Query Processing and Advanced Queries

Query Optimization (3)
Nested-Loop Joins

- We now consider algorithms for the join operator.
- The simplest one is the nested-loop join, a one-and-a-half pass algorithm.
- One table is read once, the other one multiple times.
- It is not necessary that one relation fits in main memory.
- Perform the join through two nested loops over the two input relations.
**Nested-Loop Joins**

- **Tuple-based nested-loop join**
  
  natural join $R \bowtie S$, join attribute $C$

  for each $r \in R$ do
    
    for each $s \in S$ do
      
      if $r.C = s.C$ then output ($r,s$)

- **Outer relation** $R$, **inner relation** $S$.
- One buffer for outer relation, one buffer for inner relation.
- $M = 2$.
- I/O cost is $T(R) \times T(S)$. 
Nested-Loop Joins

Example

Relations not clustered

T(R1) = 10,000    T(R2) = 5,000

R1 as the outer relation

Cost for each R1 tuple t1:

read tuple t1 + read relation R2

Total I/O cost is 10,000 (1+5,000)=50,010,000
Nested-Loop Joins

- Can do much better by organizing access to both relations by blocks.
- Use as much buffer space as possible (M-1) to store tuples of the outer relation.

**Block-based nested-loop join**

for each chunk of M-1 blocks of R do
  read these blocks into the buffer;
  for each block b of S do
    read b into the buffer;
    for each tuple t of b do
      find the tuples of R that join with t and output the join results
Nested-Loop Joins

Example

- R1 as the outer relation
- \( T(R1) = 10,000, \quad T(R2) = 5,000 \)
- \( S(R1) = S(R2) = 1/10 \) block (each block 10 tuples)
- \( M = 101, \quad 100 \) buffers for R1, 1 buffer for R2
- 10 R1 chunks
- cost for each R1 chunk:
  - read chunk: 1,000 IOs
  - read R2: 5,000 IOs
- total I/O cost is \( 10 \times 6,000 = 60,000 \) IOs
Nested-Loop Joins

- Can do even better by reversing the join order. 
  \[ \text{R2} \Join \text{R1} \]
- \( T(R1) = 10,000, \quad T(R2) = 5,000 \)
- \( S(R1) = S(R2) = 1/10 \) block (each block 10 tuples) 
- \( M = 101, \quad 100 \) buffers for \( R2 \), 1 buffer for \( R1 \)
- 5 \( R2 \) chunks
- cost for each \( R2 \) chunk: 
  - read chunk: 1,000 IOs 
  - read \( R1 \): 10,000 IOs 
- total I/O cost is \( 5 \times 11,000 = 55,000 \) IOs
Finally, performance is dramatically improved when input relations are clustered (read by block).

With clustered relations, for each R2 chunk:

- read chunk: 100 IOs
- read R1: 1,000 IOs

Total I/O is 5 x 1,100 = 5,500 IOs.

Note that the IO cost for a one-pass join (which has the minimum IO of any join algorithm) in this example is 1,000 + 500 = 1,500 IOs.

For a comparison, the one-pass join requires $M=501$ buffer blocks, which is optimal.
Two-Pass Algorithms Based on Sorting

- If the input relations are sorted, the efficiency of duplicate elimination, set-theoretic operations and join can be greatly improved.
- Reserve one buffer for each of the input relations R and S and another buffer for the output.
- Scan both relations simultaneously in sort order, merging matching tuples.
- For example, for set intersection: repeatedly consider the tuple t that is least in the sort order (w.r.t. primary key) among all tuples in the input buffer. If it appears in both R and S, output t.
Two-Pass Algorithms Based on Sorting

- In the following, we present a simple sort-merge join algorithm.
- It is called merge-join, if step (1) can be skipped, since the input relations R1 and R2 are already sorted.

Sort-merge join

1. if R1 and R2 not sorted, sort them
2. \( i \leftarrow 1; j \leftarrow 1; \)

while (\( i \leq T(R1) \)) \( \land \) (\( j \leq T(R2) \)) do
   - if \( R1\{ i \}.C = R2\{ j \}.C \) then outputTuples
   - else if \( R1\{ i \}.C > R2\{ j \}.C \) then \( j \leftarrow j+1 \)
   - else if \( R1\{ i \}.C < R2\{ j \}.C \) then \( i \leftarrow i+1 \)
Two-Pass Algorithms Based on Sorting

- Procedure `outputTuples` produces all pairs of tuples from R1 and R2 with \( C = R1 \{ i \} . C = R2 \{ j \} . C \).

- In the worst case, need to match each pairs of tuples from R1 and R2 (nested-loop join).

**Procedure `outputTuples`**

```plaintext
While (R1 \{ i \} . C = R2 \{ j \} . C) \land (i \leq T(R1)) do
    [jj \leftarrow j;
     while (R1 \{ i \} . C = R2 \{ jj \} . C) \land (jj \leq T(R2)) do
         [output pair R1 \{ i \}, R2 \{ jj \};
          jj \leftarrow jj+1 ]
     i \leftarrow i+1 ]
```

Two-Pass Algorithms Based on Sorting

Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{i}.C</th>
<th>R2{j}.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>7</td>
</tr>
</tbody>
</table>
Example
Both R1, R2 ordered by C; relations clustered.

Memory

R1
R2

R1
R2

Total cost: read R1 cost + read R2 cost
= 1,000 + 500 = 1,500 IOs
What if input relations are not yet in the required sort order?

Do Two-Phase, Multiway Merge-Sort (2PMMS).

Phase 1: Sort each block of relation R separately in main memory, write sorted sublists back to disk.

Phase 2: Merge all the B(R) sorted sublists.
Each sorted sublist has a length of $M$ blocks.

Number of sublists is $B(R)/M$.

Therefore, $B(R)/M \leq M - 1$, i.e. $B(R) \leq M^2 - M \leq M^2$.

This means we require $M \geq \sqrt{B(R)}$.

In phase 1, each tuple is read and written once. In phase 2, each tuple is read again. We ignore the cost of writing the results to disk.

Thus, the IO cost is $3B(R)$. 

Two-Pass Algorithms Based on Sorting

- IO cost is $4B(R)$, if sorting is used as a first step of sort-join and the results must be written to the disk.
- If relation $R$ is too big, apply the idea recursively.
- Divide $R$ into chunks of size $M(M-1)$, use 2PMMS to sort each one, and take resulting sorted lists as input for a third (merge) phase.
- This leads to *Multi-Phase, Multiway Merge Sort*. 
Two-Pass Algorithms Based on Sorting

Example \( M = 101 \)

(i) For each 100 blk chunk of \( R \):
- read chunk
- sort in memory
- write to disk

\[ \text{R1} \rightarrow \text{Memory} \rightarrow \text{sorted chunks} \]

\[ \text{R2} \rightarrow \\]
(ii) Read all chunks + merge + write out
Two-Pass Algorithms Based on Sorting

Sort cost: each tuple is read, written, read, written

Join cost: each tuple is read

Sort cost R1: $4 \times 1,000 = 4,000$

Sort cost R2: $4 \times 500 = 2,000$

Total cost = sort cost + join cost

= $6,000 + 1,500 = 7,500$ IOs

Total IO Cost: $5(B(R1) + B(R2))$

Running Example:

$T(R1) = 10,000$

$T(R2) = 5,000$

$S(R1) = S(R2) = 1/10$ block (each block 10 tuples)

$M = 101$

100 buffers for R2, 1 buffer for R1
Two-Pass Algorithms Based on Sorting

- **Nested loop join** (best version discussed above) needs only 5,500 IOs, i.e. outperforms sort-join.

- However, the situation changes for the following scenario:
  
  - R1 = 10,000 blocks, clustered
  - R2 = 5,000 blocks, not ordered

  - R1 is 10,000 blocks, sorting needs \( M \geq 100 \).
  - R2 is 5,000 blocks, sorting needs \( M \geq 70.7 \).

  I.e., need at least \( M=71 \) buffers.
Two-Pass Algorithms Based on Sorting

- Nested-loops join:
  \[
  5000 \times (100+10,000) = 50 \times 10,100 \\
  \frac{100}{100} \\
  = 505,000 \text{ IOs}
  \]

- Sort-join:
  \[
  5(10,000+5,000) = 75,000 \text{ IOs}
  \]

- Sort-join clearly outperforms nested-loop join!
Two-Pass Algorithms Based on Sorting

- Simple sort-join costs $5(B(R) + B(S))$ IOs.
- It requires $M \geq \sqrt{B(R)}$ and $M \geq \sqrt{B(S)}$.
- It assumes that tuples with the same join attribute value fit in $M$ blocks.
- If we do not have to worry about large numbers of tuples with the same join attribute value, then we can combine the second phase of the sort with the actual join (merge).
- We can save the writing to disk in the sort step and the reading in the merge step.
Two-Pass Algorithms Based on Sorting

- This algorithm is an advanced *sort-merge join*.
- Repeatedly find the least C-value c among the tuples in all input buffers.
- Instead of writing a sorted output buffer to disk, and reading it again later, identify all the tuples of both relations that have C=c.
- Cost is only $3(B(R) + B(S))$ IOs.
- Since we have to simultaneously sort both input tables and keep them in memory, the memory requirements are getting larger:
  \[ M \geq \sqrt{B(R) + B(S)}. \]