Access Path Selection in a Relational DBMS

Presentation: Stephen
Discussion: Alfred

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.

• Use these statistics in conjunction with
  – Predicates
  – Interesting Orders
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 sort-merge joins generate interesting orders
- We must consider WHEN we generate the interesting order into the cost of a plan
- Ordering it first may be cheaper than sorting later even though it is initially cheaper to leave it unsorted

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

- An N-Join is really just a sequence of 2-Joins
  - 2-join becomes a single composite relation
- Important fact: The method to join to composite is independent of the ordering of the composite
- 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

<table>
<thead>
<tr>
<th>EMP</th>
<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>
<td></td>
</tr>
<tr>
<td>Jones</td>
<td>50</td>
<td>5</td>
<td>15000</td>
<td></td>
</tr>
<tr>
<td>Doe</td>
<td>51</td>
<td>5</td>
<td>9500</td>
<td></td>
</tr>
</tbody>
</table>

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

<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

Example Search Tree

2 Relations Nested Loop

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