COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 18
QUERY OPTIMIZATION

QUERY OPTIMZATION: OUTLINE

Overview

Transformation of Relational Expressions

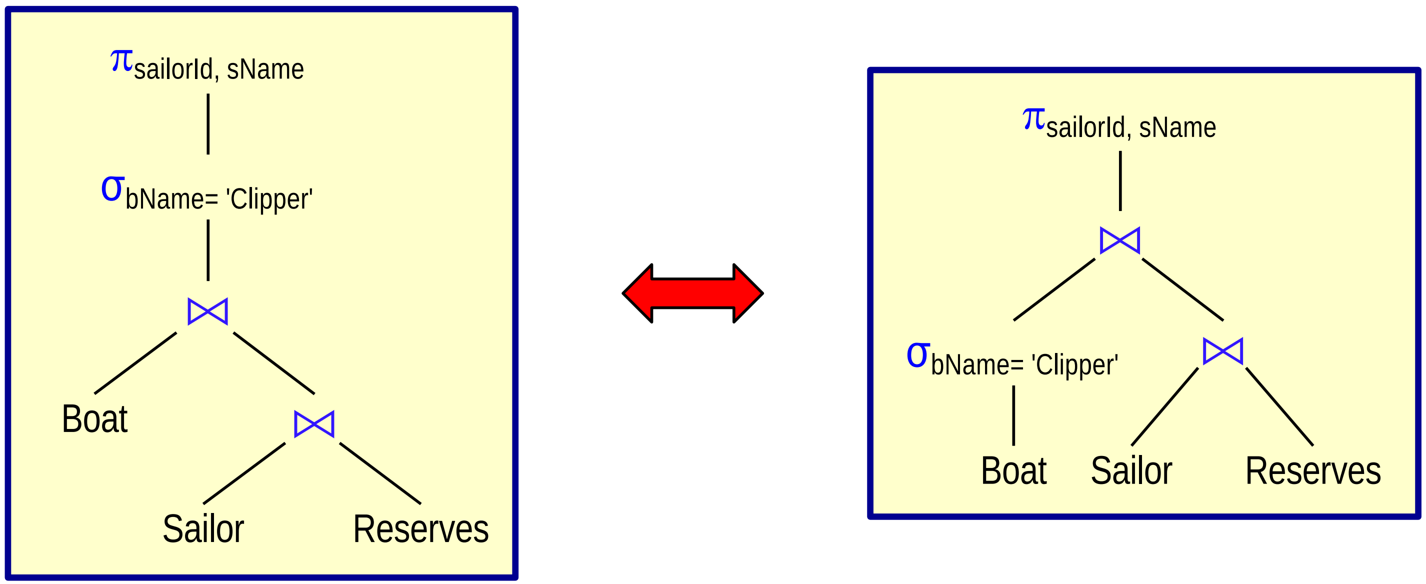
Estimating Statistics of Expression Results

OVERVIEW

Query optimization **is the process of selecting the *most efficient* query-evaluation plan from among many strategies.**

There may be several alternative ways of evaluating a given query.

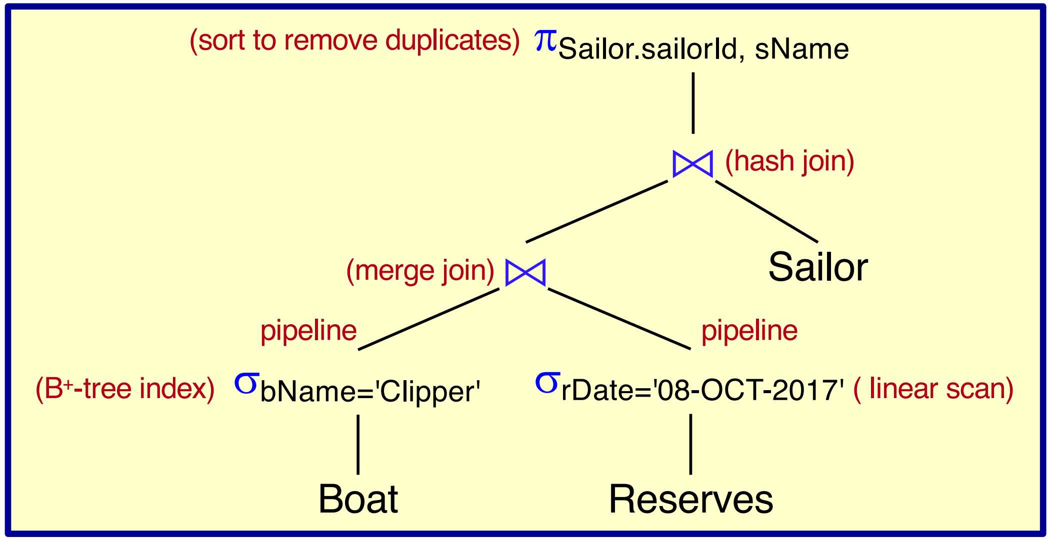
- Equivalent expressions: Find the ids and names of sailors who have reserved the boat named "Clipper".
- Different algorithms could be used for each operation.



OVERVIEW (CONT'D)

- An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

  ☞ **The cost difference between evaluation plans can be enormous (e.g., seconds versus days in some cases).**



Most DBMSs allow you to view the evaluation plans for a query.

OVERVIEW (CONT'D)

Cost Based Optimization

1. Generate logically equivalent evaluation plans using equivalence rules for relational algebra expressions.
2. Estimate the cost of each plan.
3. Execute the plan with the minimum expected cost.

- Estimation of the plan cost is based on:
  - Statistical information about relations.
    - The number of tuples, number of distinct values for an attribute, etc.
  - Statistics estimation for intermediate results.
    - To compute the cost of complex expressions.
  - Cost formulae for algorithms, computed using statistics.

OVERVIEW (CONT'D)

Heuristic Optimization

1. Perform the cheap operations first.
   - Perform selections early (i.e., push selections down)
     - Reduces the number of tuples.
2. Remove unneeded attributes early.
   - Perform projections early
     - Reduces the number of attributes.
3. Try to utilize existing indexes.
   - Perform most restrictive selection and join operations before other similar operations.

- Transform the relational algebra tree by using a set of rules that typically (but not in all cases) improves execution performance.

☞ **Some systems use only heuristics, others combine heuristics with partial cost-based optimization.**

TRANSFORMATION OF RELATIONAL EXPRESSIONS

- Two relational algebra expressions are equivalent if they *generate the same set of tuples* on *every legal* database instance.
  - The order of the tuples generated is irrelevant.
  - We do not care if they generate different results on databases that violate integrity constraints.
- SQL inputs and outputs are multisets of tuples (i.e., contain duplicates).
  - Two expressions in the multiset version of the relational algebra are equivalent if they generate the same multiset of tuples on every legal database instance.
- An equivalence rule states that two forms of an expression are equivalent.

☞ **Can replace an expression of the first form by one of the second, or vice versa.**

EQUIVALENCE RULES: SINGLE OPERATIONS

Cascading of projections: $\pi_{a1}(R) \equiv \pi_{a1}(\pi_{a2}\ldots(\pi_{an}(R))\ldots)$ where $a_1 \subseteq a_2 \ldots \subseteq a_n$

Example: $\pi_{sName}(Sailor) \equiv \pi_{sName}(\pi_{sName,\ sailorId}(Sailor))$ $\Longrightarrow$ only the final projection matters

Cascading of selections: $\sigma_{c1 \wedge c2 \ldots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}\ldots(\sigma_{cn}(R))\ldots)$

Example: $\sigma_{age=20 \wedge rating>7}(Sailor) \equiv \sigma_{rating>7}(\sigma_{age=20}(Sailor))$

Commutativity of selections: $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

Example: $\sigma_{rating>7}(\sigma_{age=20}(Sailor)) \equiv \sigma_{age=20}(\sigma_{rating>7}(Sailor))$

Commutativity of joins / Cartesian products: $R \bowtie S \equiv S \bowtie R$

Associativity of joins / Cartesian products: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

☞ **Can perform joins in any order:** $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

EQUIVALENCE RULES: MULTIPLE OPERATIONS

Commutativity of selections with projections: $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$

☞ **Applicable only if the selection condition includes only attributes of the projection list.**

Legal Example: $\pi_{sName,\ age}(\sigma_{age=20}(Sailor)) \equiv \sigma_{age=20}(\pi_{sName,\ age}(Sailor))$ (age$\subset$sName, age)

Illegal Example: $\pi_{sName,\ rating}(\sigma_{age=20}(Sailor)) \equiv \sigma_{age=20}(\pi_{sName,\ rating}(Sailor))$ (age$\not\subset$sName, rating)

Commutativity of selections with joins / Cartesian products:

$\sigma_c(R \text{ Join } S) \equiv (\sigma_c R) \text{ Join } S$

☞ **Applicable if *c* involves only attributes of R.**

Legal Examples:

$\sigma_{sName='Joe' \wedge rating=7}(Sailor \text{ Join } Reserves) \equiv (\sigma_{sName='Joe' \wedge rating=7} Sailor) \text{ Join } Reserves$

$\sigma_{sName='Joe' \wedge boatId=100}(Sailor \text{ Join } Reserves) \equiv (\sigma_{sName='Joe} Sailor) \text{ Join } (\sigma_{boatId=100} Reserves)$

Equivalence Rules: Multiple Operations (cont'd)

Projection Distributes Over Join: $\pi_{a}(R \text{ Join } S) \equiv (\pi_{a}(\pi_{a1}R) \text{ Join } (\pi_{a2}S))$

- where
  i. a_1 is the subset of a that belongs to R *plus* the join attribute,
  ii. a_2 is the subset of a that belongs to S *plus* the join attribute.

Example: $\pi_{\text{sName, rDate}}(\text{Sailor Join Reserves}) \equiv$
$\pi_{\text{sName, rDate}}((\pi_{\text{sName, sailorId}}\text{Sailor}) \text{ Join } (\pi_{\text{rDate, sailorId}}\text{Reserves}))$

- If the join attribute is included in the list of projection attributes a, then: $\pi_{a}(R \text{ Join } S) \equiv (\pi_{a1}R) \text{ Join } (\pi_{a2}R)$

Example: $\pi_{\text{sailorId, rDate}}(\text{Sailor Join Reserves}) \equiv (\pi_{\text{sailorId}}\text{Sailor}) \text{ Join } (\pi_{\text{sailorId, rDate}}\text{Reserves})$

JOIN ORDERING

- A good ordering of join operations is important for reducing the size of intermediate results.

- Consider finding the best join-order for r_1 Join r_2 Join ... Join r_n.
  - For this expression there are $(2(n-1))!/(n-1)!$ different join orders!
    - For $n = 3$, the number is 12.
    - For $n = 7$, the number is 665,280.
    - For $n = 10$, the number is greater than 176 billion!
  - We also need to consider other operations (e.g., selections) that may exist in the query.

- There is no need to generate all the join orders.
  - Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots, r_n\}$ can be computed only once and stored for future use.

JOIN ORDERING: DYNAMIC PROGRAMMING

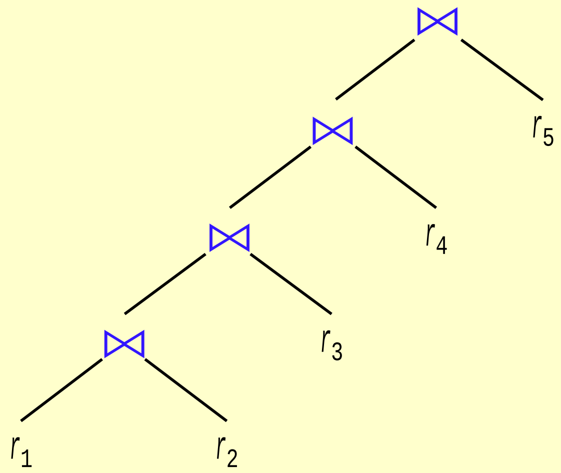
- To find the best plan for a set S of n relations, consider all possible plans of the form:
  - S_1 JOIN $(S - S_1)$ where S_1 is any non-empty subset of S.
- Recursively compute the costs for joining subsets of S to find the cost of each plan.
- Choose the cheapest of the $2^n - 1$ alternatives.
- When the plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it.

JOIN ORDERING: OPTIMIZATION ALGORITHM

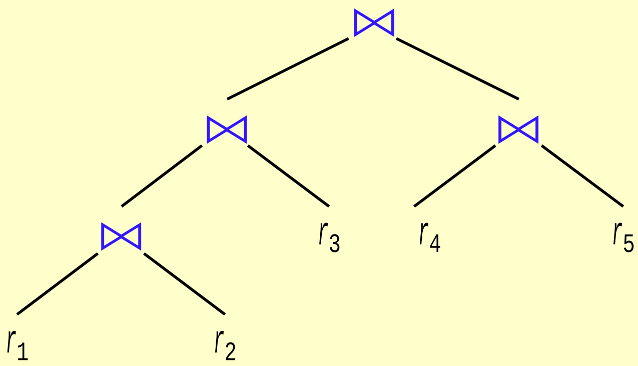
procedure FindBestPlan(S)
    if (*bestplan*[S].*cost* $\neq \infty$) /* *bestplan*[S] already computed */
      return *bestplan*[S]
    if (S contains only 1 relation)
        set *bestplan*[S].*plan* and *bestplan*[S].*cost* based on best way of accessing S
    else for each non-empty subset S_1 of S such that $S_1 \neq S$
        P_1= FindBestPlan(S_1)
        P_2= FindBestPlan(S - S_1)
        A = best algorithm for joining results of P_1 and P_2
        cost = P_1.*cost* + P_2.*cost* + cost of A
        if *cost* < *bestplan*[S].*cost*
            bestplan[S].*cost* = cost
            bestplan[S].*plan* = "execute P_1.*plan*; execute P_2.*plan*; join results of P_1 and P_2 using A"
    return *bestplan*[S]

LEFT DEEP JOIN TREES

- 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 since they facilitate pipelining. **Why?**



Left-deep join tree



Non-left-deep join tree

INTERESTING SORT ORDERS

- Consider the expression (r_1 Join r_2 Join r_3) Join r_4 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 Join r_2 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 to r_1 and r_2 is not useful.
  - Using merge-join to compute (r_1 Join r_2 Join r_3) may be costlier than hash join, but may provide an output sorted in an interesting order.
- Need to find the best join order for each subset of the set of *n* relations, *for each interesting sort order*.
  - Simple extension of earlier dynamic programming algorithms.
  - Usually, the number of interesting orders is quite small and doesn't affect time/space complexity significantly.