CS540: Assignment 4

Ga Young Lee
Department of Electrical Engineering and Computer Science
Oregon State University
leegay@oregonstate.edu

Jeffrey M. Young
Department of Electrical Engineering and Computer Science
Oregon State University
youngjef@oregonstate.edu

March 2, 2021

1 Problem 1: Query Processing Algorithms

Problem Description:
Consider the natural join of the relation *(A,B)* and *S(A,C)* on attribute A. Neither relations have any indexes built on them. Assume that R and S have 80,000 and 20,000 blocks, respectively. The cost of a join is the number of its block I/Os accesses. If the algorithms need to sort the relations, they must use two-pass multi-way merge sort. You may choose the join algorithms in your answers from the ones taught in class.

1.a

Problem Description:
Assume that there are 10 blocks available in the main memory. What is the fastest join algorithm for computing the join of R and S? What is the cost of this algorithm?

From the problem description we have:

$$\begin{aligned} B(R) &\triangleq \text{number of blocks of relation } R && = 80,000 \\ B(S) &\triangleq \text{number of blocks of relation } S && = 20,000 \\ \text{M} &\triangleq \text{number of blocks of main memory} && = 10 \\ \text{Cost} &\triangleq \text{total cost of the join} && = \text{number of block I/Os accesses} \end{aligned}$$

If both relations can fit in main memory, we can use internal memory join algorithms. However, both B and S do not fit into main memory, so we must use external memory join. Unfortunately, they do not satisfy the requirement for External Sort-Merge Join, $B(B) \leq M^2$ and $B(S) \leq M^2$. Thus, we cannot use Sort-Merge Join which only takes a linear cost of $5B(R)$+$5B(S)$. Furthermore, neither of the relations have indexes, so joins such as a zig-zag join are similarly ruled out.

Hence, we are left with Improved Nested Loop Join that uses up the available memory buffers, M. When using Improved Nested Loop Join, it is beneficial to have the smaller relation outside, so we choose *S* as the outer relation of the Nested Loop Join since the loop for *S* encloses the loop for *R*. In other words, we read 8(= M-2) blocks of the outer relation, *S*, at a time. Then, when we each block of the inner relation, *R*, we join it with all the M-2 blocks of the outer relation. This improvement reduces the number of scans of the inner relation from $B(S)$ to $[\frac{B(S)}{M-2}]$. Therefore, the cost of Improved Nested Loop Join, S $\bowtie$ R, is

$$Cost_{Nest-LoopJoin} = B(S) + [\frac{B(S)}{M-2}] * B(R)$$

Plugging in the numbers, we get $Cost = B(S)+[\frac{B(S)}{M-2}]B(R) = 20,000+[\frac{20,000}{8}]*80,000 = 200,020,000$.

1.b

Problem Description:
Assume that there are 350 blocks available in the main memory. What is the fastest join algorithm to compute the join of R and S? What is the cost of this algorithm?

$B(R) \triangleq$ number of blocks of relation R $= 80,000$

$B(S) \triangleq$ number of blocks of relation S $= 20,000$

M $\triangleq$ number of blocks of main memory $= 350$

Cost $\triangleq$ total cost of the join $=$ number of block I/Os accesses

Both relations are too large to fit into main memory. Therefore, we need to use external memory join. Since $B(R)+B(S) = 100,000 \leq M^2$ = 122,500, we can use Optimized Sort-Merge Join algorithm to join these relations. The cost of the algorithm is

$$Cost_{Sort-MergeJoin} = 3B(S) + 3B(R)$$

Plugging in the numbers, we get $Cost = 3B(R) + 3B(S) = 3(80,000 + 20,000) = 300,000$.

1.c

Problem Description:
Assume that there are 200 blocks available in the main memory. What is the fastest join algorithm to compute the join of R and S? What is the cost of this algorithm?

B(R) $\triangleq$ number of blocks of relation R $= 80,000$

B(S) $\triangleq$ number of blocks of relation S $= 20,000$

M $\triangleq$ number of blocks of main memory $= 200$

cost $\triangleq$ total cost of the join $=$ number of block I/Os accesses

Both relations are too large to fit into the main memory. Therefore, we need to use the external memory. Hash Join and Optimized Sort-Merge Join are the fastest algorithms in the given setting and have the equal cost of $3B(R) + 3B(S)$. However, the given relations do not satisfy Optimized Sort-Merge Join where $B(R) + B(S) \leq M^2$, so we cannot use Optimized Sort-Merge Join algorithm.

On the other hand, the relations meet the Hash Join condition, $min(B(R), B(S)) = B(S) \leq M^2 = 40,000$, so we can Hash Join. The cost of running this join algorithm is

$$Cost_{HashJoin} = 3B(S) + 3B(R)$$

Plugging in the numbers, we get $Cost = 3B(R) + 3B(S) = 3(80,000 + 20,000) = 300,000$.

2 Problem 2: Query Processing

2.a

Problem Description:
Assume that the entire relation *R(A,B)* fits in the available memory but relation *S(A,C)* is too large to fit in the main memory. Find a fast join algorithm, i.e., an algorithm with the lowest number of I/O access, for the natural join of R and S. Justify that your proposed algorithm is the fastest possible join algorithm to compute the natural join of R and S. Next, assume that there is a clustered index on attribute A of relation S. Explain whether or how this will change your answer.

In the given setting, we cannot fit both relations in main memory, so we need to find an external memory join algorithm. Since $min(B(R), B(S)) = B(R) \leq M^2$, we can use Partitioned Hash Join in the external memory. In this case, we partition *S* and *R* into *M* buckets, which takes a complete reading of both relations and a subsequent writing back. This operation takes $2B(R) + 2B(S)$. Then, the build and probe phases read each of the partitions once, which requires $B(R) + B(S)$. Thus, the cost of Partitioned Hash Join is

$$Cost_{Partitioned-HashJoin} = 3B(R) + 3B(S)$$

Given that there's a clustered index on attribute A of relation S, we can use Index-Based (Zig-Zag) Join algorithm. When only S has an index over the join attribute, we read R for each tuple of R and fetch matching tuples in S. The cost of this Index-Based Join algorithm is

$$Cost_{Index-BasedJoin} = B(R) + \frac{T(R)T(S)}{V(S, A)}$$

where *V(S, A)* is the number of distinct values of attribute A in S and *T(R) and T(S)* are the numbers of tuples in R and S respectively.

2.b

Problem Description:
Consider relation *R(A,B)* and *S(A,C)* that each have 1 million tuples and are too large to fit in main memory. A data scientist wants to compute 10,000 (sample) tuples of the natural join of R and S very fast. Since it is too time-consuming to compute the full natural join of R and S, the data scientist selects 1% of relation *R* and 1% of relation *S* and computes their join. Explain whether this algorithm returns the desired results. If it does not, propose an efficient algorithm that returns the desired result without computing the full natural join of R and S.

From the problem description we know that both relations are too large to fit in main memory. However, we want to get the first 10,000 tuples from a join. The suggested algorithm, performing a selection of size n on both relations and then performing the join, is unlikely to work. By the problem description, we have a natural join, and thus when we perform the selection of n tuples *before* performing the join we may simply be unlucky and select *exactly* the n tuples from each relation where $R.A \neq S.A$ *even* if there exist tuples in R and tuples in S, such that, $R.A == S.A$. Thus the proposed algorithm is not sound and we must perform the join before selecting the first n tuples, i.e., we must process this query in the reverse order suggested by the problem description. This answers the first part of the question.

As an alternative, we can create a clustered indexes (B+ tree) over R.A and S.A and use Index-Based Join algorithm. With Index-Based Join, we start from the initial block(s) of R using its index and find the block(s) with matching tuples in S using its index. We continue this step alternating between S and R until every block is exhausted. The cost of clustered Index-Based Join Algorithm can be significantly smaller than Hash Join and reduce the time required to process the joins.

In short, the suggested algorithm of selecting 10,000 tuples from each relation and joining the samples will not yield the desirable results because we don't know the distribution of the tuples, and a counterexample exists where the selected tuples might not always match. As a result the join outcome is significantly smaller than 10,000. To mitigate this problem efficiently, we propose clustered Index-Based Join algorithm to sort the relations and use the hash function to fetch matching tuples.