Lecture 22: Query Optimization (2)

Friday, November 19, 2010

Outline

- Search space
- Algorithms for enumerating query plans
- Estimating the cost of a query plan

Key Decisions

Logical plan

- What logical plans do we consider (left-deep, bushy ?); *Search Space*
- Which algebraic laws do we apply, and in which context(s) ?; *Optimization rules*
- In what order do we explore the search space ?; *Optimization algorithm*

Key Decisions

Physical plan

- What physical operators to use?
- What access paths to use (file scan or index)?

Optimizers

- Heuristic-based optimizers:
  - Apply greedily rules that always improve
    - Typically: push selections down
  - Very limited: no longer used today
- Cost-based optimizers
  - Use a cost model to estimate the cost of each plan
  - Select the “cheapest” plan

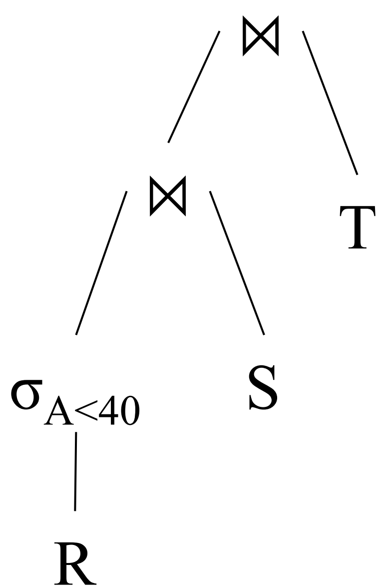
The Search Space

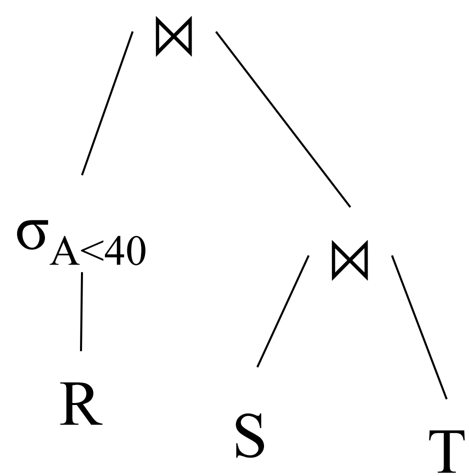
- Complete plans
- Bottom-up plans
- Top-down plans

Complete Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```





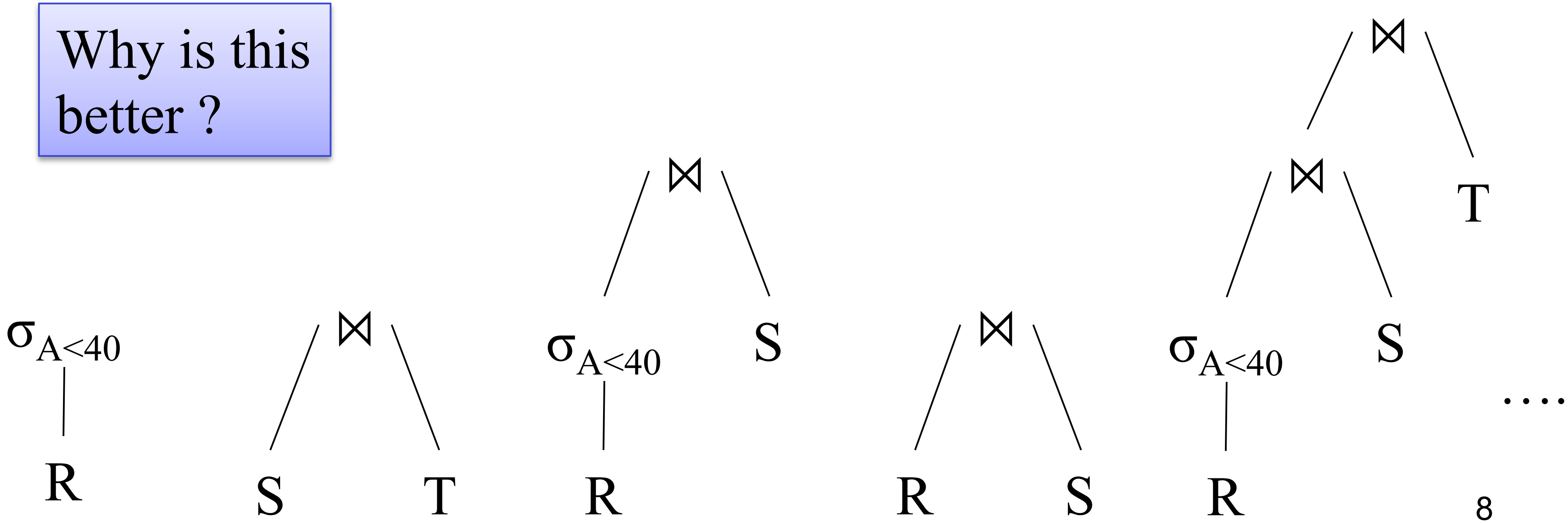
Why is this search space inefficient ?

Bottom-up Partial Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

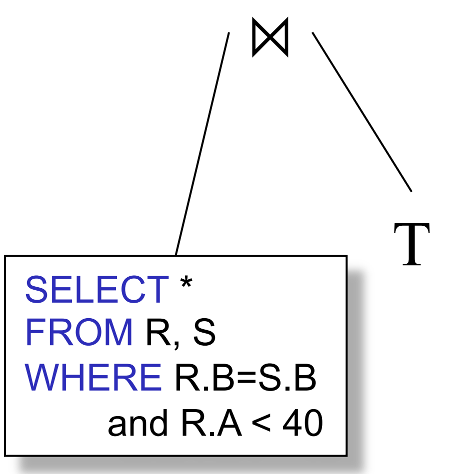
Why is this better ?

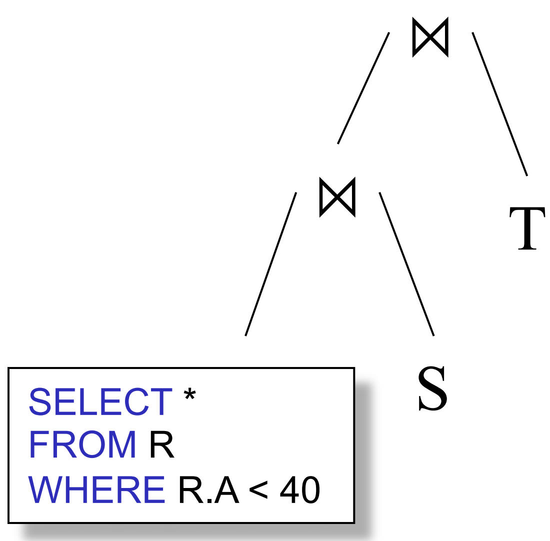


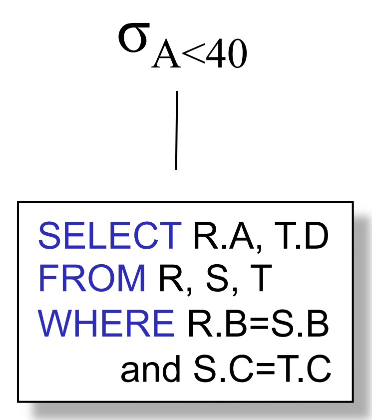
Top-down Partial Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```







.....

Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Classical algorithm [1979]
  - Limited to joins: *join reordering algorithm*
  - Bottom-up
- Rule-based algorithm (will not discuss)
  - Database of rules (=algebraic laws)
  - Usually: dynamic programming
  - Usually: top-down

Dynamic Programming

Originally proposed in System R [1979]

- Only handles single block queries:

```
SELECT list
FROM   R1, …, Rn
WHERE  cond1 AND cond2 AND . . . AND condk
```

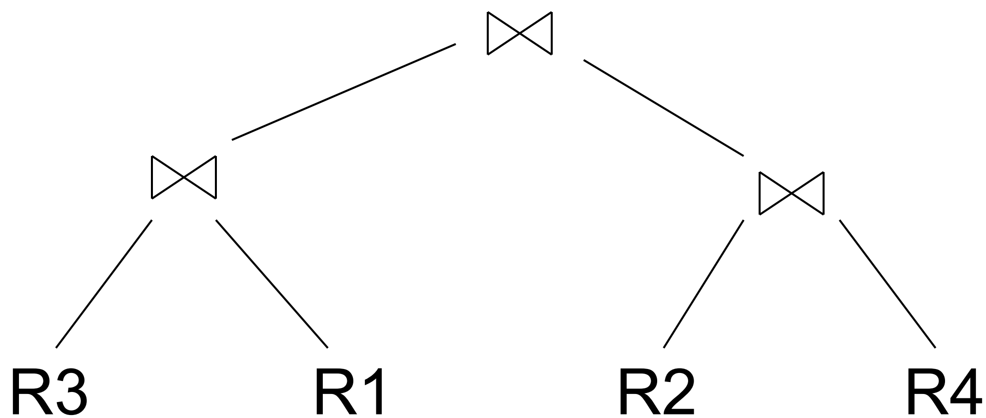
- Heuristics: selections down, projections up

Dynamic Programming

- Search space = join trees
- Algebraic laws = commutativity, associativity
- Algorithm = dynamic programming ☺

Join Trees

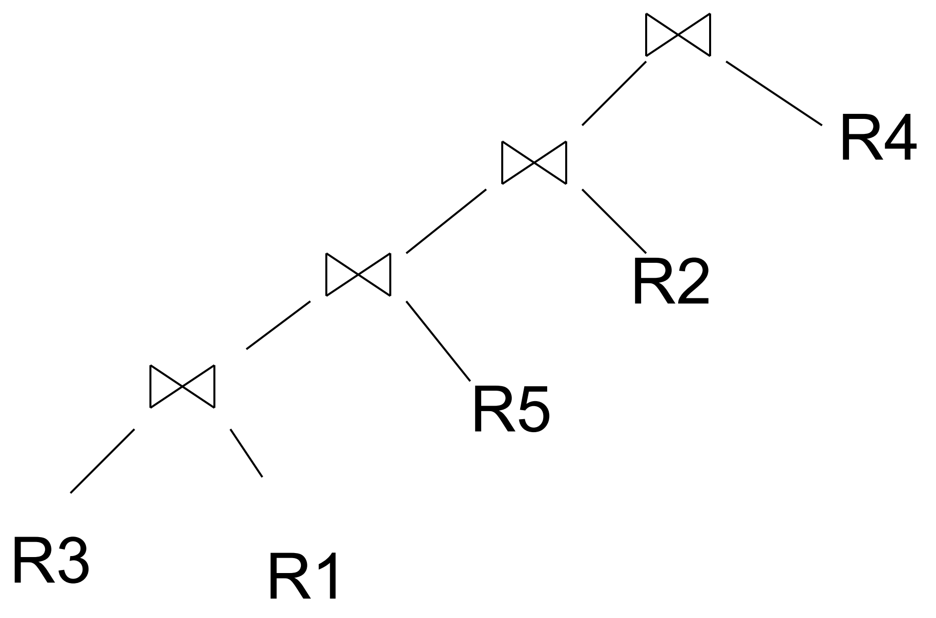
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



- A plan = a join tree
- A partial plan = a subtree of a join tree

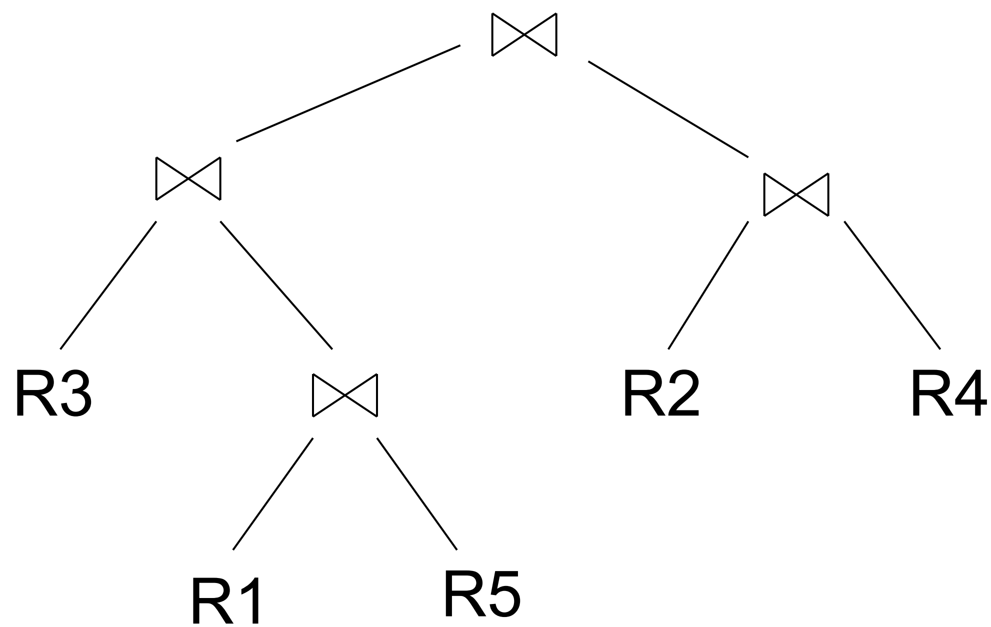
Types of Join Trees

- Left deep:



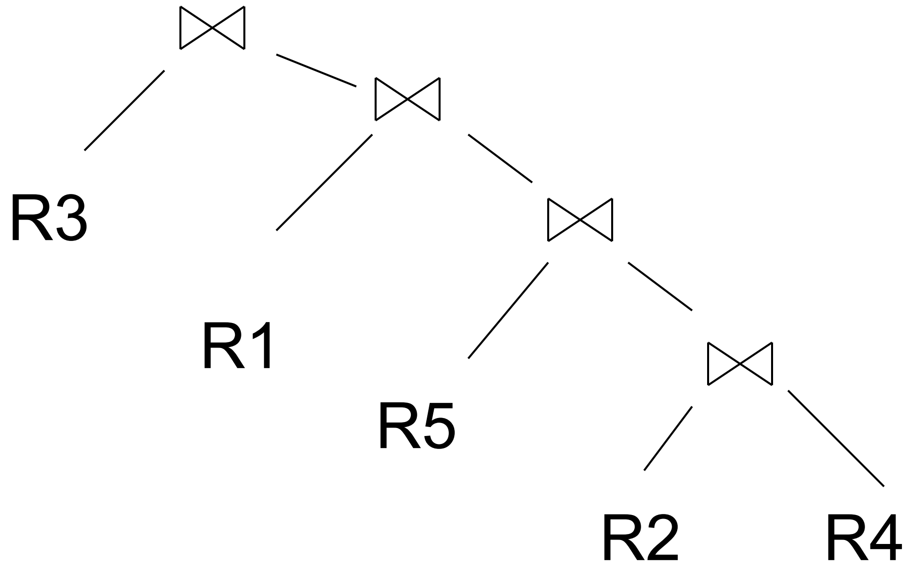
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



```
SELECT list
FROM   R1, …, Rn
WHERE cond1 AND cond2 AND . . . AND condk
```

Dynamic Programming

Join ordering:

- Given: a query R1 ⋈ R2 ⋈ . . . ⋈ Rn
- Find optimal order

- Assume we have a function cost() that gives us the cost of every join tree

```
SELECT list
FROM   R1, …, Rn
WHERE cond1 AND cond2 AND . . . AND condk
```

Dynamic Programming

- For each subquery Q ⊆{R1, …, Rn} compute the following:
  - Size(Q) = the estimated size of Q
  - Plan(Q) = a best plan for Q
  - Cost(Q) = the estimated cost of that plan

```
SELECT list
FROM   R1, …, Rn
WHERE cond_1 AND cond_2 AND . . . AND cond_k
```

Dynamic Programming

- **Step 1**: For each $\{R_i\}$, set:
  - Size($\{R_i\}$) = B(R_i)
  - Plan($\{R_i\}$) = R_i
  - Cost($\{R_i\}$) = (cost of scanning R_i)

SELECT list
FROM R1, …, Rn
WHERE $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

Dynamic Programming

- **Step 2**: For each Q $\subseteq \{R_1, \ldots, R_n\}$ involving *i* relations:
  - Size(Q) = estimate it recursively
  - For every pair of subqueries Q', Q'' s.t. Q = Q' ∪ Q'' compute cost(Plan(Q') ⋈ Plan(Q''))
    - Cost(Q) = the smallest such cost
    - Plan(Q) = the corresponding plan

What's a reasonable estimate?

SELECT list
FROM R1, …, Rn
WHERE $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

Dynamic Programming

- **Step 3**: Return Plan($\{R_1, \ldots, R_n\}$)

Example

To illustrate, ad-hoc cost model (from the book ☺):

- $\text{Cost}(P_1 \bowtie P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) + \text{size}(\text{intermediate results for } P_1, P_2)$
- Cost of a scan = 0

```
SELECT *
FROM   R, S, T, U
WHERE cond1 AND cond2 AND . . .
```

Example

- R ⋈ S ⋈ T ⋈ U
- Assumptions:

All join selectivities = 1%

T(R) = 2000
T(S) = 5000
T(T) = 3000
T(U) = 1000

T(R ⋈ S) = 0.01*T(R)*T(S)
T(S ⋈ T) = 0.01*T(S)*T(T)
etc.

T(R) = 2000
T(S) = 5000
T(T) = 3000
T(U) = 1000

Subquery	Size	Cost	Plan
RS			
RT			
RU			
ST			
SU			
TU			
RST			
RSU			
RTU			
STU			
RSTU			

T(R) = 2000
T(S) = 5000
T(T) = 3000
T(U) = 1000

Subquery	Size	Cost	Plan
RS	100k	0	RS
RT	60k	0	RT
RU	20k	0	RU
ST	150k	0	ST
SU	50k	0	SU
TU	30k	0	TU
RST	3M	60k	(RT)S
RSU	1M	20k	(RU)S
RTU	0.6M	20k	(RU)T
STU	1.5M	30k	(TU)S
RSTU	30M	60k +50k=110k	(RT)(SU)

Reducing the Search Space

- Restriction 1: only left linear trees (no bushy)

Why?

- Restriction 2: no trees with cartesian product

R(A,B) ⋈ S(B,C) ⋈ T(C,D)

Plan: (R(A,B)⋈T(C,D)) ⋈ S(B,C)
has a cartesian product.
Most query optimizers will not consider it

Dynamic Programming: Summary

- Handles only join queries:
  - Selections are pushed down (i.e. early)
  - Projections are pulled up (i.e. late)

- Takes exponential time in general, BUT:
  - Left linear joins may reduce time
  - Non-cartesian products may reduce time further

Rule-Based Optimizers

- ***Extensible*** collection of rules
  - Rule = Algebraic law with a direction
- Algorithm for firing these rules
  - Generate many alternative plans, in some order
  - Prune by cost

- Volcano (later SQL Sever)
- Starburst (later DB2)

Completing the Physical Query Plan

- Choose algorithm for each operator
  - How much memory do we have ?
  - Are the input operand(s) sorted ?
- Access path selection for base tables
- Decide for each intermediate result:
  - To materialize
  - To pipeline

Access Path Selection

- **Access path**: a way to retrieve tuples from a table
  - A file scan
  - An index *plus* a matching selection condition

- Index matches selection condition if it can be used to retrieve just tuples that satisfy the condition
  - Example: Supplier(sid,sname,scity,sstate)
  - B+-tree index on (scity,sstate)
    - matches scity='Seattle'
    - does not match sid=3, does not match sstate='WA'

Access Path Selection

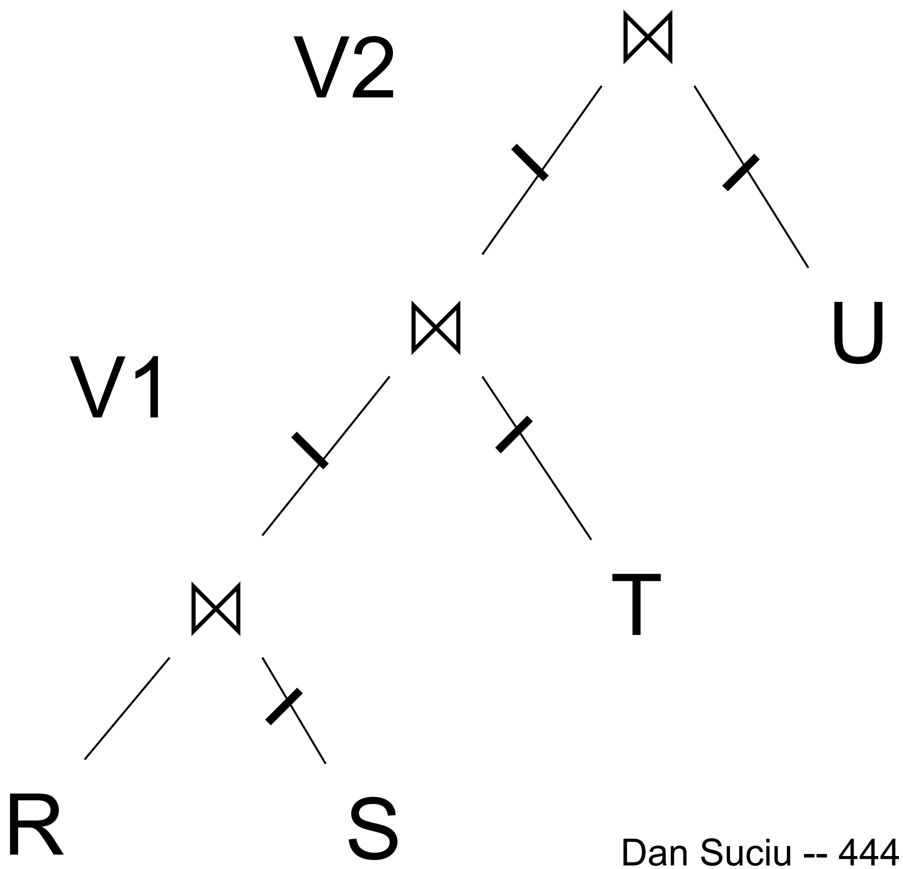
- Supplier(sid,sname,scity,sstate)
- Selection condition: sid > 300 ∧ scity='Seattle'
- Indexes: B+-tree on sid and B+-tree on scity
- Which access path should we use?
- We should pick the **most selective** access path

Access Path Selectivity

- **Access path selectivity is the number of pages retrieved if we use this access path**
  - Most selective retrieves fewest pages

- As we saw earlier, **for equality predicates**
  - Selection on equality: $\sigma_{a=v}(R)$
  - V(R, a) = # of distinct values of attribute a
  - 1/V(R,a) is thus the reduction factor
  - Clustered index on a: cost B(R)/V(R,a)
  - Unclustered index on a: cost T(R)/V(R,a)
  - (we are ignoring I/O cost of index pages for simplicity)

Materialize Intermediate Results Between Operators




```
HashTable ← S
repeat    read(R, x)
          y ← join(HashTable, x)
          write(V1, y)

HashTable ← T
repeat    read(V1, y)
          z ← join(HashTable, y)
          write(V2, z)

HashTable ← U
repeat    read(V2, z)
          u ← join(HashTable, z)
          write(Answer, u)
```

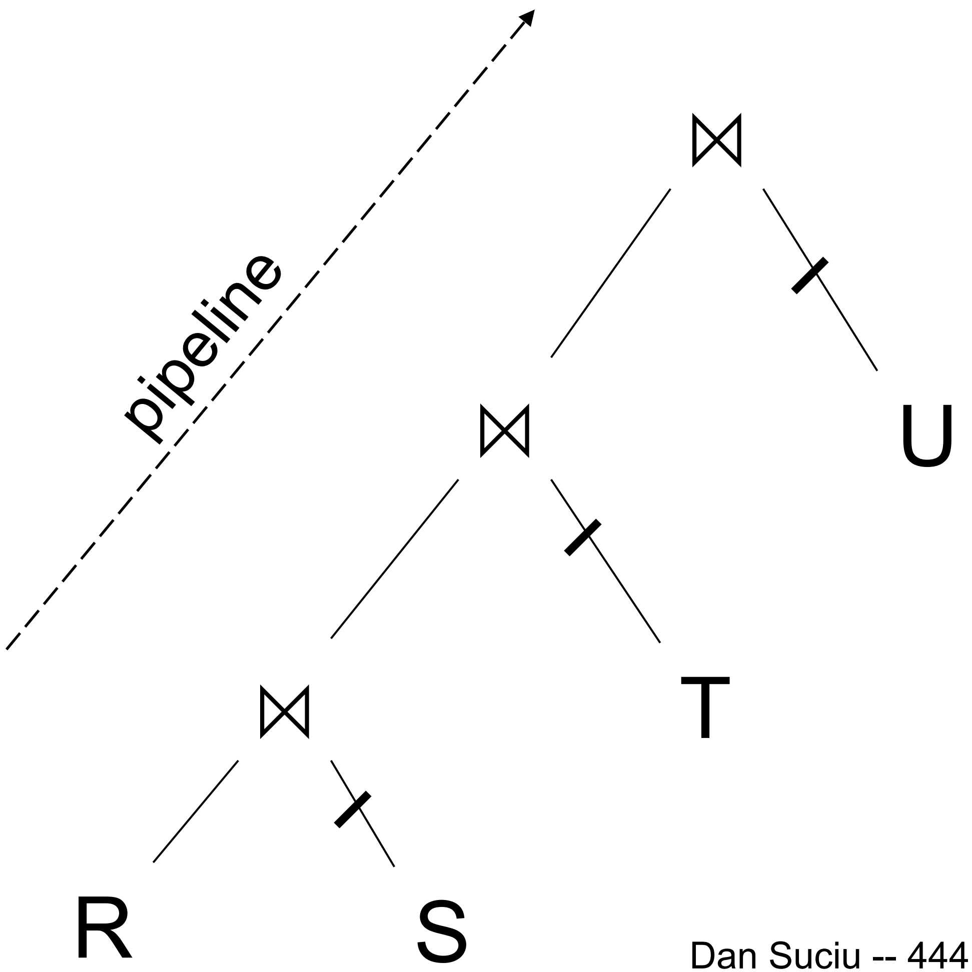
Materialize Intermediate Results Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline Between Operators




```
HashTable1 ← S
HashTable2 ← T
HashTable3 ← U
repeat    read(R, x)
          y ← join(HashTable1, x)
          z ← join(HashTable2, y)
          u ← join(HashTable3, z)
          write(Answer, u)
```

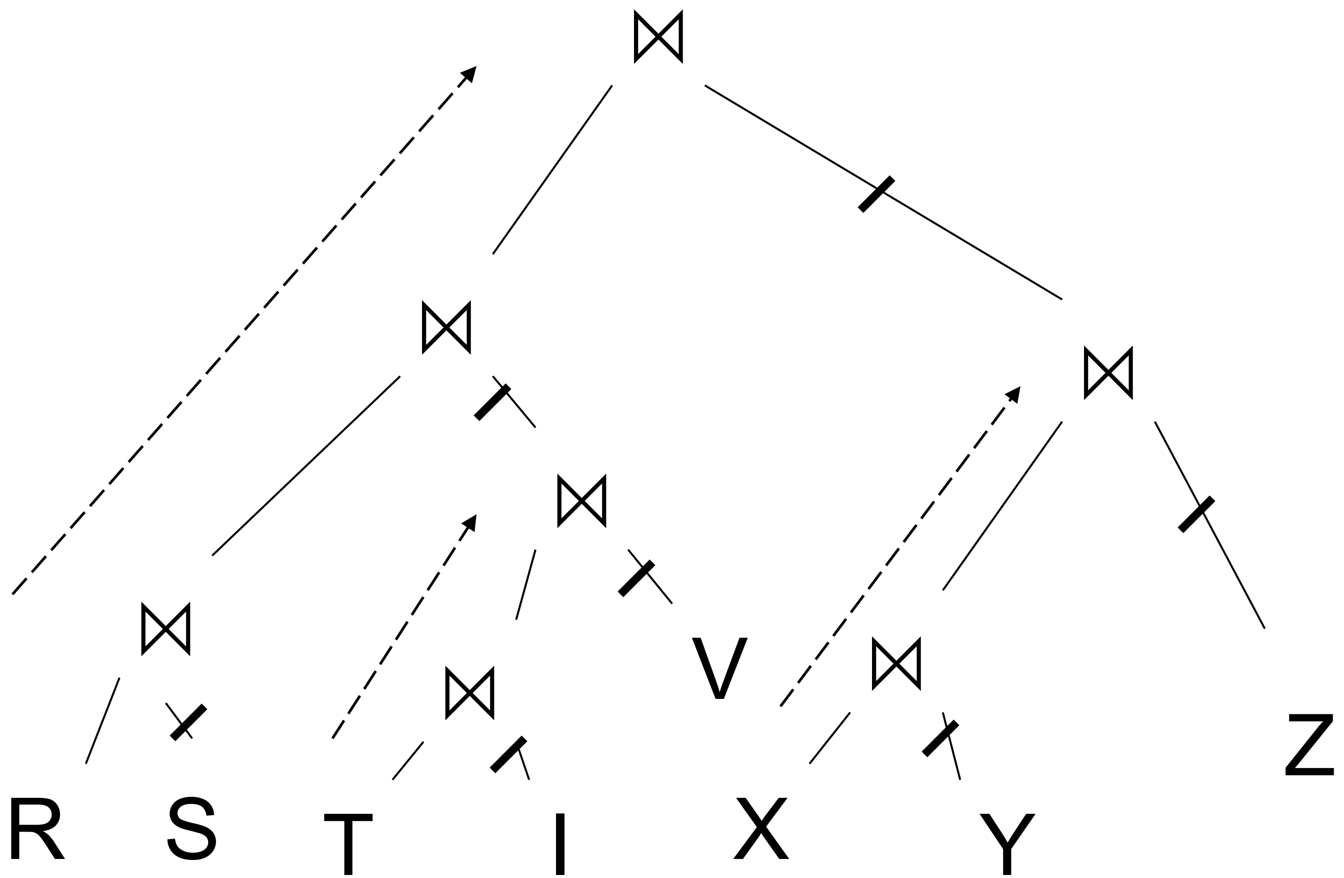
Pipeline Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

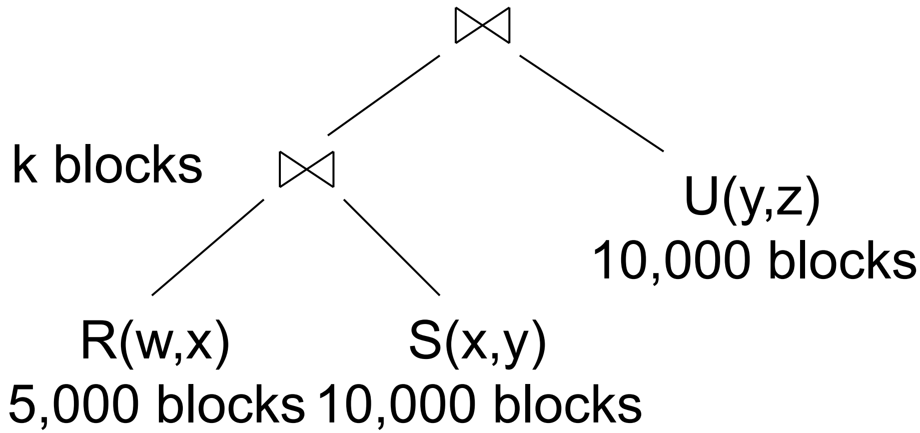
- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline in Bushy Trees



Example

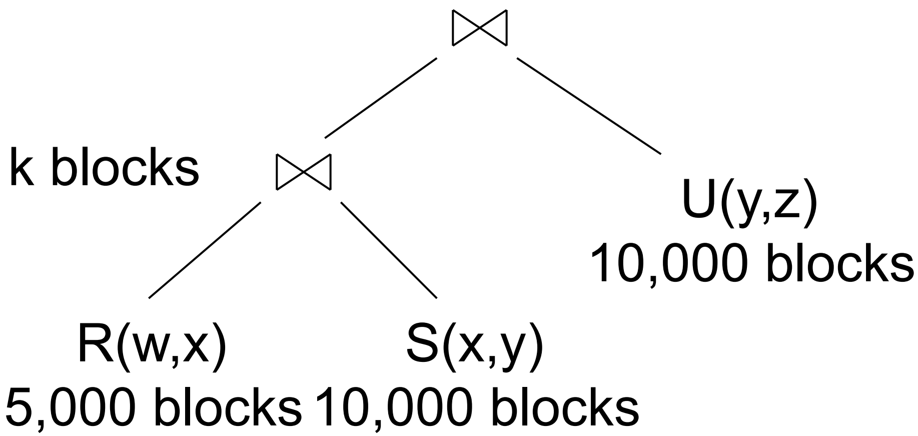
- Logical plan is:



- Main memory M = 101 buffers

Example

M = 101

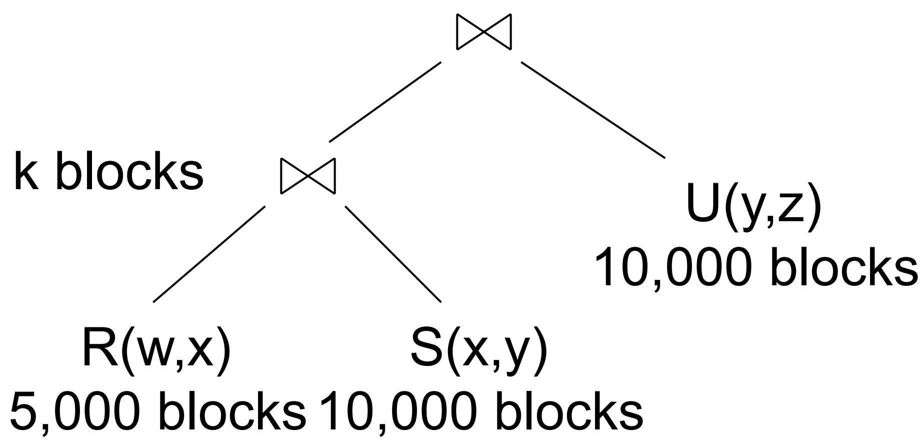


Naïve evaluation:

- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k

Example

M = 101

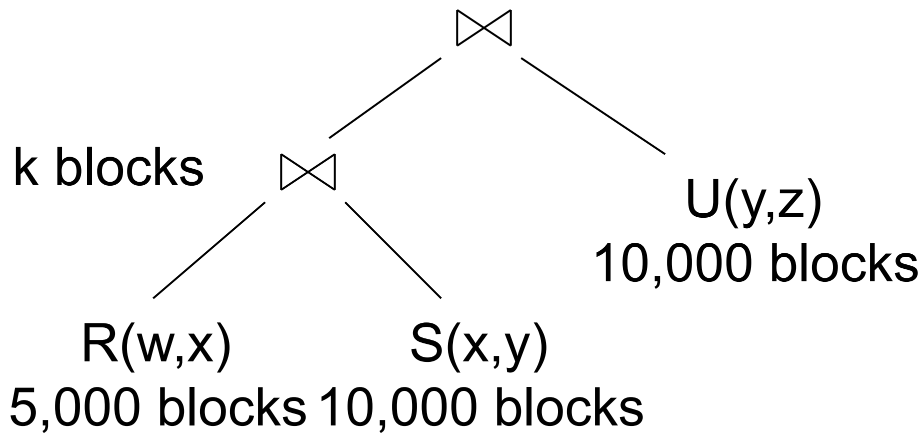


Smarter:

- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each R_i in memory (50 buffer) join with S_i (1 buffer); hash result on y into 50 buckets (50 buffers) -- here we *pipeline*
- Cost so far: 3B(R) + 3B(S)

Example

M = 101

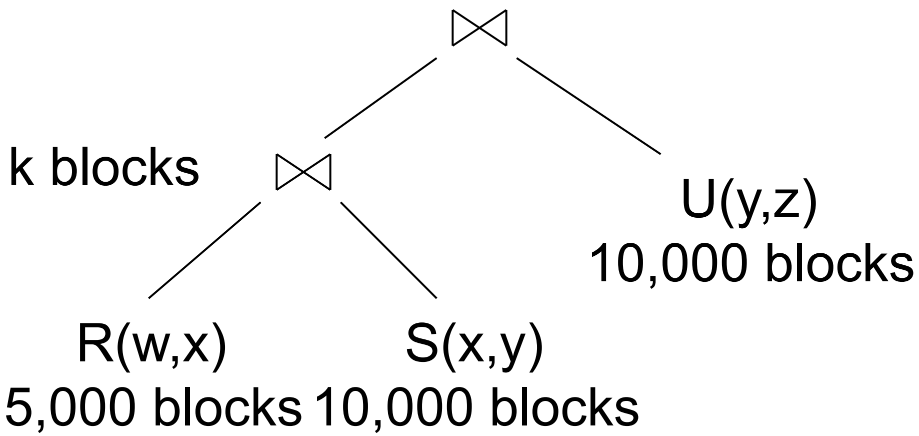


Continuing:

- How large are the 50 buckets on y ? Answer: k/50.
- If k <= 50 then keep all 50 buckets in Step 3 in memory, then:
- Step 4: read U from disk, hash on y and join with memory
- Total cost: 3B(R) + 3B(S) + B(U) = 55,000

Example

M = 101

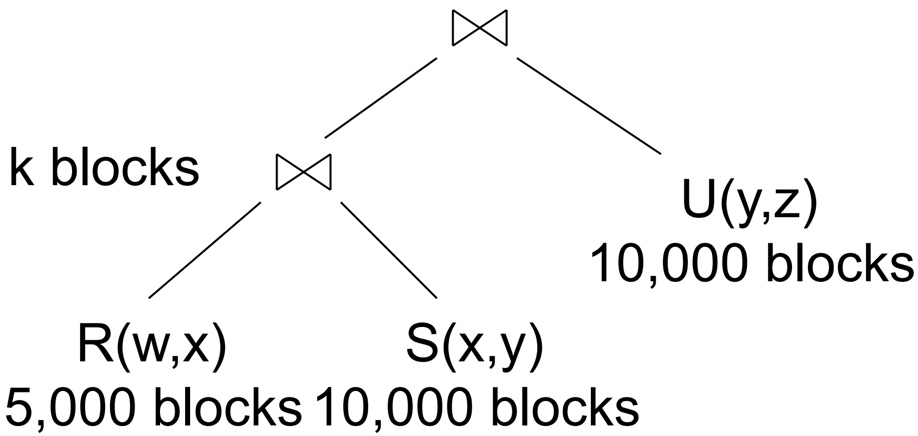


Continuing:

- If 50 < k <= 5000 then send the 50 buckets in Step 3 to disk
  - Each bucket has size k/50 <= 100
- Step 4: partition U into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: 3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k

Example

M = 101



Continuing:

- If k > 5000 then materialize instead of pipeline
- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k