ mainly due to the space complexity which can of this order of magnitude.

4. Consider the query Q_5

```
select p.a
from   P p
where  p.a not in (select r.a
from   R r
where  r.b not in (select q.b
                   from   Q q));
```

(a) Translate and optimize query Q_5 and call it Q_6. Express Q_6 as an RA SQL query just as was done for Q_2 in Example 1. Compare queries Q_5 and Q_6 in a similar way as we did in Example 1. However, now you should experiment with different sizes or properties for `P`, `R` and `Q`. You might consider how `P` and `Q` interact with `R`. [4]

??? **Solution**:

The query plan for Q_5 is

```
Hash Anti Join
  Hash Cond: (p.a = r.a)
  ->  Seq Scan on p
  ->  Hash
        ->  Seq Scan on r
              Filter: (NOT (hashed SubPlan 1))
              SubPlan 1
                ->  Seq Scan on s
```

The complexity of Q_5 is $O(|P| + |R| + |S|)$. So we can expect very fast query processing.

Q_6 can be written as an optimized RA SQL query as follows.[5]

```
select p.a
from   P p
where  p.a not in (select distinct r.a
                   from   R r
                   where  r.b not in (select s.b
                                      from   S s));
```

The query plan for Q_6 is as follows:

```
Seq Scan on p
  Filter: (NOT (hashed SubPlan 2))
  SubPlan 2
    ->  Seq Scan on r
          Filter: (NOT (hashed SubPlan 1))
          SubPlan 1
            ->  Seq Scan on s
```

The complexity of Q_6 is $O(|P| + |R| + |S|)$. So we can expect very fast query processing.

Here are some experimental comparison of Q_5 and Q_6.

[4]For example, if the '`a`' attribute in `R` is a foreign key referencing the '`a`' attribute in `P`, then Q_5 can be optimized in a different way than if this were not the case. Another interesting case is when $|P|$ or $|Q|$, or both, are small relative to $|\pi_{\texttt{a}}(\texttt{R})|$ or $|\pi_{\texttt{b}}(\texttt{R})|$, respectively.

[5]Note the anti semi joins (NOT IN clauses). In standard RA notation, Q_6 can be expressed as $P \overline{\ltimes} (R \overline{\ltimes} S)$.

```
 r |  r_n   | r_l1 |  r_u1   | r_l2 | r_u2 | p | p_n  | p_l |  p_u   | s | s_n | s_l | s_u |   q5    |    q6
---+--------+------+---------+------+------+---+------+-----+--------+---+-----+-----+-----+---------+-----------
 R |    252 |    1 |      20 |    1 |   20 | P |   11 |   1 |     20 | S |  14 |   1 |  20 |   0.085 |     0.076
 R |  90637 |    1 |  100000 |    1 |    5 | P |  995 |   1 | 100000 | S |   3 |   1 |   5 |  18.759 |    20.354
 R | 906333 |    1 | 1000000 |    1 |    5 | P |  994 |   1 | 100000 | S |   4 |   1 |   5 | 167.199 |   256.048
 R | 975401 |    1 | 1000000 |    1 |   20 | P | 9558 |   1 | 100000 | S |  12 |   1 |  20 | 248.941 |   441.849
 R | 975445 |    1 | 1000000 |    1 |   20 | P | 9539 |   1 | 100000 | S |  14 |   1 |  20 | 212.353 |   372.721
```

(b) What conclusions do you draw from the results of these experiments regarding the effectiveness of query optimization in PostgreSQL and/or by hand?

Solution: As expected from the analysis, the queries run very fast (nearly linear) and, relative to each other, they run in approximately the same time. Indeed, Q_5 and Q_6 have very similar query plans, i.e., that for the RA expression $P\overline{\ltimes}(R\overline{\ltimes}S)$. We would expect therefore very little difference in execution time. That is substantiated by some experiments. Actually, Q_5 performs slightly better.

5. Repeat Problem 4 but now when Q_5 is expressed as

```
select p.a
from   P p
where  true = all (select p.a != r.a or true = some (select r.b = q.b
                                                      from   Q q)
                   from   R r);
```

6. Consider the query Q_7

```
select p.a
from   P p
where  not exists (select 1
                   from   Q q
                   where  (p.a, q.b) not in (select r.a, r.b
                                             from   R r));
```

(a) Translate and optimize query Q_7 and call it Q_8. Express Q_8 as an RA SQL query just as was done for Q_2 in Example 1.
Compare queries Q_7 and Q_8 in a similar way as we did In Example 1. However, now you should experiment with different sizes for `P`, `R` and `Q`. You might consider how `P` and `Q` interact with `R`. (See footnote 4.)
Solution: The query plan for Q_7 is the following:

```
Nested Loop Anti Join
  Join Filter: (NOT (SubPlan 1))
  ->  Seq Scan on p
  ->  Materialize
        ->  Seq Scan on s
  SubPlan 1
    ->  Seq Scan on r
          Filter: (p.a = a)
```

The complexity of the query plan is $O(|P||S||R|)$.
Query Q_7 can be translated to the RA SQL query Q_8:

```
select p.a
from   P p
except
select distinct a
from (select a, b
      from   P cross join S
      except
      select a, b
      from   R) q;
```

This query corresponds to the RA expression

$$P - \pi_a((P \times S) - R).$$

The query plan for Q_8 is as follows:

```
HashSetOp Except
  ->  Append
        ->  Subquery Scan on "*SELECT* 1"
              ->  Seq Scan on p
        ->  Subquery Scan on "*SELECT* 2"
              ->  Unique
                    ->  Subquery Scan on q
                          ->  SetOp Except
                                ->  Sort
                                      Sort Key: "*SELECT* 1_1".a, "*SELECT* 1_1".b
                                      ->  Append
                                            ->  Subquery Scan on "*SELECT* 1_1"
                                                  ->  Nested Loop
                                                        ->  Seq Scan on s
                                                        ->  Materialize
                                                              ->  Seq Scan on p p_1
                                            ->  Subquery Scan on "*SELECT* 2_1"
                                                  ->  Seq Scan on r
```

The complexity of this query plan $O(|R| + |P||S|)$.
Here are some experimental results

```
 r |  r_n   | r_l1 | r_u1 | r_l2 | r_u2 | p | p_n | p_l | p_u  | s | s_n | s_l | s_u  |    q7    |   q8
---+--------+------+------+------+------+---+-----+-----+------+---+-----+-----+------+----------+---------
 R | 95111 |    1 | 1000 |    1 | 1000 | P |  92 |    1 | 1000 | S |  95 |    1 | 1000 |  458.608 |  56.085
 R | 95209 |    1 | 1000 |    1 | 1000 | P | 181 |    1 | 1000 | S | 173 |    1 | 1000 |  829.009 |  77.706
 R | 95114 |    1 | 1000 |    1 | 1000 | P | 322 |    1 | 1000 | S | 326 |    1 | 1000 | 1479.932 | 151.463
 R | 95185 |    1 | 1000 |    1 | 1000 | P | 548 |    1 | 1000 | S | 554 |    1 | 1000 | 2467.343 | 361.015
 R | 139213 |    1 | 1000 |    1 | 1000 | P | 635 |    1 | 1000 | S | 634 |    1 | 1000 | 4193.396 | 498.034
 R | 156310 |    1 | 1000 |    1 | 1000 | P | 640 |    1 | 1000 | S | 620 |    1 | 1000 | 4858.072 | 499.583
 R | 181310 |    1 | 1000 |    1 | 1000 | P | 657 |    1 | 1000 | S | 661 |    1 | 1000 | 5674.192 | 546.535
 R | 221344 |    1 | 1000 |    1 | 1000 | P | 631 |    1 | 1000 | S | 629 |    1 | 1000 | 7232.525 | 525.214
 R | 236540 |    1 | 1000 |    1 | 1000 | P | 637 |    1 | 1000 | S | 618 |    1 | 1000 | 7487.404 | 504.277
 R | 259059 |    1 | 1000 |    1 | 1000 | P | 648 |    1 | 1000 | S | 644 |    1 | 1000 | 8543.512 |  559.26
```

(b) What conclusions do you draw from the results of these experiments?
Clearly optimization has helped. Instead of $O(|P||S||R|)$ performance we get $O(|P||S| + |R|)$ performance.

7. Give a brief comparison of your results for Problem 4 and Problem 6. In particular, explain why you notice differences or similarities when you make this comparison. Considering the query plans for the queries involved may aid in this comparison.

??

Below are some experimental results comparing Q_5, Q_6, Q_7, Q_8. Recall the various complexities:

- Q_5: $O(|R| + |P| + |S|)$
- Q_6: (Q_5 optimized) $O(|R| + |P| + |S|)$
- Q_7: $O(|R||P||S|)$
- Q_85: $O(|R| + |P||S|)$

```
r |  r_n   | r_l1 |  r_u1   | r_l2 | r_u2 | p | p_n | p_l |  p_u   | s | s_n | s_l | s_u  |   q5   |   q6    |    q7    |   q8
---+--------+------+---------+------+------+---+-----+-----+--------+---+-----+-----+------+--------+---------+----------+---------
R |  95146 |    1 |    1000 |    1 | 1000 | P | 562 |   1 |   1000 | S | 583 |   1 | 1000 |  20.31 |  19.935 | 2493.125 | 382.874
R |  95196 |    1 |    1000 |    1 | 1000 | P | 544 |   1 |   1000 | S | 556 |   1 | 1000 | 20.859 |  20.388 | 2369.925 | 357.332
R |  90647 |    1 |  100000 |    1 |    5 | P | 992 |   1 | 100000 | S |   3 |   1 |    5 | 18.571 |  31.278 | 4222.486 |  50.681
R | 906205 |    1 | 1000000 |    1 |    5 | P | 993 |   1 | 100000 | S |   3 |   1 |    5 | 220.13 | 398.748 |  43864.4 | 365.652
```

The most interesting comparison is between Q_6 and Q_8. Q_6 is a optimized formulation of an formulation of the query

$$\{a|P(a)\ \ \&\ \ R(a) \subseteq S\}$$

i.e., a subset (all) set semi-join and Q_8 is a optimized formulation of an formulation of the query

$$\{a|P(a)\ \ \&\ \ S \subseteq R(a)\}$$

i.e., a superset (only) set semi-join. Even though these queries appear alike, in terms of the theoretical complexity, Q_6 exhibits a additive performance in terms of $|P|$ and $|S|$ (i.e. easy to evaluate), whereas Q_8 exhibits a multiplicative performance in terms of $|P|$ and $|S|$ (i.e. expensive evaluation). So the 'all' set semi-join query is considerable easier to evaluate that the 'only' set semi-join. This behavior is especially clear when S is large. When S is small, the behavior of Q_6 and Q_8 is similar.

3 Object Relational Programming

The following problems require you to write object relational programs.

8. Consider the relation schema `V(node int)` and `E(source int, target int)` representing the schema for storing a directed graph G with nodes in `V` and edges in `E`.

   Now let G be a directed graph that is **acyclic**, i.e., a graph without cycles.[6]

   A *topological sort* of an acyclic graph G is a list of **all** nodes $(n_1, n_1, \ldots, n_k)$ in `V` such that for each edge (m, n) in `E`, node m occurs before node n in this list.

   (a) Write a PostgreSQL program `someTopologicalSort()` that returns some topological sort of G.

   (b) Write a PostgreSQL program `allTopologicalSorts()` that returns a table of all topological sorts of G.

   For this problem, your solution can use arrays.

9. Suppose you have a weighted (directed) graph G stored in a ternary table with schema

   ```
   Graph(source int, target int, weight int)
   ```

   A triple (s, t, w) in Graph indicates that G has an edge (s, t) whose edge weight is w. (In this problem, we will assume that each edge weight is a positive integer.)

   Below is an example of a graph G.

`Graph` G

source	target	weight
0	1	2
1	0	2
0	4	10
4	0	10
1	3	3
3	1	3
1	4	7
4	1	7
2	3	4
3	2	4
3	4	5
4	3	5
4	2	6

[6] A cycle is a path $(n_0, \ldots, n_l)$ where $n_0 = n_l$.

Without using arrays[7], implement Dijkstra's Shortest Path Algorithm as a PostgreSQL function `Dijkstra(s integer)` to compute the shortest path lengths (i.e., the distances) from some input vertex s in G to all other vertices in G. `Dijkstra(s integer)` should accept an argument s, the source vertex, and outputs a table which represents the pairs (t, d) where d is the shortest distance from s to t in graph G. To test your procedure, you can use the graph shown above.

For example, when you apply `Dijkstra(0)`, you should obtain the following table:

target	shortestDistance
0	0
1	2
2	9
3	5
4	9

10. Consider the following relational schemas. (You can assume that the domain of each of the attributes in these relations is `int`.)

```
partSubpart(pid,sid,quantity)
basicPart(pid,weight)
```

A tuple (p, s, q) is in `partSubPart` if part s occurs q times as a **direct** subpart of part p. For example, think of a car c that has 4 wheels w and 1 radio r. Then $(c, w, 4)$ and $(c, r, 1)$ would be in `partSubpart`. Furthermore, then think of a wheel w that has 5 bolts b. Then $(w, b, 5)$ would be in `partSubpart`.

A tuple (p, w) is in `basicPart` if basic part p has weight w. A basic part is defined as a part that does not have subparts. In other words, the pid of a basic part does not occur in the pid column of `partSubpart`.

(In the above example, a bolt and a radio would be basic parts, but car and wheel would not be basic parts.)

We define the *aggregated weight* of a part inductively as follows:

(a) If p is a basic part then its aggregated weight is its weight as given in the `basicPart` relation

(b) If p is not a basic part, then its aggregated weight is the sum of the aggregated weights of its subparts, each multiplied by the quantity with which these subparts occur in the `partSubpart` relation.

Example tables: The following example is based on a desk lamp with `pid` 1. Suppose a desk lamp consists of 4 bulbs (with `pid` 2) and a frame

[7] I.e., you can only use relations that contain tuples whose components are integer values.

(with `pid` 3), and a frame consists of a post (with `pid` 4) and 2 switches (with `pid` 5). Furthermore, we will assume that the weight of a bulb is 5, that of a post is 50, and that of a switch is 3.

Then the `partSubpart` and `basicPart` relation would be as follows:

partSubPart

pid	sid	quantity
1	2	4
1	3	1
3	4	1
3	5	2

basicPart

pid	weight
2	5
4	50
5	3

Then the aggregated weight of a lamp is $4 \times 5 + 1 \times (1 \times 50 + 2 \times 3) = 76$.

(a) **Without using arrays** (see footnote 6), write a **recursive** function `recursiveAggregatedWeight(p integer)` that returns the aggregated weight of a part `p`. Test your function for all parts.

(b) **Without using arrays** (see footnote 6), write a **non-recursive** function `nonRecursiveAggregatedWeight(p integer)` that returns the aggregated weight of a part `p`. Test your function for all parts.

11. Consider the relation schema `personSkills(pid int, skills text[])` representing a relation of pairs (p, S) where p is the unique pid of a person and S denotes the set of skills of that person.

Let **Skill** denote the set of all possible skills and let t be a non-negative integer (i.e, $t \geq 0$) denoting a *threshold*. Let $X \subseteq \textbf{Skill}$. We say that X is t-frequent if

$$|\{p \mid (p, S) \in \texttt{personSkill} \text{ and } X \subseteq S\}| \geq t$$

In other words, X is *t-frequent* if there are at least t persons who each have all the job skills in X.

Write a PostgreSQL program `frequentSets(t int)` that returns the set of all t-frequent sets.

In a good solution for this problem, you should use the following rule: if X is not t-frequent then any set Y such that $Y \supseteq X$ is not t-frequent either. In the literature, this is called the *Apriori* rule of the frequent itemset mining problem. This rule can be used as a pruning rule. In other words, if you have determined that a set X in not t-frequent then you no longer have to consider any of X's supersets Y.

To learn more about this problem you can visit the site
`https://en.wikipedia.org/wiki/Apriori_algorithm`.

Test your function `frequentSets` for thresholds 0 through 5.

For this problem, your solution can use arrays.

12. Consider a directed graph G stored in a relation `Graph(source int, target int)`. We say that G is *Hamiltonian* if G has a cycle $(n_1, \dots n_k)$ such that each node n in G occurs once, but only once, as a node in this cycle.

    (a) Write a **recursive** function `recursiveHamiltonian()` that returns `true` if the graph stored in `Graph` is Hamiltonian, and `false` otherwise. Test your function.

    (b) Write a **non-recursive** function `nonRecursiveHamiltonian` that returns `true` if the graph stored in `Graph` is Hamiltonian, and `false` otherwise. Test your function.

    For this problem, your solution can use arrays.

13. Consider the relation schema `Graph(source int, target int)` representing the schema for storing a directed graph G of edges.

    Let 'red', 'green', and 'blue' be 3 colors. We say that G is *3-colorable* if it is possible to assign to each vertex of G one of these 3 colors provided that, for each edge (s, t) in G, the color assigned to s is different than the color assigned to t.

    Write a PostgreSQL program `threeColorable()` that returns true if G is 3-colorable, and false otherwise. Test your function.

    For this problem, your solution can use arrays.

14. For this problem, first read about the k-means clustering problem in

    `http://stanford.edu/~cpiech/cs221/handouts/kmeans.html`

    Additionally, look at the k-means algorithm there described. Your task is to implement this algorithm in PostgreSQL for a dataset that consists of a set points in a 2-dimensional space.

    Assume that the dataset is stored in a ternary relation with schema

    `dataSet(p int, x float, y float)`

    where `p` is an integer uniquely identifying a point (`x`, `y`).

    Without using arrays (see footnote 6), write a PostgreSQL program `kMeans(k integer)` that returns a set of k points that denote the centroids for the points in `dataSet`. Note that k is an input parameter to the `kMeans` function.

    You will need to reason about how to determine when the algorithm terminates. (For hints, consider the above website.)

    Test your function for different values of k.

4 Key-value Stores (MapReduce and Spark)

All the problems in this section are practice problems.

4.1 MapReduce

Consider the document "MapReduce and the New Software Stack" available in the module on MapReduce.[8] In that document, you can, in Sections 2.3.3-2.3.7, find descriptions of algorithms to implement relational algebra operations in MapReduce. (In particular, look at the mapper and reducer functions for various RA operators.)

Remark 1 *Even though MapReduce as a top-level programming language is only rarely used, it still serves as an underlying programming environment to which other languages compile. Additionally, the programming techniques of applying maps to key-value stores and reducing (accumulating, aggregating) intermediate and final results is an important feature of parallel and distributed data processing. Additionally, the MapReduce framework forces one to reason about modeling data towards key-value stores. Finally, the fact that the MapReduce programming model can be entirely simulated in the PostgreSQL object-relational system underscores again the versatility of this system for a broad range of database programming and application problems.*

In the following problems, you are asked to write MapReduce programs that implement some RA operations and queries with aggregation in PostgreSQL. In addition, you need to add the code which permits the PostgreSQL simulations for these MapReduce programs.

Discussion A crucial aspect of solving these problems is to develop an appropriate data representation for the input to these problems. Recall that in MapReduce the input is a **single** binary relation of $(key, value)$ pairs.

We will now discuss a general method for representing (encoding) a relational database in a single key-value store. Crucial in this representation is the utilization of `json` objects.[9]

Consider a relation `R(a,b,c)`. For simplicity, we will assume that the domain of the attributes of `R` is integer.[10]

```
create table R (a int, b int, c int);
insert into R values (1,2,3), (4,5,6), (1,2,4);
table R;
```

[8]This is Chapter 2 in *Mining of Massive Datasets* by Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman.

[9]Incidentally, this modeling technique is independent of MapReduce and can also be used to map relational data to other systems and programming languages that center around `json` objects.

[10]However, this approach can be generalized for other domains such as string, booleans, etc.

```
 a | b | c
-----+---+---
 1 | 2 | 3
 4 | 5 | 6
 1 | 2 | 4
```

Starting from this relation `R` we can, using `jsonb`[11] functions and operations on `jsonb` objects, come up with an encoding of `R` as a key-value store. Consider the tuple

$$(1, 2, 3)$$

in `R`. We will represent (encode) this tuple as the key-value pair

`('R',{"a":1, "b":2, "c":3})`.

So the key of this pair is the relation name 'R' and the `jsonb` object `{"a": 1, "b":2, "c": 1}` represents the tuple $(1, 2, 3)$. Based on this idea of representing tuples of `R`, we can generate the entire key-value store for `R` using an object-relational SQL query.[12] To that end, we can use the `jsonb_build_object` PostgreSQL function.

```
create table encodingofR (key text, value jsonb);

insert into encodingofR
  select 'R' as key, jsonb_build_object('a', r.a, 'b', r.b, 'c', r.c) as value
  from   R r;
```

This gives the following encoding for `R`.

```
table encodingofR;

key |            value
-----+----------------------------
R   | {"a" : 1, "b" : 2, "c" : 3}
R   | {"a" : 4, "b" : 5, "c" : 6}
R   | {"a" : 1, "b" : 2, "c" : 4}
```

Note that we can also "decode" the `encodingofR` key-value store to recover `R` by using the following object-relational SQL query. To that end, we can use the `jsonb` selector function `->`.

```
select p.value->'a' as a, p.value->'b' as b, p.value->'c' as c
from   encodingofR p;

a | b | c
-----+---+---
1 | 2 | 3
4 | 5 | 6
1 | 2 | 4
```

An important aspect of this encoding strategy is that it is possible to put multiple relations, possible with different schemas and arities, into the same key-value store. Besides `R`, let us also consider a binary relation `S(a,d)`.

[11] PostgreSQL support both `json` and `jsonb` objects. For this assignment, you should use the `jsonb` object type since it comes with more functionality and offers more efficient computation.

[12] Notice that this strategy works in general for any relation, independent of the number of attributes of the relation.

```
create table S (a int, d int);
insert into S values (1,2), (5,6), (2,1), (2,3);
table S;

a | d
-----+
1 | 2
5 | 6
2 | 1
2 | 3
(4 rows)
```

We can now encode both R and S into a single key-value store encodingofRandS as follows:

```
create table encodingofRandS(key text, value jsonb);

insert into encodingofRandS
  select 'R' as key, jsonb_build_object('a', r.a, 'b', r.b, 'c', r.c) as value
  from   R r
  union
  select 'S' as key, jsonb_build_object('a', s.a, 'd', s.d) as value
  from   S s
  order by 1, 2;

table encodingofRandS;

key |          value
-----+-------------------------
R   | {"a": 1, "b": 2, "c": 3}
R   | {"a": 1, "b": 2, "c": 4}
R   | {"a": 4, "b": 5, "c": 6}
S   | {"a": 1, "d": 2}
S   | {"a": 2, "d": 1}
S   | {"a": 2, "d": 3}
S   | {"a": 5, "d": 6}
(7 rows)
```

Furthermore, we can decode this key-value store using 2 object-relational SQL queries and recover R and S.

```
select p.value->'a' as a, p.value->'b' as b, p.value->'c' as c
from   encodingofRandS p
where  p.key = 'R';

a | b | c
-----+---+---
1 | 2 | 3
4 | 5 | 6
1 | 2 | 4
(3 rows)

select p.value->'a' as a, p.value->'d' as d
from   encodingofRandS p
where  p.key = 'S';

a | d
-----+
1 | 2
5 | 6
2 | 1
2 | 3
(4 rows)
```

Example 2 *Consider the following problem. Write, in PostgreSQL, a basic MapReduce program, i.e., a* `mapper` *function and a* `reducer` *function, as well as a 3-phases simulation that implements the set intersection of two unary relations* `R(a)` *and* `S(a)`*, i.e., the relation* `R` $\cap$ `S`*. You can assume that the domain of the attribute '*`a`*' is integer.*

```
-- EncodingOfRandS;
drop table R; drop table S;

create table R(a int);
insert into R values (1),(2),(3),(4);
create table S(a int);
insert into S values (2),(4),(5);

drop table EncodingOfRandS;
create table EncodingOfRandS(key text, value jsonb);

insert into EncodingOfRandS
   select 'R' as key, jsonb_build_object('a', r.a) as value
   from   R r
   union
   select 'S' as key, jsonb_build_object('a', s.a) as value
   from   S s
   order by 1;

table EncodingOfRandS;

 key |  value
-----+----------
 R   | {"a": 1}
 R   | {"a": 4}
 R   | {"a": 2}
 R   | {"a": 3}
 S   | {"a": 4}
 S   | {"a": 5}
 S   | {"a": 2}
(7 rows)

-- mapper function
CREATE OR REPLACE FUNCTION mapper(key text, value jsonb)
RETURNS TABLE(key jsonb, value text) AS
$$
    SELECT value, key;
$$ LANGUAGE SQL;

-- reducer function
CREATE OR REPLACE FUNCTION reducer(key jsonb, valuesArray text[])
RETURNS TABLE(key text, value jsonb) AS
$$
    SELECT 'R intersect S'::text, key
    WHERE  ARRAY['R','S'] <@ valuesArray;
```

```
$$ LANGUAGE SQL;

-- 3-phases simulation of MapReduce Program followed by a decoding step
WITH
Map_Phase AS (
    SELECT m.key, m.value
    FROM   encodingOfRandS, LATERAL(SELECT key, value FROM mapper(key, value)) m
),
Group_Phase AS (
    SELECT key, array_agg(value) as value
    FROM   Map_Phase
    GROUP  BY (key)
),
Reduce_Phase AS (
    SELECT r.key, r.value
    FROM   Group_Phase, LATERAL(SELECT key, value FROM reducer(key, value)) r
)
SELECT p.value->'a' as a FROM Reduce_Phase p
order by 1;

a
---
2
4
(2 rows)
```

We now turn to the problems for this section.

15. **Practice problem–not graded**.

    Write, in PostgreSQL, a basic MapReduce program, i.e., a `mapper` function and a `reducer` function, as well as a 3-phases simulation that implements the symmetric difference of two unary relations `R(a)` and `S(a)`, i.e., the relation $(\texttt{R}-\texttt{S})\cup(\texttt{S}-\texttt{R})$. You can assume that the domain of the attribute '`a`' is integer.

16. **Practice problem–not graded**.

    Write, in PostgreSQL, a basic MapReduce program, i.e., a mapper function and a reducer function, as well as a 3-phases simulation that implements the anti semijoin of two relations R(A,B) and S(A,B,C), i.e., the relation $R \overline{\ltimes} S$. You can assume that the domains of A, B, and C are integer. Use the encoding and decoding methods described above.

17. **Practice problem–not graded**. Write, in PostgreSQL, a basic MapReduce program, i.e., a mapper function and a reducer function, as well as a 3-phases simulation that implements the natural join $R \bowtie S$ of two relations R(A, B) and S(B,C). You can assume that the domains of A, B, and C are integer. Use the encoding and decoding methods described above.

18. **Practice problem–not graded**.

    Write, in PostgreSQL, a basic MapReduce program, i.e., a mapper function and a reducer function, as well as a 3-phases simulation that implements the SQL query

```
SELECT r.A, array_agg(r.B), sum(r.B)
FROM   R r
GROUP  BY (r.A)
HAVING COUNT(r.B) < 3;
```

Here R is a relation with schema (A, B). You can assume that the domains of A and B are integers. Use the encoding and decoding methods described above.

4.2 Spark

We now turn to some problems that relate to query processing in `Spark`. Note that in `Spark` it is possible to operate on multiple key-value stores.

19. Let $R(K, V)$ and $S(K, W)$ be two binary key-value pair relations. You can assume that the domains of K, V, and W are integers. Consider the cogroup transformation `R.cogroup(S)` introduced in the lecture on `Spark`.

    (a) **Practice problem–not graded**. Define a PostgreSQL view `coGroup` that computes a complex-object relation that represent the co-group transformation `R.cogroup(S)`. Show that this view works.

    (b) **Practice problem–not graded**. Write a PostgreSQL query that use this `coGroup` view to compute the anti semijoin $R \overline{\ltimes} S$, in other words compute the relation $R - R \bowtie \pi_K(S)$.

    (c) **Practice problem–not graded**. Write a PostgreSQL query that uses this `coGroup` view to implement the SQL query

    ```
    SELECT distinct r.K as rK, s.K as sK
    FROM   R r, S s
    WHERE  ARRAY(SELECT r1.V
                 FROM   R r1
                 WHERE  r1.K = r.K) <@ ARRAY(SELECT s1.W
                                            FROM   S s1
                                            WHERE  s1.K = s.K);
    ```

20. Let `A(x)` and `B(x)` be the schemas to represent two set of integers A and B. Consider the `cogroup` transformation introduced in the lecture on `Spark`. Using an approach analogous to the one in Problem 19 solve the following problems:[13]

    (a) **Practice problem–not graded**. Write a PostgreSQL query that uses the cogroup transformation to compute $A \cap B$.

[13] An important aspect of this problem is to represent A and B as a key-value stores.

(b) **Practice problem–not graded**. Write a PostgreSQL query that uses the cogroup operator to compute the symmetric difference of A and B, i.e., the expression

$$(A - B) \cup (B - A).$$

5 Graph query languages

All the problems in this section are practice problems.

21. Consider the following relational database schema.[14]

```
Person(pid, pname, city)
Company(cname,headquarter)
Skill(skill)
worksFor(pid, cname, salary)
companyLocation(cname,city)
personSkill(pid,skill)
hasManager(eid,mid)
Knows(pid1,pid2)
```

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 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`

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

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`

(a) **Practice problem–not graded**. Specify an Entity-Relationship Diagram that models this database schema.

Solution:

The ER diagram has 4 *entities*:

- `Person` with attributes `pid`, `name`, and `birthyear`; `pid` is its primary key.
- `Company` with attribute `cname`; `cname` is its primary key.
- `jobSkill` with attribute `skill`; `skill` is its primary key.
- `Location` with attribute `city`; `city` is its primary key.

The ER diagram has 3 *binary many-to-many relationships*:

- `Knows` with participating entity `Person` in two roles: `pid1` and `pid2`.
- `personSkill` with participating entities `Person` and `jobSkill`.
- `companyLocation` with participating entities `Company` and `Location`.

The ER diagram has 2 *binary many-to-one relationships (i.e. binary relationships that are functions)*:

- `worksFor` from `Person` to `Company`. `worksFor` has as attribute `salary`.
- `livesIn` from `Person` to `Location`.

(b) **Practice problem–not graded**. Specify the node and relationship types of a Property Graph for this database schema. In addition, specify the properties, if any, associated with each such type.

Solution

The solution of this problem mirrors that for Problem 21a. The property graph model has 4 *node types*:

- `Person` with properties `pid`, `name`, and `birthyear`.
- `Company` with property `cname`.
- `jobSkill` with property `skill`.
- `Location` with property `city`.

The property graph model has 5 *relationship types*:

- `Knows` from `Person` to `Person`.
- `personSkill` from `Person` to `jobSkill`.
- `companyLocation` from `Company` to `Location`.
- `worksFor` from `Person` to `Company`; `worksFor` has property `salary`.
- `livesIn` from `Person` to `Location`.

22. Using the Property Graph model in Problem 21b, formulate the following queries in the Cypher query language:

(a) **Practice problem–not graded**. Find the types of the relationships associated with `Person` nodes.

Solution:

```
MATCH (p: Person) -[r]-> (n)
RETURN type(r)
```

(b) **Practice problem–not graded**. Find each person (node) whose name is ‘John’ and has a salary that is at least 50000.

Solution:

```
MATCH (p: Person {name: 'John'}) -[w: worksFor]-> (c: Company)
WHERE w.salary >= 50000
RETURN id(p)
```

(c) **Practice problem–not graded**. Find each Skill (node) that is the skill of a person who knows a person who works for 'Amazon' and who has a salary that is at least 50000.

Solution:

```
MATCH (j:jobSkill) <-[:personSkill]- (p1:Person) -[:Knows]-> (p2:Person) -[w:worksFor]->(:Company {cname: 'Amazon'})
WHERE w.salary >= 50000
RETURN j.skill
```

(d) **Practice problem–not graded**. Find each person (node) who knows directly or indirectly (i.e., recursively) another person who works for Amazon.
Solution:

```
MATCH (p1:Person) -[:Knows*]->(p2:Person) -[:worksFor]-> (:company {cname:'Amazon'})
RETURN id(p1)
```

(e) **Practice problem–not graded**. Find for each company node, that node along with the number of persons who work for that company and who have both the Databases and Networks skills.
Solution:

```
MATCH (c:Company) <-(w:worksFor) -(p:Person)-> [personSkill]-> (:jobSkill {skill:'Databases'}),
      (c:Company) <-(w:worksFor) -(p:Person)-> [personSkill]-> (:jobSkill {skill:'Networks'})
RETURN id(c.cname), count(p)
```