Evaluation of Relational Operations

Chapter 14, Part A (Joins)

Relational Operations

- We will consider how to implement:
  - **Selection** ($\sigma$): Selects a subset of rows from relation.
  - **Projection** ($\pi$): Deletes unwanted columns from relation.
  - **Join** ($\Join$): Allows us to combine two relations.
  - **Set-difference** ($\setminus$): Tuples in reln. 1, but not in reln. 2.
  - **Union** ($\cup$): 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, name: 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

- We will consider more complex join conditions later.
- **Cost metric**: # of I/Os. We will ignore output costs.

Simple Nested Loops Join

- foreach tuple r in R do
-   foreach tuple s in S do
-     if r.sid == s.sid then add <r, s> to result

- For each tuple in the outer relation R, we scan the entire inner relation S.
  - Cost: $M + p_R \times M \times N = 1000 + 100^2 \times 500$ 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: $M + M^2 N = 1000 + 100^2 \times 500 + 500 \times 1000$

Index Nested Loops Join

- foreach tuple r in R do
-   foreach tuple s in S where r.sid == s.sid do
-     add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it inner and exploit the index.
  - Cost: $M + (M^2 p_R) + \text{cost of finding matching S tuples}$
- 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: up to 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 = [ # of pages of outer / blocksize ]
  - With Reserves (R) as outer, and 100 pages of R:
    - 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.
    - If space for just 90 pages of R, we would scan S 12 times.
  - 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.

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

**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.
  - Advane scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= 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.

- 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.)

**Example of Sort-Merge Join**

<table>
<thead>
<tr>
<th>sid</th>
<th>snome</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>

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

**Refinement of Sort-Merge Join**

- We can combine the merging phases in the sorting of R and S with the merging required for the join.
  - With \(B > \sqrt{L}\), where L is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge` while checking the join condition.
  - **Cost:** read + write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.

- In practice, cost of sort-merge join, like the cost of external sorting, is **linear**.
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 \( h_2 \). Scan matching partition of \( S \), search for matches.

Observations on Hash-Join

- \# partitions \( k < B-1 \) (why?), and \( B-2 > \) size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing \( k \), we get:
  - \( k = B-1 \), and \( M/(B-1) < B-2 \), i.e., \( B \) must be \( > \sqrt{M} \)
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more \( R \) partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this \( R \)-partition with corresponding \( S \)-partition.

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 vs. Hash Join:
  - Given a minimum amount of memory (what is this, for each?) both have a cost of \( 3(M+N) \) I/Os. Hash Join superior on this count 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.