Lecture 22: Query Optimization (2)
Friday, November 19, 2010

Outline
- Search space
- Algorithms for enumerating query plans
- Estimating the cost of a query plan

Key Decisions
Logical plan
- What logical plans do we consider (left-deep, bushy ?); Search Space
- Which algebraic laws do we apply, and in which context(s) ?; Optimization rules
- In what order do we explore the search space ?; Optimization algorithm

Physical plan
- What physical operators to use?
- What access paths to use (file scan or index)?

Optimizers
- Heuristic-based optimizers:
  - Apply greedily rules that always improve
    - Typically: push selections down
  - Very limited: no longer used today
- Cost-based optimizers
  - Use a cost model to estimate the cost of each plan
  - Select the “cheapest” plan

The Search Space
- Complete plans
- Bottom-up plans
- Top-down plans
Complete Plans

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

Why is this search space inefficient?

Bottom-up Partial Plans

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

Why is this better?

Top-down Partial Plans

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

Dynamic Programming

Originally proposed in System R [1979]

- Only handles single block queries:
  
  ```
  SELECT list
  FROM R1, ..., Rn
  WHERE cond1 AND cond2 AND ... AND condn
  ```

- Heuristics: selections down, projections up

Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Classical algorithm [1979]
  - Limited to joins: join reordering algorithm
  - Bottom-up

- Rule-based algorithm (will not discuss)
  - Database of rules (=algebraic laws)
  - Usually: dynamic programming
  - Usually: top-down

Dynamic Programming

- Search space = join trees
- Algebraic laws = commutativity, associativity
- Algorithm = dynamic programming ©
Join Trees

- \( R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \)
- Join tree:
  \[
  \begin{array}{c}
  & \searrow & \\
  \nearrow & & \searrow \\
  & R_3 & R_1 & R_2 & R_4 \\
  \end{array}
  \]
- A plan = a join tree
- A partial plan = a subtree of a join tree

Join ordering:

- Given: a query \( R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \)
- Find optimal order
- Assume we have a function \( \text{cost()} \) that gives us the cost of every join tree

Dynamic Programming

- For each subquery \( Q \subseteq \{R_1, \ldots, R_n\} \) compute the following:
  - \( \text{Size}(Q) \) = the estimated size of \( Q \)
  - \( \text{Plan}(Q) \) = a best plan for \( Q \)
  - \( \text{Cost}(Q) \) = the estimated cost of that plan
Dynamic Programming

• **Step 1:** For each \{R_i\}, set:
  – Size(\{R_i\}) = B(\{R_i\})
  – Plan(\{R_i\}) = R_i
  – Cost(\{R_i\}) = (cost of scanning \{R_i\})

Dynamic Programming

• **Step 2:** For each \(Q \subseteq \{R_1, \ldots, R_n\}\) involving i relations:
  – Size(\(Q\)) = estimate it recursively
  – For every pair of subqueries \(Q', Q''\) s.t. \(Q = Q' \cup Q''\) compute cost(Plan(\(Q'\)) \(\Join\) Plan(\(Q''\)))
  – Cost(\(Q\)) = the smallest such cost
  – Plan(\(Q\)) = the corresponding plan

Dynamic Programming

• **Step 3:** Return Plan(\{R_1, \ldots, R_n\})

Example

To illustrate, ad-hoc cost model (from the book ©):

• Cost(\(P_1 \Join P_2\)) = Cost(\(P_1\)) + Cost(\(P_2\)) + size(intermediate results for \(P_1, P_2\))

• Cost of a scan = 0

Example

• \(R \Join S \Join T \Join U\)

• Assumptions:
  
  All join selectivities = 1%

  
  \begin{align*}
  T(R) &= 2000 \\
  T(S) &= 5000 \\
  T(T) &= 3000 \\
  T(U) &= 1000
  \end{align*}

  
  \begin{align*}
  T(R \times S) &= 0.01 \times T(R) \times T(S) \\
  T(S \times T) &= 0.01 \times T(S) \times T(T) \\
  &\text{etc.}
  \end{align*}

Example

Subquery | Size | Cost | Plan
--- | --- | --- | ---
RS | | | 
RT | | | 
RU | | | 
ST | | | 
SU | | | 
TU | | | 
RST | | | 
RSU | | | 
RTU | | | 
STU | | | 
RSTU | | |
Reducing the Search Space

- Restriction 1: only left linear trees (no bushy)
- Restriction 2: no trees with cartesian product

$$\text{R(A,B)} \bowtie \text{S(B,C)} \bowtie \text{T(C,D)}$$

Plan: $$(\text{R(A,B)}\bowtie \text{T(C,D)}) \bowtie \text{S(B,C)}$$

Most query optimizers will not consider it

Why?

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 Server)
- Starburst (later DB2)

Completing the Physical Query Plan

- Choose algorithm for each operator
  - How much memory do we have?
  - Are the input operand(s) sorted?
- Access path selection for base tables
- Decide for each intermediate result:
  - To materialize
  - To pipeline

Access Path Selection

- **Access path**: a way to retrieve tuples from a table
  - A file scan
  - An index plus a matching selection condition
- Index matches selection condition if it can be used to retrieve just tuples that satisfy the condition
  - Example: Supplier(sid,sname,scity,sstate)
    - B+-tree index on (scity,sstate)
    - matches scity='Seattle'
    - does not match sid=3, does not match sstate='WA'
Access Path Selection

• Supplier(sid, sname, scity, sstate)
• Selection condition:(sid > 300 ∧ scity = 'Seattle')
• Indexes: B+-tree on sid and B+-tree on scity
• Which access path should we use?
• We should pick the most selective access path

Access Path Selectivity

• Access path selectivity is the number of pages retrieved if we use this access path
  – Most selective retrieves fewest pages
• As we saw earlier, for equality predicates
  – Selection on equality: \( \sigma_{a=v}(R) \)
  – \( V(R, a) \) is the number of distinct values of attribute \( a \)
  – \( 1/V(R, a) \) is thus the reduction factor
  – Clustered index on \( a \): cost \( B(R)/V(R, a) \)
  – Unclustered index on \( a \): cost \( T(R)/V(R, a) \)
  – (we are ignoring I/O cost of index pages for simplicity)

Materialize Intermediate Results Between Operators

<table>
<thead>
<tr>
<th>V1</th>
<th>R</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- HashTable ← S Repeat
  - read(R, x)
  - y ← join(HashTable, x)
  - write(V1, y)
- HashTable ← T Repeat
  - read(V1, y)
  - z ← join(HashTable, y)
  - write(V2, z)
- HashTable ← U Repeat
  - read(V2, z)
  - u ← 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

<table>
<thead>
<tr>
<th>V1</th>
<th>R</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- HashTable1 ← S Repeat
  - read(R, x)
  - y ← join(HashTable1, x)
  - z ← join(HashTable2, y)
  - u ← 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:

\[ \begin{align*}
& \text{k blocks} \\
& \text{U(y,z)} \\
& \text{10,000 blocks} \\
& \text{R(w,x) } 5,000 \text{ blocks} \\
& \text{S(x,y) } 10,000 \text{ blocks} \\
\end{align*} \]

• Main memory M = 101 buffers

Example

M = 101

\[ \begin{align*}
& \text{k blocks} \\
& \text{U(y,z)} \\
& \text{10,000 blocks} \\
& \text{R(w,x) } 5,000 \text{ blocks} \\
& \text{S(x,y) } 10,000 \text{ blocks} \\
\end{align*} \]

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

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 \( R \) in memory (50 buffer) join with \( S \) (1 buffer); hash result on \( y \) into 50 buckets (50 buffers) -- here we pipeline
• Cost so far: \( 3B(R) + 3B(S) \)

Example

M = 101

\[ \begin{align*}
& \text{k blocks} \\
& \text{U(y,z)} \\
& \text{10,000 blocks} \\
& \text{R(w,x) } 5,000 \text{ blocks} \\
& \text{S(x,y) } 10,000 \text{ blocks} \\
\end{align*} \]

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

\[ \begin{align*}
& \text{k blocks} \\
& \text{U(y,z)} \\
& \text{10,000 blocks} \\
& \text{R(w,x) } 5,000 \text{ blocks} \\
& \text{S(x,y) } 10,000 \text{ blocks} \\
\end{align*} \]

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

\[
\begin{array}{c}
\text{k blocks} \\
\text{bc} \\
\text{R(k,x)} \\
\text{S(x,y)} \\
10,000 \text{ blocks} \\
\end{array}
\]

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