Evaluation of Relational Operations

Yanlei Diao
UMass Amherst
March 6 - 25, 2007
Relational Operations

- We will consider how to implement:
  - **Selection** \((\sigma)\) Selects a subset of rows from relation.
  - **Join** \((\bowtie)\) Allows us to combine two relations.
  - **Projection** \((\pi)\) Deletes unwanted columns from relation.

- **Union** \((\cup)\) Tuples in either reln. 1 or reln. 2.
- **Intersection** \((\cap)\) Tuples in both reln. 1 and reln. 2.
- **Set-difference** \((\setminus)\) Tuples in reln. 1, but not in reln. 2.

- **GROUP BY** and **Aggregation** (SUM, MIN, etc.)
Outline

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

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

- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.

- **Reserves:**
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- **Cost metric:** # I/Os
Using an Index for Selections

Cost of selection =
- finding data entries (often small) +
- cost of retrieving records (could be large w/o clustering).

Cost factors: \# qualifying tuples and clustering.
- ‘rating > 8’: if 20% of tuples qualify, we have 500/5=100 pages and 80*100=8,000 tuples. Cost ≈
  - 100 I/Os with a clustered index;
  - otherwise, up to 8,000 I/Os!
General Selections

- A boolean combination of predicates connected using AND and OR.
  - Conjunctive Normal Form (CNF), e.g.
    - ‘(pred1 OR pred2) AND (pred3 OR pred4)’
- A file scan always works for general selections.
- An index scan works when a predicate is a conjunct of the CNF.
  - E.g. an index matching pred 1 can be used for
    - ‘pred1 AND (pred3 OR pred4)’
General Selections w/o Disjunction

- CNF without OR: e.g. ‘pred 1 AND pred 2 AND pred 3’
  
1. Find the most selective access path, retrieve tuples using it
   - An index or file scan that is expected to require the smallest # I/Os.

2. Apply remaining terms that don’t match index on the fly.
   - Other terms only discard retrieved tuples, but do not affect I/O cost.

‘day<8/9/94 AND bid=5 AND sid=3’

- A B+ tree index on day can be used; then, bid=5 and sid=3 must be checked for each retrieved tuple.

- A hash index on <bid, sid> could be used; day<8/9/94 must then be checked on the fly.
Improvement: Intersection of Rids

- 2+ matching indexes (Alt 2 or 3 for data entries):
  1. Get sets of rids of data records using each index.
  2. Intersect these sets of rids.
  3. Retrieve the records and apply any remaining terms.

‘day<8/9/94 AND bid=5 AND sid=3’

Given a B+ tree index on day and an index on sid, both using Alt 2:

1. retrieve rids of records satisfying day<8/9/94 using the first, rids of records satisfying sid=3 using the second,
2. intersect these rids,
3. retrieve records and check bid=5.
Outline

- Evaluation of selections
- Evaluation of joins
- Evaluation of projections
- Evaluation of other operations
Equality Joins With One Join Column

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

- R ⨝ S, natural join. Very common operation!
- Semantics: cross product (\( \times \)) followed by selection (\( \sigma \))
  - If \( R \times S \) is large, \( R \times S \) followed by a selection is inefficient.
  - Must be carefully optimized.
- **Cost metric:** \# of I/Os. Ignore output cost in analysis.
  - R: M pages, \( p_R \) tuples per page
  - S: N pages, \( p_S \) tuples per page.
Schema for Examples

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

- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.

- **Reserves:**
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- **Cost metric:** # I/Os
Simple Nested Loops Join (NLJ)

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 \( R \), scan the entire inner relation \( S \).
  - Cost: \( M + (p_R * M) * N = 1000 + 100*1000*500 = 1,000 + 5 * 10^7 \) I/Os.
  - Assuming 10 ms per I/O, the join will take about 140 hours!
Page-Oriented Nested Loops Join

- How can we improve Simple NLJ?

```plaintext
foreach page of R do
  foreach page of S do
    write out each matching pair <r, s>
    // r is in R-page, s is in S-page
```

- Cost: \( M + M \times N = 1000 + 1000 \times 500 = 501,000 \) I/Os.
- If 10 ms per I/O, the join will take 1.4 hours.

- Which relation should be the outer?
  - If smaller relation (S) is the outer:
    cost = 500 + 500 \times 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, hence reducing # times of reading the inner.

- Block NLJ:
  - Take the smaller reln, say R, as outer, the other as inner.
  - Buffer allocation:
    - 1 buffer for scanning the inner S
    - 1 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
  for each matching tuple r in R-block, s in S-page do
    add <r, s> to result
Cost of Block Nested Loops Join

- Cost: Scan of outer + #outer blocks * scan of inner
  - B buffer pages available
  - Cost = size of outer + [size of outer / B-2] * size of inner

- E.g. B=102, Sailors S = 500 pages, Reserves R = 1000 pages.
  - What is the cost if S is outer, R is inner?
    - A block = B-2 = 100 pages
    - Cost = 500 + [500/100] * 1000 = 5,500 I/Os.
  - What is the cost if we swap R and S?
    - Cost = 1000 + [1000/100] * 500 = 6,000 I/Os.
  - Which relation should be the outer for smaller cost?
Index Nested Loops Join

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

```plaintext
foreach tuple r in R do
    foreach tuple s in S where r == s (via index lookup)
        add <r, s> to result
```

- Cost: \( M + (M \times p_R \times \text{cost of finding matching S tuples}) \)
  - For each R tuple, cost of probing S index:
    - About 1.2 I/O for hash index, 2-4 I/O’s for B+ tree.
  - Cost of finding all S matches (assuming Alt. 2 or 3):
    - Clustered index: 1 I/O (typical).
    - Unclustered: up to 1 I/O per matching S tuple.
Examples of Index Nested Loops

Sailors ▶◁_{sid} Reserves
- Sailors: Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Reserves: Each tuple is 40 bytes, 100 tuples per page, 1000 pages.

- Hash-index (Alt. 2) on \textit{sid of Sailors} (as inner):
  - Scan \textit{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 \textit{(exactly one)} matching Sailors tuple.
  - Total: 1000+ 100*1000*2.2 = 221,000 I/Os.
Examples of Index Nested Loops

Sailors $\triangleright\triangleleft_{\text{sid}}$ Reserves

- Sailors: Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- Reserves: Each tuple is 40 bytes, 100 tuples per page, 1000 pages.

- 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** \((R \bowtie S)_{i=j}\)

- **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 (called R-group and S-group of the current value).
  - Output \(<r, s>\) for all pairs of such tuples.
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>yummy</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: $M \log M + N \log N + \text{Merging\_cost}$
  - Merging\_cost $\in [M+N, M*N]$  
  - M*N: possible but uncommon. When?  
  - M+N: *foreign key join* with the referenced reln. as inner.

- With 35 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost is 7500 (assuming M+N).
Refinement of Sort-Merge Join

- Key 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 and S
    - 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.
Merging in 2-Pass Sort-Merge

Relation R

Run1 of R
Run2 of R
RunK of R

Relation S

Run1 of S
Run2 of S
RunK of S

B Main memory buffers

OUTPUT

Join Results
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"). So,
    
    \[
    \frac{U}{2B} \leq \text{# of runs per relation} \leq \frac{U}{2B}.
    \]
  - Merging pass holds sorted runs of both relations and an output buffer, merges while checking join condition.
    
    \[
    2 \left( \frac{U}{2B} \right) + 1 \leq B \rightarrow B > \sqrt{U}
    \]

- 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.
**Hash-Join**

- **Key idea**: Partition both R and S using a hash function s.t. R tuples will only match S tuples in partition i.

- **Phase 1 Partitioning**: Partition both relations using hash fn $h$ (R_i tuples will only match with S_i tuples).
Hash-Join

- **Phase 2 Probing:**
  - Read in partition $R_i$, build hash table on $R_i$ using $h_2$ ($<> h!$).
  - Scan partition $S_i$, one page at a time, search for matches.

![Diagram of Hash-Join process]

- **Partitions of R & S**
  - Disk
  - Hash function $h_2$

- **Hash table for partition $R_i$ (k ≤ B-2 pages)**
  - Input buffer for $S_i$
  - Output buffer
  - B main memory buffers

- **Join Result**
  - Disk
  - Output buffer
  - $h_2$
Memory Requirement

- Partitioning: # partitions in memory \( \leq B-1 \),
  - Probing: to fit in memory, size of partition \( \leq B-2 \).
  - A little more memory needed to build hash table, but ignored here.

- Assuming uniformly sized partitions, \( L = \min(M, N) \):
  - \( \frac{L}{(B-1)} < (B-2) \Rightarrow B > \sqrt{L} \)
  - Hash-join works if the smaller relation satisfies above! Use the smaller relation as the building relation in the probing phase.

- 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 I/Os.
- Total cost = 3(M+N).
  - In our running example, a total of 4500 I/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) I/Os.
  - Hash Join superior on this count if relation sizes differ greatly. Assuming M<N, what if sqrt(M) < B < sqrt(N)?
  - Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equalities over several attributes (e.g., \textit{R.sid=S.sid AND R.rname=S.sname}): 
  - Block NL works fine.
  - For Index NL, build index on \textit{<sid, sname>} (if S is inner); or use existing indexes on \textit{sid} or \textit{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.
General Join Conditions

- 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.
  - Block NL often works well.
- Hash Join, Sort Merge Join not applicable.
Outline

- Evaluation of selections
- Evaluation of joins
- Evaluation of projections
- Evaluation of other operations
The Projection Operation

- 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.

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
**Projection Based on Sorting**

- **Pass 0 of sorting:** modified to remove 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.)

- **Merging passes:** modified to eliminate duplicates.
  - # result tuples smaller than input. (Difference depends on # of duplicates.)

- **Cost \( \leq 3M \) (input relation size M, ignoring output cost)
  - Pass 0: read relation, write out same num. of smaller tuples.
  - Merging passes, fewer tuples written out in each pass.
  - Reserves: 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25.
Projection Based on Hashing

- **Partitioning phase**: Partition input relation using $h1$; for each tuple, discard unwanted fields.
  - Result is B-1 partitions (of tuples with only wanted fields). 2 tuples from different partitions guaranteed to be distinct.

- **Duplicate elimination phase**: Read each partition, build an in-memory hash table using $h2$ on all fields, discard duplicates:
  - For each tuple $t$, probe the hash table.
    - If we find a match, discard $t$;
    - Otherwise, output $t$ and add it to the hash table.
  - If partition does not fit in memory, apply hash-based projection algorithm recursively to this partition.

- **Cost $\leq 3M$**
  - For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.
Outline

- Evaluation of selections
- Evaluation of joins
- Evaluation of projections
- Evaluation of other operations
  - Set operators
  - Aggregation and Group By
Set Operations

- **Intersection**: Tuples in both reln. 1 and reln. 2.

  Equality join on *all* fields!
Set Operations

- **Union**: Tuples in either reln. 1 or reln. 2.
  - Selects *distinct* values only.
  - **Sorting** based (over two relations):
    - Sort both relations on combination of *all* attributes.
    - Scan sorted relations, merge them and remove duplicates.
  - **Hashing** based (over two relations):
    - Partition R and S using $h1$ on *all* attributes.
    - For each R-partition, (1) write it out (as part of the answer) and (2) build an in-memory hash table using $h2$. For each tuple in S-partition, probe the hash table.
      - If the S tuple is in the hash table, discard it.
      - Otherwise, output it.
Set Operations

- **Set-difference**: Tuples in reln. 1, but not in reln. 2.
  - Sorting-based solution, similar to Union.
  - Hashing-based solution, similar to Union.

What changes are needed?
Aggregate Operations (AVG, MIN, etc.)

```
SELECT min(S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

- Aggregation without grouping
  - **File scan**: in general, requires scanning the relation.
  - **Index only scan**: if we have an index whose search key includes all attributes in the SELECT and WHERE clauses, e.g. B+tree on <rating, age>.
Aggregate Operations (contd.)

SELECT \( \min(S.\text{age}) \)
FROM Sailors S
WHERE S.rating > 5
GROUP BY S.rating

- Aggregation with grouping (GROUP BY)
- Single-relation sorting: sort by group-by attribute(s); compute aggregate for each group in last merging phase.
- Single-relation hashing: hash on group-by attribute(s); compute aggregate using the in-memory hash table for each partition.
- Index only scan: if we have a tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, e.g. B+tree on <rating, age>.
Summary

- 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 implementations 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.