Evaluation of relational operators

Kathleen Durant
CS 3200 Lecture 17
Why is it important?

• Now that we know about the benefits of indexes, how does the DBMS know when to use them?
• An SQL query can be implemented in many ways, but which one is best?
  – Perform selection before or after join etc.
  – Many ways of physically implementing a join (or other relational operator), how to choose the right one?
• The DBMS does this automatically, but we need to understand it to know what performance to expect
Query Evaluation

• SQL query is implemented by a query plan
  – Tree of relational operators
    • Each internal node operates on its children
    • Can choose different operator implementations

• Two main issues in query optimization:
  – For a given query, what plans are considered?
    • Algorithm to search plan space for cheapest (estimated) plan.
  – How is the cost of a plan estimated?

• Ideally: Want to find best plan.
  – Practically: Avoid worst plans!
Tree of relational operators

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

SELECT sid
FROM Sailors NATURAL JOIN Reserves
WHERE bid = 100 AND rating > 5;

\[\pi_{sid} (\sigma_{bid=100 \text{ AND} \ rating>5} (\text{Sailors} \bowtie\text{Reserves}))\]

RA expressions are represented by an expression tree.

An algorithm is chosen for each node in the expression tree.
Approaches to Evaluation

• 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.

* Watch for these techniques as we discuss query evaluation during this lecture
Statistics and Information Schema

- Need information about the relations and indexes involved. Catalog typically contains:
  - #tuples (NTuples) and #pages (NPages) for each relation.
  - #distinct key values (NKeys), INPages index pages, and low/high key values (ILow/IHigh) for each index.
  - Index height (IHeight) for each tree index.
  - Catalog data stored in tables; can be queried.

- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; costs are approximate anyway, so slight inconsistency expected.

- More detailed information (e.g., histograms of the values in some field) sometimes stored.
Access Paths : Method for retrieval

• Access path = way of retrieving tuples:
  – File scan, or index that matches a selection (in the query)
  – Cost depends heavily on access path selected

• A tree index matches (a conjunction of) conditions that involve only attributes in a prefix of the search key.

• A hash index matches (a conjunction of) conditions that has a term attribute = value for every attribute in the search key of the index.

• Selection conditions are first converted to conjunctive normal form (CNF):
  – E.g., (day<8/9/94 OR bid=5 OR sid=3 ) AND (rname=‘Paul’ OR bid=5 OR sid=3)
### Matching an index

<table>
<thead>
<tr>
<th>Search key &lt;a, b, c&gt;</th>
<th>Tree Index</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. a=5 and b= 3?</td>
<td>1. Yes</td>
<td>1. No</td>
</tr>
<tr>
<td>2. a &gt; 5 and b &lt; 3</td>
<td>2. Yes</td>
<td>2. No</td>
</tr>
<tr>
<td>3. b=3</td>
<td>3. No</td>
<td>3. No</td>
</tr>
<tr>
<td>4. a=7 and b=5 and</td>
<td>4. Yes</td>
<td>4. Yes</td>
</tr>
<tr>
<td>c=4 and d&gt;4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. a=7 and c=5</td>
<td>5. Yes</td>
<td>5. No</td>
</tr>
</tbody>
</table>

Index matches (part of) a predicate if:

- Conjunction of terms involving only attributes (no disjunctions)
- Hash: only equality operation, predicate has all index attributes.
- Tree: Attributes are a prefix of the search key, any ops.
Selectivity of access path

• Selectivity = #pages retrieved (index + data pages)
• Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don’t match the index:
  – Most selective path – fewer I/O
  – Terms that match the index reduce the number of tuples retrieved
  – Other terms are used to discard some retrieved tuples, but do not affect number of tuples fetched.
  – Consider “day < 8/9/94 AND bid=5 AND sid=3”.
    • Can use B+ tree index on day; then check bid=5 and sid=3 for each retrieved tuple
    • Could similarly use a hash index on <bid,sid>; then check day < 8/9/94
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* (\( \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
  – *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.
Relational Operators to Evaluate

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

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

• Sailors:
  – Each tuple is 50 bytes long,
  – 80 tuples per page
  – 500 pages. $\sim$40,000 tuples

• Reserves:
  – Each tuple is 40 bytes long,
  – 100 tuples per page,
  – 1000 pages. $\sim$100,000 tuples
Equality Joins With One Join Column

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

- 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.
- **Cost metric**: # of I/Os. Ignore output cost in analysis.
Simple Nested Loops Join (NLJ)

- For each tuple in the outer relation R, scan the entire inner relation S.
  - Cost: \( M + (p_R \times M) \times N = 1000 + 100 \times 1000 \times 500 = 1,000 + (5 \times 10^7) \) 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 Nested Loop 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: \( M + M \times N = 1000 + 1000\times500 = 501,000 \) 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\times1000 = 500,500 \) I/Os.
• How many buffers do we need?
Block Nested Loops Join

• How can we utilize additional buffer pages?
  – If the smaller relation 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 Nested Loops Join:
  – Take the smaller relation, 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.
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
Examples of Block Nested Loops

- **Cost:** Scan of outer table + \#outer blocks * scan of inner table
  - \#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.

- **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.
Disk Behavior in Block NLJ

- What is the *disk behavior* in Block Nested Loop Join (NLJ)?
  - Reading outer: *sequential* for each block
  - Reading inner: *sequential* if output does not interfere; o.w., *random*.

- Optimization for *sequential reads* of the inner table
  - Read S also in a block-based fashion.
  - May result in more passes, but reduced *seeking* time.
Index Nested Loops Join

- If there is an index on the join column of one relation (say S), can make it the **inner** and exploit the index.
  - Cost: \( M + ( (M \times p_R) \times \text{cost of finding matching S tuples}) \)

- 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.

```plaintext
foreach tuple r in R do
  foreach tuple s in S where r_i == s_j do
    add <r, s> to result
```
Example 1 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.

• 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.
Example 2 of Index Nested Loops

• 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 (clustered) or 2.5 I/Os (uncluster).
  – Total: 500+80*500*(2.2~3.7) = 88,500~148,500 I/Os.

• 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.
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 >= current S tuple,
  - Advance scan of S until current S-tuple >= 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?**

  R is scanned once, each S partition scanned once per matching R tuple.
Example of Sort-Merge Join

- Cost: \( M \log M + N \log N + \text{merging\_cost} \ (\in [M+N, M*N]) \)
  - The cost of merging could be \( M*N \) (but quite unlikely). When?
  - \( M+N \) is guaranteed in \textit{foreign key join}; treat the referenced relation as inner
  - As with sorting, \( \log M \) and \( \log N \) are small numbers, e.g. 3, 4.

- With 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost is 7500 (assuming \( M+N \)).
  More on external sort next week
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.
**Hash Join**

- **Partitioning**: Partition both relations using hash fn $h$: $R_i$ tuples will only match with $S_i$ tuples.

- **Probing**: Read in partition $i$ of $R$, build hash table on $R_i$ using $h_2 (<> h!)$. Scan partition $i$ of $S$, search for matches.
Hash Join Memory Requirement

• **Partitioning:** # partitions in memory ≤ B-1,
  
  **Probing:** size of largest partition (to fit in memory) ≤ 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) → B > \( \sqrt{L} \)
  
  – Hash-join works if the *smaller* relation satisfies above size restriction

• 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 relations; $2(M+N)$. Probing reads both relations; $M+N$ I/Os.

Total cost $= 3(M+N)$.

– In our running example, a total of 4,500 I/Os using hash join, less than 1 min (compared to 140 hours w. Nested Loop Join).

• Sort-Merge Join vs. Hash Join:
  – Given a minimum amount of memory both have a cost of $3(M+N)$ I/Os.
  – Hash Join superior if relation sizes differ greatly
  – 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 Nested Loop, 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 Nested Loop, 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 Nested Loop 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 $\text{gpa} > 3.0$, 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 rids 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 *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.
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 *day*<8/9/94 AND *bid*=5 AND *sid*=3. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can:
    - retrieve rids of records satisfying *day*<8/9/94 using the first, rids of records satisfying *sid*=3 using the second,
    - intersect these rids,
    - retrieve records and check *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 $h1$ 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 $h2 (<> h1)$ 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: Query plan

• 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.