Lecture 26:
Query Optimization

Monday, December 4th, 2006
Outline

• Cost-based optimization 16.5, 16.6

• Cost estimation: 16.4
Cost-based Optimizations

• Main idea: apply algebraic laws, until estimated cost is minimal

• Practically: start from partial plans, introduce operators one by one
  – Will see in a few slides

• Problem: there are too many ways to apply the laws, hence too many (partial) plans
Cost-based Optimizations

Approaches:

• **Top-down**: the partial plan is a top fragment of the logical plan

• **Bottom up**: the partial plan is a bottom fragment of the logical plan
Dynamic Programming

Originally proposed in System R

- Only handles single block queries:

  ```sql
  SELECT list
  FROM   list
  WHERE cond_1 AND cond_2 AND ... AND cond_k
  ```

- Heuristics: selections down, projections up
- Dynamic programming: *join reordering*
Join Trees

- $R_1 \times R_2 \times \ldots \times R_n$
- Join tree:

\[
\begin{array}{c}
\text{R3} \\
\text{R1} \\
\text{R2} \\
\text{R4}
\end{array}
\]

- A plan = a join tree
- A partial plan = a subtree of a join tree
Types of Join Trees

- Left deep:
Types of Join Trees

- Bushy:

```
      /
     /|
    /  |  
   R3  /
     /  |
    /    |
   R1    R5
```

```
      /
     /|
    /  |  
   R2  /
     /  |
    /    |
   R4    R1
```
Types of Join Trees

• Right deep:

```
  R3  
   / 
  /   
R1   R5
   /   /   
  /   /   
R2 R4
```
Dynamic Programming

• Given: a query $R_1 \times R_2 \times \ldots \times R_n$
• Assume we have a function cost() that gives us the cost of every join tree
• Find the best join tree for the query
Dynamic Programming

• Idea: for each subset of \{R_1, \ldots, R_n\}, compute the best plan for that subset

• In increasing order of set cardinality:
  – Step 1: for \{R_1\}, \{R_2\}, \ldots, \{R_n\}
  – Step 2: for \{R_1,R_2\}, \{R_1,R_3\}, \ldots, \{R_{n-1}, R_n\}
  – …
  – Step n: for \{R_1, \ldots, R_n\}

• It is a bottom-up strategy

• A subset of \{R_1, \ldots, R_n\} is also called a subquery
Dynamic Programming

• For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ compute the following:
  – Size($Q$)
  – A best plan for $Q$: Plan($Q$)
  – The cost of that plan: Cost($Q$)
Dynamic Programming

• **Step 1:** For each \( \{R_i\} \) do:
  
  - \( \text{Size}(\{R_i\}) = B(R_i) \)
  
  - \( \text{Plan}(\{R_i\}) = R_i \)
  
  - \( \text{Cost}(\{R_i\}) = (\text{cost of scanning } R_i) \)
Dynamic Programming

• **Step i:** For each $Q \subseteq \{R_1, \ldots, R_n\}$ of cardinality $i$ do:
  - Compute $\text{Size}(Q)$ (later…)
  - For every pair of subqueries $Q', Q''$ s.t. $Q = Q' \cup Q''$
    - compute $\text{cost}(\text{Plan}(Q') \times \text{Plan}(Q''))$
  - $\text{Cost}(Q) = \text{the smallest such cost}$
  - $\text{Plan}(Q) = \text{the corresponding plan}$
Dynamic Programming

- Return Plan(\{R_1, \ldots, R_n\})
Dynamic Programming

To illustrate, we will make the following simplifications:

- \( \text{Cost}(P_1 \times P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) + \text{size( intermediate result(s)})} \)

- Intermediate results:
  - If \( P_1 \) = a join, then the size of the intermediate result is \( \text{size}(P_1) \), otherwise the size is 0
  - Similarly for \( P_2 \)

- Cost of a scan = 0
Dynamic Programming

- Example:
  - Cost(R5 |×| R7) = 0 (no intermediate results)
  - Cost((R2 |×| R1) |×| R7)
    = Cost(R2 |×| R1) + Cost(R7) + size(R2 |×| R1)
    = size(R2 |×| R1)
Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: $T(A \times B) = 0.01 \times T(A) \times T(B)$
<table>
<thead>
<tr>
<th>Subquery</th>
<th>Size</th>
<th>Cost</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RSTU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subquery</td>
<td>Size</td>
<td>Cost</td>
<td>Plan</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>RS</td>
<td>100k</td>
<td>0</td>
<td>RS</td>
</tr>
<tr>
<td>RT</td>
<td>60k</td>
<td>0</td>
<td>RT</td>
</tr>
<tr>
<td>RU</td>
<td>20k</td>
<td>0</td>
<td>RU</td>
</tr>
<tr>
<td>ST</td>
<td>150k</td>
<td>0</td>
<td>ST</td>
</tr>
<tr>
<td>SU</td>
<td>50k</td>
<td>0</td>
<td>SU</td>
</tr>
<tr>
<td>TU</td>
<td>30k</td>
<td>0</td>
<td>TU</td>
</tr>
<tr>
<td>RST</td>
<td>3M</td>
<td>60k</td>
<td>(RT)S</td>
</tr>
<tr>
<td>RSU</td>
<td>1M</td>
<td>20k</td>
<td>(RU)S</td>
</tr>
<tr>
<td>RTU</td>
<td>0.6M</td>
<td>20k</td>
<td>(RU)T</td>
</tr>
<tr>
<td>STU</td>
<td>1.5M</td>
<td>30k</td>
<td>(TU)S</td>
</tr>
<tr>
<td>RSTU</td>
<td>30M</td>
<td>60k+50k=110k</td>
<td>(RT)(SU)</td>
</tr>
</tbody>
</table>
Reducing the Search Space

- Left-linear trees v.s. Bushy trees

- Trees without cartesian product

Example: $R(A,B) \times S(B,C) \times T(C,D)$

Plan: $(R(A,B) \times T(C,D)) \times S(B,C)$ has a cartesian product – most query optimizers will not consider it
Dynamic Programming: Summary

• Handles only join queries:
  – Selections are pushed down (i.e. early)
  – Projections are pulled up (i.e. late)

• Takes exponential time in general, BUT:
  – Left linear joins may reduce time
  – Non-cartesian products may reduce time further
Rule-Based Optimizers

- *Extensible* collection of rules
  Rule = Algebraic law with a direction
- Algorithm for firing these rules
  Generate many alternative plans, in some order
  Prune by cost

- Volcano (later SQL Sever)
- Starburst (later DB2)
Completing the Physical Query Plan

• Choose algorithm to implement each operator
  – Need to account for more than cost:
    • How much memory do we have?
    • Are the input operand(s) sorted?

• Decide for each intermediate result:
  – To materialize
  – To pipeline
Materialize Intermediate Results Between Operators

HashTable $\leftarrow S$
repeat
read(R, x)
y $\leftarrow$ join(HashTable, x)
write(V1, y)
HashTable $\leftarrow T$
repeat
read(V1, y)
z $\leftarrow$ join(HashTable, y)
write(V2, z)
HashTable $\leftarrow U$
repeat
read(V2, z)
u $\leftarrow$ join(HashTable, z)
write(Answer, u)
Materialize Intermediate Results Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

• What is the total cost of the plan?
  – Cost =

• How much main memory do we need?
  – M =
Pipeline Between Operators

Pipeline Between Operators

HashTable1 $\leftarrow$ S
HashTable2 $\leftarrow$ T
HashTable3 $\leftarrow$ U
repeat
  read(R, x)
  y $\leftarrow$ join(HashTable1, x)
  z $\leftarrow$ join(HashTable2, y)
  u $\leftarrow$ join(HashTable3, z)
write(Answer, u)
Pipeline Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan?
  - Cost =

- How much main memory do we need?
  - M =
Pipeline in Bushy Trees
Example

- Logical plan is:

```
               k blocks
              /     \
             /       \
          U(y,z)     \
          /      /     \
         /    / 10,000 blocks
        /  /     \
      R(w,x) S(x,y)
        5,000 blocks 10,000 blocks
```

- Main memory M = 101 buffers
Example

M = 101

Naïve evaluation:
• 2 partitioned hash-joins
• Cost $3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k$
Example

\[ M = 101 \]

```
M = 101

k blocks

R(w,x) 5,000 blocks

S(x,y) 10,000 blocks

U(y,z) 10,000 blocks
```

Smarter:
- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each \( R_i \) in memory (50 buffer) join with \( S_i \) (1 buffer); hash result on y into 50 buckets (50 buffers) -- here we *pipeline*
- Cost so far: \( 3B(R) + 3B(S) \)
Example

$M = 101$

\[
\begin{array}{c}
\text{k blocks} \\
\text{R}(w,x) \quad \text{S}(x,y) \\
5,000 \text{ blocks} \quad 10,000 \text{ blocks} \\
\text{U}(y,z) \\
10,000 \text{ blocks}
\end{array}
\]

Continuing:

• How large are the 50 buckets on y? Answer: $k/50$.
• If $k \leq 50$ then keep all 50 buckets in Step 3 in memory, then:
• Step 4: read $U$ from disk, hash on $y$ and join with memory
• Total cost: $3B(R) + 3B(S) + B(U) = 55,000$
Example

M = 101

```
  k blocks
 /
\-
R(w,x)  S(x,y)
5,000 blocks 10,000 blocks
```

U(y,z)
10,000 blocks

Continuing:
- If $50 < k \leq 5000$ then send the 50 buckets in Step 3 to disk
  - Each bucket has size $k/50 \leq 100$
- Step 4: partition $U$ into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: $3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k$
Example

M = 101

Continuing: 5,000 blocks 10,000 blocks
• If $k > 5000$ then materialize instead of pipeline
• 2 partitioned hash-joins
• Cost $3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k$
Example

Summary:

- If $k \leq 50$, \hspace{1cm} \text{cost} = 55,000
- If $50 < k \leq 5000$, \hspace{1cm} \text{cost} = 75,000 + 2k
- If $k > 5000$, \hspace{1cm} \text{cost} = 75,000 + 4k
Size Estimation

The problem: Given an expression $E$, compute $T(E)$ and $V(E, A)$

- This is hard without computing $E$
- Will ‘estimate’ them instead
Size Estimation

Estimating the size of a projection

- Easy: $T(\Pi_L(R)) = T(R)$
- This is because a projection doesn’t eliminate duplicates
Size Estimation

Estimating the size of a selection

- \( S = \sigma_{A=c}(R) \)
  - \( T(S) \) can be anything from 0 to \( T(R) - V(R,A) + 1 \)
  - Estimate: \( T(S) = T(R)/V(R,A) \)
  - When \( V(R,A) \) is not available, estimate \( T(S) = T(R)/10 \)

- \( S = \sigma_{A<c}(R) \)
  - \( T(S) \) can be anything from 0 to \( T(R) \)
  - Estimate: \( T(S) = (c - \text{Low}(R, A))/(\text{High}(R,A) - \text{Low}(R,A))T(R) \)
  - When Low, High unavailable, estimate \( T(S) = T(R)/3 \)
Size Estimation

Estimating the size of a natural join, $R \times_{|A} S$

- When the set of $A$ values are disjoint, then $T(R \times_{|A} S) = 0$
- When $A$ is a key in $S$ and a foreign key in $R$, then $T(R \times_{|A} S) = T(R)$
- When $A$ has a unique value, the same in $R$ and $S$, then $T(R \times_{|A} S) = T(R) \cdot T(S)$
Size Estimation

Assumptions:

- **Containment of values**: if $V(R, A) \leq V(S, A)$, then the set of $A$ values of $R$ is included in the set of $A$ values of $S$
  - Note: this indeed holds when $A$ is a foreign key in $R$, and a key in $S$

- **Preservation of values**: for any other attribute $B$, 
  $V(R \mid \times \!|_A S, B) = V(R, B)$ (or $V(S, B)$)
Size Estimation

Assume $V(R,A) \leq V(S,A)$

- Then each tuple $t$ in $R$ joins \textit{some} tuple(s) in $S$
  - How many?
  - On average $T(S)/V(S,A)$
  - $t$ will contribute $T(S)/V(S,A)$ tuples in $R \times |_A S$

- Hence $T(R \mid \times |_A S) = T(R) \cdot T(S) / V(S,A)$

In general: $T(R \mid \times |_A S) = T(R) \cdot T(S) / \max(V(R,A),V(S,A))$
Size Estimation

Example:

- $T(R) = 10000$, $T(S) = 20000$
- $V(R,A) = 100$, $V(S,A) = 200$
- How large is $R \times_A S$?

Answer: $T(R \times_A S) = 10000 \times 20000/200 = 1M$
Size Estimation

Joins on more than one attribute:

• \( T(R | \times |_{A,B} S) = \)

\[
T(R) \frac{T(S)}{(\max(V(R,A), V(S,A)) \times \max(V(R,B), V(S,B)))}
\]
Histograms

• Statistics on data maintained by the RDBMS

• Makes size estimation much more accurate (hence, cost estimations are more accurate)
Histograms

Employee(ssn, name, salary, phone)
• Maintain a histogram on salary:

<table>
<thead>
<tr>
<th>Salary:</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuples</td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

• $T(\text{Employee}) = 25000$, but now we know the distribution
Histograms

**Ranks(rankName, salary)**

- Estimate the size of Employee $| \times |$ Salary Ranks

<table>
<thead>
<tr>
<th>Employee</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>800</td>
<td>5000</td>
<td>12000</td>
<td>6500</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ranks</th>
<th>0..20k</th>
<th>20k..40k</th>
<th>40k..60k</th>
<th>60k..80k</th>
<th>80k..100k</th>
<th>&gt; 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>100</td>
<td>2</td>
</tr>
</tbody>
</table>
### Histograms

- **Eqwidth**

<table>
<thead>
<tr>
<th></th>
<th>0..20</th>
<th>20..40</th>
<th>40..60</th>
<th>60..80</th>
<th>80..100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>104</td>
<td>9739</td>
<td>152</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Eqdepth**

<table>
<thead>
<tr>
<th></th>
<th>0..44</th>
<th>44..48</th>
<th>48..50</th>
<th>50..56</th>
<th>55..100</th>
</tr>
</thead>
</table>