

B561 Assignment 8
Relational Programming

1. Consider the relation schema **A(x int)** representing a schema for storing a set of integers A .

Using arrays to represent sets, write a PostgreSQL program

superSetsOfSet(X int[])

that returns each subset of A that is a superset of X , i.e., each set Y such that $X \subseteq Y \subseteq A$.

For example, if $X = \{\}$, then **superSetsofSets(X)** should return each element of the powerset of A .

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

A path in G is a sequence of vertices (v_0, v_1, \dots, v_n) such that, for each $i \in [0, n-1]$, (v_i, v_{i+1}) is an edge of G . We say that such a path connects v_0 to v_n via an n -length path.

- (a) Write a PostgreSQL program **connectedByEvenLengthPath()** that returns the pairs of vertices (s, t) of G that are connected via an *even*-length path in G .
- (b) Write a PostgreSQL program **connectedByOddLengthPath()** that returns the pairs of vertices (s, t) of G that are connected via an *odd*-length path in G .

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

Now let G be a directed graph that is acyclic, a graph without cycles.¹

A *topological sort* of a graph is a list (array) of **all** its vertices (v_1, v_2, \dots, v_n) such that for each edge (s, t) in G , vertex s occurs before vertex t in this list.

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

4. 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 .

¹A cycle is a path (v_0, \dots, v_k) where $v_0 = v_k$.

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

(Hint: use a backtracking algorithm that finds a 3-color assignment to the vertices of G if such an assignment exists.)

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

We say that G is *Hamiltonian* if there exists a permutation (v_1, \dots, v_n) of all the vertices of G such that (a) for each $i \in [1, n - 1]$, (v_i, v_{i+1}) is an edge of G and (b) (v_n, v_1) is an edge in G . Such a permutation is called an *Hamiltonian cycle* of G .

Write a PostgreSQL program `Hamiltonian()` that returns true if G is Hamiltonian, and false otherwise.

(Hint: use a backtracking algorithm that finds an Hamiltonian cycle of G if such a cycle exists.)

6. 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 \mathbf{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

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

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 each 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.

7. 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) .

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.)

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			Parts	
pid	sid	quantity	pid	weight
1	2	4	2	5
1	3	1	4	50
3	4	1	5	3
3	5	2		

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

Write a PostgreSQL function `aggregatedWeight(p integer)` that returns the aggregated weight of a part `p`.

9. Suppose you have a weighted undirected 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.)

Since the graph is undirected, whenever there is an weighted edge (s, t, w) in G , then (t, s, w) is also a weighted edge in the G . 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
2	4	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 `Paths` which represents the pairs (t, d) where d is the shortest distance from s to t . To test your procedure, you can use the graph shown above.

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

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

Hint: You can find the details of Dijkstra's Algorithm in the attached pdf document, but you are not required to exactly follow the pseudocode.

10. Consider the document *Introduction_to_MapReduce* in the Lecture Notes on MapReduce. In that document, you can find in Sections 2.3.3-2.3.7 how certain relational algebra operations can be implemented in MapReduce. In particular, look at the map and reduce functions for various RA operators.

In the following problems, you are asked to write simulations of MapReduce in PostgreSQL. To do this, you should follow the strategy described in the document *word_count_mapreduce.pdf*.²

- (a) Write a simulation in PostgreSQL of a MapReduce program that implements $\pi_A(R)$ where $R(A, B)$ is a relation. You can assume that the domains of A and B are integer. (Recall that in a projection, duplicates are eliminated.)
- (b) Write a simulation in PostgreSQL of a MapReduce program that implements the set difference of two unary relations $R(A)$ and $S(A)$, i.e., the relation $R(A) - S(A)$. You can assume that the domain of A is integer.
- (c) Write a simulation in Postgres of a MapReduce program 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.

²This document is also in the Lecture Notes on MapReduce.