Lecture #14: Implementation of Relational Operations (R&G ch. 12 and 14)

Outline

• introduction
• selection
• projection
• join
• set & aggregate operations

Introduction

• Today’s topic: QUERY PROCESSING
• Some database operations are expensive
• Can greatly improve performance by being “smart”
  – e.g., can speed up 1,000,000x over naïve approach
**Introduction (cnt’d)**

- Main weapons are:
  - clever implementation techniques for operators
  - exploiting “equivalencies” of relational operators
  - using statistics and cost models to choose among these.

**A Really Bad Query Optimizer**

- For each Select-From-Where query block
  - do cartesian products first
  - then do selections
  - etc, i.e.:
    - GROUP BY, HAVING
    - projections
    - ORDER BY
- Incredibly inefficient
  - Huge intermediate results!

**Cost-based Query Sub-System**

Queries → Query Parser → Query Optimizer → Plan Generator → Plan Cost Estimator → Catalog Manager → Schema Statistics → Query Plan Evaluator

Usually there is a heuristics-based rewriting step before the cost-based steps.
The Query Optimization Game

- “Optimizer” is a bit of a misnomer…
- Goal is to pick a “good” (i.e., low expected cost) plan.
  - Involves choosing access methods, physical operators, operator orders, …
  - Notion of cost is based on an abstract “cost model”

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
- Recall: ops can be composed!
- Later, we’ll see how to optimize queries with many ops

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.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - N=500, p_s=80.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - M=1000, p_r=100.
Outline

• introduction
• selection
• projection
• join
• set & aggregate operations

Simple Selections

• Of the form $\sigma_{R.\text{attr} \ op \ value} (R)$
• Question: how best to perform?

```
SELECT *
FROM Reserves R
WHERE R.fname < 'C%'
```
Simple Selections

- Size of result approximated as
  \[ \text{size of } R \times \text{reduction factor} \]
  - “reduction factor” is also called selectivity.
  - estimate of reduction factors is based on statistics – we will discuss shortly.

Alternatives for Simple Selections

- With no index, unsorted:
  - Must essentially scan the whole relation
  - cost is \( M \times \text{#pages in } R \). For “reserves” = 1000 I/Os.

Simple Selections (cnt’d)

- With no index, sorted:
  - cost of binary search + number of pages containing results.
  - For ‘reserves’ = 10 I/Os + \( \lceil \text{selectivity} \times \text{#pages} \rceil \)
Simple Selections (cnt’d)

- With an index on selection attribute:
  - Use index to find qualifying data entries,
  - then retrieve corresponding data records.
  - (Hash index useful only for equality selections.)

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost:
    - finding qualifying data entries (typically small)
    - plus cost of retrieving records (could be large w/o clustering).

Selections using Index (cnt’d)
Selections using Index (cnt’d)

– In example “reserves” relation, if 10% of tuples qualify (100 pages, 10,000 tuples).
  • With a *clustered* index, cost is little more than 100 I/Os;
  • if *unclustered*, could be 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. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

General Selection Conditions

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

• Q: What would you do?
General Selection Conditions

\( (\text{day}<8/9/94 \text{ AND } \text{rname}='\text{Paul}') \text{ OR bid}=5 \text{ OR sid}=3 \)

- **Q:** What would you do?
- **A:** try to find a selective (clustering) index. Specifically:

\( (\text{day}<8/9/94 \text{ OR bid}=5 \text{ OR sid}=3 ) \text{ OR (rname}='\text{Paul}' \text{ OR bid}=5 \text{ OR sid}=3) \)

- Convert to conjunctive normal form (CNF):
  - \( (\text{day}<8/9/94 \text{ OR bid}=5 \text{ OR sid}=3 ) \text{ AND (rname}='\text{Paul}' \text{ OR bid}=5 \text{ OR sid}=3) \)
- We only discuss the case with no ORs (a conjunction of terms of the form \text{attr op value}).

General Selection Conditions

- A **B-tree** index matches (a conjunction of) terms that involve only attributes in a **prefix** of the search key.
  - Index on \( <a, b, c> \) matches \( a=5 \text{ AND } b=3 \), but not \( b=3 \).
- For **Hash** index, must have **all** attributes in search key
Two Approaches to General Selections

• **#1**: Find the cheapest access path, retrieve tuples using it, and apply any remaining terms that don’t match the index
• **#2**: get rids from first index; rids from second index; intersect and fetch.

Cheapest Access Path - Example

• **Consider** `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.  
• Similarly, a hash index on `<bid, sid>` could be used;  
  – *Then*, `day<8/9/94` must be checked.
Cheapest Access Path - cnt’d

• Consider \( \text{day} < 8/9/94 \ AND \ \text{bid}=5 \ AND \ \text{sid}=3 \).

• \text{How about a B+tree on} \ <\text{rname,day}>?\ 
• \text{How about a B+tree on} \ <\text{day, rname}>?\ 
• \text{How about a Hash index on} \ <\text{day, rname}>?\ 

Intersection of RIDs

• Approach #2. if we have 2 or more matching indexes (w/Alternatives (2) or (3) for data entries):
  – Get sets of rids of data records using each matching index.
  – Then intersect these sets of rids.
  – Retrieve the records and apply any remaining terms.

Intersection of RIDs (cnt’d)

• EXAMPLE: Consider \( \text{day}<8/9/94 \ AND \ \text{bid}=5 \ AND \ \text{sid}=3 \).
• With a \( \text{B+ tree index on day} \) and an index on \( \text{sid} \),
• we can retrieve rids of records satisfying \( \text{day}<8/9/94 \) using the first,
• rids of recs satisfying \( \text{sid}=3 \) using the second,
• intersect,
• retrieve records and check \( \text{bid}=5 \).
Outline

• introduction
• selection
• projection
  • join
  • set & aggregate operations

The Projection Operation

• Issue is removing duplicates.
• Basic approach: sorting
  – 1. Scan R, extract only the needed attrs (why?)
  – 2. Sort the resulting set
  – 3. Remove adjacent duplicates
Cost: Reserves with size ratio 0.25 = 250 pages. With 20 buffer pages can sort in 2 passes, so
  \[1000 + 250 + 2 \times 250 + 250 = 2500\text{ I/Os} \]

Projection

• Can improve by modifying external sort algorithm (see chapter 13):
  – Modify Pass 0 of external sort to eliminate unwanted fields.
  – Modify merging passes to eliminate duplicates.
Cost: for above case: read 1000 pages, write out 250 in runs of 40 pages, merge runs = 1000 + 250 + 250 = 1500.
Projection with Hashing

- we saw it earlier:
- Phase 1: partition
- Phase 2: reHash (now: dup. elim.)

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.

  ![Diagram]

  **Diagram Description**
  - Original Relation
  - INPUT
  - Hash function $h_1$
  - OUTPUT
  - Partitions
  - Disk
  - B main memory buffers
  - B-1

DupElim with Hashing (cnt’d)

- **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.
DupElim with Hashing (cnt’d)

- Cost:
  - assuming partitions fit in memory (i.e. #bufs \( \geq \) square root of the #of pages of projected tuples)
  - read 1000 pages and write out partitions of projected tuples (250 pages)
  - Do dup elim on each partition (total 250 page reads)
  - Total : 1500 I/Os.

Discussion of Projection

- Sort-based approach: better handling of skew and result is sorted.
- If enough buffers, both have same I/O cost: \( M + 2T \) where \( M \) is #pgs in \( R \), \( T \) is #pgs of \( R \) with unneeded attributes removed.
  - Although many systems don’t use the specialized sort.

Discussion of Projection

- 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!)
Discussion of Projection

- If an ordered (i.e., 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.

Outline

- introduction
- selection
- projection
- join
  - set & aggregate operations

Joins

- Joins are very common.
- Joins can be very expensive (cross product in worst case).
- Many approaches to reduce join cost.
Joins

• Join techniques we will cover:
  – Nested-loops join
  – Index-nested loops join
  – Sort-merge join
  – Hash join

Equality Joins With One Join Column

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

• In algebra: R \times S. Common! Must be carefully optimized. R \times S is large; so, R \times S followed by a selection is inefficient.
• Remember, join is associative and commutative.

Equality Joins

• Assume:
  – M pages in R, \( p_R \) tuples per page, \( m \) tuples total
  – N pages in S, \( p_S \) tuples per page, \( n \) tuples total
  – 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

- Algorithm #0: (naive) nested loop \((SLOW!)\)

\[
\begin{array}{c}
R(A_1, \ldots) \\
\downarrow \\
m \\
\end{array}
\quad
\begin{array}{c}
S(A_1, \ldots) \\
\downarrow \\
n \\
\end{array}
\]

for each tuple \(r\) of \(R\)
for each tuple \(s\) of \(S\)
print, if they match
Simple Nested loops

- Algorithm #0: why is it bad?
- how many disk accesses (‘M’ and ‘N’ are the number of blocks for ‘R’ and ‘S’)?

\[ M + m \times N \]

M pages, m tuples
R(A,...)

N pages, n tuples
S(A,...)

Simple Nested Loops Join

- Actual number
- \((p_R \times M) \times N + M = 100 \times 1000 \times 500 + 1000\) I/Os.
  - At 10ms/IO, Total: ???
- What if smaller relation (S) was outer?
- What assumptions are being made here?
Simple Nested Loops Join

- Actual number
  - \((p_R \times M) \times N + M = 100 \times 1000 \times 500 + 1000\) I/Os.
    - At 10ms/IO, Total: \(~6\) days (!)
- What if smaller relation (S) was outer?
  - slightly better
- What assumptions are being made here?
  - 1 buffer for each table (and 1 for output)

(Blocked) Nested loops

- Algorithm #1: Blocked nested-loop join
  - read in a block of R
    - read in a block of S
      - print matching tuples
    COST?

Nested loops

- Algorithm #1: Blocked nested-loop join
  - read in a block of R
    - read in a block of S
      - print matching tuples
    COST = \(M + M \times N\)
Nested loops

• Which one should be the outer relation?

\[ \text{COST} = M + M \times N \]

- M pages, m tuples
- N pages, n tuples

A: the smallest (page-wise)

\[ \text{Cost} = 1000 + 1000 \times 500 = 501,000 \]

\[ \approx 5010 \text{ sec} \sim 1.4h \]
Nested loops

- M = 1000, N = 500 - if smaller is outer:
- Cost = 500 + 1000 * 500 = 500,500
- = 5005 sec ~ 1.4h

COST = N + M * N

• What if we have B buffers available?
- A: give B - 2 buffers to outer, 1 to inner, 1 for output
Nested loops

- Algorithm #1: Blocked nested-loop join
  - read in $B-2$ blocks of $R$
  - read in a block of $S$
  - print matching tuples

\[
\text{COST} = \frac{M}{B-2} \times N
\]
Nested loops

- If smallest (outer) fits in memory
- (ie., B = N+2),
- Cost = ?

\[ \text{COST} = \frac{M}{B-2} \times N + M \]

Nested loops

- If smallest (outer) fits in memory
- (ie., B = N+2),
- Cost = N + M (minimum)

\[ \text{COST} = \frac{M}{B-2} \times N + M \]

Nested loops - guidelines

- pick as outer the smallest table (= fewest pages)
- fit as much of it in memory as possible
- loop over the inner
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join

Index NL join

- use an existing index, or even build one on the fly
- cost: $M + m \cdot c$ (c: look-up cost)

\[ \begin{align*}
M & \text{ pages,} \\
m & \text{ tuples} \\
\downarrow & \text{ } \\
R(A_{..}) \\
\uparrow & \text{ } \\
S(A, ......) \\
\downarrow & \text{ } \\
N & \text{ pages,} \\
n & \text{ tuples}
\end{align*} \]
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join

Sort-merge join

- sort both on joining attributed
- scan each and merge
- Cost, given B buffers?

\[
\text{Cost} \approx 2 \times M \times \log M / \log B + 2 \times N \times \log N / \log B + M + N
\]
Sort-Merge Join

- Useful if
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes
- “Merge” phase can require some backtracking if duplicate values appear in join column

Example of Sort-Merge Join

<table>
<thead>
<tr>
<th>sid</th>
<th>sid</th>
<th>rating</th>
<th>age</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>28</td>
<td>7</td>
<td>45.0</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>9</td>
<td>35.0</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>8</td>
<td>55.5</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>lubber</td>
</tr>
<tr>
<td>44</td>
<td>31</td>
<td>5</td>
<td>35.0</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>58</td>
<td>10</td>
<td>35.0</td>
<td>dustin</td>
</tr>
</tbody>
</table>
Example of Sort-Merge Join

• With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
• (while BNL cost: 2,500 to 15,000 I/Os)

Sort-merge join

• Worst case for merging phase?
• Cost?

Refinements

• All the refinements of external sorting
• plus overlapping of the merging of sorting with the merging of joining.
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join

Hash joins

- hash join: use hashing function \( h() \)
  - hash ‘R’ into (0, 1, ..., ‘max’) buckets
  - hash ‘S’ into buckets (same hash function)
  - join each pair of matching buckets

Hash join - details

- how to join each pair of partitions \( Hr_i, Hs_i \) ?
- A: build another hash table for \( Hs_i \), and probe it with each tuple of \( Hr_i \)
Hash join - details

• In more detail:
• Choose the (page-wise) smallest - if it fits in memory, do ~NL
  – and, actually, build a hash table (with h2() != h())
  – and probe it, with each tuple of the other

Hash join details

• what if Hs-i is too large to fit in main-memory?
• A: recursive partitioning
• more details (overflows, hybrid hash joins): in book
• cost of hash join? (if we have enough buffers:)
  \(3(M + N)\) (why?)

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.
Hash join details

- [cost of hash join? (if we have enough buffers:)]
  \[3(M + N)\]
- What is ‘enough’? sqrt(N), or sqrt(M)?

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.
Sort-Merge vs Hash join

• **Hash Join Pros:**
  – ??
  – ??
  – ??

• **Sort-Merge Join Pros:**
  – ??

**Hash Join Pros:**
  – Superior if relation sizes differ greatly
  – Shown to be highly parallelizable (*beyond scope of class*)

**Sort-Merge Join Pros:**
  – ??

**Sort-Merge vs Hash join**

• **Hash Join Pros:**
  – Superior if relation sizes differ greatly
  – Shown to be highly parallelizable (*beyond scope of class*)

• **Sort-Merge Join Pros:**
  – Less sensitive to data skew
  – Result is sorted (may help “upstream” operators)
  – goes faster if one or both inputs already sorted
General Join Conditions

- Equalities over several attributes (e.g., $R.sid=S.sid$ and $R.rname=S.sname$): What to do?

- Equalities over several attributes (e.g., $R.sid=S.sid$ and $R.rname=S.sname$):
  - all previous methods apply, using the composite key

General Join Conditions

- Inequality conditions (e.g., $R.rname < S.sname$):
  - which methods still apply?
    - NL
    - index NL
    - Sort merge
    - Hash join
General Join Conditions

- Inequality conditions (e.g., $r_{name} < s_{name}$):
  - which methods still apply?
    - NL (probably, the best!)
    - index NL (only if clustered index)
    - Sort merge (does not apply!) (why?)
    - Hash join (does not apply!) (why?)

Outline

- introduction
- selection
- projection
- join
  - set & aggregate operations

Set Operations

- Intersection and cross-product: special cases of join
- Union (Distinct) and Except: similar; we’ll do union (with dup. elim.):
- Effectively: concatenate; use sorting or hashing
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - Alternative: Merge runs from Pass 0 for both relations.
Set Operations, cont’d

• Hash based approach to union:
  – Partition R and S using hash function \( h \).
  – For each S partition, build in-memory hash table (using \( h2 \)), scan corresponding R partition and add tuples to table while discarding duplicates.

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.

```sql
select count(*)
from takes;
```

Aggregate Operations (AVG, MIN, etc.)

• With grouping: how would you do it, using:
  – Sorting
  – Hashing
  – A suitable tree index

```sql
select ssn, count(*)
from takes
group by ssn;
```
Aggregate Operations (AVG, MIN, etc.)

• With grouping:
  – 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: similarly.
  – 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.

Impact of Buffering

• If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.
• Repeated access patterns interact with buffer replacement policy.
  – e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (sequential flooding).
  – Does replacement policy matter for Block Nested Loops?
  – What about Index Nested Loops?

Summary

• A virtue of relational DBMSs:
  – queries are composed of a few basic operators
  – The implementation of these operators can be carefully tuned
    – Important to do this!
• Many alternative implementation techniques for each operator
  – No universally superior technique for most operators.

“it depends” [Guy Lohman (IBM)]
Summary cont’d

• Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc.
  – Part of the broader task of optimizing a query composed of several ops.