Evaluation of Relational Operations

Yanlei Diao
UMass Amherst
March 01, 2007

Slides Courtesy of R. Ramakrishnan and J. Gehrke

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 (\( \bowtie \)) 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
  - Order By Returns tuples in specified order.
- 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.

Outline

- Evaluation of joins
- Evaluation of selections
- Evaluation of projections
- Evaluation of other operations
**Schema for Examples**

- **Sailors** `(sid: integer, sname: string, rating: integer, age: real)`
- **Reserves** `(sid: integer, bid: integer, day: date, rname: string)`
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

**Equality Joins With One Join Column**

- In algebra: R \( \bowtie \) S, natural join, common operation!
  - R \( \times \) S is large; R \( \times \) S followed by a selection is inefficient.
  - Must be carefully optimized.
- Assume: M pages in R, \( p_R \) tuples per page, N pages in S, \( p_S \) tuples per page.
- **Cost metric**: # of I/Os. Ignore output cost in analysis.

**SELECT** *
**FROM** Reserves R, Sailors S
**WHERE** R.sid = S.sid

**Simple Nested Loops Join (NLJ)**

```plaintext
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 tuple in the outer relation R, scan the entire inner relation S.
  - Cost: \( M + (p_R \times M) \times N = 1000 + 100 \times 1000 \times 500 = 1,000 + 100 \times 10^7 \) I/Os.
  - Assuming each I/O takes 10 ms, the join will take about 140 hours!
Page-Oriented Nested Loops Join

- How can we improve Simple NLJ?
- 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: \( M + M \times N = 1000 + 1000 \times 500 = 501,000 \) I/Os.
  - If each I/O takes 10 ms, the join will take 1.4 hours.
- Which relation should be the outer?
  - The smaller relation (S) should be the outer:
    - cost = 500 + 500*1000 = 500,500 I/Os.
- How many buffers do we need?

Block Nested Loops Join

- How can we utilize additional buffer pages?
  - If the smaller reln fits in memory, use it as outer, read the inner only once.
  - Otherwise, read a big chunk of it each time, resulting in reduced # times of reading the inner.
- Block NLJ:
  - Take the smaller reln, say R, as outer, the other as inner.
  - Buffer allocation: one buffer for scanning the inner S, one buffer for output, all remaining buffers for holding a “block” of outer R.

Block Nested Loops Join (Contd.)

```
foreach block in R do
    build a hash table on R-block
foreach S page
    foreach matching tuple r in R-block, s in S-page do
        add <r, s> to result
```
Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = [ # pages of outer / block size]
  - Given available buffer size B, block size is at most B-2.
- With Sailors (S) as outer, a block has 100 pages of S:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5 * 1000 = 5,500 I/Os.

Disk Behavior in Block NLJ

- What is the disk behavior in Block NLJ?
  - Reading outer: sequential for each block
  - Reading inner: sequential if output does not interfere; o.w., random. (Differ from the data access pattern in QLSM!)
- Optimization for sequential reads of the inner
  - Read S also in a block-based fashion.
  - May result in more passes, but reduced seeking time.

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*p_d) * 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: 1000 + 100*1000*2.2 = 221,000 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.
    - If uniform distribution, 2.5 reservations per sailor (100,000 / 40,000).
    - Cost of retrieving them is 1 or 2.5 I/Os (cluster?).
  - Total: 500 + 80*500*(2.2~3.7) = 88,500~148,500 I/Os.

Sort-Merge Join \((R \bowtie S)\)

- Sort R and S on join column using external sorting.
- Merge R and S on join column, output result tuples.

Repeat until either R or S is finished:

- Scanning:
  - Advance scan of R until current R-tuple \(\geq\)current S-tuple,
  - Advance scan of S until current S-tuple \(\geq\)current R tuple;
  - Do this until current R tuple = current S tuple.

- Matching:
  - Match all R tuples and S tuples with same value, output \(<r, s>\) for all pairs of such tuples.

Data access patterns for R and S?

Example of Sort-Merge Join

```
<table>
<thead>
<tr>
<th>sid</th>
<th>snme</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: \(M \log M + N \log N + \text{merging cost} (\in [M+N, M^2N])\)
  - The cost of merging could be \(M^2N\) (but quite unlikely). When?
  - \(M^2N\) is guaranteed in foreign key join; treat the referenced reln. as inner!
  - As with sorting, log M and log N are small numbers, e.g. 3, 4.
  - With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost is 7500 (assuming M+N).
    - \(\text{BNL cost: } 2500 (B=300), 5500 (B=100), 15000 (B=35)\)
Refinement of Sort-Merge Join

- **Idea:**
  - Sorting of R and S has respective merging phases
  - Join of R and S also has a merging phase
  - Combine all these merging phases!
- **Two-pass algorithm** for sort-merge join:
  - Pass 0: sort subfiles of R, S individually
  - Pass 1: merge sorted runs of R, merge sorted runs of S, and merge the resulting R and S files as they are generated by checking the join condition.

2-Pass Sort-Merge Algorithm

2-Pass Sort-Merge Algorithm

Memory Requirement and Cost

- **Memory requirement for 2-pass sort-merge:**
  - Assume \( U \) is the size of the larger relation, \( U = \max(M, N) \).
  - Sorting pass produces sorted runs of length up to 2B ("replacement sort")
  - \# of runs per relation \( \leq U/2B \).
  - Merging pass holds sorted runs of both relations and an output buffer, merges while checking join condition.
    \( 2(U/2B) < B \rightarrow B \geq U \)
- **Cost:**
  - read & write each relation in Pass 0
  - read each relation in merging pass
  - (+ writing result tuples, ignore here) = \( 3(M+N) \)
  - In example, cost goes down from 7500 to 4500 I/Os.
Idea: Partition both $R$ and $S$ using a hash function \( s.t. \) $R$ tuples will only match $S$ tuples in partition $i$.

**Hash-Join**

- **Partitioning**: Partition both relations using hash fn $h$: $R$ tuples will only match with $S$ tuples.
- **Probing**: Read in partition $i$ of $R$, build hash table on $R_i$ using $h_2 (<> h_1)$. Scan partition $i$ of $S$, search for matches.

**Memory Requirement**

- **Partitioning**: # partitions in memory $\leq B-1$,
- **Probing**: size of largest partition (to fit in memory) $\leq B-2$.
  - A little more memory is needed to build hash table, but ignored here.
- Assuming uniformly sized partitions, $L = \min(M, N)$:
  - $L / (B-1) < (B-2) \rightarrow B > \sqrt{L}$.
  - Hash-join works if the smaller relation satisfies above!
- What if hash fn $h$ does not partition uniformly and one or more $R$ partitions does not fit in memory?
  - Can apply hash-join technique recursively to do the join of this $R$-partition with the corresponding $S$-partition.

**Cost of Hash-Join**

- **Partitioning reads+writes both relns; 2(M+N).**
- **Probing reads both relns; M+N 1/Os.**
- **Total cost = 3(M+N).**
  - In our running example, a total of 4500 1/Os using hash join, less than 1 min (compared to 140 hours w. NLJ).
- **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.
  - Assuming $M+N$, what if $\sqrt{M} < B < \sqrt{N}$? Also, Hash Join is 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 and check the other join condition on the fly.
  - 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 B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins (clustered index is much preferred).
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be a winner here.

Outline

- Evaluation of joins
  - Evaluation of selections
  - Evaluation of projections
  - Evaluation of other operations

Using an Index for Selections

- Cost depends on # qualifying tuples, and clustering.
  - Cost of finding data entries (often small) + cost of retrieving records (could be large w/o clustering).
  - For $gpa > 3.0$, if 10% of tuples qualify (100 pages, 10,000 tuples), cost $\approx 100$ I/Os with a clustered index; otherwise, up to 10,000 I/Os.

- Important refinement for unclustered indexes:
  1. Find qualifying data entries.
  2. Sort the rid's of the data records to be retrieved.
  3. Fetch rid's in order.

  *Each data page is looked at just once, although # of such pages likely to be higher than with clustering.*
**Approach 1 to General Selections**

- (1) Find the most selective access path, retrieve tuples using it, and (2) apply any remaining terms that don’t match the index on the fly.
  - **Most selective access path**: An index or file scan that is expected to require the smallest # I/Os.
    - Terms that match this index reduce the number of tuples retrieved;
    - Other terms are used to discard some retrieved tuples, but do not affect I/O cost.
  - **Consider** \( \text{day}<8/9/94 \text{ AND bid}=5 \text{ AND sid}=3 \).  
    - A B+ tree index on \( \text{day} \) can be used; then, \( \text{bid}=5 \) and \( \text{sid}=3 \) must be checked for each retrieved tuple.
    - A hash index on \( <\text{bid, sid}> \) could be used; \( \text{day}<8/9/94 \) must then be checked on the fly.

**Approach 2: Intersection of Rids**

- If we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries:
  - Get sets of rids of data records using each matching index.
  - **Intersect** these sets of rids.
  - Retrieve the records and apply any remaining terms.
  - **Consider** \( \text{day}<8/9/94 \text{ AND bid}=5 \text{ AND sid}=3 \). If we have a B+ tree index on \( \text{day} \) and an index on \( \text{sid} \), both using Alternative [2], we can:
    - retrieve rids of records satisfying \( \text{day}<8/9/94 \) using the first, rids of records satisfying \( \text{sid}=3 \) using the second,
    - intersect these rids,
    - retrieve records and check \( \text{bid}=5 \).

**The Projection Operation**

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```

- Projection consists of two steps:
  - Remove unwanted attributes (i.e., those not specified in the projection).
  - Eliminate any duplicate tuples that are produced, if **DISTINCT** is specified.
- Algorithms: single relation sorting and hashing based on all remaining attributes.
Projection Based on Sorting

- Modify Pass 0 of external sort to eliminate unwanted fields.
  - Runs of about 2B pages are produced,
  - But tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)
- Modify merging passes to eliminate duplicates.
  - # result tuples smaller than input. Difference depends on # of duplicates.
- Cost: In Pass 0, read input relation (size M), write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass.
  - Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25.

Projection Based on Hashing

- Partitioning phase: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h_1$ to choose one of $B-1$ output buffers.
  - Result is $B-1$ partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.
- Duplicate elimination phase: For each partition, read it and build an in-memory hash table, using hash fn $h_2 (<> h_1)$ on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.
- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

Discussion of Projection

- Sort-based approach is the standard; better handling of skew and result is sorted.
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller?)
- If a tree index contains all wanted attributes as prefix of search key can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
  - E.g. projection on $<$sid, age>, search key on $<$sid, age, rating>.
Set Operations

- Intersection and cross-product special cases of join.
  - Intersection: equality on all fields.
- Union (Distinct) and Except similar; we'll do union.
- **Sorting** based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them, removing duplicates.
- **Hashing** based approach to union:
  - Partition R and S using hash function h.
  - For each R-partition, build in-memory hash table (using h2).
  - Scan S-partition. For each tuple, probe the hash table. If the
tuple is in the hash table, discard it; o.w. add it to the hash
table.

Aggregate Operations (AVG, MIN, etc.)

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the
SELECT or WHERE clauses, can do index-only scan.
- With grouping (GROUP BY):
  - Sort on group-by attributes, then scan relation and compute
aggregate for each group. (Can improve upon this by
combining sorting and aggregate computation.)
  - **Hashing** on group-by attributes also works.
  - Given tree index whose search key includes all attributes in
SELECT, WHERE and GROUP BY clauses: can do index-only scan;
if group-by attributes form prefix of search key, can
retrieve data entries/tuples in group-by order.

Summary

- A virtue of relational DBMSs: *queries are composed of*
  *a few basic operators*; the implementation of these
operators can be carefully tuned.
- Algorithms for evaluating relational operators use
some simple ideas extensively:
  - **Indexing**: Can use WHERE conditions to retrieve small
set of tuples (selections, joins).
  - **Iteration**: Sometimes, faster to scan all tuples even if
there is an index. (And sometimes, we can scan the data
entries in an index instead of the table itself.)
  - **Partitioning**: By using sorting or hashing, we can
partition the input tuples and replace an expensive
operation by similar operations on smaller inputs.
Summary (Contd.)

- Many implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on:
  - system state (e.g., memory) and
  - statistics (table size, # tuples matching value k).
- This is part of the broader task of optimizing a query composed of several ops.