Access Path Selection in a Relational DBMS

Original Slides by
Presentation: Stephen Ingram
Modified by: Rachel Pottinger, Sarah Elhammadi
Why bother to optimize?

- Queries must be executed and execution takes time
- There are multiple execution plans for most queries
- Some plans cost less than others
Simple Example

- SELECT * FROM A,B,C WHERE A.n = B.n AND B.m = C.m
- A = 100 tuples
- B = 50 tuples
- C = 2 tuples
- Which plan is cheaper?
  - Join( C, Join( A, B ) )
  - Join( A, Join( B, C ) )
How did we find the right one?

1. Measure the **cost** of each query
2. Enumerate possibilities
3. Pick the least expensive one

• Is that all?
But the search space is too big

• Just for this simple join example, we have a factorial search space ( n! )

• Just to remind you,
  – 20! = 2,432,902,008,176,640,000

• So now what do we do?
Use Statistics

• For each relation keep track of
  – Cardinality of tuples
  – Cardinality of pages
  – Etc.
• For each index keep track of
  – number of distinct keys in index I.
  – the number of pages in index I.
• Use these statistics in conjunction with
  – Predicates
  – Interesting Orders
This all seems like a bit of a black art. And yet it largely works. Does this surprise you? Why or why not?
Predicates

- Predicates like =, >, NOT, etc. reduce the number of tuples

- THUS: Evaluate predicates as early as possible
Interesting Orders

- GROUP BY and ORDER BY or Join column order are interesting orders.
- To find the cheapest plan we examine the the cheapest access path that produces tuples in interesting order or cheapest unordered plan.
- Sorted orders by merge scan can significantly reduce the cost of subsequent joining even though nested loop join could cheaper for the current join.
But…

• Statistics alone cannot save us
  – Expensive to compute
  – Can’t keep track of all joint statistics

• Compromise on statistics
  – Periodically update stats for each relation

• Compromise on search
  – Dynamic programming approach
Dynamic programming (Wikipedia)

• *Optimal substructure* means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem.

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.
Optimal Substructure in Joins

- Exploits principle of optimality of cost model.
- An N-Join is really just a sequence of 2-Joins
  - 2-join becomes a single composite relation
- Find the cheapest join of a subset of the N tables and store (memoization)
- This costs $2^n$, which is $<< n!$
From the Top

• Enumerate access paths to each relation
  – Sequential scans
  – Interesting orders

• Enumerate access paths to join a second relation
to these results (if there is a predicate to do so)
  – Nested loop (unordered)
  – Merge (interesting order)

• Compare with equivalent solutions found so far but
  only keep the cheapest
### Example Schema

#### EMP

<table>
<thead>
<tr>
<th>NAME</th>
<th>DNO</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>50</td>
<td>12</td>
<td>8500</td>
</tr>
<tr>
<td>JONES</td>
<td>50</td>
<td>5</td>
<td>15000</td>
</tr>
<tr>
<td>DOE</td>
<td>51</td>
<td>5</td>
<td>9500</td>
</tr>
</tbody>
</table>

#### DEPT

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>MFG</td>
<td>DENVER</td>
</tr>
<tr>
<td>51</td>
<td>BILLING</td>
<td>BOULDER</td>
</tr>
<tr>
<td>52</td>
<td>SHIPPING</td>
<td>DENVER</td>
</tr>
</tbody>
</table>

#### JOB

<table>
<thead>
<tr>
<th>JOB</th>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>CLERK</td>
</tr>
<tr>
<td>6</td>
<td>TYPIST</td>
</tr>
<tr>
<td>9</td>
<td>SALES</td>
</tr>
<tr>
<td>12</td>
<td>MECHANIC</td>
</tr>
</tbody>
</table>
Example Query

```
SELECT NAME, TITLE, SAL, DNAME
FROM EMP, DEPT, JOB
WHERE TITLE='CLERK'
AND LOC='DENVER'
AND EMP.DNO=DEPT.DNO
AND EMP.JOB=JOB.JOB
```
Example Initial Access Paths

EMP:
- Index
- EMP.DNO
- N₁
  - C(EMP.DNO)
- EMP.JOB
- N₁
  - C(EMP.JOB)
- Segment
- Scan on
- EMP
- N₁
  - C(EMP seg. scan)
  - Pruned

DEPT:
- Index
- DEPT.DNO
- N₂
  - C(DEPT.DNO)
- Segment
- Scan on
- DEPT
- N₂
  - C(DEPT seg. scan)
  - Pruned

JOB:
- Index
- JOB.JOB
- N₃
  - C(JOB.JOB)
- Segment
- Scan on
- JOB
- N₃
  - C(JOB seg. scan)
Example Search Tree

```
EMP
  └── Index
      └── EMP.DNO
          └── N5
              └── C(EMP.DNO)
                  └── DNO order

EMP
  └── Index
      └── EMP.JOB
          └── N1
              └── C(EMP.JOB)
                  └── JOB order

DEPT
  └── Index
      └── DEPT.DNO
          └── N2
              └── C(DEPT.DNO)
                  └── DNO order

JOB
  └── Index
      └── JOB.JOB
          └── N3
              └── C(JOB.JOB)
                  └── JOB order

JOB
  └── segment scan
      └── N3
          └── C(JOB seg. scan)
              └── unordered
```
2 Relations Nested Loop

(C(E.DNO) + N₁Cₑ(D.DNO) DNO order)

(C(E.JOB) + N₁Cₑ(D.DNO) JOB order)

(C(E.DNO) + N₁Cₑ(J.JOB) DNO order)

(C(E.JOB) + N₁Cₑ(J.JOB) JOB order)

(C(D.DNO) + N₂C_D(E.DNO) DNO order)

(C(J.JOB) + N₃C_J(E.JOB) JOB order)

(C(J seg scan) + N₃C_J(E.JOB) unordered)
2 Relations Merge Join
Prune and 3 Relations
Major Contributions of Paper

• Cost based optimization
  – Statistics
  – CPU utilization (for sorts, etc.)
• Dynamic programming approach
• Interesting Orders
Discussion

We saw a lot of these ideas still in use 20 years later in the previous paper. Are you surprised on how much was kept, or not?