Query Execution

Nested Loop Algorithms

II. Nested loop algorithms.

- Join

Nested-Loop joins

One-pass join algorithm can only be used if one of the the relations is small enough to fit in main memory.

If both relations are large, then, one pass will not be enough. However, we can devise algorithms that read one of the two relations only once. This family of algorithms is called Nested-Loop algorithms.

Tuple-based Nested-Loop Join

Assume we are computing natural join $R(X,Y) \bowtie S(Y,Z)$. Tuple-based nested-loop join accesses relations in a tuple by tuple fashion, and for each pair of tuples checks if the join condition holds. It can be represented as follows:

```
Algorithm TupleJoin(R,S)

  for each s in S do
     for each r in R do
        if  (s.Y == r.Y)  output (r.X,r.Y,s.Z);

end Algorithm
```

We can build iterators based on tuple-based join.

```
Algorithm Open(R,S)
  Open(S);
  Open(R);
  s: = GetNext(s);  // set the first tuple from S
end Algorithm


Algorithm GetNext(R,S)

 do
    r:= GetNext(R);
    if (NOT found)    // if R is at the end, advance the tuple in S
       { Close(R);
         s:= GetNext(S);
 if (NOT found) return;   // if at the end of S, return
         Open(R);     // restart R for the new tuple from S
}

 while (s.Y != r.Y);

 return (r.X, r.Y, s.Z); // return the next joined tuple

end Algorithm



Algorithm Close(R,S)
 Close(R);
 Close(S);
end Algorithm
```

Block Nested-Loop Join

Tuple-based nested-loop join "pretends" to load tuples one-by-one into main memory. If it is allowed to proceed this way, its cost is $O(T(R) \cdot T(S)$ — very high.

Block Nested-Loop Join reads data in blocks to fill as much main memory as possible. This allows for computations of large portions of the final join without any extra I/O manipulation.

Let us assume that $B(R) > B(S) > M$, i.e., neither R nor S fits main memory. The block nested-loop join approach is as follows:

- Break S into "chunks" of size $M - 1$ each[1].
- organize the algorithm as a nested loop.
- The outter loop loads the next chunk from S into main memory.
- The inner loop scans R and joins tuples from R with tuples from S.

[1] The last chunk may have a smaller size.

The pseudocode for this algorithm is below.

```
Algorithm BlockNestedLoopJoin(R,S)

 NumChunks := (B(S) div (M-1)) + 1;  // compute the number of chunks

 for i = 1 to NumChunks do
     read M-1 blocks of S into buffer;   // load next chunk of S
     index these blocks on S.Y;          // index it

     for j = 1 to B(R) do                // read next block from R
        read block  Block_j from R;
        for each tuple r in Block_j do
     for each tuple s in main memory, s.t. s.Y == r.Y do   // use indexing to find
        output (r.X, r.Y, s.Z);                            // these tuples
     end for;
 end for;

end Algorithm
```

Evaluation

Algorithm	TupleJoin	BlockNestedLoopsJoin
Constraints	NONE!	
I/O Cost	$T(R) \cdot T(S)$	$O\left(\frac{B(S) \cdot B(R)}{M}\right)$
Memory Footprint	O(1)	M

Notes. For the tuple nested-loop join, we provide the worst-case estimate. For the block nested-loop join, the actual estimate of I/O costs is $B(S) + \frac{B(S) \cdot B(R)}{M-1}$. This formula explains why the smaller relation should be in the *outter loop*.