Query Optimization

Objectives:
• an overview of the general structure of complete optimization process
• optimization of join order by dynamic programming

Reading:
- Ch. 16.5, 16.6
- At this point you are responsible for all of Ch. 16

•Slide thank yous: some from my colleague Don Batory, various text slides

What do we know today?

- Query expression trees
- Adorned with
  - choice of physical operator
  - structural data info. e.g. relation schema
  - statistical information concerning size and data distribution of the arguments
- Cost functions associated with operators connect the costs between nodes of the expression tree.
- Logically correct manipulation of those trees can result in query plans with different execution times.

Query Optimization

So far,
- logical [query] expression trees
  - manipulate using the identities of relational algebra
  - estimate size of results
- physical operators
  - often have a choice of operator
  - actual costs, CPU & I/O,
    as a function of detailed statistics of the data & machines

Lecture Approach

- Examine in detail a core, only slightly idealized optimization problem - join order
  - example
  - start, a host of terminology
  - specific algorithms

- Speak in generalities to the structure of the system as a whole.

System R: Classic Algorithm

- System R optimized retrieval-join expressions (only) using dynamic programming algorithm

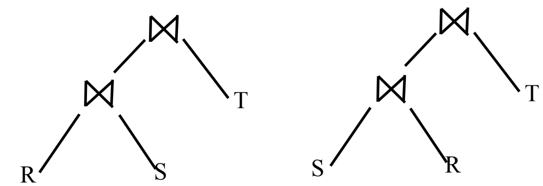
"Access path selection in a relational database management system. "

P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. In SIGMOD Conference, pages 23--34, 1979.

Commutativity of Join

$\bowtie$

- $(R \bowtie S) \bowtie T = (S \bowtie R) \bowtie T$)



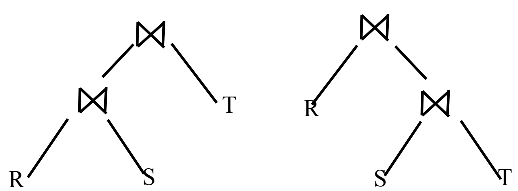
- Question: Given a tree represented as pointers in memory could you write a program that would recognize commutativity

and change the pointers around to form

Associativity of Join

□ $\bowtie$

- $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T$)
- □



- Question: Given a tree represented as pointers in memory could you write a program that would recognize, associativity

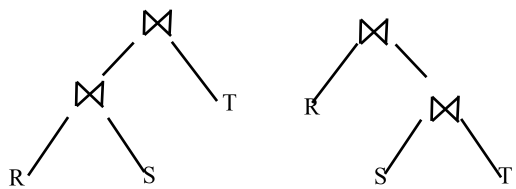
and change the pointers around to form

<u>Join Selectivity</u>

$$\text{join selectivity} = \frac{|R \bowtie S|}{|R \times S|}$$

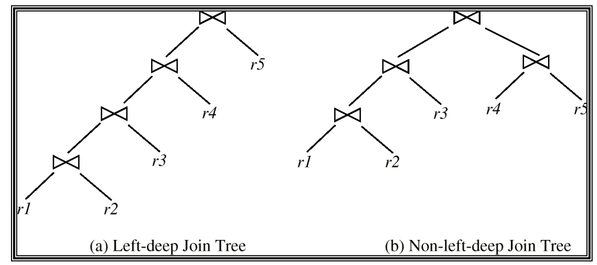
Consider the Size of Intermediate Results

- Suppose T, is very small, join selectivity = 0.1
  - T(T) = 100, (sorry)
  - T(R) = T(S) = 10,000



Left [Right] Deep Join Trees vs. Bushy

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.

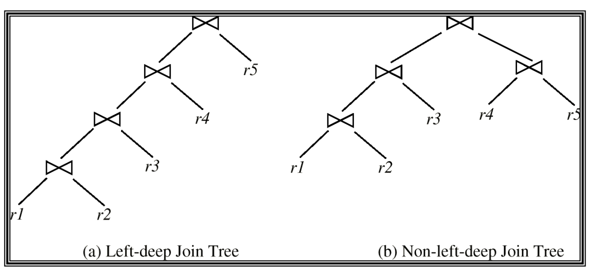


(a) Left-deep Join Tree (b) Non-left-deep Join Tree

A bushy join tree

Left [Right] Deep

- Maximize pipelining
- Minimize memory needed
- 1 topology → no choices when searching

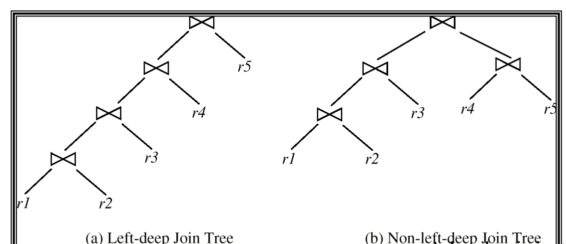


(a) Left-deep Join Tree (b) Non-left-deep Join Tree

A bushy join tree

Bushy Plans

- More choices (topology)
  - Can get better plans
  - Harder search (more combinations)
- More opportunity for parallelism
  - Execute plans in subtrees independently



(a) Left-deep Join Tree (b) Non-left-deep Join Tree

A bushy join tree

Choose the Optimal Join Order

- Just determining the best logical join order for a left/right deep tree is NP-Hard

Query Optimizers Combine

- Simplying assumptions
- Greedy algorithms
- Search methods

All of it a heavy emphasis on heuristics.

Beginner Heuristics:

1. Restrict to Left[/Right] Deep Trees
   - minimize intermediate storage
   - how many join orders, still, (ans: n!)
2. Given left-deep tree, sort relations by size.
   - when might it fail?

Pushing Selects & Projects

- Execute selects and projects
  - as early as possible, (before the joins)
  - use O(n) operations to reduce n for the O(n^2) operation
- Formally: Apply relational transformations to a query expression tree, such that select and project occur before the joins.

"Dynamic Programming" Algorithm for Join Ordering

- Consider A, B, C, D
  - Logical plan optimization
  - Number of rows and selectivity
- Bottom-up
  - first consider all two way joins
  - record cost & order in matrix. // for class, size of the output

AB	AC	AD	BC	BD	CD
A	B	C	D		

Fill in Table

- to fill "ABC"
- consider cost all

possible plans:

Min(
  - AB⋈C
  - AC⋈B
  - BC⋈A

)

ABC					
AB	AC	AD	BC	BD	CD
A	B	C	D		

- Record cost and "winner"

For class focus is logical plan – minimize total size of intermediate results in rows

- to fill "ABC"
- consider cost all

possible plans:

Min(
  - AB⋈C
  - AC⋈B
  - BC⋈A

)

A(BC)					
AB	AC	AD	BC	BD	CD
A	B	C	D		

1. Since all result in the same size intermediate result, really want Min_size

Search Saving Aspect

- to fill "ABD"
- consider:

Min(
  - AB⋈D
  - AD⋈B
  - BD⋈A

)

ABC	ABD				
AB	AC	AD	BC	BD	CD
A	B	C	D		

- Cost of AB computed once,

but used to fill both ABC & ABD cells

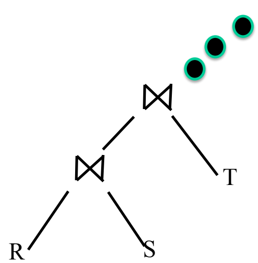
Each of these computed once used twice

ABC	ABD	ACD	BCD		
AB	AC	AD	BC	BD	CD
A	B	C	D		

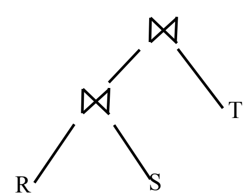
Even restricted to left-deep n! possible sequences, found in 2^n-1 steps

ABCD					
ABC	ABD	ACD	BCD		
AB	AC	AD	BC	BD	CD
A	B	C	D		

But, do we really want to evaluate all possible permutations?

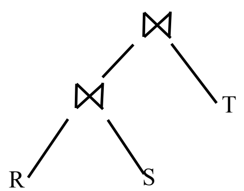


What if there are *no joinable* columns between R and S?



- What does R $\bowtie$ S have to produce to achieve a correct result?

What if there are *no joinable* columns between R and S?



- What does R $\bowtie$ S have to produce to achieve a correct result?
- → Cartesian product
- How big is that?

Query Graph

A query graph for query Q, is a labeled graph (V, E),

- where each vertex Vi corresponds to a relation Ri in Q, and
- there is an edge from Vi to Vj, if there is a predicate, in the query P(Ri, Rj).

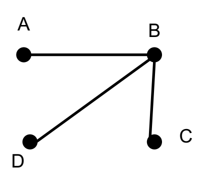
What about transitivity? (a = b & b = c)

Example Query Graph

(aside, if the graph is acyclic neat things can happen)

- select *
  from A, B, C, D
  where A.a=4 and
  B.b = 3 and
  C.c = 3 and
  D.d = 5 and
  A.x=B.x and
  B.y=C.y and
  B.z=D.z

Query Graph
nodes = relations
edges = join predicates



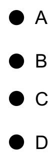
Query-Graph Heuristic

Instead of exhaustively trying every remaining relation
- Use query graph to avoid cartesian products

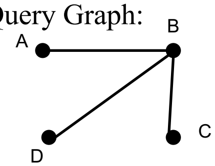
- For each vertex, V from the query graph
  - remove it from query graph
  - add it to the join plan
- Until query graph is empty, for each plan, {
  - choose each vertex remaining in query graph but connected (edgewise) to the plan.
  - (if none, but query graph is nonempty, choose smallest vertex whose relation is smallest) }

From Query Graph Perspective

- 1-relation
  - connected subgraphs with 1 node

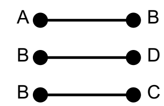


- Query Graph:

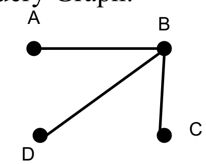


From Query Graph Perspective

- Two relations
  - 2-way joins only
  - Two relations on an edge



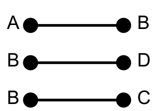
- Query Graph:



why not D to A? D to C?

From Query Graph Perspective

- Two relations
  - 2-way joins only
  - Two relations on an edge

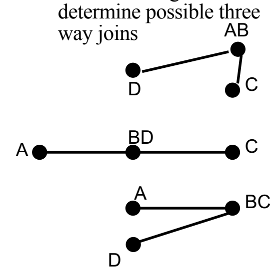


AB	~~AC~~	~~AD~~	**BC**	**BD**	~~CD~~
A	B	C	D		

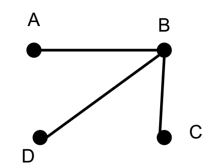
why not D to A? D to C?

From Query Graph Perspective

- Cheapest 3-relation queries
  - remove an edge to determine possible three way joins

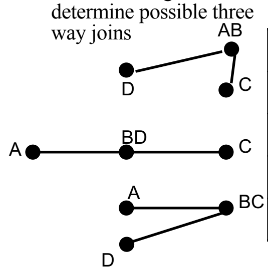


- Query Graph:



From Query Graph Perspective

- Cheapest 3-relation queries
  - remove an edge to determine possible three way joins
- Query Graph:



ABD	ABC	BDC	BCD		
AB	~~AC~~	~~AD~~	**BC**	**BD**	~~CD~~
A	B	C	D		

Advantages of the Query Graph

- avoid considering Cartesian products

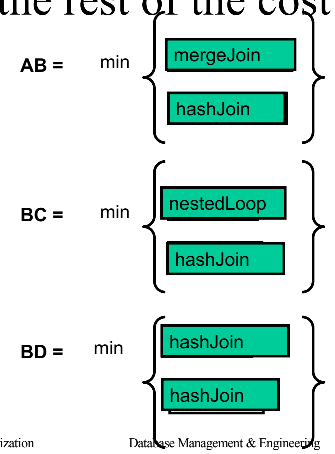
In conjunction with dynamic programming approach

- net, very large reduction in search.
- can degenerate [I believe] to $O(n^2)$

But 2^n not so bad

- But we were only considering
  - logical alternative
  - cost model: sum of the sizes of intermediate results

Can consider physical operators and the rest of the cost equations



AB denotes most efficient way to join relations A, B

BC denotes most efficient way to join relations B, C

etc.

Two most important heuristics

1. Push down selects and projects
   - so reliable, blindly done first - period.
2. Optimize logical plan, then physical plan
   - join order
   - index-based access paths (mainting sort properties)

Practice?

- Pipeline of heuristics

1. push down selects and projects
2. Some rule-based systems
   - e.g. if |R| > |S| transform $S \bowtie R$ to $R \bowtie S$
3. Logical plan optimization
4. Physical optimization

Volcano [dewitt]

- Pipeline of heurstics

1. push down selects and projects
2. Logical plan optimization
   - Decompose logical plan into subplans
   - Map/optimize subplans to physical plans
3. Build complete physical plans from "library" of physical subplans.

Cutting Edge

- Adaptive query plans
  - monitor/measure a query while it is executing
  - replan in the background based on measurements
  - if warranted, dynamically reorganize the plan.
- Several papers in the last 2 or 3 years

"Eddies"

- First paper on adaptive query optimization, "Eddies", Avnur & Hellerstein, 2000,
  - http://db.cs.berkeley.edu/papers/sigmod00-eddy.pdf