Overview of Query Evaluation

Chapter 12
Join:

```
SELECT  S.sid, S.name, R.bid
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid
```

- Join is the most *common* and most *expensive* query operator
- Joins are widely studied and systems support several join algorithms
- A straightforward way for the join is an exhaustive nested loop

For each tuple r in R do
  For each tuple s in S
    if r.sid == s.sid do
      add <r, s> to result
Schema for Examples

Sailors \((\text{sid}: \text{integer}, \text{sname}: \text{string}, \text{rating}: \text{integer}, \text{age}: \text{real})\)
Reserves \((\text{sid}: \text{integer}, \text{bid}: \text{integer}, \text{day}: \text{dates}, \text{rname}: \text{string})\)

- **Sailors:**
  - No. of tuples: 40,000
  - No. of pages \(N\): 500
  - No. of tuples/page \(p_S\): 80

- **Reserves:**
  - No. of tuple: 100,000
  - No. of pages \(M\): 1,000
  - No. of tuples/page \(p_R\): 100

- Retrieving a page through hashing costs 1.2 I/O
- **Cost metric:** # of I/Os. We will ignore output costs.
Simple Nested Loops Join

For each tuple \( r \) in \( R \) do
  for each tuple \( s \) in \( S \) do
    if \( r_i = s_j \) then add \( <r, s> \) to result

- For each tuple in the outer relation \( R \), we scan the entire inner relation \( S \).
  - Cost: \( M + p_R \cdot M \cdot N = 1000 + 100 \cdot 1000 \cdot 500 \) I/Os.
    \[ = 50,001,000 \] I/Os.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages</td>
<td>( N = 500 )</td>
<td>( M = 1,000 )</td>
</tr>
<tr>
<td>Tuples/page</td>
<td>( p_S = 80 )</td>
<td>( p_R = 100 )</td>
</tr>
</tbody>
</table>
Page-Oriented Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_j then add <r, s> to result

- For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: \( M + M \times N = 1000 + 1000 \times 500 = 51,000 \)

- If smaller relation (S) is outer:
  - Cost: \( N + M \times N = 500 + 1000 \times 500 = 50,500 \)

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages</td>
<td>N=500</td>
<td>M=1,000</td>
</tr>
<tr>
<td>Tuples/page</td>
<td>( p_S = 80 )</td>
<td>( p_R = 100 )</td>
</tr>
</tbody>
</table>
Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold `block’’ of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.
Cost of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = \text{\lceil # of pages of outer / blocksize\rceil}

- With Reserves (R) as outer
  - Block size 100 \(\Rightarrow\) \(1,000 + (1,000/100) \times 500 = 6,000\)
  - Block size 90 \(\Rightarrow\) \(1,000 + \text{Ceil}(1,000/90) \times 500 = 7,000\)

- With 100-page block of Sailors as outer:
  - Block size 100: \(\Rightarrow\) \(500 + (500/100) \times 1,000 = 5,500\)
  - Block size 90: \(\Rightarrow\) \(500 + (500/90) \times 1,000 = 6,500\)

- With \textit{sequential reads} considered, analysis changes:
  - may be best to divide buffers evenly between R and S.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages</td>
<td>(N=500)</td>
<td>(M=1,000)</td>
</tr>
<tr>
<td>Tuples/page</td>
<td>(p_S = 80)</td>
<td>(p_R = 100)</td>
</tr>
</tbody>
</table>
Index Nested Loops

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.

- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.
Cost of Index Nested Loops

✓ Reserve is outer, Hash-index (Alt. 2) on sid of Sailors (as inner):
  ▪ Scan Reserves: 1000 page I/Os.
  ▪ For each Reserves tuple (100*1000 tuples): 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple.
  ▪ Cost: 1000 + 100,000 * 2.2 = 221,000 I/Os.

✓ Does it matter if the index is clustered or not?

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages</td>
<td>N=500</td>
<td>M=1,000</td>
</tr>
<tr>
<td>Tuples/page</td>
<td>p_S = 80</td>
<td>p_R = 100</td>
</tr>
</tbody>
</table>
Cost of Index Nested Loops

- Sailors is outer, Hash-index (Alt. 2) on sid of Reserves (as inner):
  - Scan Sailors: 500 page I/Os
  - For each Sailors tuple (80*500 tuples): 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples.
  - Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000).
  - Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
  - Clustered Index $\Rightarrow 50 + 40,000 \times 2.2 = 88,500$
  - Unclustered Index $\Rightarrow 500 + 40,000 \times 3.7 = 148,500$
**Sort-Merge Join** \((R \bowtie S)_{i=j}\)

- Sort \(R\) and \(S\) on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of \(R\) until current \(R\)-tuple \(\geq\) current \(S\) tuple, then advance scan of \(S\) until current \(S\)-tuple \(\geq\) current \(R\) tuple; do this until current \(R\) tuple = current \(S\) tuple.
  - At this point, all \(R\) tuples with same value in \(R_i\) (current \(R\) group) and all \(S\) tuples with same value in \(S_j\) (current \(S\) group) match; output \(<r, s>\) for all pairs of such tuples.
  - Then resume scanning \(R\) and \(S\).

- \(R\) is scanned once; each \(S\) group is scanned once per matching \(R\) tuple. (Multiple scans of an \(S\) group are likely to find needed pages in buffer.)
Join: Sort-Merge ($R \Join S_{i=j}$)

- Sort $R$ and $S$ on the join column, then scan them to do a "merge" (on join col.), and output result tuples.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke
Cost of Sort-Merge Join

- **Sorting** takes two passes, for each pass, we need to scan (read and write) each data record:
  - Cost for sorting Reserves: $2 \times 2 \times 1000 = 4000$
  - Cost for sorting Sailors: $2 \times 2 \times 500 = 2000$

- **Merging** needs only one global pass over the two tables with read only
  - Merging cost $= 1000 + 500 = 1500$

- **Total cost** $= 4000 + 2000 + 1500 = 7500$
**Hash-Join**

- Partition both relations using hash function $h$: R tuples in partition $i$ will only match S tuples in partition $i$.

- Read in a partition of R, hash it using $h_2$ ($<> h$). Scan matching partition of S, search for matches.
Cost of Hash-Join

- In partitioning phase, read+write both relns; $2(M+N)$. In matching phase, read both relns; $M+N$ I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Hash Join superior if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equalities over several attributes (e.g., \( R.sid = S.sid \) AND \( R.rname = S.sname \)):
  - For Index NL, build index on \(<sid, sname>\) (if S is inner); or use existing indexes on \(sid\) or \(sname\).
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., \( R.rname < S.sname \)):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.