CS411
Database Systems

09: Query Optimization

Optimization

- At the heart of the database engine
- Step 1: convert the SQL query to a logical plan
- Step 2: find a better logical plan, find an associated physical plan
- (Feed the physical plan into the query processor.)

SQL –> Logical Query Plans

Converting from SQL to Logical Plans

Select a1, …, an
From R1, …, Rk
Where C

$\Pi_{a1,\ldots,an}(\sigma_C(R1 \bowtie R2 \bowtie \ldots \bowtie Rk))$

Select a1, …, an, aggs
From R1, …, Rk
Where C
Group by b1, …, bl

$\Pi_{a1,\ldots,an}(\gamma_{b1, \ldots, bm, aggs}(\sigma_C(R1 \bowtie R2 \bowtie \ldots \bowtie Rk)))$

- 16.3.1

Removing Subqueries from conditions

```
Select distinct product.name
From product
Where product.maker in (Select company.name
                        From company
                        where company.city="Urbana")
```

```
Select distinct product.name
From product, company
Where product.maker = company.name AND
      company.city="Urbana"
```

- 16.3.2

Optimization: Logical Query Plan

- Now we have one logical plan. Let's try to make it better.

- Algebraic laws: what are the ways in which an expression or tree may be rewritten without changing the meaning

- Optimizations: if there are multiple ways to write the same query, which one to choose.
  - Rule-based (heuristics): apply laws that *seem* to result in cheaper plans
  - Cost-based: estimate size and cost of intermediate results, search systematically for best plan

The three components of an optimizer

- Algebraic laws
- An optimization algorithm
- A cost estimator

Algebraic Laws

- 16.2

Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$, $R \cap (S \cap T) = (R \cap S) \cap T$
  - $R \bowtie S = S \bowtie R$, $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

- Distributive Laws
  - $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Algebraic Laws

- Laws involving selection:
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
  - $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
    - When C involves only attributes of R
    - What if it involves attributes of R and S?
  - $\sigma_C(R - S) = \sigma_C(R) - S$
  - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
  - $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}(R \bowtie_{D=E} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$?

Algebraic Laws

- Laws involving projections
  - $\Pi_M(\Pi_N(R)) = \Pi_{M \cap N}(R)$
  - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
    - Where N, P, Q are appropriate subsets of attributes of M

- Example R(A,B,C,D), S(E, F, G)
  - $\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_?(R) \bowtie \Pi_?(S))$

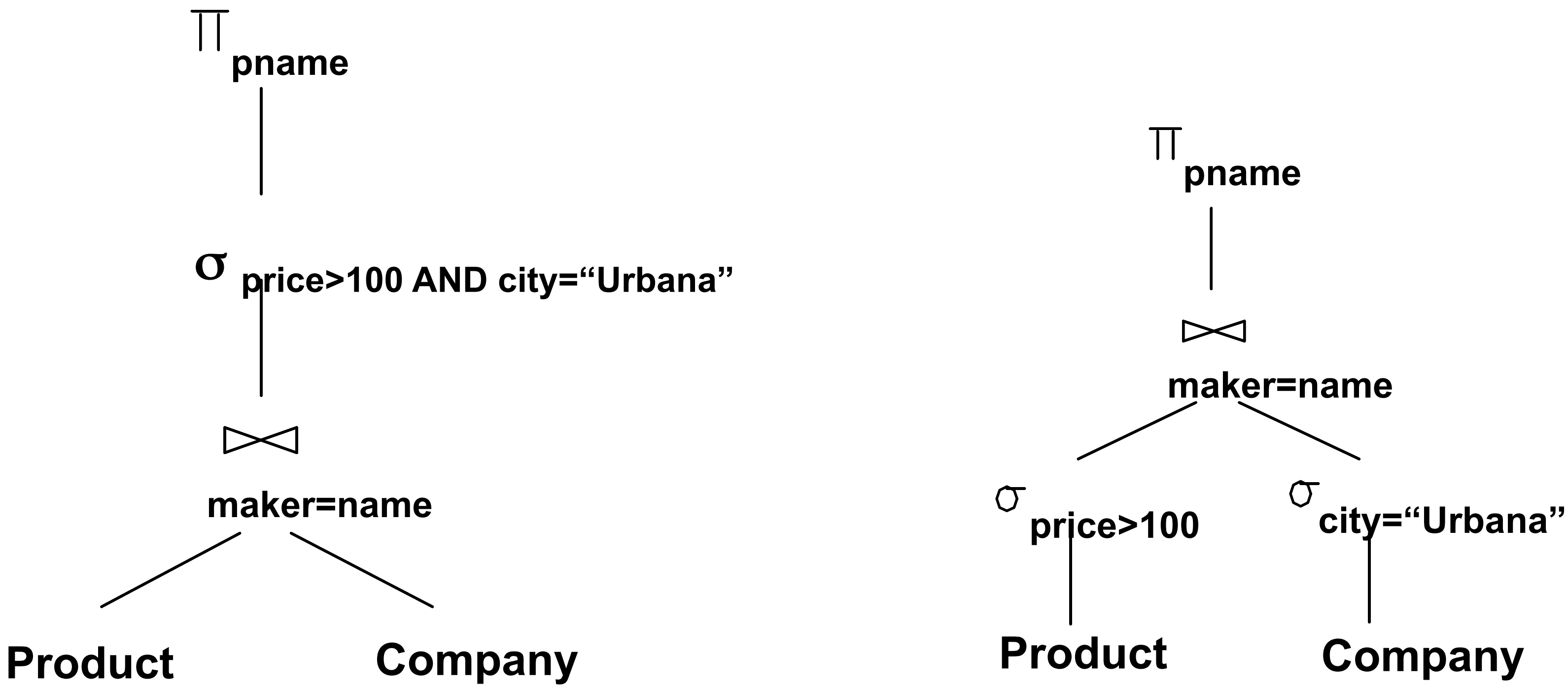
Rule-based Optimization

- 16.2.3

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Heuristic number 1:
  - Push selections down
- Heuristic number 2:
  - (In some cases) push selections up, then down

Pushing selection down



The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to lose an important ordering of the tuples, if we use indexes).

Pushing selection down

```
Select   y.name, Max(x.price)
From     product x, company y
Where    x.maker = y.name
GroupBy  y.name
Having   Max(x.price) > 100
```

⇨

```
Select  y.name, Max(x.price)
From    product x, company y
Where   x.maker=y.name  and
        x.price > 100
GroupBy  y.name
Having  Max(x.price) > 100
```

- *For each company, find the maximal price of its products.*
- *But only display (company, maxprice) pair if maxprice > 100*
- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- Won't work if we replace Max by Min.

Pushing selection up?

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 5
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
```

Pushing selection up

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 5
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 5
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
```

… and then down

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 5
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 5
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
  AND x.price < 5
```

Cost-based Optimization

- 16.5

Cost-based Optimizations

- Main idea: apply algebraic laws, until estimated cost is minimal
- Start from partial plans, build more complete plans
  - Will see in a few slides
- Problem: there are too many ways to apply the laws, hence too many (partial) plans

- 16.5.4

Too many plans: e.g., Join trees

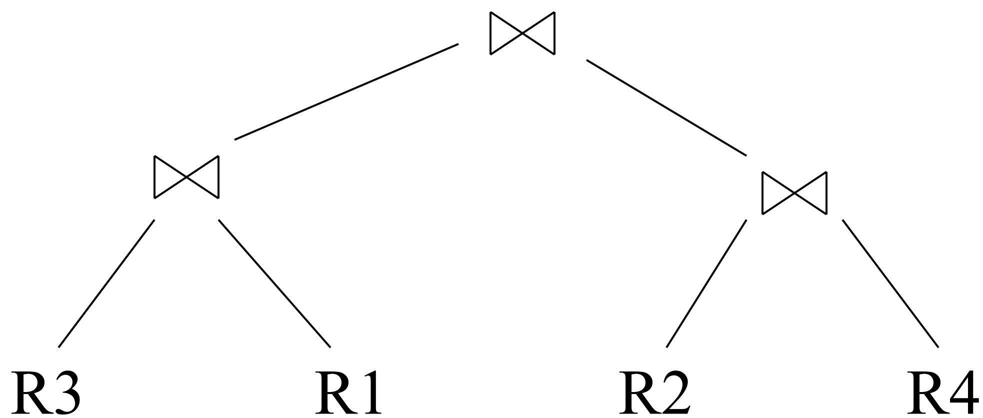
Search Strategies

- **Branch-and-bound**:
  - Remember the cheapest complete plan P seen so far, and its cost C
  - Stop generating partial plans whose cost is > C
  - If a cheaper complete plan is found, replace P, C

- **Hill climbing**:
  - Remember the cheapest partial plan seen so far
  - Make a small change to that plan, see if it is cheaper than the current one; if so, "move" to the new plan, else try again

Join Trees

- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:

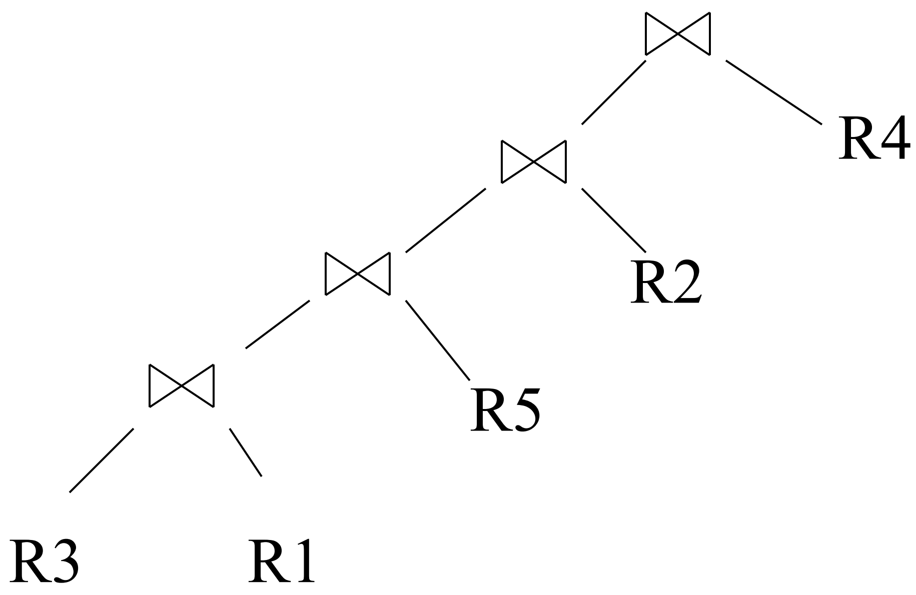


- A plan = a join tree.
- Lots of these !
- A partial plan = a subtree of a join tree

• 16.6

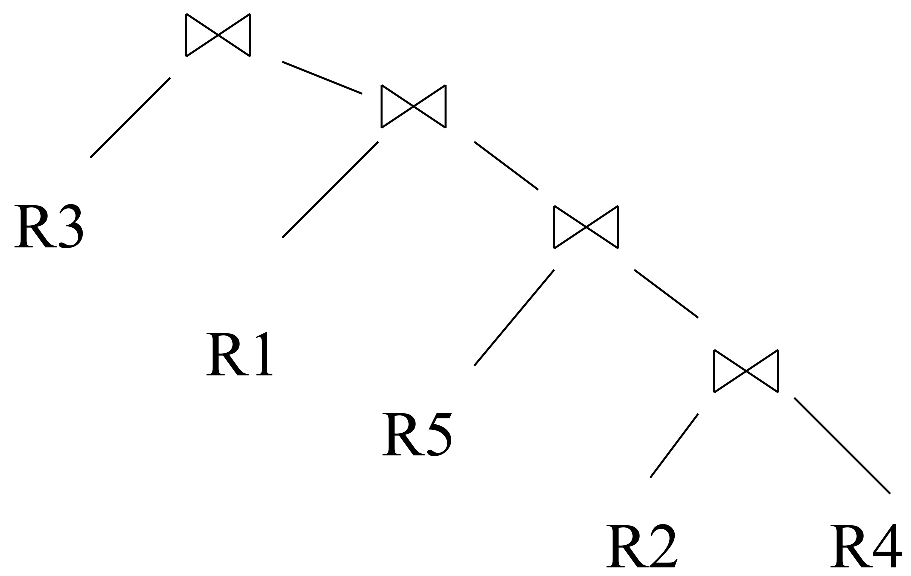
Types of Join Trees

- Left deep:



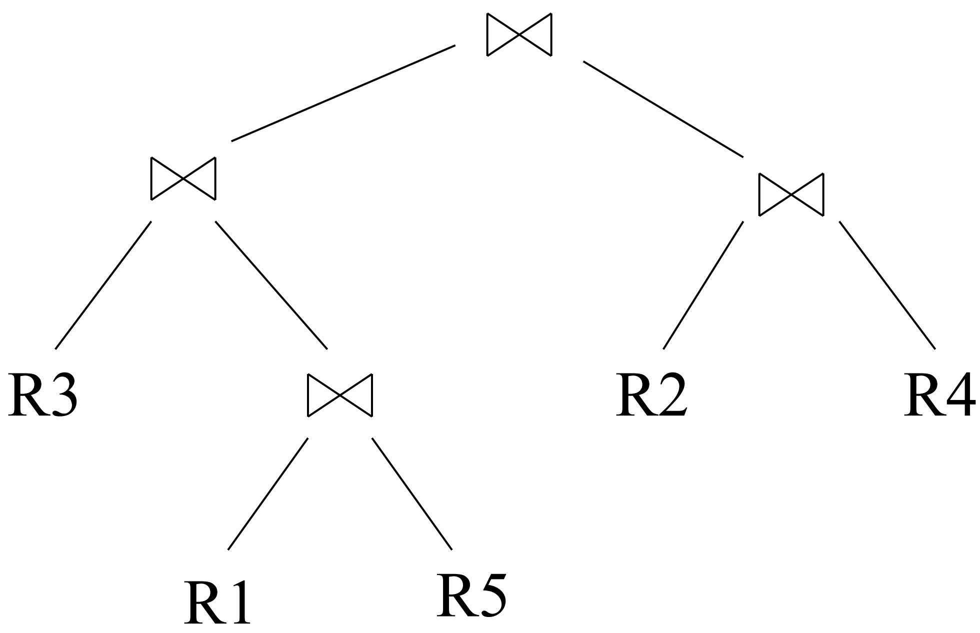
Types of Join Trees

- Right deep:



Types of Join Trees

- Bushy:



Problem

- Given: a query $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
- 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

Dynamic Programming

- For each subset $Q \subseteq \{R1, \ldots, Rn\}$ compute the following:
  - Size(Q): estimated size of the join of the relations in Q
  - A best plan for Q: Plan(Q), a particular join tree.
  - The cost of that plan: Cost(Q). The cost is going to be the size of intermediate tables used by the plan.

Dynamic Programming

- **Step 1**: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = 0.

- **Step 2**: For each {Ri, Rj} do:
  - Size({Ri,Rj}) = estimate of size of join (we'll see later)
  - Plan({Ri,Rj} = either Ri $\bowtie$ Rj, or Rj $\bowtie$ Ri
  - Cost = 0. (no intermediate tables used)

Dynamic Programming

- **Step i**: For each Q $\subseteq$ {R1, …, Rn} of cardinality i do:
  - Compute Size(Q)   (later…)
  - For every pair of subqueries Q’, Q’’
    s.t. Q = Q’ $\cup$ Q’’
    compute cost(Q’) + cost(Q’’) + size(Q’) + size(Q’’)
  - Cost(Q) = the smallest such sum
  - Plan(Q) = the corresponding plan

- **In the end,** Return Plan({R1, …, Rn})

Dynamic Programming

- Example:
- Cost(R5 $\bowtie$ R7) = 0 (no intermediate results)
- Cost((R2 $\bowtie$ R1) $\bowtie$ R7)

  = Cost(R2 $\bowtie$ R1) + Cost(R7) + size(R2 $\bowtie$ R1)

  = size(R2 $\bowtie$ R1)

Dynamic Programming

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

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	UR
ST	150k	0	TS
SU	50k	0	US
TU	30k	0	UT
RST	3M	60k	S(RT)
RSU	1M	20k	S(UR)
RTU	0.6M	20k	T(UR)
STU	1.5M	30k	S(UT)
RSTU	30M	110k	(US)(RT)

Dynamic Programming

- Summary: computes optimal plans for subqueries:
  - Step 1: {R1}, {R2}, …, {Rn}
  - Step 2: {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
  - …
  - Step n: {R1, …, Rn}
- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next time)
  - heuristics for Reducing the Search Space
    - Restrict to left linear trees
    - Restrict to trees "without cartesian product": R(A,B), S(B,C), T(C,D) (R join T) join S has a cartesian product

Completing Physical Query Plan

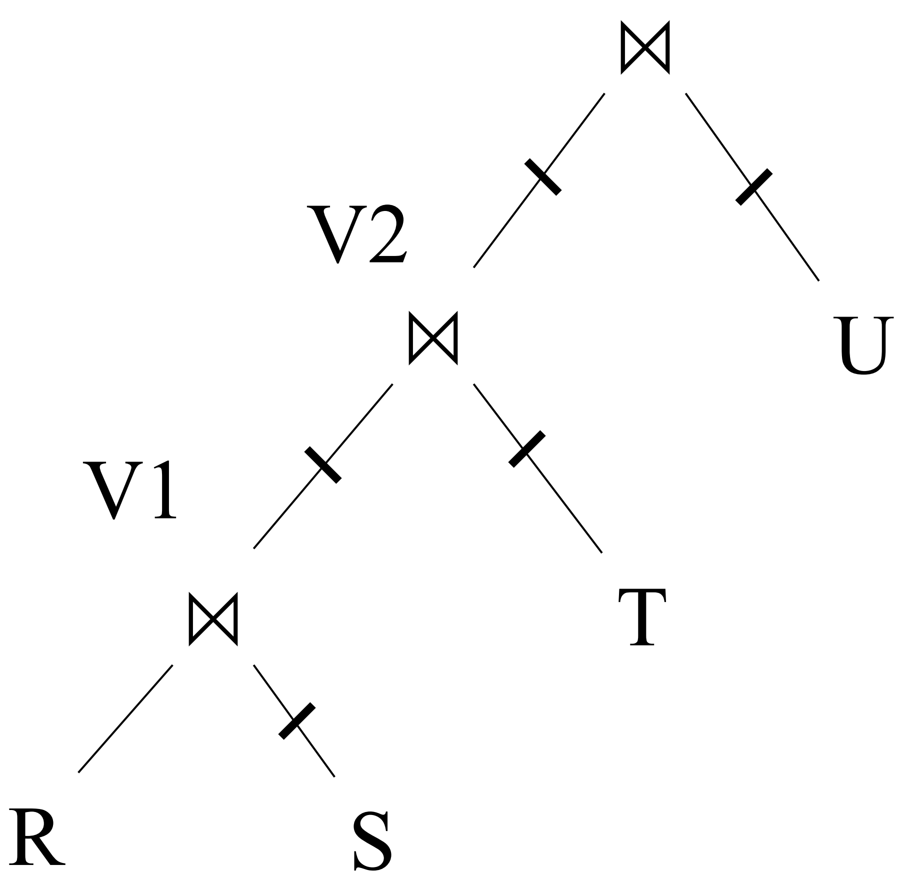
Completing the Physical Query Plan

- 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: create entirely and store on disk
  - To pipeline: create in parts and move on to next operation; entire result may never be available at the same time, not stored on disk.

Materialize Intermediate Results Between Operators

- 16.7.3



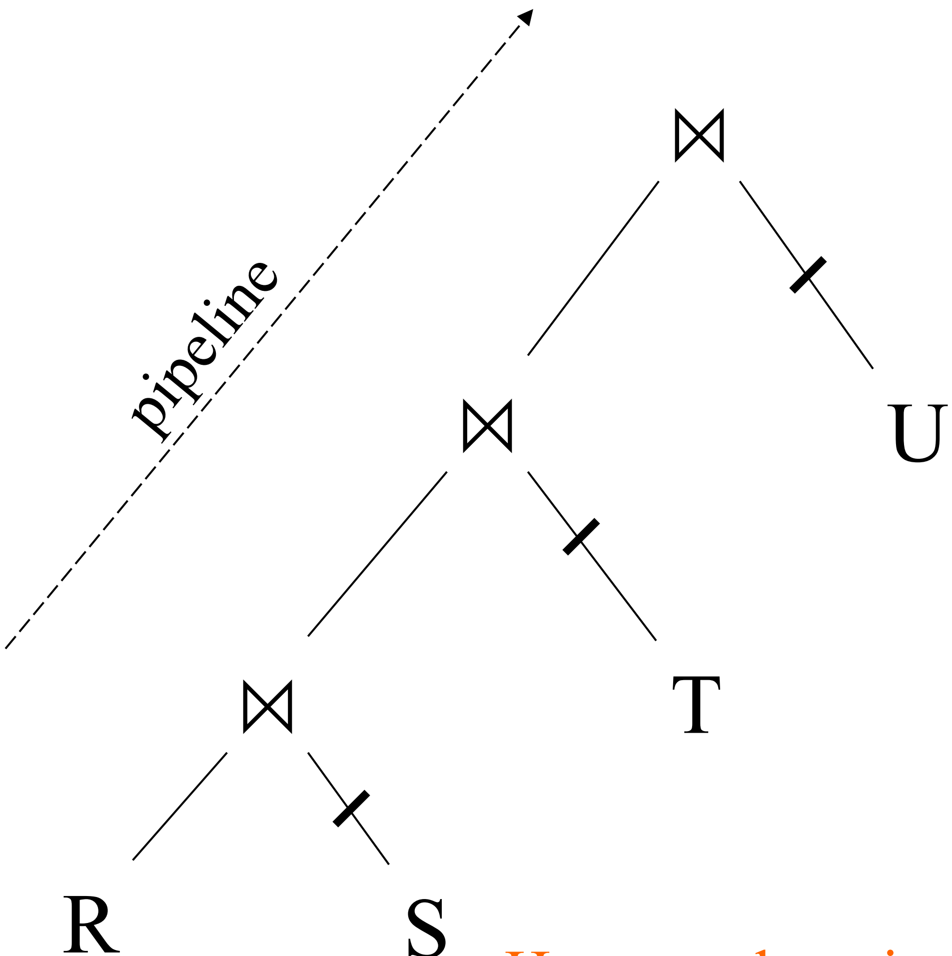

```
HashTable ← S
repeat    read(R, x)
          y ← join(HashTable, x)
          write(V1, y)

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

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)
```

How much main memory do we need ? M =

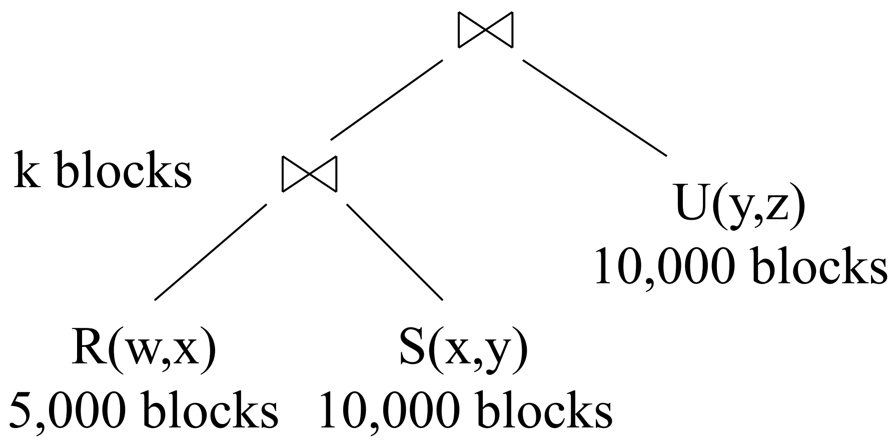
Pipeline Between Operators

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 =

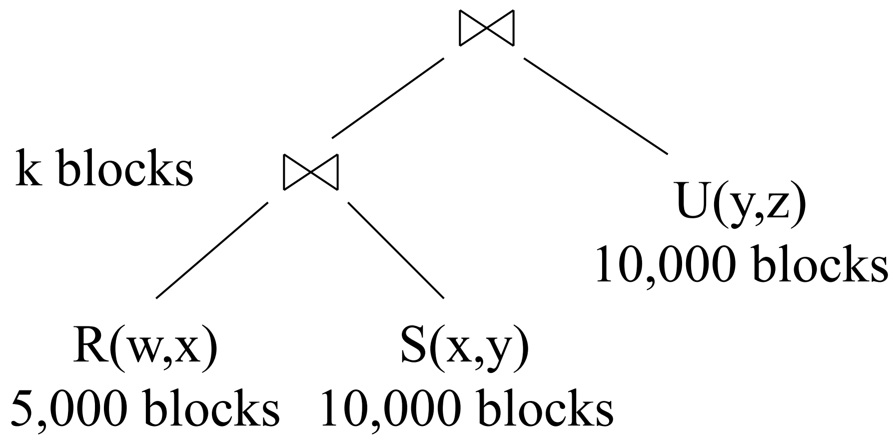
Example

- Logical plan is:



- Main memory M = 101 buffers

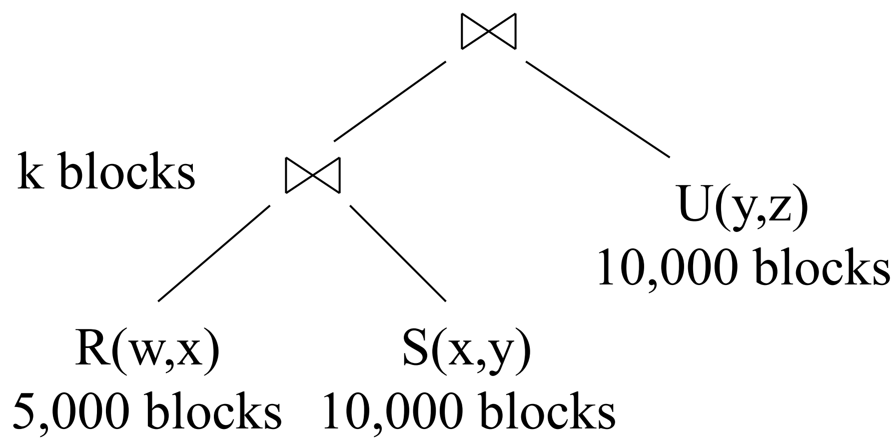
Example



Naïve evaluation:

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

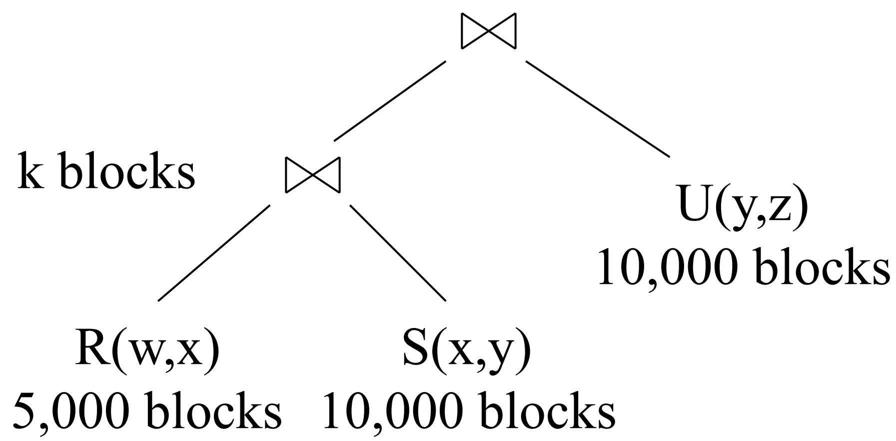
Example



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 Ri in memory (50 buffer) join with Si (1 buffer); hash result on y into 50 buckets (using the 50 remaining buffers)
- Cost so far: 3B(R) + 3B(S)

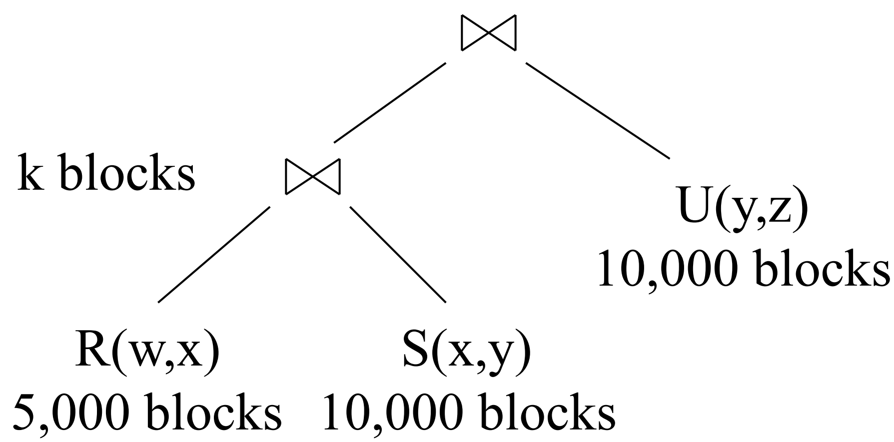
Example



Continuing:

- If k <= 50 then keep all 50 buckets in Step 3 in memory, then:
- Step 4: read U from disk (one block at a time), hash on y and join with memory
- Total cost: 3B(R) + 3B(S) + B(U) = 55,000

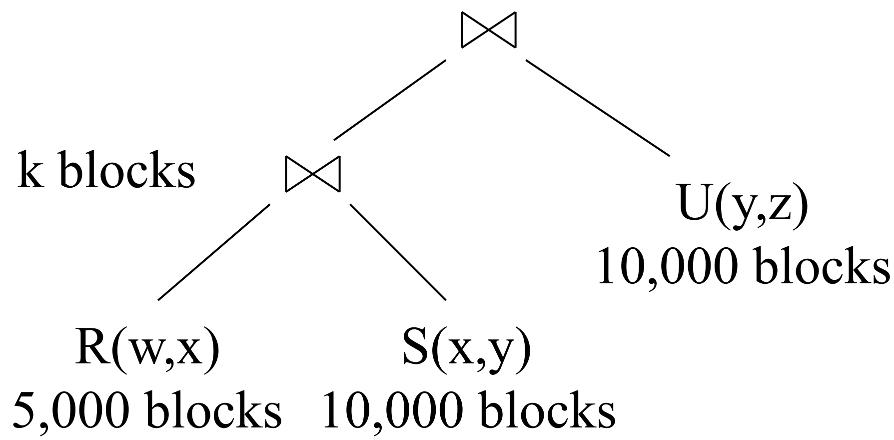
Example



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 bucket of (R $\bowtie$ S) into memory, read corresponding bucket of U (block by block) and join in memory
- Total cost: 3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k

Example



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 = $75{,}000 + 4k$

  - Read Example 16.36.

Estimating Sizes

- 16.4.1

Estimating Sizes

- Need size in order to estimate cost
- Example:
  - Cost of partitioned hash-join $E1 \bowtie E2$ is $3B(E1) + 3B(E2)$
  - $B(E1) = T(E1)/$ block size
  - $B(E2) = T(E2)/$ block size
  - So, we need to estimate $T(E1)$, $T(E2)$

Estimating Sizes

Estimating the size of a projection

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

Estimating Sizes

Estimating the size of a selection

- $S = \sigma_{A=c}(R)$
  - T(S) can be anything from 0 to T(R) – V(R,A) + 1
  - Mean value: T(S) = T(R)/V(R,A)
- $S = \sigma_{A<c}(R)$
  - T(S) can be anything from 0 to T(R)
  - Heuristic: T(S) = T(R)/3

Estimating Sizes

Estimating the size of a natural join, $R \bowtie_A S$

- When the set of A values are disjoint, then
  $T(R \bowtie_A S) = 0$
- When A is a key in S and a foreign key in R, then
  $T(R \bowtie_A S) = T(R)$

Estimating Sizes

Simplifying 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 holds when A is a foreign key in R, and a key in S

- *Preservation of values sets*: for any other attribute B, $V(R \bowtie_A S, B) = V(R, B)$ (or $V(S, B)$)

Estimating Sizes

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

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

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

Estimating Sizes

Example:

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

Answer: $T(R \bowtie_A S) = 10000 * 20000/200 = 1M$

Estimating Sizes

Joins on more than one attribute:

- $T(R \bowtie_{A,B} S) =$

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

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 $\bowtie_{Salary}$ 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

- Assume:
  - $V(Employee, Salary) = 200$
  - $V(Ranks, Salary) = 250$
- Then $T(Employee \bowtie_{Salary} Ranks) =$

  $= \Sigma_{i=1,6} T_i T_i' / 250$

  $= (200x8 + 800x20 + 5000x40 +$
  $12000x80 + 6500x100 + 500x2)/250$

  $= \ldots.$