CSIT5300: Advanced Database Systems

L11: Query Optimization

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
• Practical query optimizers incorporate elements of the following two broad approaches:
  
  - **Cost-based optimization**: search all the plans and choose the best plan in a cost-based fashion
    
    **General idea:**
    1] Generate “all” possible evaluation plans
    2] Estimate the cost of each plan
    3] Execute the plan with the minimum expected cost
  
  - **Heuristic optimization**: Use heuristics to choose a plan
    
    **General idea:**
    1] Perform the cheap operations first (i.e., push selections down)
    2] Try to utilize existing indexes
    3] Remove the useless attributes early
Consider finding the best join-order for $R_1 \text{ JOIN } R_2 \text{ JOIN } \ldots \text{ JOIN } R_n$

There are $(2(n-1))!/(n-1)!$ different join orders for the above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{R_1, R_2, \ldots, R_n\}$ is computed only once and stored for future use.
• To find the best join plan for a set $S$ of $n$ relations:
  - Consider all possible plans of the form: $S_1 \text{ JOIN } (S - S_1)$
    where $S_1$ is any non-empty subset of $S$
  - Recursively compute costs for joining subsets of $S$ to find the cost of each plan.
    Choose the cheapest of the $2^n - 1$ alternatives.
  - When a plan for any subset is computed, store it and reuse it when it is required again, instead of re-computing it
    • Dynamic programming
### Procedure `findbestplan(S)`

```plaintext
if (bestplan[S].cost ≠ ∞)
    return bestplan[S]
// else bestplan[S] has not been computed earlier, compute it now
for each non-empty subset S1 of S such that S1 ≠ S
    P1 = findbestplan(S1)
    P2 = findbestplan(S - S1)
    A = best algorithm for joining results of P1 and P2
    cost = P1.cost + P2.cost + cost of A
    if cost < bestplan[S].cost
        bestplan[S].cost = cost
        bestplan[S].plan = “execute P1.plan, execute P2.plan,
                          join results of P1 and P2 using A”
    return bestplan[S]
```
In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join. Some optimizers only consider left-deep plans (they facilitate pipelining).

(a) Left-deep Join Tree  
(b) Non-left-deep Join Tree
• Consider the expression \((r_1 \text{ JOIN } r_2 \text{ JOIN } r_3) \text{ JOIN } r_4 \text{ JOIN } r_5\)

• An **interesting sort order** is a particular sort order of tuples that could be useful for a *later operation*
  - Generating the result of \((r_1 \text{ JOIN } r_2 \text{ JOIN } r_3)\) sorted on the attributes common with \(r_4\) or \(r_5\) may be useful, but generating it sorted on the attributes common only \(r_1\) and \(r_2\) is not useful
  - Using merge-join to compute \((r_1 \text{ JOIN } r_2 \text{ JOIN } r_3)\) may be costlier, but may provide an output sorted in an interesting order

• **Must find the best join order for each subset, for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn’t affect time/space complexity significantly
Cost-based optimization is expensive, even with dynamic programming.

Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.

Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

- Perform selections early (reduces the number of tuples)
- Perform projections early (reduces the number of attributes)
- Perform most restrictive selections and join operations before other similar operations.

Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
• **Cascading of selections:** \( \sigma_{c_1 \land c_2 \land \ldots \land c_n}(R) = \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R)))) \)

  Example: \( \sigma_{\text{Age}=20 \land \text{Rating}>7}(\text{Sailors}) = \sigma_{\text{Rating}>7}(\sigma_{\text{Age}=20}(\text{Sailors})) \)

• **Commutativity of selections:** \( \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R)) \)

  Example: \( \sigma_{\text{Rating}>7}(\sigma_{\text{Age}=20}(\text{Sailors})) = \sigma_{\text{Age}=20}(\sigma_{\text{Rating}>7}(\text{Sailors})) \)

• **Cascading of projections:** \( \pi_{a_1}(R) = \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R)))) \)

  where \( a_1 \subseteq a_2 \subseteq \ldots \subseteq a_n \)

  Example: \( \pi_{\text{Name}}(\text{Sailors}) = \pi_{\text{Name}}(\pi_{\text{Name}}, \text{Sid}(\text{Sailors})) \)

• **Commutativity of joins (and Cartesian products):** \( R \JOIN S = S \JOIN R \)

• **Associativity of joins (and Cartesian products):**

  \( (R \JOIN S) \JOIN T = R \JOIN (S \JOIN T) \)

• From these two properties of joins (and Cartesian products):

  \( R \JOIN (S \JOIN T) = (T \JOIN R) \JOIN S \)

  - That is, we can take any order of joins
• **Commute selections with projections:** \( \pi_a(\sigma_c(R)) = \sigma_c(\pi_a(R)) \)

Only if the selection condition includes only attributes of the projection list

**Legal Example:**

\[ \pi_{\text{Name}, \text{Age}}(\sigma_{\text{Age}=20}(\text{Sailors})) = \sigma_{\text{Age}=20}(\pi_{\text{Name}, \text{Age}}(\text{Sailors})) \]

**Illegal Example:**

\[ \pi_{\text{Name}, \text{Rating}}(\sigma_{\text{Age}=20}(\text{Sailors})) = \sigma_{\text{Age}=20}(\pi_{\text{Name}, \text{Rating}}(\text{Sailors})) \]

• **Commute selections with joins (and Cartesian products):**

\[ \sigma_c(R \ JOIN \ S) = (\sigma_c R) \ JOIN \ S \]

Applicable if \( c \) involves only attributes of \( R \)

**Legal Examples:**

\[ \sigma_{\text{Name}=\text{Joe} \land \text{Rating}=7}(\text{Sailors} \ JOIN \ \text{Reserves}) = (\sigma_{\text{Name}=\text{Joe} \land \text{Rating}=7} \text{Sailors}) \ JOIN \ \text{Reserves} \]

\[ \sigma_{\text{Name}=\text{Joe} \land \text{Bid}=100}(\text{Sailors} \ JOIN \ \text{Reserves}) = (\sigma_{\text{Name}=\text{Joe}} \text{Sailors}) \ JOIN \ (\sigma_{\text{Bid}=100} \text{Reserves}) \]
\[ \pi_a(R \Join S) = \pi_a((\pi_{a_1}R) \Join (\pi_{a_2}S)) \]

Where (i) \( a_1 \) is the subset of projected attributes \( a \) that belong to \( R \) + the join attribute, (ii) \( a_2 \) is the subset of \( a \) that belong to \( S \) + the join attribute

Example:
\[ \pi_{Name, \ Date}(\text{Sailors} \ Join \text{ Reserves}) = \]
\[ \pi_{Name, \ Date}((\pi_{Name,Sid} \text{Sailors}) \Join (\pi_{Date,Sid} \text{Reserves})) \]

\[ \pi_a(R \ Join S) = (\pi_{a_1}R) \ Join (\pi_{a_2}R) \]
If the join attribute is included in the list of projected attributes \( a \)

Example:
\[ \pi_{Sid, \ Date}(\text{Sailors} \ Join \text{ Reserves}) = \]
\[ (\pi_{Sid} \text{Sailors}) \ Join (\pi_{Date,Sid} \text{Reserves}) \]