CS4432: Database Systems II

Lecture #17
Join Processing Algorithms (cont).

Professor Elke A. Rundensteiner
Join: \[ \text{R1} \bowtie \text{R2}, \text{R2} \bowtie \text{R1} \]

- Iteration (nested loops)
- Merge join
- Join with index
- Hash join
**Nested-Loop Join:** \( R1 \ JOIN \ R2 \)

- Use one page as input buffer for scanning inner \( R2 \).
- Use one page as the output buffer.
- Use all remaining pages to hold `block` of outer \( R1 \).
So far

not contiguous

\{ 
\text{Iterate R2} \times \text{R1} \quad 50,010,000 \\
\text{Be memory access aware:} \\
\text{Iterate R2} \times \text{R1} \quad 55,000 \\
\}

contiguous

\{ 
\text{Blocked-access:} \\
\text{Iterate R2} \times \text{R1} \quad 5,500 \\
\}
Join: \[ R1 \Join R2, R2 \Join R1 \]

- Iteration (nested loops)
- Sort-Merge join
- Join with index
- Hash join
Merge join (conceptually)

(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 \( i \leftarrow i+1 \)
Procedure Output-Tuples

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 \]
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: Merge Join

- Both R1, R2 ordered by C; relations contiguous

**Memory**

```
R1  
R2  
```

```
R1  
.....  
```

```
R2  
.....  
```

**Total cost:** Read R1 cost + read R2 cost  
= 1000 + 500 = 1,500 IOs
Example  Merge Join

• R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first.... HOW?
One way to sort: Merge Sort

(i) For each 100 blk chunk of R:
- Read chunk
- Sort in memory
- Write to disk
(ii) Read all chunks + merge + write out

Sorted file

Memory

\[ \ldots \]

Sorted Chunks
Cost: Sort

Each tuple is read, written, read, written

so...

Sort cost R1: \( 4 \times 1,000 = 4,000 \)
Sort cost R2: \( 4 \times 500 = 2,000 \)
Example Merge Join

R1, R2 contiguous, but unordered

Total cost = sort cost + join cost

\[ = 6,000 + 1,500 = 7,500 \text{ IOs} \]
Comparison?

Iteration cost = 5,500 IOs
Merge cost = 7,500 IOs

Conclusion: so merge join does not pay off?

True sometimes? or always???
Next a case

Where sort-merge join beats the iteration join.
For:  
R1 = 10,000 blocks    contiguous
R2 = 5,000 blocks not ordered

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

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

Merge Join (with sort) WINS!
How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks

\[ \text{R1} \quad \begin{array}{c}
\text{100 chunks} \Rightarrow \text{to merge, need 100 blocks!}
\end{array} \]
In general:

Say k blocks in memory
x blocks for relation sort
# chunks = \( \frac{x}{k} \)  size of chunk = k

# chunks \leq buffers available for merge

so… (x/k) \leq k
or \( k^2 \geq x \) or \( k \geq \sqrt{x} \)

Size of buffer \geq \sqrt{file}
In our example

R1 is 1000 blocks, \( k \geq 31.62 \)

R2 is 500 blocks, \( k \geq 22.36 \)

Need at least 32 buffers
Can we improve on merge join?

Hint: do we need the fully sorted files?

R1

R2

sorted runs

Join?

--> Memory requirement?
Cost of improved merge join:

\[ C = \text{Read } R1 + \text{write } R1 \text{ into runs} + \text{read } R2 + \text{write } R2 \text{ into runs} + \text{join (via merging lists)} = 2000 + 1000 + 1500 = 4500 \]
Join: \( R1 \bowtie R2, R2 \bowtie R1 \)

- Iteration (nested loops)
- Merge join
- Join with index
- Hash join
Join with index (Conceptually)

Assume R2.C index

For each \( r1 \in R1 \) do

\[
\begin{align*}
    & X \leftarrow \text{index-lookup}(R2.C) \\
    & \text{for each } r2 \in X \text{ do} \\
    & \quad \text{output } (r1,r2) \text{ pair}
\end{align*}
\]
Example: Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered
- Assume R1.C index fits in memory
Cost:

Reads R2: 500 IOs
for each R2 tuple:
- probe index: free
- if match, read R1 tuple: 1 IO
What is expected # of matching tuples?

(a) say R1.C is key, R2.C is foreign key
   then expect = 1
(b) say V(R1, C) = 5000, T(R1) = 10,000
   with uniform assumption
   expect = 10,000/5,000 = 2
(c) say DOM(R1, C)=1,000,000,
    T(R1) = 10,000

   expect = 10,000/1,000,000 = 1/100
Total cost with index join

(a) Total cost = 500 + 5000(1)1 = 5,500

(b) Total cost = 500 + 5000(2)1 = 10,500

(c) Total cost = 500 + 5000(1/100)1 = 550
What if index does not fit in memory?

Example: say R1.C index is 201 blocks

• Keep root + 99 leaf nodes in memory
• Expected cost of each probe is
  \[ E = \left(\frac{0}{200}\right)^{99} + \left(\frac{1}{200}\right)^{101} \approx 0.5 \]
Total cost (including probes)

For case b:

\[ = 500 + 5000 \text{ [Probe + get records]} \]
\[ = 500 + 5000 \times [0.5 + 2] \text{ uniform assumption} \]
\[ = 500 + 12,500 = 13,000 \]

For case c:

\[ = 500 + 5000 \times [0.5 \times 1 + (1/100) \times 1] \]
\[ = 500 + 2500 + 50 = 3050 \text{ IOs} \]
What if?

- R2 was sorted on R2.C
- R1.C index was primary (clustered) index
- Buffer holds R1.C index
What if?

- R1.C index was clustered index
  - then relation R1 sorted on C
- R2 was sorted on R2.C
  - Then scan R2 retrieves R2 tuples for index matching in order of R1.C index access
- Buffer holds R1.C index
  - No IO cost for index lookup

\[
\text{Read R2} + \text{“Scan-Index”} + \text{“Read R1”} \\
500 + 200 + (1000?) = 1,700 \text{ IOs}
\]
### So far

<table>
<thead>
<tr>
<th>Iteration Type</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate R2 $\Join$ R1</td>
<td>55,000 (best)</td>
</tr>
<tr>
<td>Merge Join</td>
<td>______</td>
</tr>
<tr>
<td>Sort+ Merge Join</td>
<td>______</td>
</tr>
<tr>
<td>R1.C Index</td>
<td>______</td>
</tr>
<tr>
<td>R2.C Index</td>
<td>______</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Type</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate R2 $\Join$ R1</td>
<td>5500</td>
</tr>
<tr>
<td>Merge join</td>
<td>1500</td>
</tr>
<tr>
<td>Sort+Merge Join</td>
<td>7500 → 4500</td>
</tr>
<tr>
<td>R1.C Index</td>
<td>5500 → 3050 → 550</td>
</tr>
<tr>
<td>R2.C Index</td>
<td>______</td>
</tr>
</tbody>
</table>
Join:

- Iteration (nested loops)
- Merge join
- Join with index
- Hash join
Hash-Join

- Partition both relations using hash fct $h$.

- NOTE: R1 tuples in partition $i$ will only match R2 tuples in partition $i$. 

![Diagram showing the process of hash-join]

Original Relation  | B main memory buffers  | OUTPUT  | Partitions  
--- | --- | --- | ---
Disk  |  |  |  
INPUT  |  |  |  
hash function $h$  |  |  |  

1  | 2  | \(B-1\)  |  

Disk  |  |  |  

Hash-Join

- Partition both relations using hash fn \( h \)

- Read in a partition of \( R_1 \), hash it using \( h_2 \) (\( <> h! \)).

- Scan matching partition of \( R_2 \), and search for matches.
Query Optimizer needs ...

- File manager (if file is sorted, or contiguous)
- Index manager (if index exists and its properties)
- Buffer manager (how much space can be dedicated to the query process)
- Statistics manager (to provide estimates for expected matches)
Summary

- Iteration ok for “small” relations (relative to memory size)
- For equi-join, where relations not sorted and no indices exist, hash join is usually best.
- Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use merge join
- If index exists, it could be useful (depends on expected result size)
• Sort + merge join good for non-equi-join (e.g., R1.C > R2.C)
• If relations already sorted, use merge join
• If index exists, it could be useful (depends on expected result size)