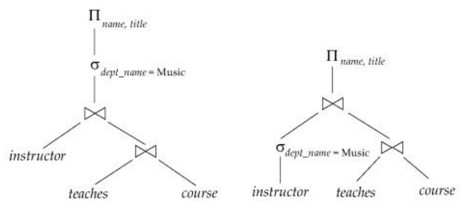
Query Optimization

Database System Concepts, 6th Ed.

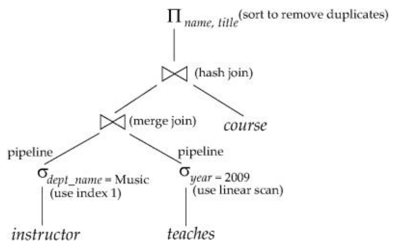
Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



Introduction (Cont.)

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



Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\Pi_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

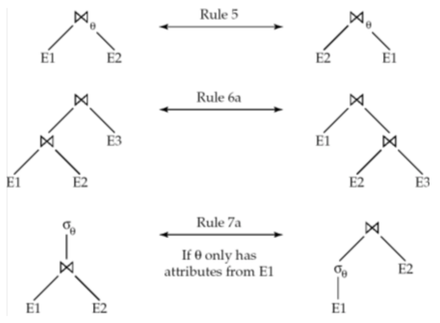
Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
$$s_{q_1 Ù q_2}(E) = s_{q_1}(s_{q_2}(E))$$
2. Selection operations are commutative.
$$s_{q_1}(s_{q_2}(E)) = s_{q_2}(s_{q_1}(E))$$
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$
4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \text{ X } E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.
$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$
6. (a) Natural join operations are associative:
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$
   (b) Theta joins are associative in the following manner:
$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$
   where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Join Ordering Example

- For all relations $r_1, r_2,$ and r_3,
$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$
  (Join Associativity)
- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose
$$(r_1 \bowtie r_2) \bowtie r_3$$
  so that we compute and store a smaller temporary relation.

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- **The above approach is very expensive in space and time**
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Cost Estimation

- Cost of each operator computed
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $(2(n - 1))!/(n - 1)!$ different join orders for 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.

Dynamic Programming in Optimization

- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ 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 $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

Search Space

The resulting search space is **enormous**:

Possible bushy join trees joining *n* relations

number of relations n	C_{n-1}	join trees
2	1	2
3	2	12
4	5	120
5	14	1,680
6	42	30,240
7	132	665,280
8	429	17,297,280
10	4,862	17,643,225,600

Dynamic Programming

Example (Four-way join of tables $R_{1,\ldots,4}$)

Pass 1 (best 1-relation plans)
Find the best **access path** to each of the R_i individually (considers index scans, full table scans).

Pass 2 (best 2-relation plans)
For each **pair** of tables R_i and R_j, determine the best order to join R_i and R_j (use $R_i \bowtie R_j$ or $R_j \bowtie R_i$?):

$$optPlan(\{R_i, R_j\}) \leftarrow \text{best of } R_i \bowtie R_j \text{ and } R_j \bowtie R_i \ .$$

$\rightarrow$ 12 plans to consider.

Pass 3 (best 3-relation plans)
For each **triple** of tables R_i, R_j, and R_k, determine the best three-table join plan, using sub-plans obtained so far:

$$optPlan(\{R_i, R_j, R_k\}) \leftarrow \text{best of } R_i \bowtie optPlan(\{R_j, R_k\}), \\ optPlan(\{R_j, R_k\}) \bowtie R_i, \quad R_j \bowtie optPlan(\{R_i, R_k\}), \ldots \ .$$

$\rightarrow$ 24 plans to consider.

Dynamic Programming

Example (Four-way join of tables $R_{1,\ldots,4}$ (cont'd))

Pass 4 (best 4-relation plan)
For each set of **four** tables R_i, R_j, R_k, and R_l, determine the best four-table join plan, using sub-plans obtained so far:

$$optPlan(\{R_i, R_j, R_k, R_l\}) \leftarrow \text{best of } R_i \bowtie optPlan(\{R_j, R_k, R_l\}), \\ optPlan(\{R_j, R_k, R_l\}) \bowtie R_i, \quad R_j \bowtie optPlan(\{R_i, R_k, R_l\}), \ldots, \\ optPlan(\{R_i, R_j\}) \bowtie optPlan(\{R_k, R_l\}), \ldots \ .$$

$\rightarrow$ 14 plans to consider.

- Overall, we looked at only **50** (sub-)plans (instead of the possible 120 four-way join plans; ↗ slide 12).
- All decisions required the evaluation of **simple** sub-plans only (**no need to re-evaluate** $optPlan(\cdot)$ for already known relation combinations $\Rightarrow$ use lookup table).

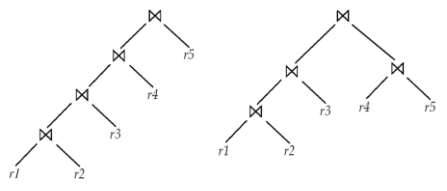
Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S  /* Using selections on S and indices on S */
    else 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]
```

* Some modifications to allow indexed nested loops joins on relations that have selections (see book)

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.



(a) Left-deep join tree (b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!