Evaluation of Relational Operations

Chapter 14, Part A (Joins)
Relational Operations

- We will consider how to implement:
  - **Selection** (σ) Selects a subset of rows from relation.
  - **Projection** (π) Deletes unwanted columns from relation.
  - **Join** (⋈) Allows us to combine two relations.
  - **Set-difference** (―) Tuples in reln. 1, but not in reln. 2.
  - **Union** (∪) Tuples in reln. 1 and in reln. 2.
  - **Aggregation** (SUM, MIN, etc.) and GROUP BY

- Since each op returns a relation, ops can be composed!
  After we cover the operations, we will discuss how to optimize queries formed by composing them.
Schema for Examples

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

- Similar to old schema; rname added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
Equality Joins With One Join Column

SELECT * 
FROM Reserves R1, Sailors S1 
WHERE R1.sid=S1.sid

- In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.

- Assume: M tuples in $R$, $p_R$ tuples per page, N tuples in $S$, $p_S$ tuples per page.
  - In our examples, $R$ is Reserves and $S$ is Sailors.

- We will consider more complex join conditions later.

- **Cost metric**: # of I/Os. We will ignore output costs.
Simple Nested Loops Join

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: \( M + p_R \times M \times N = 1000 + 100 \times 1000 \times 500 \) I/Os.
- Page-oriented Nested Loops join: 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: \( M + M \times N = 1000 + 1000 \times 500 \)
  - If smaller relation (S) is outer, cost = \( 500 + 500 \times 1000 \)
Index Nested Loops Join

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

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: \( M + (M \times p_R) \times \text{cost of finding matching S tuples} \)
- 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.
Examples of Index Nested Loops

- Hash-index (Alt. 2) on sid of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.

- Hash-index (Alt. 2) on sid of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 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.
**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.
Examples of Block Nested Loops

- Cost: Scan of outer + \#outer blocks * scan of inner
  - \#outer blocks = \left\lfloor \frac{\# of pages of outer}{blocksize} \right\rfloor

- With Reserves (R) as outer, and 1000 pages of R:
  - Block size is 100
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.

- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
Sort-Merge Join (R $\bowtie$ S)  

- 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 Ri (current R group) and all S tuples with same value in Sj (current S group) match; output $<r, s>$ for all pairs of such tuples.
  - Then resume scanning R and S.
Example of Sort-Merge Join

<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>

- **Cost:** \( M \log M + N \log N + (M+N) \)
  - The cost of scanning, \( M+N \), could be \( M*N \) (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
**Hash-Join**

- **Partitioning Phase:**
  Partition both relations using hash fn \( h \): R tuples in partition \( i \) will only match S tuples in partition \( i \).

- **Probing Phase:**
  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.

→ Total Cost = $3(M + N)$

- In our running example, this is a total of 4500 I/Os.

- Sort-Merge Join vs. Hash Join:
  - Hash Join shown to be highly parallelizable. How?
  - Sort-Merge is less sensitive to data skew; result is sorted.