B561 Assignment 7

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

Dirk Van Gucht

This assignment focuses on problems related to Lecture 9 and Lectures 18 through 22.

- Lecture 18: Algorithms for RA operations
- Lecture 19: Query processing and query plans
- Lecture 20: Object-relational database programming
- Lecture 20: Key-value stores. NoSQL in MapReduce style
- Lecture 21: Key-value stores; NoSQL in Spark style
- Lecture 22: 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 13: Object-Relational databases and queries
- Lecture 14: Nested Relational, Semi-structured Databases, Document Databases

This assignment has problems that are required to be solved. Others, identified as such, are practice problems that you should attempt since the 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. (Do not include solutions for the practice problems.) 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.

1 Analysis of Queries Using Query Plans

Consider Lecture 19 on Query Processing: Query Planning in (Object) Relational Systems. Consider the analysis, using query plans, for the `SOME` quantifier.

1. Assume the relation schemas $P(x)$, $Q(x)$, $R(x, y)$ and $S(x, z)$.

   Consider the `NOT ALL` generalized query

   $$\{(p.x, q.x) | P(p) \wedge Q(p) \wedge R(p.x) \not\supset S(q.x)\}$$

   where

   $$\begin{array}{rcl} R(p.x) & = & \{r.y \mid R(r) \wedge r.x = p.x\} \\ S(q.x) & = & \{s.z \mid S(s) \wedge s.x = q.x\} \end{array}$$

   Consider Lecture 19 on *Query Processing: Query Planning in (Object) Relational Systems* and in particular the analysis, using query plans, for the `SOME` generalized quantifier.

   Now to the problem. In analogy with the analysis for the `SOME` generalized quantifier, do an analysis for the `NOT ALL` generalized quantifier.

   Solution: The solution can be found in the Lecture on Query Processing. There, I added some of the cases for this situation.

2 Experiments to Test the Effectiveness of Query Optimization

In the following problems, you will conduct experiments in PostgreSQL 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? 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.

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. 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 assignments, we will use three relations:[1]

[1] A typical case could be where `P` is `Person`, `R` is `Knows`, and `S` 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 `S` is the set of skills of persons who live in Bloomington. Etc.

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

To generate `P` or `S`, 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
    $$
        select floor(random() * (u_1-l_1+1) + l_1)::int as x,
               floor(random() * (u_2-l_2+1) + l_2)::int as y
        from   generate_series(1,n)
        group by (x,y) order by 1,2;
     $$ 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*. Here 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* $u_1 = u_2 = 1000$.[2]

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

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*.*

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*. (Why?)*

```
            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
```

[2] All the experiments where done on a MacMini.

We now turn to the problems for this section.

2. Consider query Q_3

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

Intuitively, if we view R as a graph, and P and S as node types (properties), then Q_3 determines each P-node in the graph from which there emanates a path of length 3 that ends at a S-node.[3] I.e., a P-node n_0 is in the answer if there exists sequence of nodes (n_0, n_1, n_2, n_3) such that (n_0, n_1), (n_1, n_2), and (n_2, n_3) are edges in R and n_3 is a S-node.

Query Plan for Q_3.

```
Unique
   ->  Sort
         Sort Key: p.a
         ->  Hash Join
               Hash Cond: (r2.a = r1.b)
               ->  Hash Join
                     Hash Cond: (r3.a = r2.b)
                     ->  Hash Join
                           Hash Cond: (s.b = r3.b)
                           ->  Seq Scan on s
                           ->  Hash
                                 ->  Seq Scan on r r3
                     ->  Hash
                           ->  Seq Scan on r r2
               ->  Hash
                     ->  Hash Join
                           Hash Cond: (p.a = r1.a)
                           ->  Seq Scan on p
                           ->  Hash
                                 ->  Seq Scan on r r1
```

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

[3] Such a query is typical in Graph Databases.

Solution:

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

```
select p.a
from   P p
where  p.a in
    (select r.a
     from   R r
     where  r.b in
          (select r.a
           from   R r
           where  r.b in
            (select r.a
             from   R r
             where  r.b in (select s.b from S))));
```

(b) 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:

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	q3	q4	
R	501	1	25	1	25	P	41	1	100	S	36	1	100	182.173	0.578	
R	6362	1	100	1	100	P	40	1	100	S	38	1	100	2039.291	6.613	
R	6291	1	100	1	100	P	41	1	100	S	41	101	200	159.948	1.133	-- S does not overlap
R	6332	1	100	1	100	P	42	101	200	S	40	1	100	175.464	6.599	-- P does not overlap
R	6312	1	100	1	100	P	40	101	200	S	45	101	200	158.274	1.262	- P and S do not overlap
R	18805	1	400	1	400	P	46	1	400	S	45	1	400	479.318	18.813	
R	18768	1	400	1	400	P	260	1	400	S	249	1	400	8377.092	21.117	
R	37842	1	600	1	600	P	295	1	600	S	300	1	600	55208.732	41.579	
R	48789	1	1000	1	1000	P	612	1	1000	S	634	1	1000	51061.334	55.797	
R	95123	1	1000	1	1000	P	637	1	1000	S	627	1	1000	554601.762	109.236	

(c) 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 $O(|P| + |Q| + |R|$. This is as opposed to the original query with has complexity of order $|R|^3$.

3. Consider the Pure SQL Q_5 which is an formulation of a variation of the *not subset (not only) set semijoin* query

$$\{p.a \mid \mathtt{P}(p) \wedge \mathtt{R}(p.a) \not\subseteq \mathtt{S}\}$$

where

$$\mathtt{R}(p.a) = \{r.b \mid \mathtt{R}(r) \wedge r.a = p.a\}.$$

```
select p.a
from   P p
where  exists (select 1
               from   R r
               where  r.a = p.a and
                      not exists (select 1 from S s where r.b = s.b));
```

(a) Translate and optimize this query and call it Q_6. Then write Q_6 as an RA SQL query just as was done for Q_2 in Example 1.

Solution:

Q_6 can be written as an optimized RA SQL query as follows. (Note the semi joins (IN clause) and the anti semi join (NOT IN clause).) The complexity of this query is $O(|P| + |R| + |S|)$. So we can expect very fast query processing.

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

The query plan for $Q6$ is as follows:

```
Hash Semi 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
```

(b) An alternative way to write a query equivalent with Q_5 is as the object-relational query

```
with nestedR as (select P.a, array_agg(R.b) as bs
                 from   P natural join R
                 group by (P.a)),
     Ss as (select array(select b from S) as bs)
select a
from   nestedR
where  not (bs <@ (select bs from Ss));
```

Call this query Q_7.

Compare queries Q_5, Q_6, and Q_7 in a similar way as we did in Example 1. However, now you should experiment with different sizes for `P`, `R` and `S` as well as consider how `P` and `S` interact with `R`.

Solution: The query plan for Q_7 is as follows:

```
Subquery Scan on nestedr
  InitPlan 2 (returns $1)
    ->  Result
          InitPlan 1 (returns $0)
            ->  Seq Scan on s
  ->  GroupAggregate
        Group Key: p.a
        Filter: (NOT (array_agg(r.b) <@ $1))
        ->  Sort
              Sort Key: p.a
              ->  Hash Join
                    Hash Cond: (p.a = r.a)
                    ->  Seq Scan on p
                    ->  Hash
                          ->  Seq Scan on r
```

Note that the construction of NestedR and Ss can be done on $O(|R|+|S|)$. The potentially most significant cost is the verification of the $\not\subseteq$ condition. We don't see the actual implementation of this condition. However, we can imagine hashing S and then for each $r.a$ probe the set of its associated r.b's in **hashed**(S). Doing this would be $(|S|+|R|)$. Hence the combined complexity for this query is $O(|P|+|R|+|S|)$. The experiments yield the following table.

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
R	373	1	20	1	20	P	20	1	20	S	20	1	20	0.106	0.082	122.906
R	90734	1	100000	1	5	P	995	1	100000	S	5	1	5	18.452	11.46	132.487
R	906459	1	1000000	1	5	P	9503	1	100000	S	5	1	5	184.527	115.174	280.61
R	975309	1	1000000	1	20	P	9554	1	100000	S	20	1	20	199.811	123.972	293.104
R	975434	1	1000000	1	20	P	9525	1	100000	S	0	1	20	477.277	363.302	295.672
R	975312	1	1000000	1	20	P	9509	1	100000	S	20	1	20	198.724	124.457	295.853
R	906112	1	1000000	1	5	P	9529	1	100000	S	5	1	5	183.913	115.544	281.143

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

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. We do see that the optimized query Q_6 performs the best, followed by Q_5 and then Q_7. The only case where Q_7 performs poorly is when P is small compared to the range for the $r.a$ values. In this case, Q_5 and Q_6 significantly reduce R when semi joined.

4. Consider the Pure SQL Q_8 which is an formulation of a variation of the *not superset, (not all) set semijoin* query

$$\{p.a \mid \mathtt{P}(p) \wedge R(p.a) \not\supseteq S\}$$

where

$$\mathtt{R}(p.a) = \{r.b \mid \mathtt{R}(r) \wedge r.a = p.a\}.$$

```
select p.a
from   P p
where  exists (select 1
               from   S s
               where  not exists (select 1 from R where p.a = r.a and r.b = s.b));
```

Solution: The query plan for Q_8 is the following:

```
Nested Loop Semi Join
  Join Filter: (NOT (alternatives: SubPlan 1 or hashed SubPlan 2))
  ->  Seq Scan on p
  ->  Seq Scan on s
  SubPlan 1
    ->  Seq Scan on r
          Filter: ((p.a = a) AND (b = s.b))
  SubPlan 2
    ->  Seq Scan on r r_1
```

Generally, SubPlan 2 is applied. This plan will create a hash table on R and it will be checked if $(p.a, s.b)$ is not in R for each $p.a$ and $s.b$ using a semi join operation. The complexity of this plan is $O(|R| + |P||S| +$ sortTime).

(a) Translate and optimize this query and call it Q_9. Then write Q_9 as an RA SQL query just as was done for Q_2 in Example 1.

Solution:

The following is an optimized query expressed in RA SQL where the anti semi join is implemented using a `NOT IN` clause. We get the query:

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

The query plan for Q_9 is as follows:

```
Unique
  ->  Sort
        Sort Key: p.a
        ->  Nested Loop
              Join Filter: (NOT (hashed SubPlan 1))
              ->  Seq Scan on s
```

```
->  Seq Scan on p
SubPlan 1
  ->  Seq Scan on r;
```

This plan is very similar to that for Q_8 except that for query Q_8 we have a nested loop semi-join. The complexity is $O(|R| + |P||S|)$.

(b) An alternative way to write a query equivalent with Q_8 is as the object-relational query

```
with nestedR as (select P.a, array_agg(R.b) as bs
                 from   P natural join R
                 group by (P.a)),
     Ss as (select array(select b from S) as bs)
select a
from   P
where  a not in (select a from nestedR) and
       not((select bs from Ss) <@ '{}')
union
select a
from   nestedR
where  not((select bs from Ss) <@ bs);
```

Call this query Q_{10}.
We actually rewrite this query as follows and also refer to it as Q_{10}:

```
select a
from   (select a, array_agg(b) as bs
        from   R
        where  a in (select a from P)
        group by (a)) q
where  not((select array(select b from S)) <@ bs)
union all
select a
from   P
where  a not in (select  a from R) and
       exists (select  1 from S);
```

Solution

The query plan for Q_{10} is as follows:

```
Append
  ->  Subquery Scan on q
        InitPlan 2 (returns $1)
          ->  Result
                InitPlan 1 (returns $0)
                  ->  Seq Scan on s
        ->  GroupAggregate
              Group Key: r.a
              Filter: (NOT ($1 <@ array_agg(r.b)))
              ->  Sort
                    Sort Key: r.a
                    ->  Hash Semi Join
                          Hash Cond: (r.a = p.a)
                          ->  Seq Scan on r
                          ->  Hash
```

```
                                ->  Seq Scan on p
->  Result
      One-Time Filter: $3
      InitPlan 4 (returns $3)
        ->  Seq Scan on s s_1
      ->  Seq Scan on p p_1
            Filter: (NOT (hashed SubPlan 3))
            SubPlan 3
              ->  Seq Scan on r r_1
```

The complexity associated with this query plan is $O(|R|+|P|+|S|+|\pi_a(R)|C)$ where C is the average time to check if $S \subseteq \{b|R(a,b)\}$ across all $a \in \pi_a(R)$. What C is depends on the data and is therefore difficult to capture in terms of $|S|$ and $|R|$. Furthermore, the analysis also depends on the algorithm used to implement `<@`.

(c) Compare queries Q_8, Q_9, and Q_{10} in a similar way as we did In Example 1. However, now you should experiment with different sizes for `P`, `R` and `S` as well as consider how `P` and `S` interact with R.

Solution:

We get some of the following experimental results with working memory set at 256MB.

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	q8	q9	q10
R	95270	1	1000	1	1000	P	95	1	1000	S	95	1	1000	175.4	190.762	256.92
R	95171	1	1000	1	1000	P	182	1	1000	S	181	1	1000	178.294	203.057	251.998
R	95231	1	1000	1	1000	P	94	1	1000	S	97	1	1000	179.957	192.72	248.344
R	95217	1	1000	1	1000	P	257	1	1000	S	264	1	1000	180.743	225.018	254.873
R	95179	1	1000	1	1000	P	863	1	1000	S	852	1	1000	179.411	598.348	286.619
R	95198	1	1000	1	1000	P	880	1	1000	S	4	1	5	177.896	200.489	285.854
R	95160	1	1000	1	1000	P	860	1	1000	S	0	1	5	3.379	5.153	227.023
R	95220	1	1000	1	1000	P	866	1	1000	S	3	1	5	177.885	186.621	283.177
R	95150	1	1000	1	1000	P	1000	1	1000	S	2	1	5	177.68	186.57	290.877
R	632459	1	1000	1	1000	P	1000	1	1000	S	1	1	5	364.195	366.889	755.613
R	632147	1	1000	1	1000	P	1000	1	1000	S	12	1	30	371.247	374.871	760.437
R	995140	1	10000	1	10000	P	1000	1	1000	S	791	1	2000	512.968	1009.046	647.928

(d) What conclusions do you draw from the results of these experiments?

Solution:

We observe that, in general, for various parameter settings, the timings for each of the 3 queries is very similar, with that for Q_8 narrowly the best. Optimization did not improve the performance. And the complex object formulation also did not improve the performance. We can conclude that the query optimizer did an excellent job optimizing the original query Q_8.

5. Give a brief comparison of your results for Problem 3 and Problem 4. In particular, where the results show significant differences, explain why you think that is the case. And, where the results show similarities, explain why you think that is the case.

Generally speaking, there is little difference between the performance for the two problems. This is surprising since the theoretical performance of the queries in Problem 4 has a term $|P||Q|$ which is absent from the queries in Problem 3. All of this might be explained by the fact that most of the query processing could still be done in main memory. Perhaps our data sets where too small.

3 Object Relational Programming

The following problems require you to write object relational programs. Many of these require program written in Postgres' `plpgsql` database programming language.

6. **Practice Problem–not graded** 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. [4]

   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. Note that a path can be stored in an array.

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

7. Consider a parent-child relation `PC(parent, child)`. (You can assume that `PC` is a rooted tree and the domain of the attributes `parent` and `child` is `int`.) An edge (p, c) in `PC` indicates that node p is a parent of node c. Now consider a pair of nodes (m, n) in `PC` (m and n maybe the same nodes.) We say that m and n are in the *same generation* when the distance from m to the root of `PC` is the same as the distance from n to the root of `PC`.

   Consider the following recursive query that computes the `sameGeneration` relation:

```
WITH RECURSIVE sameGeneration(m, n) AS
     ((SELECT parent, parent FROM PC) UNION (select child, child from PC)
     UNION
     SELECT  t1.child, t2.child
     FROM    sameGeneration pair, PC t1, pc t2
     WHERE   pair.m = t1.parent and pair.n = t2.parent)
select distinct pair.m, pair.n from sameGeneration pair order by m, n;
```

   Write a non-recursive function `sameGeneration()` in the language `plpgsql` that computes the `sameGeneration` relation.

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

8. 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 (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) Write a **recursive** function `recursiveAggregatedWeight(p integer)` that returns the aggregated weight of a part `p`. Test your function.

(b) Write a **non-recursive** function `nonRecursiveAggregatedWeight(p integer)` that returns the aggregated weight of a part `p`. Test your function.

9. **Practice problem–not graded**. Consider the heap data structure. For a description, consult

   https://en.wikipedia.org/wiki/Binary_heap.

   (a) Implement this data structure in PostgreSQL. This implies that you need to implement the `insert` and `extract` heap operations.

   In this problem, you are **not** allowed to use arrays to implement this data structure! Rather you must you relations.

   (b) Then, using the heap data structure developed in question 9a, write a PostgreSQL program `heapSort()` that implement the `Heapsort` algorithm. For a description of this algorithm, see

   https://en.wikipedia.org/wiki/Heapsort

   You are **not** allowed to use arrays to implement this the `Heapsort` algorithm!

   The input format is a list of integers stored in a binary relation `Data(index,value)`. For example, `Data` could contain the following data.

   Data

index	value
1	3
2	1
3	2
4	0
5	7

   The output of `heapSort()` should be stored in a relation `sortedData(index,value)`. On the `Data` relation above, this should be the following relation:

10. **Practice problem–not graded**. 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 *Graph* 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

Implement Dijkstra's 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.

When you apply `Dijkstra(0)`, you should obtain the following table:

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

11. Consider the relation schema `document(doc int, words text[])` representing a relation of pairs (d, W) where d is a unique id denoting a document and W denotes the set of words that occur in d.

    Let **W** denote the set of all words that occur in the documents and let t be a positive integer denoting a *threshold*. Let $X \subseteq \mathbf{W}$. We say that X is t-frequent if

    $$\texttt{count}(\{d|(d, W) \in \texttt{document} \text{ and } X \subseteq W\}) \geq t$$

    In other words, X is *t-frequent* if there are at least t documents that contain all the words in X.

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

    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 $X \subseteq Y$ 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.

    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.

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 $(n_1, \ldots n_k)$ such that each node n in G occurs once, but only once, as a node n_i 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.

4 Key-value Stores (MapReduce and Spark)

Consider the document "MapReduce and the New Software Stack" available in the module on MapReduce.[5] 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.[6]

Consider a relation `R(a,b,c)`. For simplicity, we will assume that the domain of the attributes of `R` is integer.[7]

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

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

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

[6] 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.

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

Starting from this relation `R` we can, using `jsonb`[8] 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

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

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.[9] 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)`.

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

[8] 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.

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

```
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* $\mathtt{R} \cap \mathtt{S}$*. 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.

13. **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 $(\mathtt{R}-\mathtt{S})\cup(\mathtt{S}-\mathtt{R})$. You can assume that the domain of the attribute `'a'` is integer.

14. 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 semijoin of two relations R(A,B) and S(A,B,C), i.e., the relation $R \ltimes S$. You can assume that the domains of A, B, and C are integer. Use the encoding and decoding methods described above.

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 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.

16. 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.

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.

17. 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) 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) Write a PostgreSQL query that use this `coGroup` view to compute the semi join $R \ltimes S$, in other words compute the relation $R \bowtie \pi_K(S)$.

    (c) 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  NOT ARRAY(SELECT r1.V
                 FROM   R r1
                 WHERE  r1.K = r.K) && ARRAY(SELECT s1.W
                                             FROM  S s1
                                             WHERE  s1.K = s.K);
```

18. **Practice problem–not graded**. 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 17 solve the following problems:[10]

    (a) Write a PostgreSQL query that uses the cogroup transformation to compute $A \cap B$.

    (b) Write a PostgreSQL query that uses the cogroup operator to compute the symmetric difference of A and B, i.e., the expression

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

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

5 Graph query languages

Each of the following problems is a practice problem.

19. Consider the database schema `Person`, `Company`, `companyLocation`, `Knows`, `jobSkill`, worksFor, and `personSkill`.

    (a) 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) 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 19a. 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`.

20. Using the Property Graph model in Problem 19b, formulate the following queries in the Cypher query language:

(a) Find the types of the relationships associated with `Person` nodes.
Solution:

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

(b) 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) Find each jobSkill (node) that is the job 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) 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) 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 job 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)
```