**CPTS 415**

**Big Data**

**Assignment 2**

1. [**Join operators**] (45) Read Chap 15.3, 15.4.6-15.4.8, and 15.5.5 of “Database Systems: the Complete Book” to review join operators. This sets of questions test the understanding of basic database search operators.

Consider a join R.A=S.B S. We ignore the cost of output the result, and measure the cost with the number of I/O. Given the information about relations to be joined below:

**Relation S** contains 2K tuples and has 10 tuples per block; in total 200 blocks.

**Relation R** contains 10K tuples and has 10 tuples per block; in total 1000 blocks.

Attribute B of S is the primary key of S. In total 52 Buffer pages are available in memory. Each page can hold a block.

Assume neither relation has any index.

For each question, you should try to give the algorithm with the cheapest cost among the correct ones.

1. (15) Describe a block nested join algorithm. Give the cost of joining R and S with a block nested loops join.

for each block Bs of s do begin

read these blocks into main-memory buffers;

organize their tuples into a search structure whose

search key is the common attributes of R and S.

for each block Br of r do begin

read Br into main memory

for each tuple ts in Bs do begin

for each tuple tr in Br do begin

Check if (ts, tr) satisfy the join condition

if they do, add ts \* tr to the result.

end

end

end

end

The idea of block nested join algorithm is to read the outer relation in blocks, and then for each block, scan the inner relation page-by-page. Therefore, the total cost will be minimized if the smaller relation is the outer relation. In this problem, we should choose S as the outer relation.

The outer relation is always scanned once, while the inner relation is scanned for each outer block. If we have B buffers, the number of blocks is ceil(/(B-2).

Cost = ceil(bs/(B-2)) \* br + bs block transfers = ceil(200/50) \* 1000 + 200 = 4200

1. (15) Describe a sort-merge join algorithm. Give the cost of joining R and S with a sort‐merge join.

The idea of sort-merge join algorithm is to sort both relations on their join attribute, and merge the sorted relations to join them. Join step is similar to the merge stage of the sort-merge algorithm. Main difference is handling of duplicate values in join attribute—every pair with same value on join attribute must be matched.

To be more specific, the file of size B are first read into memory and sorted. A number of passes followed, where each pass makes a (B-1)-way merge of the file. Each pass reads and writes the file. The total number of passes is 1 + ceil(ceil(n/B)).

Cost = 2bs \* (1 + ceil(ceil(n/B))) + 2br \* (1 + ceil(ceil(n/B))) + bs + br

=2\*200\*(1+ceil(4) + 2\*1000\*(1+ceil(4) + 200 + 1000

=6000

1. (15) Describe a hash-join algorithm. Give the cost of joining R and S with a hash join.

The idea of sort-merge join algorithm is to partition the relation s using hashing function h, and then partition r similarly. When partitioning a relation, one block of memory is reserved as the output buffer for each partition. For each i, load si into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h. Then read the tuples in ri from the disk one by one. For each tuple tr locate each matching tuple ts in si using the in-memory hash index. Output the concatenation of their attributes.

In this problem, we need to assume that we have at least sqrt(br) buffers or sqrt(bs) buffers that the hash function gives a uniform distribution. Given that both sqrt(200) and sqrt(1000) are smaller than 52, the assumption is satisfied.

Cost = 3(bs + br) = 3(200 + 1000) = 3600

1. **[Graph algorithms]** (30) The following questions test your understanding on basic graph algorithms.

Give a correctness proof and complexity (in Big O notation) of your algorithm. [hint: transform the weight to non‐negative numbers, e.g., ‐log r(u,v) and transform it to a familiar graph problem].

1. (10) Given a directed graph G (V, E, L) with V the node set, E the edge set and L a function that assigns to each edge e in E a label L(e). A label constrained reachability query Q(s, t, M) tests if there exists a path from a source s to a target t with a path, which consists of edges having a label from a label set M. Give an algorithm (pseudo-code) to answer query Q. [\*A straightforward way is to revise BFS or DFS traversal].

BFS traversal:

for each vertex u∈G(V, E, L):

u.color = WHITE

u.d = ∞

u.Π = NIL

s.color = GRAY

s.d = 0

s.Π = NIL

Q = ∅

LabelSet = ∅

ENQUEUE(Q, s)

while Q ≠ ∅

u=DEQUEUE(Q)

for each v ∈adjacent edges

if v.color == WHITE

v.color = GRAY

v.d = u.d + 1

v.Π = u

ENQUEUE(Q, v)

u.color = BLACK

B = B∪{u}

GetLabelSet(B)

LabelSet = ∅

for any point u, w in B

for each v ∈adjacent edges

if v.id = w.id:

LabelSet = LabelSet ∪L(u, v)

For query Q(s, t, M), we first run BFS(G, s) to see whether t is marked as black or not. If t is not marked as black, it means there is no path between t and s. Otherwise, we compare the label set M with GetLabelSet(B). If M ⊆ GetLabelSet(B), M is potentially a set from s to t. Finally, we need to test whether there exists a path from s to t within the set M.

1. (20) Consider a network G (V, E) of servers, where each edge (u, v) represents a communication channel from a server u to another server v. Each edge has an associated value r (u, v), which is a constant in [0, 1]. The value represents the reliability of the channel, i.e., the probability that the channel from u to v will not fail. Assume these probabilities are independent. **Give an algorithm (pseudo‐code) to find the most reliable path between two given servers.**

To find the most reliable path between s and t, run Dijkstra’s algorithm with edge weight w(u, v) = -log(r(u, v)) to find the shortest paths from s in O(E + VlogV) time.

As the probabilities are independent, the probability that a path will not fail is the product of the probabilities that its edges will not fail. Therefore, in order to find the most reliable path between two given servers, we want to maximize the product of reliabilities of all edges,. This is equivalent to maximizing, which is also equivalent to. As a result, the objective can be minimizing. (As r(u, v) can be 0, define log0 = -inf.) If we assign weights w (u, v) = -log r (u, v), we have the shortest-path problem.

Since logx <= 0 for 0 < x <= 1, and log0 = -inf, all weights w are nonnegative. By using Dijkstra’s algorithm, we can find the shortest paths from s in O(E + VlogV) time.

Algorithm:

Dijkstra (G, s, w):

1. for all nodes v in V do

a.d[v] ←∞; ←

2. d[s] ←0; Que ←V;

3. while Que is nonempty do

a. u ←ExtractMin(Que);

b. for all nodes v in adj(u ) do

a)if d[v] > d[u] + w(u, v) then d[v] ←d[u] + w(u, v);

Proof:

1. Initialization.

Initially, S = ∅, and so the invariant is true.

1. Maintenance.

In order to show that in each iteration u.d = δ(s, u) for the vertex is added to set S, we first make the initial vertex for u.d ≠ δ(s, u) when it is added to set S. As s is the first vertex added to the set S, we can get that s.d = δ(s, s) = 0. That is, u ≠ s. Then, we can get that S ≠ ∅ before u is added to S. Therefore, there exists paths from s to u. If there is no path from s to u, we will get the result that u.d = δ(s, u) = ∞, which violate the former assumption that u.d ≠ δ(s, u). As there is at least one path, there is shortest path p from s to u, which connect a vertex in S with a vertex in V. The path p is s-x-y-u.

In this path, x ∈ S. As u is the first vertex for u.d ≠ δ(s, u) when it is added to set S, x.d=δ(s, x) when added to S. Similarly, y.d=δ(s, y).

Now we can obtain the contradiction to prove the algorithm. As y is prior to u on the shortest path p, and all edges are non-negative, we can get that δ(s, y) <= δ(s, u). We can conclude by relax function that y.d = δ(s, y) <=δ(s, u) <= u.d. However, as u and y were in the priority queue, u.d <= y.d, which means u.d = δ(s, u). This contradicts our choice of u.

In short, u.d = δ(s, u) when u is added to S for all iterations.

1. Termination.

At the termination, Q = ∅, which implies S = V. This means that u.d = δ(s, u) for all vertices u ∈ V. As a result, the algorithm gets proved.

\*\* (bonus 5 points) How may a 2-hop cover/landmark vector index improve the efficiency of your algorithms in a and b for large G (e.g., contains millions of nodes and edges)? Give a discussion to justify your idea.

For each vertex v, we precompute a label denoted as L(v), which is a set of pairs (u, ), where u is a vertex and . We call the set of labels {L(v)}, v∈V as an index. To answer a distance query between vertices s and t, we compute and answer Query(s, t, L) defined as Query(s, t, L) = min{+|(v, ) ∈L(s), (v, ) ∈L(t)}. If L(s) and L(t) do not share any vertex, we set Query(s, t, L) to be infinity. We call L a 2-hop cover of G if Query(s, t, L) = for any pair of vertices s and t. For each vertex v, we store the label L(v) so that pairs in it are sorted by their vertices. Then, we can compute Query(s, t, L) in O(|L(s)| + |L(t)|) time using a merge-join-like algorithm.

1. [**Approximate query processing**]. (25) This question continues our discussion on using data synopsis for query processing based on data-driven approximation. You are given a vector of numbers: [127, 71, 87, 31, 59,3,43, 99, 100, 42, 0, 58, 30, 88, 72, 130], each data point records the frequency of communication of a server in a 5 minutes interval. For example, in the first 5 minutes ([0,5]), 127 contacts; in the second 5 minutes ([5, 10]), 71 contacts…
2. Give the Haar decomposition and draw a corresponding error tree for the contacts data vector.

|  |  |  |
| --- | --- | --- |
| Resolution | Averages | Detail coefficients |
| 4 | [127, 71, 87, 31, 59, 3, 43, 99, 100, 42, 0, 58, 30, 88, 72, 130] | -- |
| 3 | [99, 59, 31, 71, 71, 29, 59, 101] | [28, 28, 28, -28, 29, -29, -29, -29] |
| 2 | [79, 51, 50, 80] | [20, -20, 21, -21] |
| 1 | [65, 65] | [14, -15] |
| 0 | [65] | [0] |

Haar decomposition: [65, 0, 14, -15, 20, -20, 21, -21, 28, 28, 28, -28, 29, -29, -29, -29]

0

65

14

-15

20

-20

-21

21

-

+

+

+

+

+

+

+

-

-

-

-

-

-

-29

-29

-29

29

-28

28

28

28

-

+

- +

+

- +

- +

- +

- +

-

+

-

127 71 87 31 59 3 43 99 100 42 0 58 30 88 72 130

1. Give the process and result for reconstructing the frequency during time interval [15, 20] using Haar decomposition.

|  |  |
| --- | --- |
| Interval | Frequency |
| [0, 5] | 127 |
| [5, 10] | 71 |
| [10, 15] | 87 |
| [15, 20] | 31 |
| [20, 25] | 59 |
| [25, 30] | 3 |
| [30, 35] | 43 |
| [35, 40] | 99 |
| [40, 45] | 100 |
| [45, 50] | 42 |
| [50, 55] | 0 |
| [55, 60] | 58 |
| [60, 65] | 30 |
| [65, 70] | 88 |
| [70, 75] | 72 |
| [75, 80] | 130 |

The frequency during time interval [15, 20] is 31.

By using the synopsis [65, 0, 14, 20, 28, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] and constructing the tree on fly, frequency=31 will be returned.

1. Use Haar decomposition and error tree to compute the total number of communication between time interval [15, 30].

|  |  |
| --- | --- |
| Interval | Frequency |
| [0, 5] | 127 |
| [5, 10] | 71 |
| [10, 15] | 87 |
| [15, 20] | 31 |
| [20, 25] | 59 |
| [25, 30] | 3 |
| [30, 35] | 43 |
| [35, 40] | 99 |
| [40, 45] | 100 |
| [45, 50] | 42 |
| [50, 55] | 0 |
| [55, 60] | 58 |
| [60, 65] | 30 |
| [65, 70] | 88 |
| [70, 75] | 72 |
| [75, 80] | 130 |

Find the Haar wavelet transformation and construct the tree

Where

0

65

14

-15

20

-20

-21

21

-

+

+

+

+

+

+

+

-

-

-

-

-

-

-29

-29

-29

29

-28

28

28

28

-

+

- +

+

- +

- +

- +

- +

-

+

-

127 71 87 31 59 3 43 99 100 42 0 58 30 88 72 130

Result: (5 – 3 + 1)\*65 + (1 – 2)\*14 – 20 + 2\*(-20) – 28 + 28 – 28 = 93