Lecture 26: Query Optimization

Monday, December 4th, 2006

Outline

- Cost-based optimization 16.5, 16.6
- Cost estimation: 16.4

Cost-based Optimizations

- Main idea: apply algebraic laws, until estimated cost is minimal
- Practically: start from partial plans, introduce operators one by one
  - Will see in a few slides
- Problem: there are too many ways to apply the laws, hence too many (partial) plans

Cost-based Optimizations

Approaches:

- **Top-down**: the partial plan is a top fragment of the logical plan

- **Bottom up**: the partial plan is a bottom fragment of the logical plan

Dynamic Programming

Originally proposed in System R

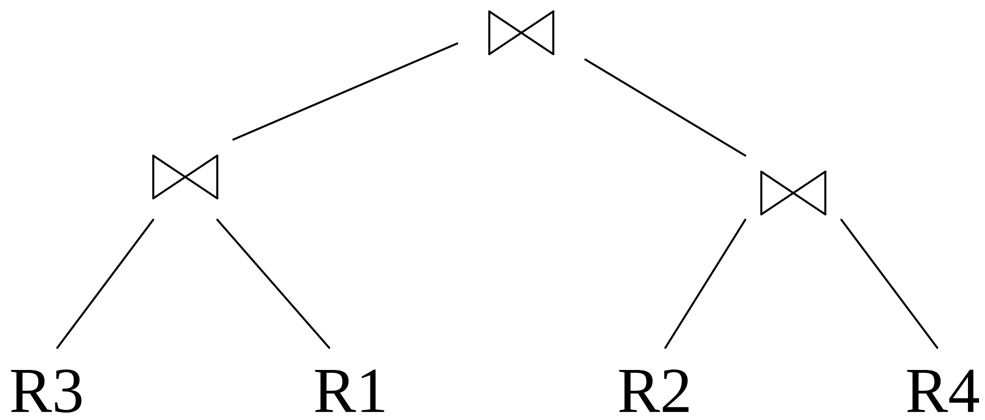
- Only handles single block queries:

```
SELECT list
FROM   list
WHERE cond1 AND cond2 AND . . . AND condk
```

- Heuristics: selections down, projections up
- Dynamic programming: ***join reordering***

Join Trees

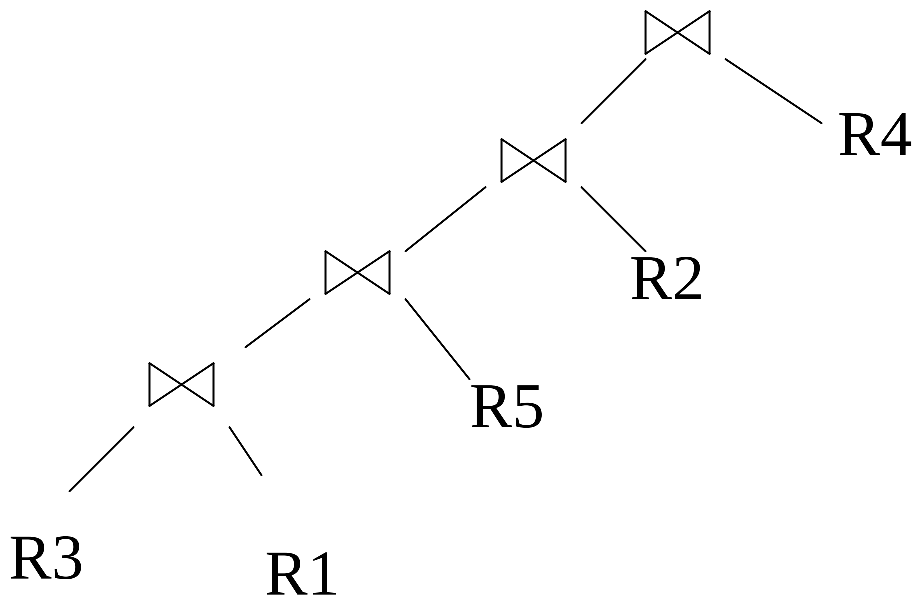
- R1 |×| R2 |×| …. |×| Rn
- Join tree:



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

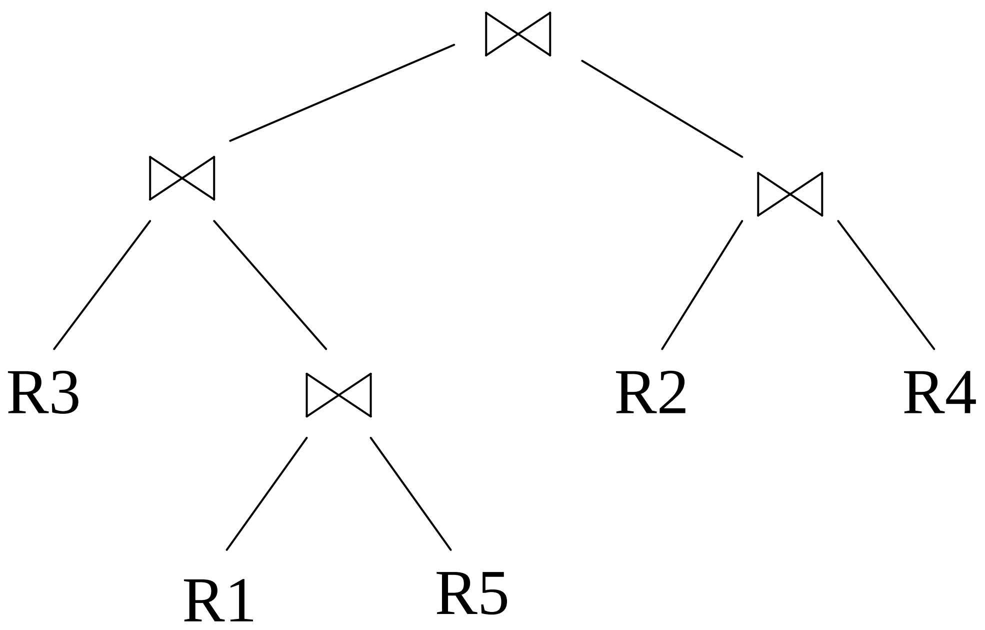
Types of Join Trees

- Left deep:



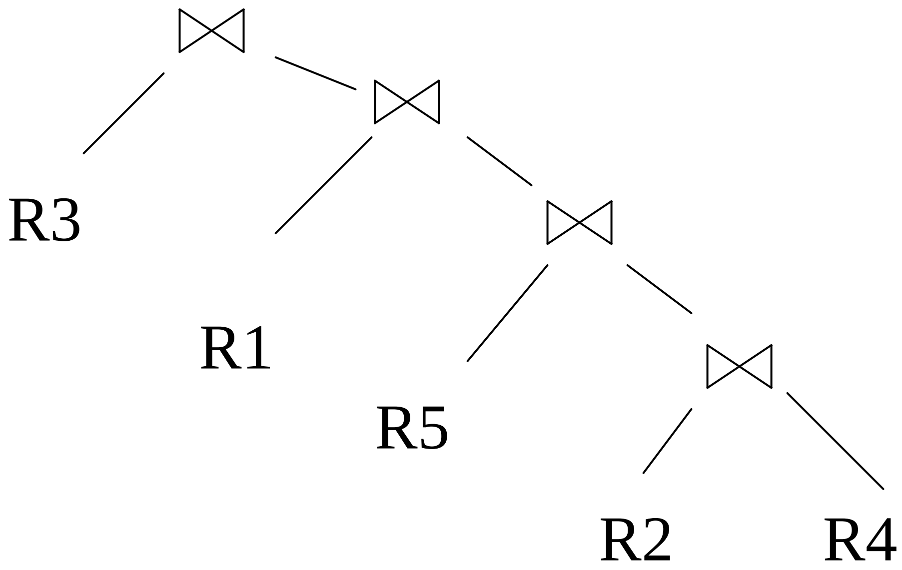
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Dynamic Programming

- Given: a query R1 |×| R2 |×| … |×| Rn
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- It is a bottom-up strategy
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery $Q \subseteq \{R1, \ldots, Rn\}$ compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

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

Dynamic Programming

- **Step i**: For each $Q \subseteq \{R_1, \ldots, R_n\}$ of cardinality i do:
  - Compute Size(Q)   (later…)
  - For every pair of subqueries Q', Q''
    s.t. $Q = Q' \cup Q''$
    compute cost(Plan(Q') $|\times|$ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan($\{R_1, \ldots, R_n\}$)

Dynamic Programming

To illustrate, we will make the following simplifications:

- $\text{Cost}(P_1 \mid\times\mid P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) +$ size(intermediate result(s))
- Intermediate results:
  - If P_1 = a join, then the size of the intermediate result is size(P_1), otherwise the size is 0
  - Similarly for P_2
- Cost of a scan = 0

Dynamic Programming

- Example:
- $\text{Cost}(R5 \ltimes\!\rtimes R7) = 0$ (no intermediate results)
- $\text{Cost}((R2 \ltimes\!\rtimes R1) \ltimes\!\rtimes R7)$

  $= \text{Cost}(R2 \ltimes\!\rtimes R1) + \text{Cost}(R7) + \text{size}(R2 \ltimes\!\rtimes R1)$

  $= \text{size}(R2 \ltimes\!\rtimes R1)$

Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: $T(A \mid\times\mid B) = 0.01*T(A)*T(B)$

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

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

- Left-linear trees v.s. Bushy trees

- Trees without cartesian product

Example: 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

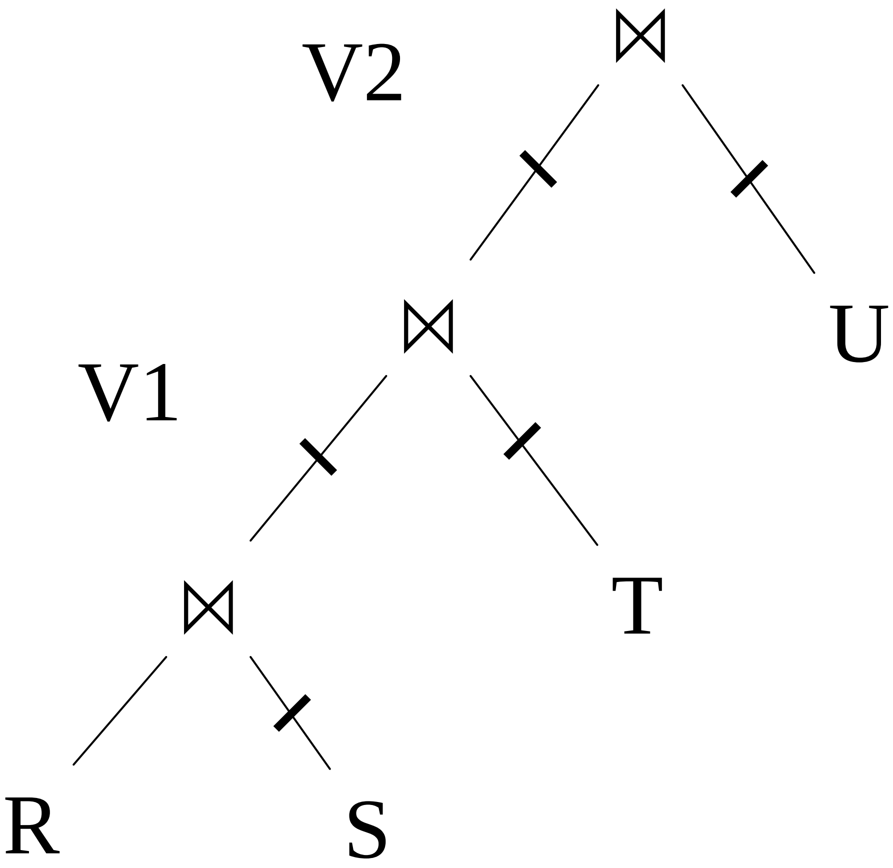
- ***Extensible*** collection of rules
  Rule = Algebraic law with a direction
- Algorithm for firing these rules
  Generate many alternative plans, in some order
  Prune by cost

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

Completing the Physical Query Plan

- Choose algorithm to implement each operator
  - Need to account for more than cost:
    - How much memory do we have ?
    - Are the input operand(s) sorted ?
- Decide for each intermediate result:
  - To materialize
  - To pipeline

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)
```

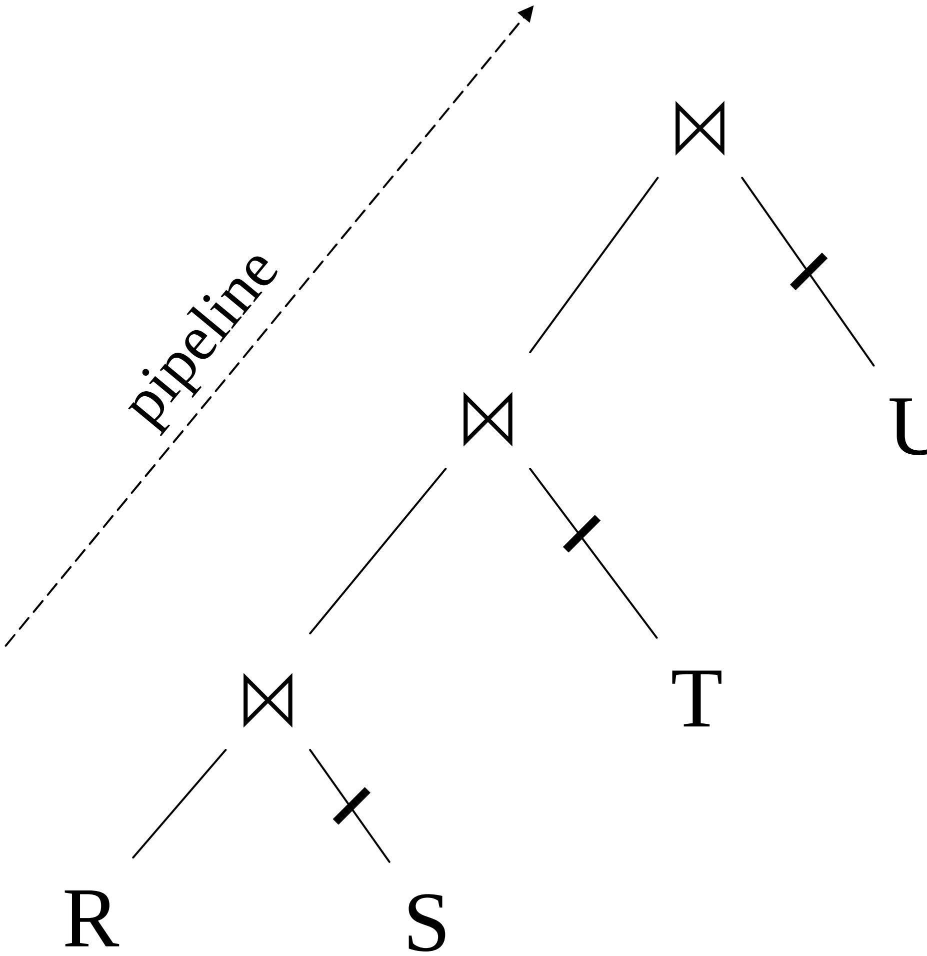
Materialize Intermediate Results Between Operators

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)
```

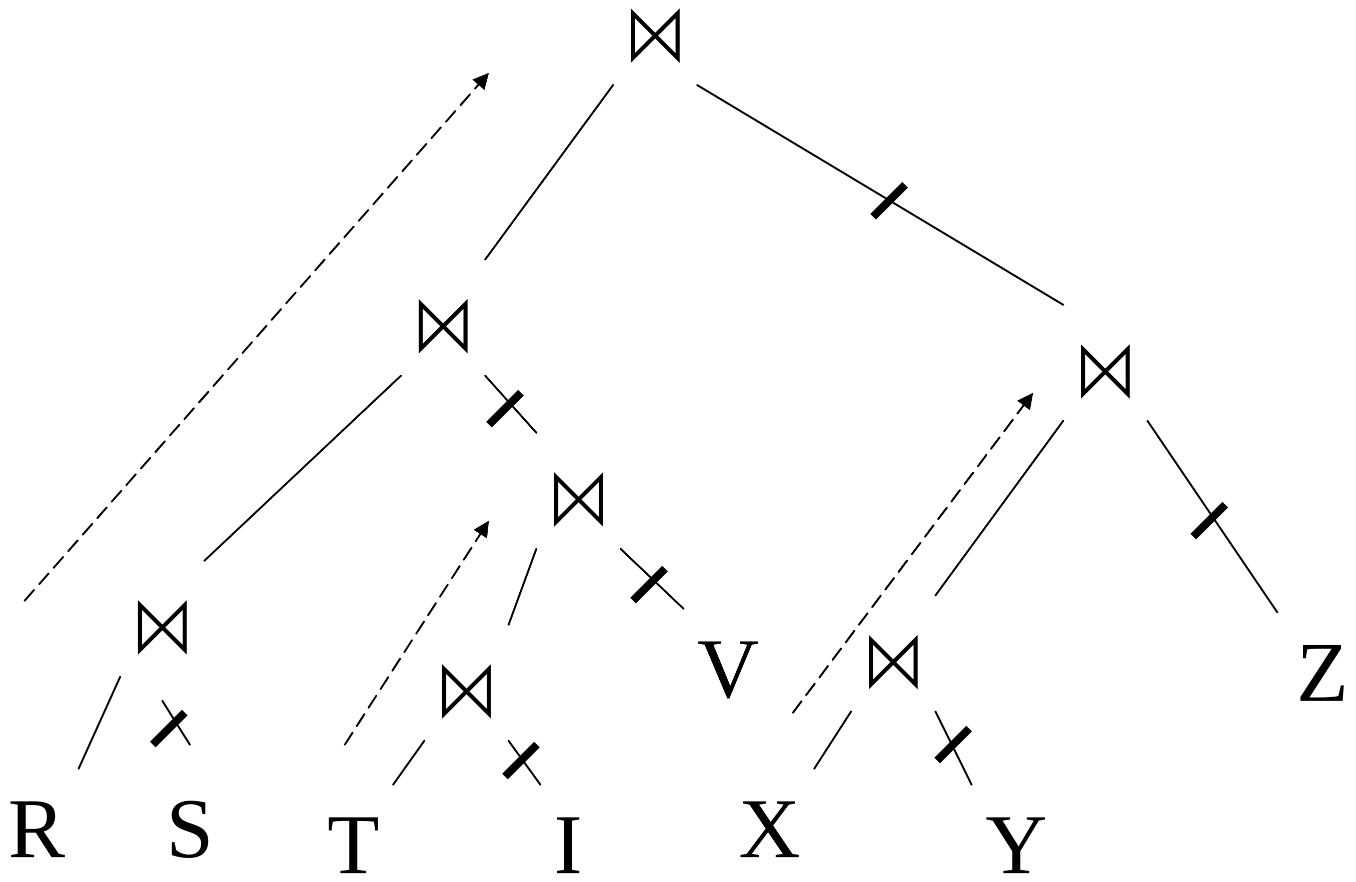
Pipeline Between Operators

Question in class

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

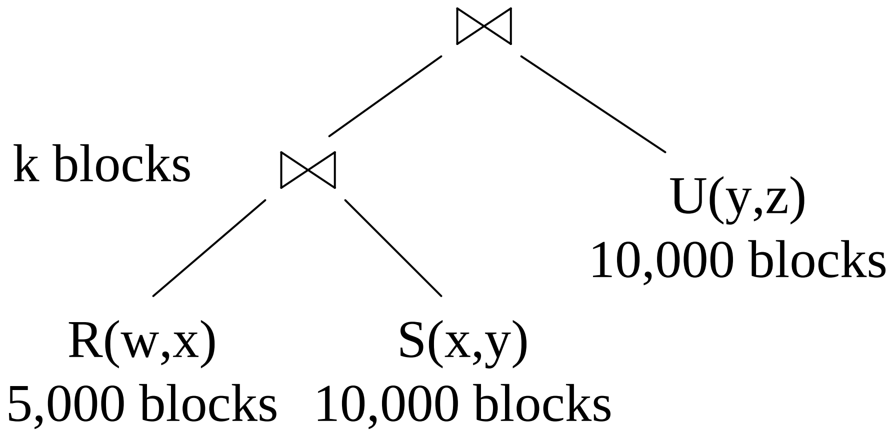
- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline in Bushy Trees



Example

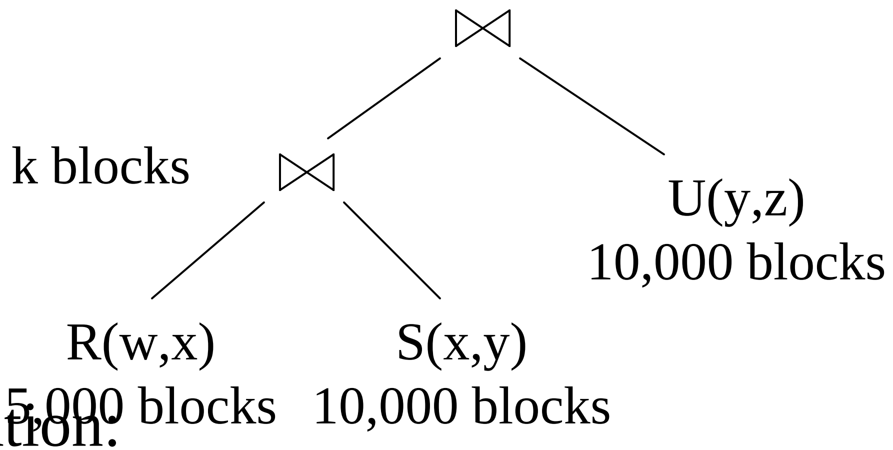
- Logical plan is:



- Main memory M = 101 buffers

Example

M = 101

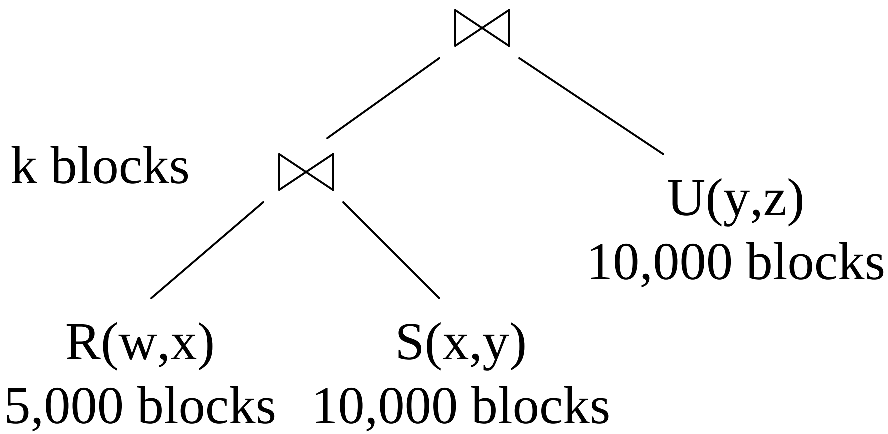


Naïve evaluation:

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

Example

M = 101

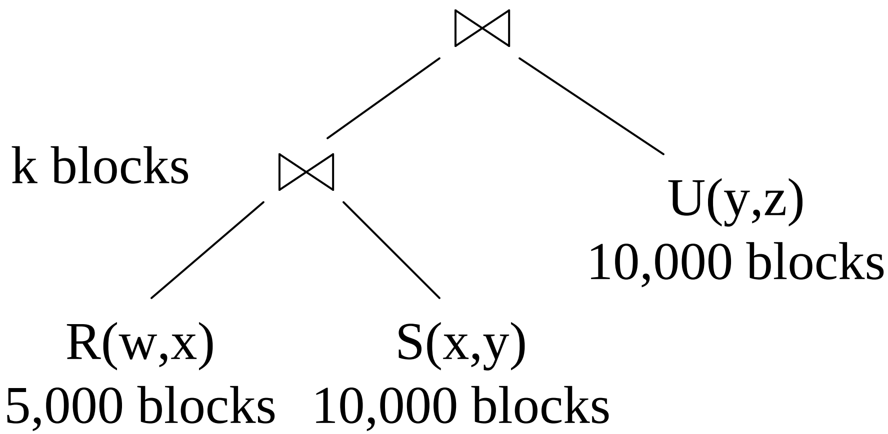


Smarter:

- 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

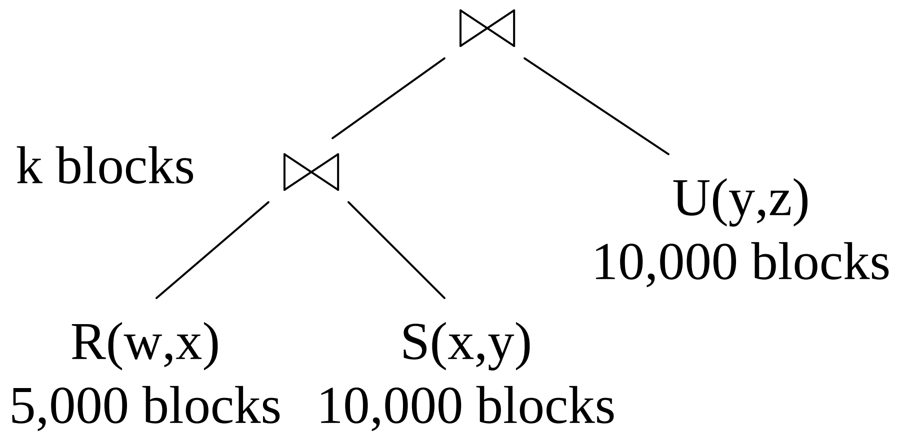


Continuing:

- 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

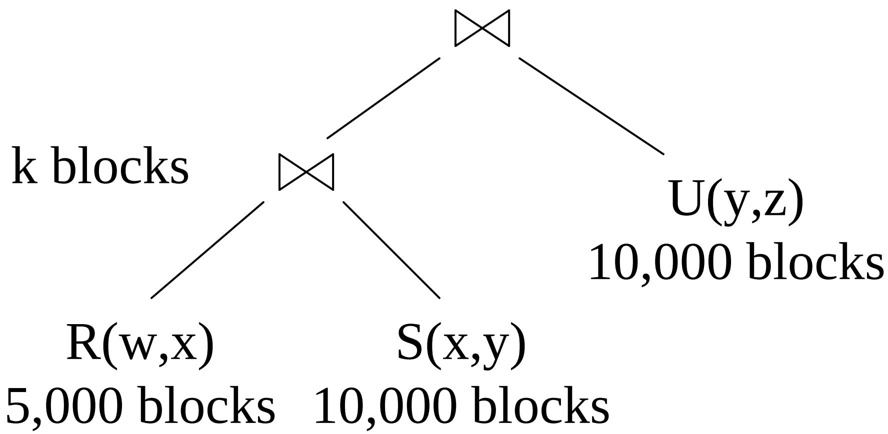


Continuing:

- 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

Example

Summary:

- If $k <= 50$, cost = 55,000
- If $50 < k <= 5000$, cost = 75,000 + 2k
- If $k > 5000$, cost = 75,000 + 4k

Size Estimation

The problem: Given an expression E, compute T(E) and V(E, A)

- This is hard without computing E
- Will 'estimate' them instead

Size Estimation

Estimating the size of a projection

- Easy: $T(\Pi_L(R)) = T(R)$
- This is because a projection doesn't eliminate duplicates

Size Estimation

Estimating the size of a selection

- $S = \sigma_{A=c}(R)$
  - T(S) san be anything from 0 to T(R) – V(R,A) + 1
  - Estimate: T(S) = T(R)/V(R,A)
  - When V(R,A) is not available, estimate T(S) = T(R)/10

- $S = \sigma_{A<c}(R)$
  - T(S) can be anything from 0 to T(R)
  - Estimate: T(S) = (c - Low(R, A))/(High(R,A) - Low(R,A))T(R)
  - When Low, High unavailable, estimate T(S) = T(R)/3

Size Estimation

Estimating the size of a natural join, $R \mid\times\mid_A S$

- When the set of A values are disjoint, then $T(R \mid\times\mid_A S) = 0$
- When A is a key in S and a foreign key in R, then $T(R \mid\times\mid_A S) = T(R)$
- When A has a unique value, the same in R and S, then $T(R \mid\times\mid_A S) = T(R)\ T(S)$

Size Estimation

Assumptions:

- *Containment of values*: if V(R,A) <= V(S,A), then the set of A values of R is included in the set of A values of S
  - Note: this indeed holds when A is a foreign key in R, and a key in S
- *Preservation of values*: for any other attribute B, $V(R \mid\times\mid_A S, B) = V(R, B)$ (or $V(S, B)$)

Size Estimation

Assume V(R,A) <= V(S,A)

- Then each tuple t in R joins *some* tuple(s) in S
  - How many ?
  - On average T(S)/V(S,A)
  - t will contribute T(S)/V(S,A) tuples in $R \mid\times\mid_A S$
- Hence $T(R \mid\times\mid_A S) = T(R)\ T(S)\ /\ V(S,A)$

In general: $T(R \mid\times\mid_A S) = T(R)\ T(S)\ /\ \max(V(R,A),V(S,A))$

Size Estimation

Example:

- $T(R) = 10000$, $T(S) = 20000$
- $V(R,A) = 100$, $V(S,A) = 200$
- How large is $R \mid\times\mid_A S$?

Answer: $T(R \mid\times\mid_A S) = 10000\ 20000/200 = 1M$

Size Estimation

Joins on more than one attribute:

- $T(R \mid\times\mid_{A,B} S) =$

  $T(R)\ T(S)/(\max(V(R,A),V(S,A))*\max(V(R,B),V(S,B)))$

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(<u>ssn</u>, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $|\times|_{\text{Salary}}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Eqwidth

0..20	20..40	40..60	60..80	80..100
2	104	9739	152	3

- Eqdepth

0..44	44..48	48..50	50..56	55..100
2000	2000	2000	2000	2000