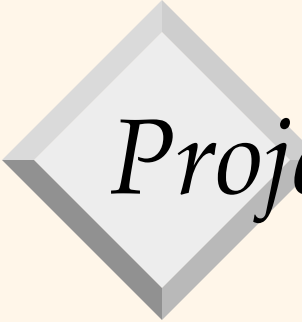



Practicum in Database Systems

Project 4 intro



Project 4 overview

- Bring together your work for P2 and P3
- Support query optimization:
 - gather data statistics
 - choose implementation of selection
 - choose join order
 - choose implementation of each join
- Contains algorithms/ data structures you may not have seen or not seen in this exact format




Architecture/format changes

- Physical Plan Builder config file goes away
- Interpreter config file contains just input/output/temp sort directory
- You need to generate and print logical and physical plans as well as query answers
- See instructions for expected format of your logical/physical plan



Step 1: gather statistics

- Write code to collect basic stats on your relations:
 - number of tuples
 - for each attribute, min and max values
- Your interpreter should do this before running queries
- Should write stats out in a file that can be accessed by DB catalog



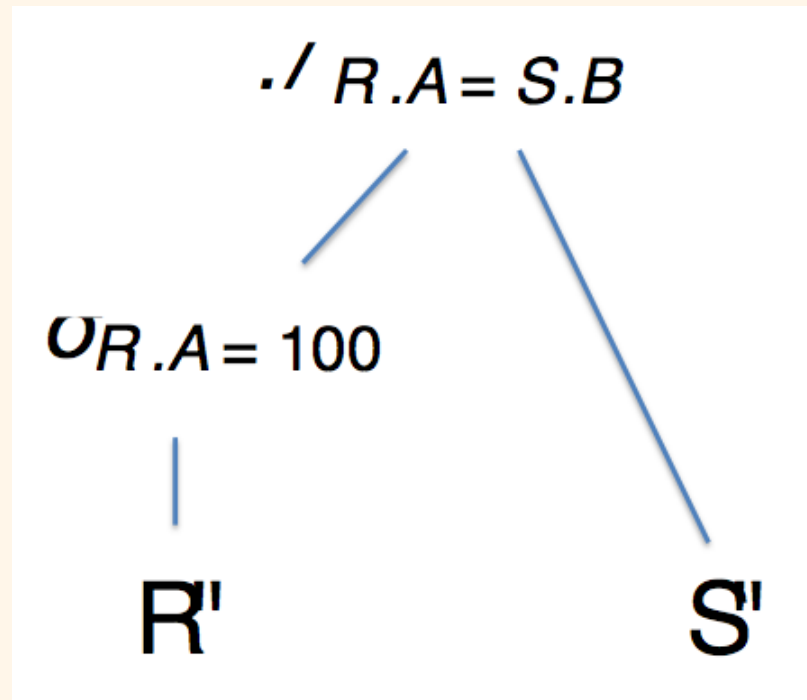
Step 2: push selections

- We already did some of this in P2
- But this time, we will push more aggressively

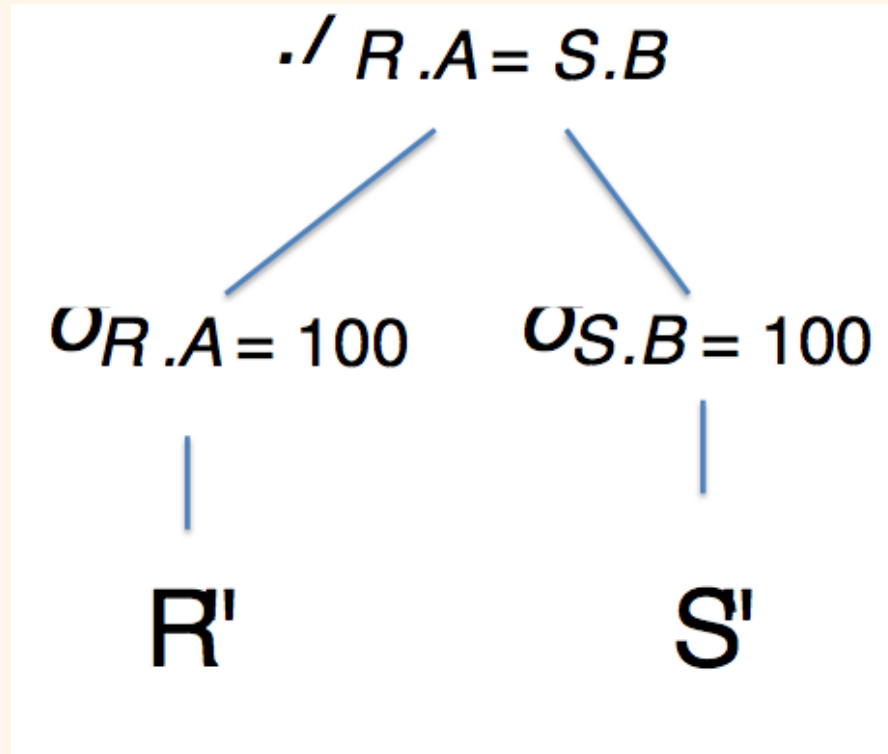
Example


SELECT * FROM R, S WHERE R.A = 100 AND R.A = S.B

- Your plan probably looks like this:



A better plan





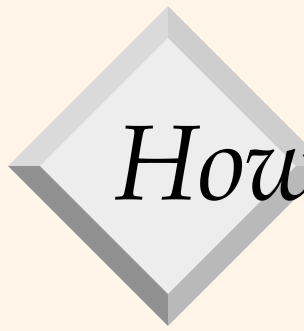
First things first

- When to push selections?
- While building logical plan
 - Always advantageous in our implementation
 - Regardless of data stats




What does your logical plan look like?

- Single relation case – same as before
- Multi-relation case – don't fix the join order
 - This will be done in the physical plan builder based on data stats
 - So you only have one join operator, with multiple children
 - Children are selections or base tables



How to push selections

- Want to "propagate" numerical constraints through equalities between attributes
- Details and scope of what you should do are specified in the instructions



The union-find data structure

- Should propagate constraints using a *union-find* data structure
 - Also sometimes called *disjoint-set*
- A union-find has a collection of elements
- Every element is a set of attributes which are constrained to be equal to each other
- May also contain numerical bounds



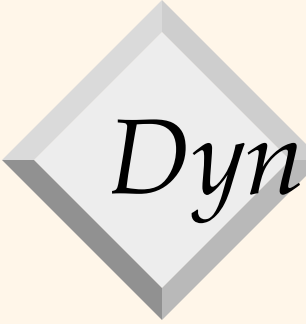
Step 3: choose implementation for each selection

- In your Physical Plan Builder, when you visit() a logical selection op
- Use data statistics and available index info
- Calculate cost for every possible way to evaluate
- Choose lowest-cost alternative
- Formulas should be familiar from 4320



Step 4: choose a join order

- Left-deep tree, but need to choose ordering
- Similar to algorithm you saw in 4320 but simpler (?)
- See instructions of textbook by Garcia-Molina, Ullman and Widom for more details in instructions)



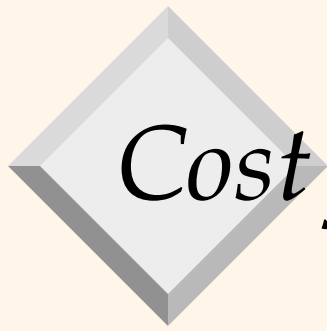
Dynamic programming algorithm

- Iterate over all subsets of relations, in increasing order of size
 - All subsets of size 2, all subsets of size 3, etc
- For each subset, find and retain only the lowest-cost join order
- This terminates with the lowest-cost join order for the entire set



Finding the lowest cost plan

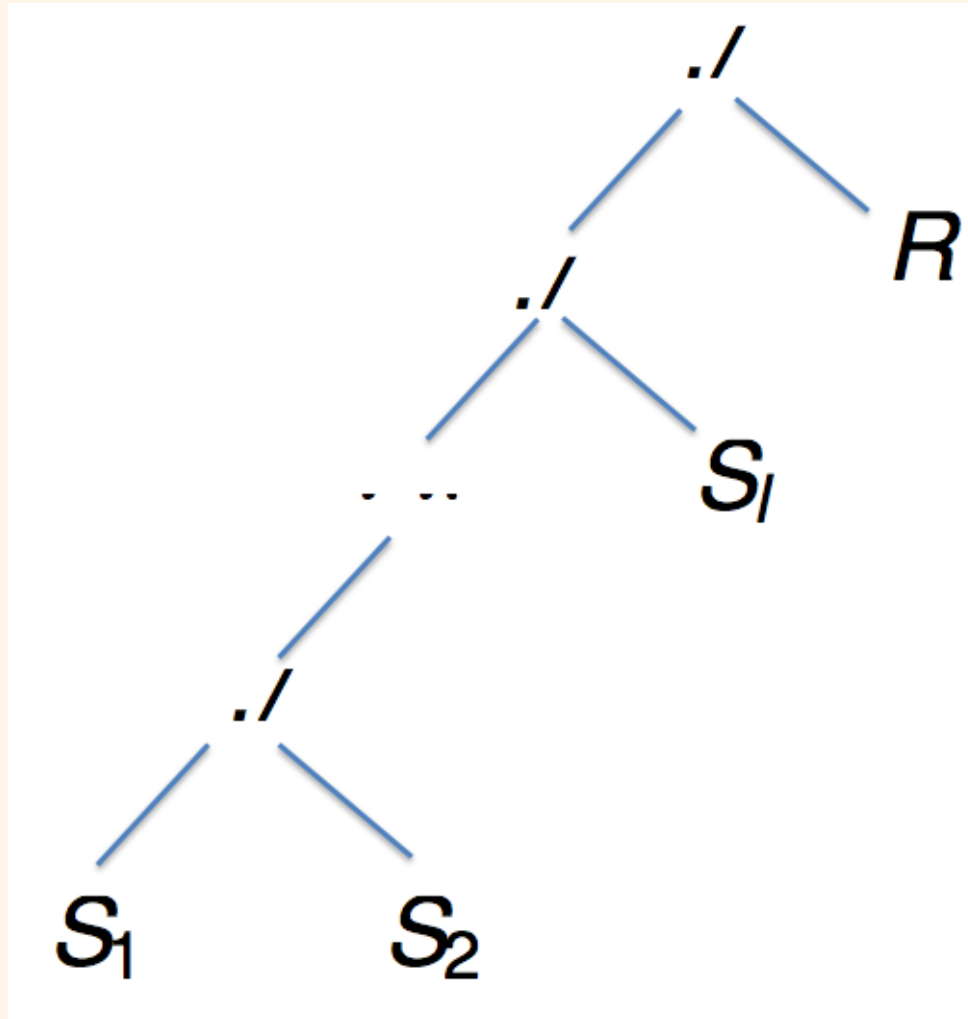
- If a subset has 2 relations, best (lowest cost) plan is the one with smaller relation as outer
- Whether you use BNLJ or SMJ, outer size influences overall cost




Cost for subsets of size ≥ 3

- Cost of join order = sum of sizes of all intermediate relations, excluding final result
- Rationale: every intermediate relation we count is outer for some join

Calculating cost of a plan





Cost of plan

- Cost = size of last intermediate relation + cost of subplan to generate that relation
- (For a plan with 2 relations assume cost = 0)



How to compute relation sizes?

- Ok, so how do we compute intermediate relation sizes?
- Let's start with the join of 2 relations, R and S
- Size could theoretically vary from 0 to the cross product size
- We estimate it using statistics



V-values

- Given a relation R and attribute A , $V(R, A) =$ number of distinct values A takes in R
- You can compute this for each relation and each attribute from your statistics
- Assume uniform distribution
- If you have a selection, include the reduction factor in your calculations

Join size

- Joining R and S on $R.A = S.A$
- Make some assumptions about R.A and S.A:
 - if $V(R,A) \leq V(S,A)$ then every value of R.A appears as a value of S.A
 - if $V(R,A) \geq V(S,A)$ then every value of S.A appears as a value of R.A
- Intuition from primary key/foreign key joins

Computing join size

- Suppose $V(R.A) \leq V(S.A)$
- Every tuple in R has a chance $1/V(S,A)$ of joining with a tuple from S
- So joins with $|S|/V(S,A)$ tuples
- So expected join size is $|R| |S|/V(S,A)$
- If $V(R.A) \geq V(S.A)$ analogous argument shows join has size $|R| |S|/V(R,A)$

Conclusion

- If joining R and S on $R.A = S.A$, join size is

$$\frac{|R||S|}{\max(V(R, A), V(S, A))}$$

- See instructions for discussion of more complex join conditions




Putting it all together

- You need to iterate over all subsets of relations
- Note: relations really means "relation instances" (may have FROM Reserves R1, Reserves R2 -> 2 instances)
- For every subset, compute:
 - best join order
 - cost of this plan
 - size of resulting relation
 - V-values for resulting relation
- The best join order for the (unique) largest subset is your answer



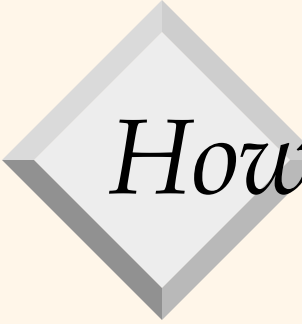
Step 5: choose an implementation for each join

- Make some choice between BNLJ and SMJ
- Don't use a "trivial" policy that always uses BNLJ (or always uses SMJ)
 - Unless of course you have not been able to implement one of these joins in P3
- Find and state some criterion for choosing between the two



Must-have requirements

- Pretty much need to implement everything the way we described it
- If you want to diverge from the instructions, need to get permission (Piazza, office hours, email)
- Will need to explain why your version is better than our suggested approach/algorithm



How we will grade

- Your stats file must match ours
- Your logical plans must match ours
- Your physical plans must be "reasonable" given the data and any special features of your implementation (that you mention in your README)
- Your query answers must still be correct