CS525: Advanced Database Organization

Notes 6: Query Optimization
Physical

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

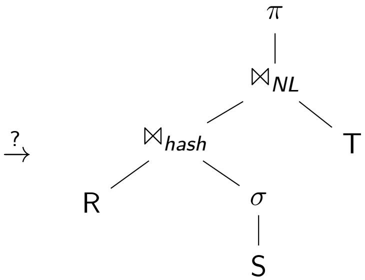
yelmehdwi@iit.edu

November 8, 2018

Slides: adapted from a courses taught by Hector Garcia-Molina, Stanford, T. Grust, Universitï£ · t Tuebingen & Distributed DBMS by M. Oezsu & P. Valduriez

Finding the “Best” Query Plan

```
SELECT  ...
FROM    ...
WHERE   ...
```



- We already saw that there may be more than one way to answer a given query.
  - Which one of the join operators should we pick? With which parameters (block size, buffer allocation,...)?
  - The task of finding the best execution plan is, in fact, the “holy grail” of any database implementation.

Overview of query optimization

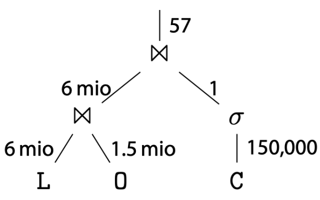
- Parser: syntactical/semantical analysis
- Rewriting: heuristic optimizations independent of the current database state (table sizes, availability of indexes, etc.). For example:
  - Apply predicates early
  - Avoid unnecessary duplicate elimination
- Optimizer: optimizations that rely on a cost model and information about the current database state
- The resulting plan is then evaluated by the system's execution engine

Cost of Query

- Parse + Analyze
  - Can parse SQL code in milliseconds
- Optimization - Find plan
  - Generating plans, costing plans
- Execution
  - Execute plan
- Return results to client
  - Can be expensive but not discussed here

Impact on Performance

- Finding the right plan can dramatically impact performance.



  - In terms of execution times, these differences can easily mean "seconds versus days."

Theory vs. Implementation

- Theory
  - The join operator is (in theory) commutative: (i.e.: symmetric in behavior)
    - $R \bowtie S = S \bowtie R$
- Practice:
  - The algorithms that implement the join operation are Asymmetric
    - i.e., the role of the first input relation is different from the role of the second input relation

Cost-Based Optimization: Overall idea

- Physical Optimization
  - Apply after applying heuristics in logical optimization

  1. enumerates all possible execution plans, (if this yields too many plans, at least enumerate the “promising” plan candidates)
  2. estimates cost of each plan
  3. chooses the best one as the final execution plan.
     - i.e., picks plan with least estimated costs

- To apply pruning in the search for the best plan
  - Steps 1 and 2 have to be interleaved
  - Prune parts of the search space
    - if we know that it cannot contain any plan that is better than what we found so far
- Before we can do so, we need to answer the question
  - What is a “good” execution plan at all?

Cost Metrics

- Database systems judge the quality of an execution plan based on a number of cost factors, e.g.,
  - the number of disk I/Os required to evaluate the plan,
  - the plan's CPU cost,
  - the overall response time observable by the database client as well as the total execution time.
- All of the above factors depend on one critical piece of information: the size of (intermediate) query results.
- Database systems, therefore, spend considerable effort into accurate result size estimates.

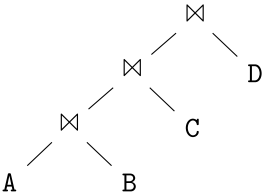
Search Space

- Search space: The set of alternative query execution plans (query trees)
  - Typically very large
  - The main issue is to optimize the joins
  - For `N` relations, there are `O(N!)` equivalent join trees that can be obtained by applying commutativity and associativity rules

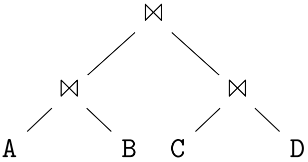
Search Space

- Restrict by means of heuristics
  - Perform unary operations before binary operations, etc.
- Restrict the shape of the join tree
  - Consider the type of trees (linear trees, vs. bushy ones)

- Linear Join Tree



- Bushy Join Tree



Plan Enumeration

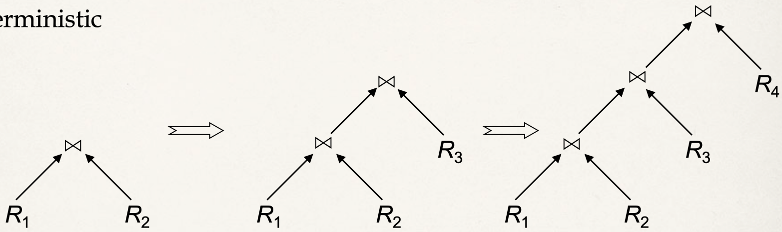
- For each operator in the query
  - Several implementation options
- Binary operators (joins)
  - Changing the order may improve performance a lot!
- consider both different implementations and order of operators in plan enumeration

Plan Enumeration Algorithms/Search Strategy

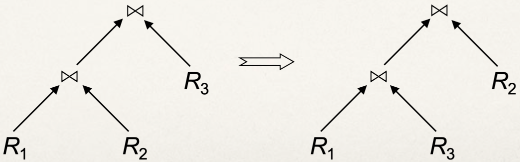
- Deterministic scan of the search space
  - Start from base relations and build plans by adding one relation at each step
  - Breadth-first strategy: build all possible plans before choosing the "best" plan (dynamic programming approach)
  - Depth-first strategy: build only one plan (greedy approach)
- Randomized
  - Search for optimal solutions around a particular starting point
  - Trade optimization time for execution time
    - Does not guarantee that the best solution is obtained, but avoid the high cost of optimization
  - The strategy is better when more than $>$ 10 relations are involved
  - Simulated annealing
  - Iterative improvement

Search Space

- Deterministic



- Randomized

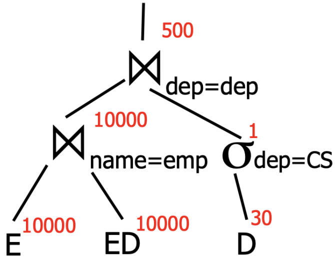


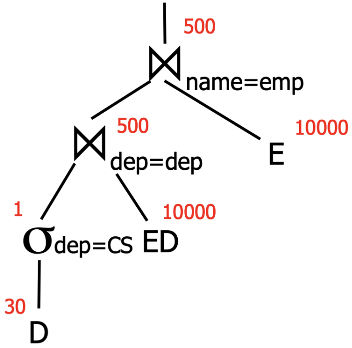
Join Order

- For a query plan that contains multiple order, we often have a choice in which order we execute the different join operations.
- Notice that a particular order assumes a suitable join predicate on all pairwise joins.
- If not such predicate exists, that pairwise join is equal to a cross product and hence it is likely the particular join order does not have to be considered for query optimization.
- Q: What may be the effect of picking the wrong join order?
  - If we pick an unsuitable join order, we may end up with a slow query plan
- Q: Is join order problem only relevant for joins?
  - No, the "join order" problem exists for all binary operations that are commutative and associative, e.g., union, intersection, and cross product

Example Join Ordering: Result Sizes

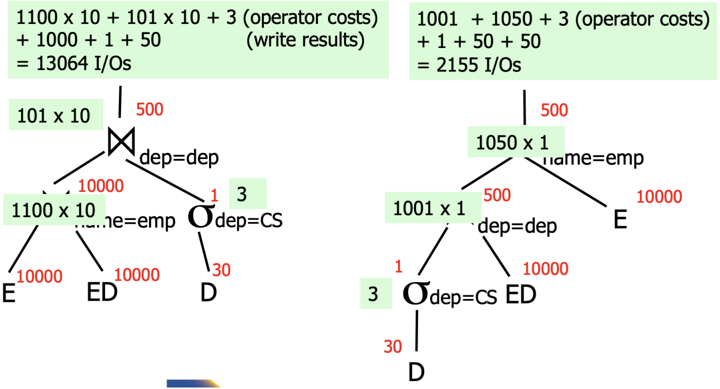
- Cost (only NL)
  - S(E)=S(ED)=S(D)=$\frac{1}{10}$ block
  - M=101





Example Join Ordering: Result Sizes

- Cost (only NL)
  - S(E)=S(ED)=S(D)=$\frac{1}{10}$ block
  - M=101
  - I/O costs only
  - No pipelining, write all results to disk



Agenda : Join Optimization

- Given some query
  - How to enumerate all plans?
- Try to avoid cross-products
- Need way to figure out if equivalences can be applied
  - Data structure: Join Graph

Queries Considered

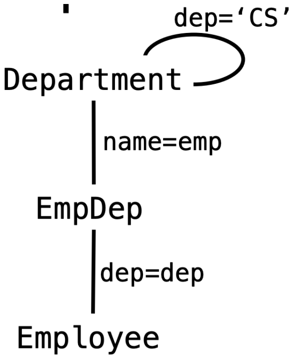
- Concentrate on join ordering, that is:
  - conjunctive queries
  - simple predicates
  - predicates have the form $a_1 = a_2$ where a_1 is an attribute and a_2 is either an attribute or a constant
- We join relations $R_1, \ldots, R_n$ where R_i can be
  - a base relation
  - a base relation including selections
  - a more complex building block or access path

Query Graph

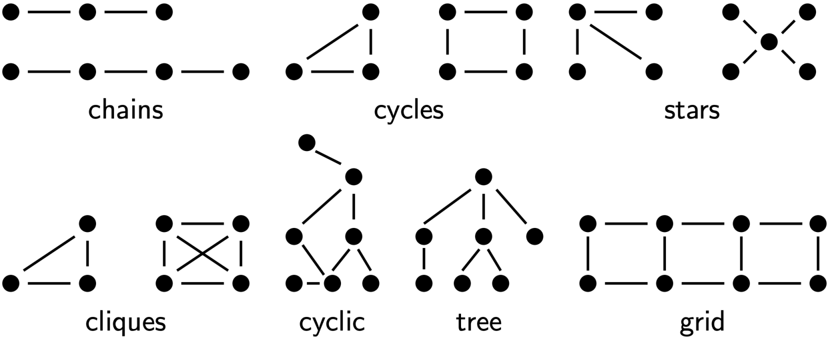
- Queries of this type can be characterized by their query graph:
  - the query graph is an undirected graph with $\mathtt{R_1,\ldots,R_n}$ as nodes
  - a predicate of the form $\mathtt{a_1 = a_2}$, where $\mathtt{a_1} \in \mathtt{R_i}$ and $\mathtt{a_2} \in \mathtt{R_j}$ forms an edge between $\mathtt{R_i}$ and $\mathtt{R_j}$ labeled with the predicate
  - a predicate of the form $\mathtt{a_1 = a_2}$, where $\mathtt{a_1} \in \mathtt{R_i}$ and $\mathtt{a_2}$ is a constant forms a self-edge on $\mathtt{R_i}$ labeled with the predicate
  - most algorithms will not handle self-edges, they have to be pushed down

Query Graph: Example

```
SELECT e.name
FROM Employee e,
     EmpDep ed,
     Department d
WHERE e.name = ed.emp
      AND ed.dep = d.dep
      AND d.dep = 'CS'
```



Shapes of Query Graphs

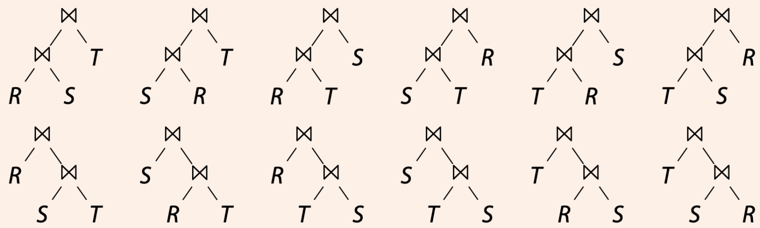


Join Trees

- A join tree is a binary tree with
  - join operators as inner nodes
  - relations as leaf nodes
- Shape of Join Trees
  - Commonly used classes of join trees:
    - left-deep tree
    - right-deep tree
    - zigzag tree
    - bushy tree
  - The first three are summarized as linear trees

How many join orders?

- Assumption
  - Use cross products (can freely reorder)
  - Joins are binary operations
    - Two inputs
    - Each input either join result or relation access
- Example: 3 relations `R,S,T`: Ways of building a 3-way join from two 2-way joins
  - 12 orders



- How many join orders? How Many Such Combinations Are There?

Finding the optimal join-tree: search space analysis

- Consider the possible join ordering when there are 3 relations:

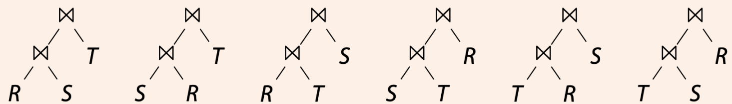
```
Form 1:
   ( R ⋈ S ) ⋈ T      ( S ⋈ R ) ⋈ T
   ( R ⋈ T ) ⋈ S      ( T ⋈ R ) ⋈ S
   ( S ⋈ T ) ⋈ R      ( T ⋈ S ) ⋈ R

Form 2:
   R ⋈ ( S ⋈ T )      R ⋈ ( T ⋈ S )
   S ⋈ ( R ⋈ T )      S ⋈ ( T ⋈ R )
   T ⋈ ( R ⋈ S )      T ⋈ ( S ⋈ R )
```

- The number of different join orderings of `n` relations is exponentially large
- Because the number of permutations is exponentially large
  - The number of possible join trees to consider is just too large
  - We need to reduce the search space.
- Some type of trees in better than others: the left-deep tree

Search space for left-Deep trees

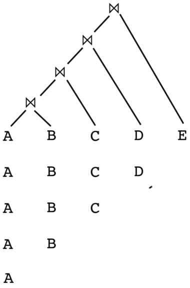
- Left-Deep Tree
  - Given a sequence of input relations $\mathtt{R_1, R_2, .., R_n}$, a left-deep tree starts by a binary join on $\mathtt{R_1}$ and $\mathtt{R_2}$. Then it iteratively adds binary joins $\mathtt{R_3, .., R_n}$.
- Example: 3 way join: $\mathtt{R} \bowtie \mathtt{S} \bowtie \mathtt{T}$
  - The possible left-deep join trees are:



  - `# left deep tree = 6 (= 3!)`
- $\Rightarrow$ Number of possible left-deep join trees for `n` input relations is `n!` (factorial)

Search space for left-Deep trees: Example

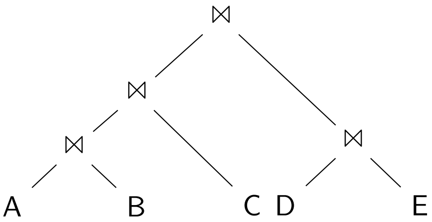
- Starting by joining two relations $A \bowtie B$ then the result joining with the next relation, and so on



- # of options in terms of structure (# of unlabeled trees (varies)) = 1
- # different order (# of leaf combinations) = n!=5!=120
- # of join left-deep trees = # of leaf combinations $\times$ # of unlabeled trees (varies) = n!$\times$ 1 = n!

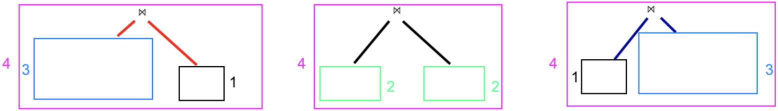
Search space for not a left-Deep trees: Bushy Tree

- Bushy Tree
  - Tree that is not left-deep, i.e., at least for one of the joins in that plan its right input is not an input relation but another join operation.
  - Notice that the union of the set of bushy trees and the set of left-deep tress forms the set of all possible trees.

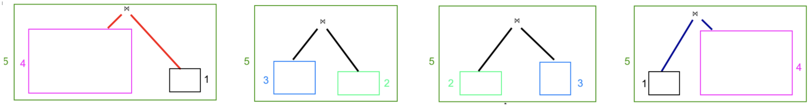


Search space for not a left-Deep trees: Bushy Tree

$\Rightarrow$ 2 options



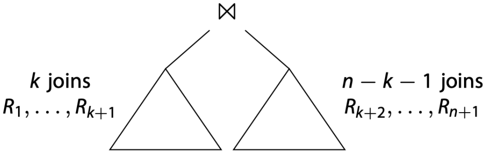
$\Rightarrow$ 5 options



$\Rightarrow$ 14 options

How Many Such Combinations Are There?

- A join over `n+1` relations $R_1, \ldots, R_{n+1}$ requires `n` binary joins.
- Its root-level operator joins sub-plans of `k` and `n-k-1` join operators ($0 \leq k \leq n-1$):



- Let C_i be the number of possibilities to construct a binary tree of `i` inner nodes (join operators):

$$C_n = \sum_{k=0}^{n-1} C_k \; C_{n-k-1}$$

Catalan Numbers

- This recurrence relation is satisfied by Catalan numbers:

$$C_n = \begin{cases} 1, & \text{if } n=0 \\ \sum_{k=0}^{n-1} C_k C_{n-k-1}, & n>0 \end{cases}$$

- describing the number of ordered binary trees with $n + 1$ leaves.
- It can be written in a closed form as $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$
- For **each** of these trees, we can **permute** the input relations $R_1, \ldots, R_{n+1}$, leading to:

Number of possible join trees for an (n+1)-way relational join

- $\frac{(2n)!}{(n+1)!n!} \times (n+1)! = \frac{(2n)!}{n!}$

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

- And we haven't yet even considered the use of `k` different join algorithms (yielding another factor of $\mathtt{k}^{(\mathtt{n-1})}$)

How many join orders?

- If for each join we consider `k` join algorithms then for `n` relations we have
    - Multiply with a factor $\mathtt{k}^{\mathtt{n-1}}$
- Example consider
    - Nested loop
    - Merge
    - Hash

What is dynamic programming?

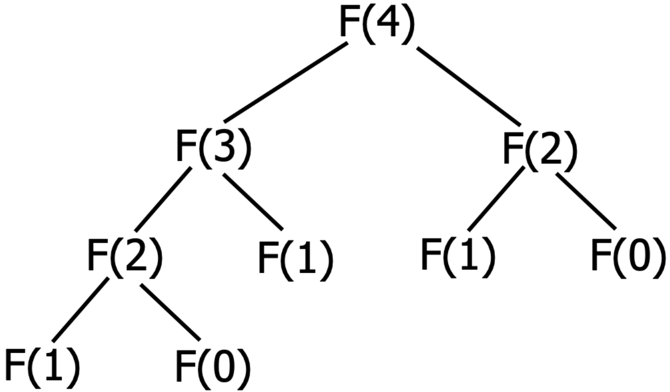
- Recall data structures and algorithms
- Consider a Divide-and-Conquer problem
  - Solutions for a problem of size `n` can be build from solutions for sub-problems of smaller size (e.g., $\frac{n}{2}$ or `n-1`)
- Memoize
  - Store solutions for sub-problems
  - Each solution has to be only computed once
  - Needs extra memory

Example Fibonacci Numbers

- F(n) = F(n-1) + F(n-2)
- F(0) = F(1) = 1

```
Fib(n){
  if (n < 0) return 0
  else if (n = 1) return 1
  else return Fib(n-1) + Fib(n-2)
}
```

Example Fibonacci Numbers

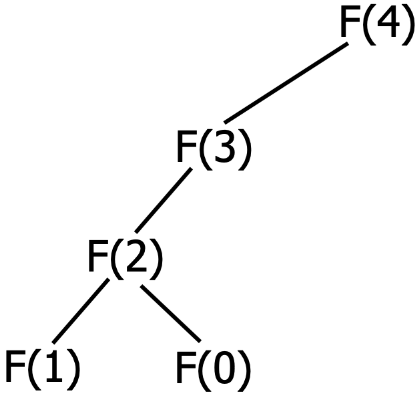


- Complexity
  - Number of calls
    - $T(n) = T(n-1) + T(n-2) + \theta(1)$
    - Another way to solve it (lower bound): $T(n) \geq 2T(n-2)$ $= \theta(2^{\frac{n}{2}})$

Using dynamic programming

```
Fib(n)
{
    int[] fib;
    fib[0] = 1;
    fib[1] = 1;
    for(i = 2; i < n; i++)
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n];
}
```

Example Fibonacci Numbers



What do we gain?

- $\theta(\mathtt{n})$ instead of $\theta(2^{\frac{\mathtt{n}}{2}})$

Dynamic Programming for Join Enumeration

- The traditional approach to master this search space is the use of dynamic programming
- **Idea**
  - Find the cheapest plan for an `n`-way join in `n` **passes**.
  - In each pass `k`, find the best plans for all `k`-relation **sub-queries**.
  - Construct the plans in pass `k` from best `i`-relation and `(k-i)`-relation sub-plans found in **earlier passes** ($1 \leq i < k$).
- **Assumption:**
  - To find the optimal **global plan**, it is sufficient to only consider the optimal plans of its **sub-queries** ("Principle of optimality").

Dynamic Programming for Join Enumeration

- Find cheapest plan for `n`-relation join in `n` passes
- For each `k` in `1...n`
  - Construct solutions of size `k` from best solutions of size $<$ `k`

DP: Example (Four-way join of tables $R_1, \ldots, R_4$)

- **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 and R_j, and R_k, determine the best three-table join plan, using sub-plans obtained so far:

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

    $\Rightarrow$ 24 plans to consider.

DP: Example-Four-way join of tables $R_1, \ldots, R_4$) cont'd

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

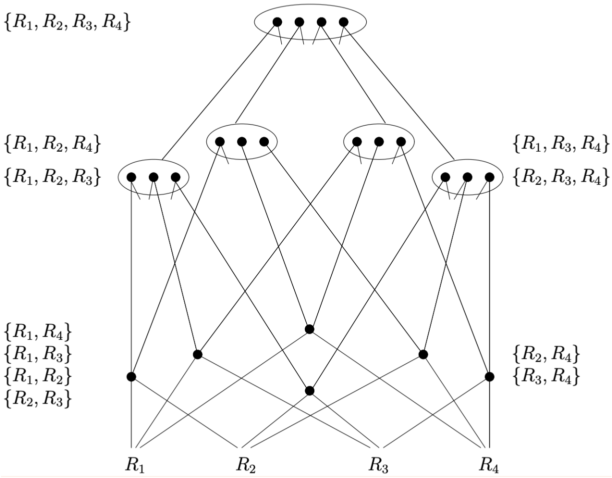
    $$\begin{aligned}&\mathit{optPlan}(\{R_i, R_j, R_k, R_l\}) \leftarrow \text{best of}\\ &R_i \bowtie \mathit{optPlan}(\{R_j, R_k, R_l\}), \mathit{optPlan}(\{R_j, R_k, R_l\}) \bowtie R_i,\\ &R_j \bowtie \mathit{optPlan}(\{R_i, R_k, R_l\}), \ldots,\\ &\mathit{optPlan}(\{R_i, R_j\}) \bowtie \mathit{optPlan}(\{R_k, R_l\})\end{aligned}$$

    $\Rightarrow$ 14 plans to consider.

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

Sharing Under the Optimality Principle

- Sharing optimal sub-plans



Dynamic Programming Algorithm

Find optimal n-way bushy join tree via dynamic programming

Function: `find_join_tree_dp` $(q(R_1, \ldots, R_n))$

for $i = 1$ **to** n **do**
  $optPlan(\{R_i\}) \leftarrow$ `access_plans` (R_i) ;
  `prune_plans` $(optPlan(\{R_i\}))$;
for $i = 2$ **to** n **do**
  foreach $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do**
    $optPlan(S) \leftarrow \varnothing$;
    foreach $O \subset S$ with $O \neq \varnothing$ **do**
      $optPlan(S) \leftarrow optPlan(S) \cup$
        `possible_joins` $\left[optPlan(O) \bowtie optPlan(S \setminus O) \right]$;
    `prune_plans` $(optPlan(S))$;
return $optPlan(\{R_1, \ldots, R_n\})$;

- *possible_joins* [R $\bowtie$ S] enumerates the possible joins between R and S (nested loops join, merge join, etc.).
- *prune_plans* (set) discards all but the best plan from set.

Dynamic Programming - Discussion

- Enumerate all non-empty true subsets of S:

```
O = S & -S;
do {
  /* perform operation on O */
  O = S & (O - S);
} while (O != S);
```

- *find_join_tree_dp* () draws its advantage from **filtering** plan candidates early in the process.
  - In our example, pruning in Pass 2 reduced the search space by a factor of 2, and another factor of 6 in Pass 3.
- Some **heuristics** can be used to prune even more plans:
  - Try to avoid **Cartesian products**.
  - Produce **left-deep plans** only (see next slides).

Left/Right-Deep vs. Bushy Join Trees

- The algorithm on slide 45 explores all possible shapes a join tree could take
- Actual systems often prefer left-deep join trees.
  - The **inner** (rhs) relation always is a **base relation**.
  - Allows the use of **index nested loops join**.
  - Easier to implement in a **pipelined** fashion.

Revisiting the assumption

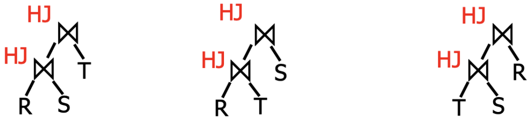
- Is it really sufficient to only look at the best plan for every sub-query?
- Cost of merge join depends whether the input is already sorted
  - A sub-optimal plan may produce results ordered in a way that reduces cost of joining above
  - Keep track of interesting orders
  - i.e., The notion of interesting orders allowed query optimizers to consider plans that could be locally sub-optimal, but produce ordered output beneficial for other operators, and thus be part of a globally optimal plan.

Interesting Orders

- Number of interesting orders is usually small
- Extend DP join enumeration to keep track of interesting orders
  - Determine interesting orders
  - For each sub-query store best-plan for each interesting order
- In $prune\_plans()$, retain
  - the cheapest "unordered" plan and
  - the cheapest plan for each interesting order.

Example Interesting Orders

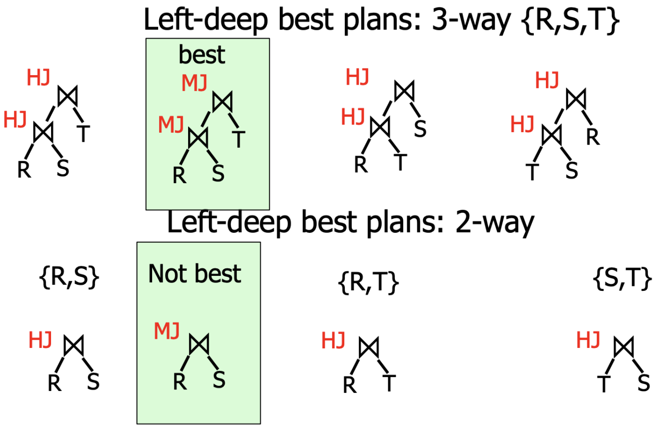
Left-deep best plans: 3-way {R,S,T}



Left-deep best plans: 2-way

{R,S} {R,T} {S,T}

Example Interesting Orders



Joining Many Relations

- Dynamic programming still has exponential resource requirements[1]:
  - time complexity: $O(3^n)$
  - space complexity: $O(2^n)$
- This may still be too expensive
  - for joins involving many relations ($\sim$ 10-20 and more),
  - for simple queries over well-indexed data (where the right plan choice should be easy to make).
- The greedy join enumeration algorithm jumps into this gap.

[1] K. Ono, G.M. Lohman, Measuring the Complexity of Join Enumeration in Query Optimization, VLDB 1990

Other join enumeration techniques

- Randomized algorithms
  - randomly rewrite the join tree one rewrite at a time; use **hill-climbing** or **simulated annealing** strategy to find optimal plan.
- Genetic algorithms

Summary: (Join) Optimization

- Find "best" query execution plan based on a cost model (considering I/O cost, CPU cost,...); data statistics (histograms); dynamic programming, greedy join enumeration; physical plan properties (interesting orders).