Evaluation of Relational Operations

Chapter 12, Part A

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 (\(-\)) 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, name: string)

- Similar to old schema; name 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

\[
\begin{align*}
\text{SELECT } * \\
\text{FROM Reserves R1, Sailors S1} \\
\text{WHERE R1.sid = S1.sid}
\end{align*}
\]

- In algebra: \(R \times S\) Common! Must be carefully optimized. \(R^n\) 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 \(r\) in \(R\) do
  - For each tuple \(s\) in \(S\) do
    - If \(r.s = s\) then add \(r, s\) to result

- For each tuple in the outer relation \(R\), we scan the entire inner relation \(S\).
  - Cost: \(M + p_R \times M \times N = 1000 + 100 \times 100 \times 300 = 1/0s\).
- 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: \(M + M \times N = 1000 + 100 \times 300 = 4000\).

Index Nested Loops Join

- For each tuple \(r\) in \(R\) do
  - For each tuple in \(S\) where \(r.s = s\) 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: \(M + (M \times p_R)\) 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 Access (2) or (3) for data entries) depends on clustering.
  - Clustered index: \(1/0\) (typical), unclustered: up to \(1 \times 0\) per matching \(S\) tuple.
Examples of Index Nested Loops

- Hash-index (Alt. 2) on sid of Sailors (as inner):
  - Scan Sailors: 1000 page I/Os, 100,000 tuples.
  - For each Sailors tuple 1:1/Os to get data entry in
    index, plus 1:1/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: 300 page I/Os, 80,000 tuples.
  - For each Sailors tuple 1:21/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
    2:1/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 in R block, 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 100/1.001; a total of 10 blocks.
  - Per page of R, we scan Sailors (S): 100,000 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 300 I/Os; a total of 5 blocks.
  - Per page of S, we scan Reserves: 5 M/1001 I/Os.
- With sequential reads considered, analysis changes:
  may be best to divide buffers evenly between R and S.

Sort-Merge Join (R = 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 == 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 R (current R
    group) and all S tuples with same value in S (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>name</th>
<th>rating</th>
<th>age</th>
<th>day</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7.0</td>
<td>45.0</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9.0</td>
<td>35.0</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8.0</td>
<td>55.5</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5.0</td>
<td>35.0</td>
<td>10/11/96</td>
<td>guppy</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10.0</td>
<td>35.0</td>
<td>11/12/96</td>
<td>rusty</td>
</tr>
</tbody>
</table>

- Cost: Mlog 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.
  (Note: a 2500 to 15000 I/Os)

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{n}, 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 4300 I/Os.
  - In practice, cost of sort-merge join, like the cost of
    external sorting, is linear.
### Cost of Hash-Join

- **In** partitioning phase, read+write both relns, 2(M+N).
- **In** matching phase, read both relns, M+N 1/Os.
- **In** our running example, this is a total of 4300 1/Os.
- **Sort-Merge Join vs. Hash Join:**
  - Given a minimum amount of memory (what is this, for each?) both have a cost of $3(M+N)$ 1/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; it matches likely to be much higher than for equality join.
  - Hash Join, Sort-Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

---

### 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.