

CS411

Database Systems

09: Query Optimization

Why Do We Learn This?

Optimization

- At the heart of the database engine
- Step 1: convert the SQL query to some logical plan
- Step 2: find a better logical plan, find an associated physical plan

SQL \rightarrow Logical Query Plans

Converting from SQL to Logical Plans

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

$$\Pi_{a1, \dots, an}(\sigma_C(R1 \times R2 \times \dots \times Rk))$$

Select a1, ..., an
From R1, ..., Rk
Where C
Group by b1, ..., bm

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

Optimization: Logical Query Plan

- Now we have one logical plan
- Algebraic laws:
 - foundation for every optimization
- Two approaches to optimizations:
 - 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

Motivating Example

Select S.name, C.instructor

From Students S, Enrollment E, Course C

Where S.dept = 'CS' and

S.sid=E.sid and E.cid = C.cid

The three components of an optimizer

We need three things in an optimizer:

- Algebraic laws
- An optimization algorithm
- A cost estimator

Algebraic Laws

Algebraic Laws

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

Q: How to prove these laws? Make sense?

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
 - $\sigma_C(R - S) = \sigma_C(R) - S$
 - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
 - $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Q: What do they mean? Make sense?

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
 - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
 - Where N, P, Q are appropriate subsets of attributes of M
 - $\Pi_M(\Pi_N(R)) = \Pi_{M \cap N}(R)$
- Example $R(A,B,C,D), S(E, F, G)$
 - $\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_{?}(R) \bowtie \Pi_{?}(S))$

Q: Again, what do they mean? Make sense?

Behind the Scene: Oracle RBO and CBO

TECHNOLOGY: Talking Tuning

Understanding Optimization

By Kimberly Floss

Improvements in the Oracle Database 10g Optimizer make it even more valuable for tuning.

Since its introduction in Oracle7, the cost-based optimizer (CBO) has become more valuable and relevant with each new release of the Oracle database while its counterpart, the rule-based optimizer (RBO), has become increasingly less so. The difference between the two optimizers is relatively clear: The CBO chooses the best path for your queries, based on what it knows about your data and by leveraging Oracle database features such as bitmap indexes, function-based indexes, hash joins, index-organized tables, and partitioning, whereas the RBO just follows established rules (heuristics). With the release of Oracle Database 10g, the RBO's obsolescence is official and the CBO has been significantly improved yet again.

As Published In

ORACLE
MAGAZINE
January/February
2005

- Oracle 7 (1992) prior (since 1979): RBO.
- Oracle 7-10: RBO + CBO.
- Oracle 10g (2003): CBO.

Behind the Scene: Oracle RBO and CBO

Rule-based optimization sometimes provided better performance than the early versions of Oracle's cost-based optimizer for specific situations. The rule-based optimizer had several weaknesses, including offering only a simplistic set of rules. The Oracle rule-based optimizer had about 20 rules and assigned a weight to each one of them. In a complex database, a query can easily involve several tables, each with several indexes and complex selection conditions and ordering. This complexity means that there were a lot of options, and the simple set of rules used by the rule-based optimizer might not differentiate the choices well enough to make the best choice.

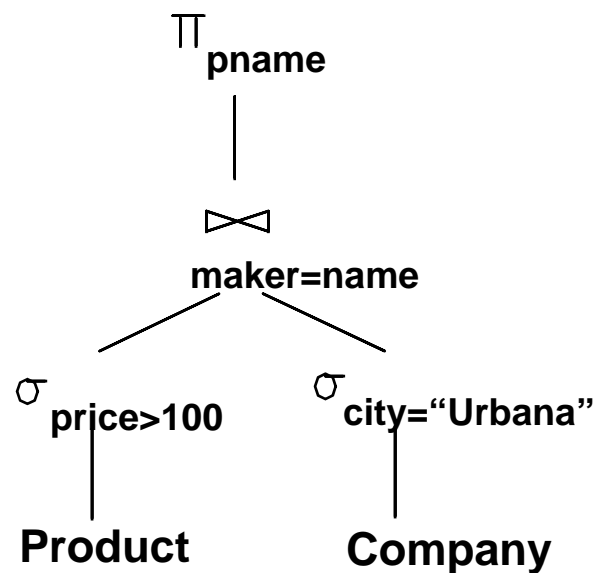
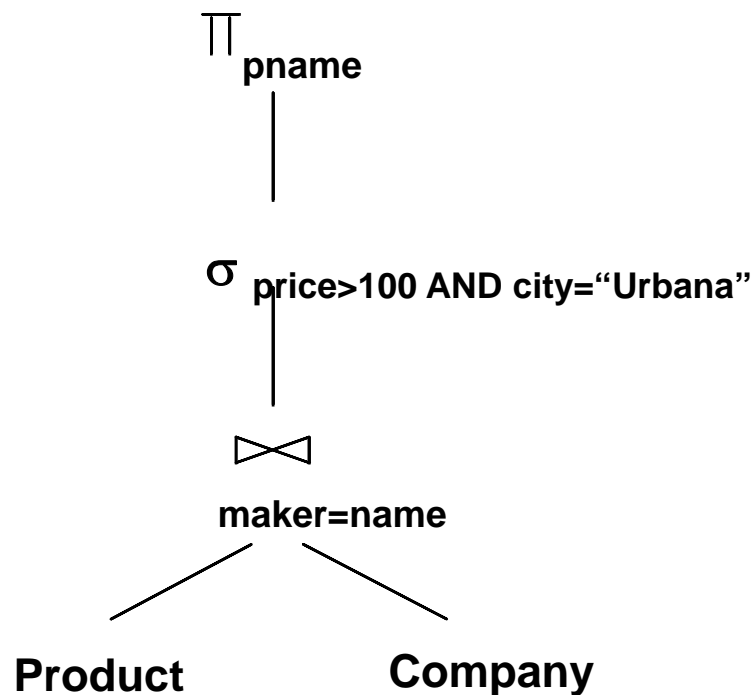
The rule-based optimizer assigned an optimization score to each potential execution path and then took the path with the best optimization score. Another weakness in the rule-based optimizer was resolution of optimization choices made in the event of a "tie" score. When two paths presented the same optimization score, the rule-based optimizer looked to the syntax of the SQL statement to resolve the tie. The winning execution path was based on the order in which the tables occur in the SQL statement.

Rule-based Optimization

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Heuristics number 1:
 - Push selections down
- Heuristics number 2:
 - Sometimes push selections up, then down

Predicate Pushdown



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).

Predicate Pushdown

```
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.*
- 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 predicates 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 < 20
GroupBy x.category
```

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

Query Rewrite: Pushing predicates 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 < 20
```



```
Create View V1 AS
Select  x.category,
        Min(x.price) AS p
From    product x
Where   x.price < 20
GroupBy x.category
```


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

Query Rewrite:

Pushing predicates 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 < 20
```



```
Create View V1 AS
Select  x.category,
        Min(x.price) AS p
From    product x
Where   x.price < 20
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 < 20
```

Cost-based Optimization

Behind the Scene: The Selinger Style!

Patricia Selinger

From Wikipedia, the free encyclopedia

Patricia Selinger is an American [computer scientist](#) and [IBM Fellow](#), best known for her work on [relational database management systems](#). She played a fundamental role in the development of [System R](#), a pioneering relational database implementation, and wrote the canonical paper on relational [query optimization](#).^[1] The dynamic programming algorithm for determining join order proposed in that paper still forms the basis for most of the query optimizers used in modern relational systems.

She was made an IBM Fellow in 1994, was elected to the [National Academy of Engineering](#) in 1999, and won the [SIGMOD Edgar F. Codd Innovations Award](#) in 2002. Before her retirement, she was the Vice President of Data Management Architecture and Technology at IBM. She received A.B., S.M., and Ph.D. degrees in [applied mathematics](#) from [Harvard University](#).

References

[\[edit\]](#)

- [^] Selinger, P. G.; Astrahan, M. M.; [Chamberlin, D. D.](#); Lorie, R. A.; Price, T. G. (1979), "Access Path Selection in a Relational Database Management System", *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 23-34, doi:10.1145/582095.582099 [↗](#)

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.



Behind the Scene: The Selinger Style!

In my view, the query optimizer was the first attempt at what we call autonomic computing or self-managing, self-tuning technology. Query optimizers have been 25 years in development, with enhancements of the cost-based query model and the optimization that goes with it, and a richer and richer variety of execution techniques that the optimizer chooses from. We just have to keep working on this. It's a never-ending quest for an increasingly better model and repertoire of optimization and execution techniques. So the more the model can predict what's really happening in the data and how the data is really organized, the closer and closer we will come [to the ideal system].

In hindsight, I didn't really have enough experience as a system designer and developer to realize that System R was going to be something that people [within IBM] were going to take almost as is, and make a productized copy of it and [sell] it. So designing control blocks, designing interfaces and APIs to programming languages---we just sort of put together something that we thought would work. The error control block, the way that you pass variables and parameters into the system---those were not designed with thought and care and elegance. They were [just]

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

Search Strategies

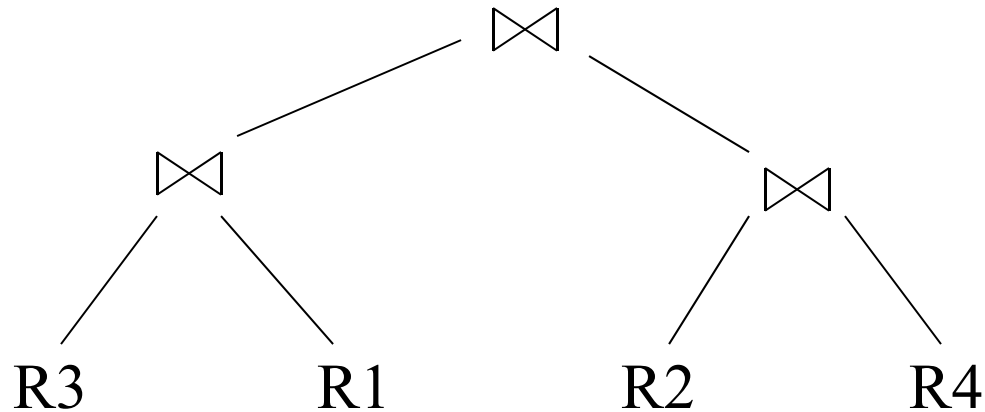
- **Branch-and-bound:**
 - Remember the cheapest complete plan P seen so far and its cost C
 - Stop generating partial plans whose cost is $> C$
 - If a cheaper complete plan is found, replace P , C
- **Hill climbing:**
 - Remember only the cheapest partial plan seen so far
- **Dynamic programming:**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

Dynamic Programming

Unit of Optimization: select-project-join

Join Trees

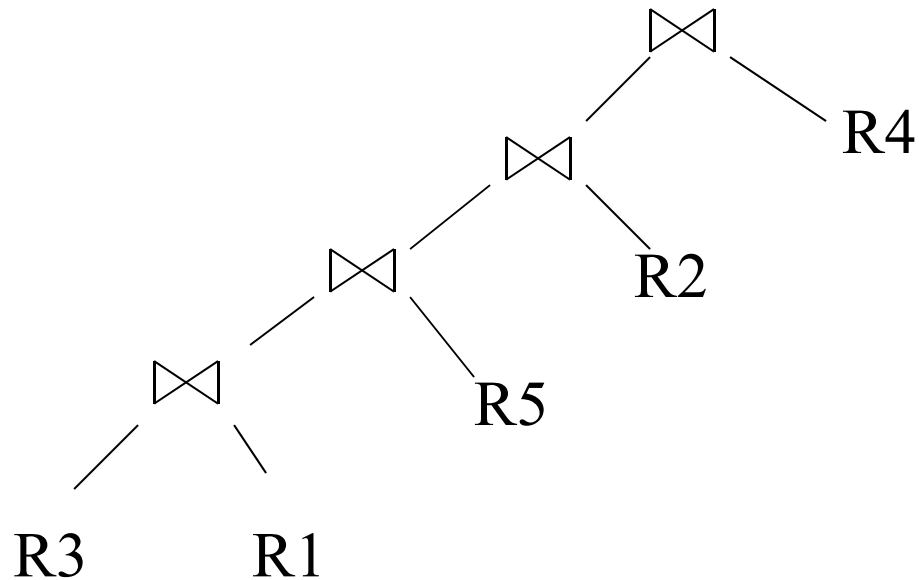
- $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$
- Join tree:



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

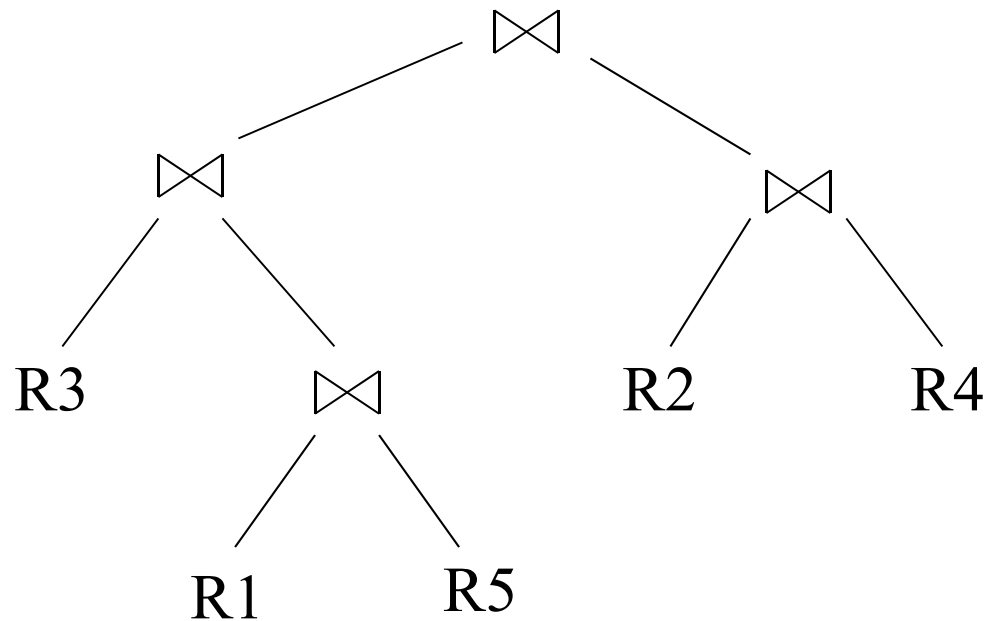
Types of Join Trees

- Left deep:



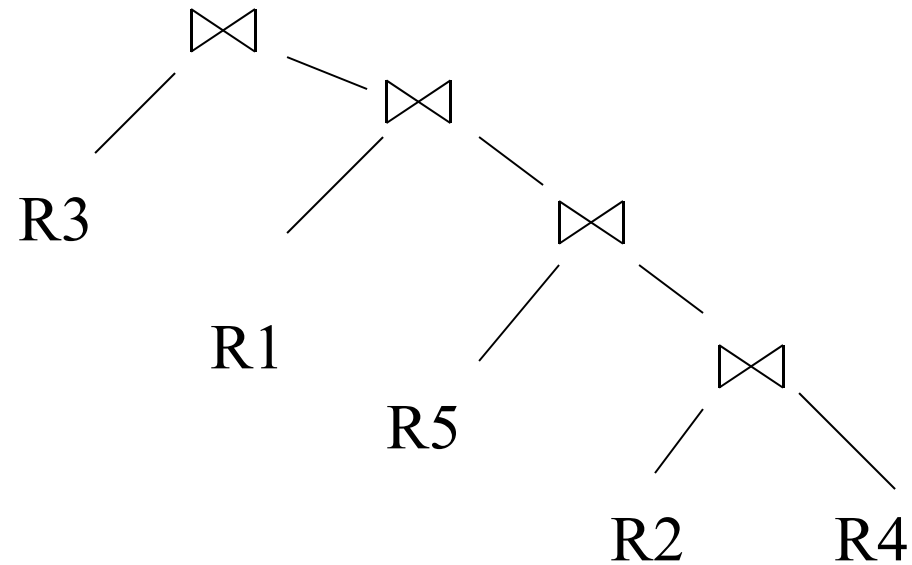
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Problem

- Given: a query $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$
- Assume we have a function $\text{cost}()$ that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of $\{R_1, \dots, R_n\}$, compute the best plan for that subset
- In increasing order of set cardinality:
 - Step 1: for $\{R_1\}, \{R_2\}, \dots, \{R_n\}$
 - Step 2: for $\{R_1, R_2\}, \{R_1, R_3\}, \dots, \{R_{n-1}, R_n\}$
 - ...
 - Step n: for $\{R_1, \dots, R_n\}$
- It is a bottom-up strategy
- A subset of $\{R_1, \dots, R_n\}$ is also called a *subquery*

Dynamic Programming

- For each subquery $Q \subseteq \{R_1, \dots, R_n\}$ compute the following:
 - $\text{Size}(Q)$
 - A best plan for Q : $\text{Plan}(Q)$
 - The cost of that plan: $\text{Cost}(Q)$

Dynamic Programming

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

Dynamic Programming

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

Dynamic Programming

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

Dynamic Programming

To illustrate, we will make the following simplifications:

- $\text{Cost}(P1 \bowtie P2) = \text{Cost}(P1) + \text{Cost}(P2) + \text{size}(\text{intermediate result})$
- Intermediate results:
 - If $P1$ = a join, then the size of the intermediate result is $\text{size}(P1)$, otherwise the size is 0
 - Similarly for $P2$
- Cost of a scan = 0

Dynamic Programming

- Example:
- $\text{Cost}(R5 \bowtie R7) = 0$ (no intermediate results)
- $\text{Cost}((R2 \bowtie R1) \bowtie R7)$
 $= \text{Cost}(R2 \bowtie R1) + \text{Cost}(R7) + \text{size}(R2 \bowtie R1)$
 $= \text{size}(R2 \bowtie R1)$

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	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)

Dynamic Programming

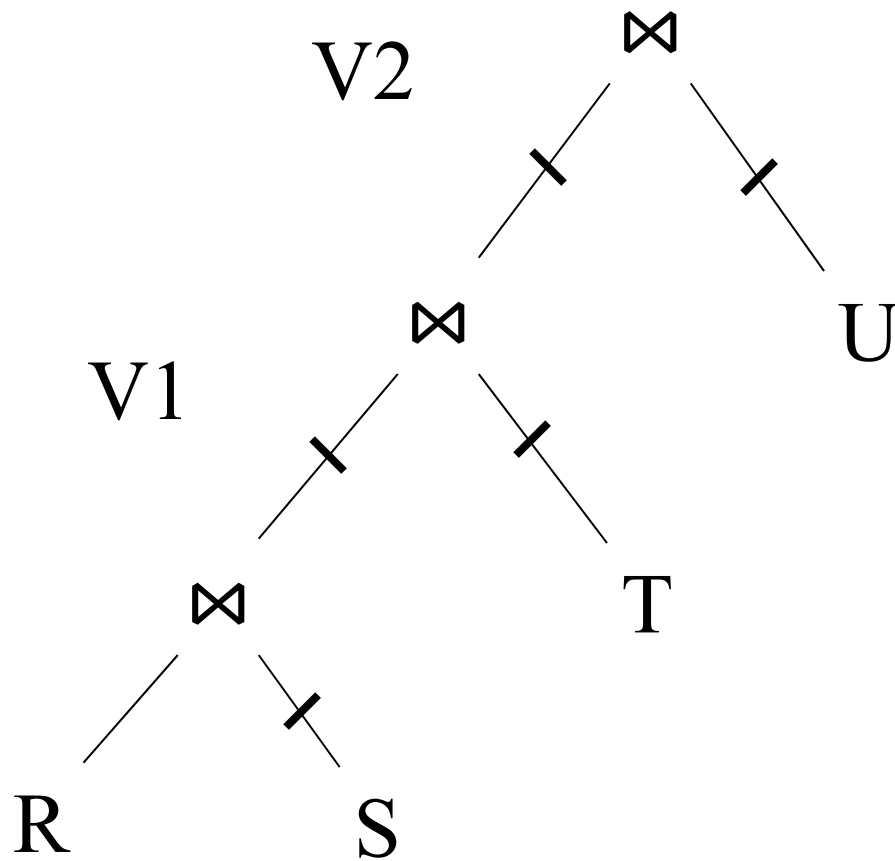
- Summary: computes optimal plans for subqueries:
 - Step 1: $\{R_1\}, \{R_2\}, \dots, \{R_n\}$
 - Step 2: $\{R_1, R_2\}, \{R_1, R_3\}, \dots, \{R_{n-1}, R_n\}$
 - ...
 - Step n: $\{R_1, \dots, R_n\}$
- We used naïve size/cost estimations
- In practice:
 - more realistic size/cost estimations (next time)
 - heuristics for Reducing the Search Space
 - Restrict to left linear trees
 - Restrict to trees “without cartesian product”: $R(A,B), S(B,C), T(C,D)$
 $(R \text{ join } T) \text{ join } S$ 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
 - 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)
```

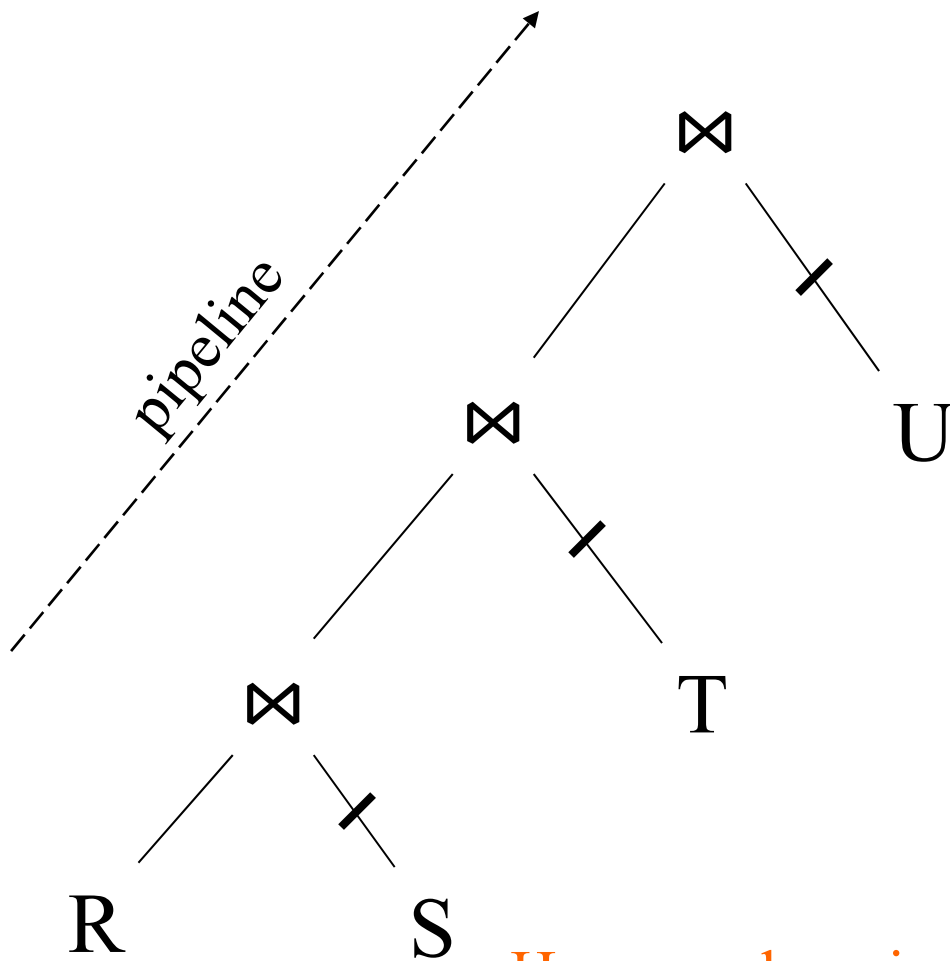
```
HashTable ← U
repeat
  read(V2, z)
  u ← join(HashTable, z)
  write(Answer, u)
```


Materialize Intermediate Results Between Operators

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

How much main memory do we need ? $M =$

Pipeline Between Operators

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

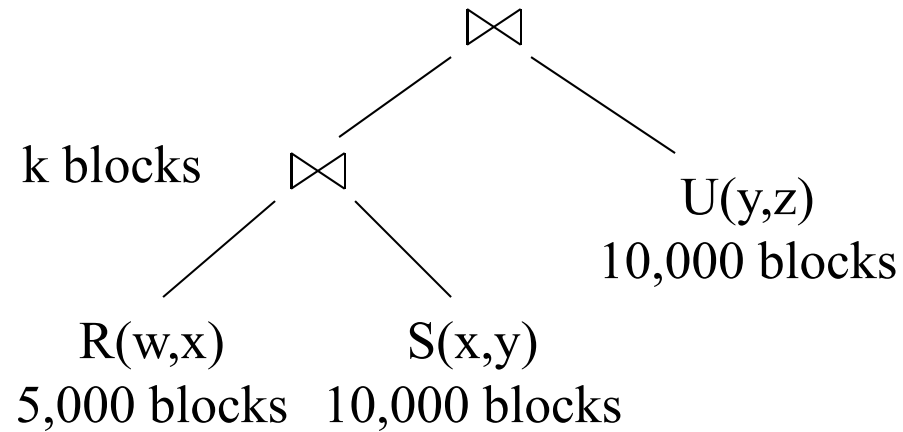
- What is the total cost of the plan ?
 - Cost =
- How much main memory do we need ?
 - M =

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

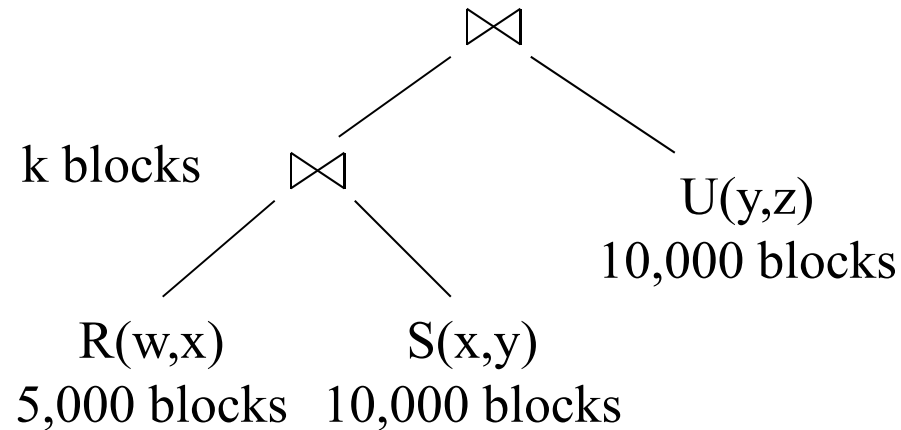
Example

- Logical plan is:



- Main memory $M = 101$ buffers

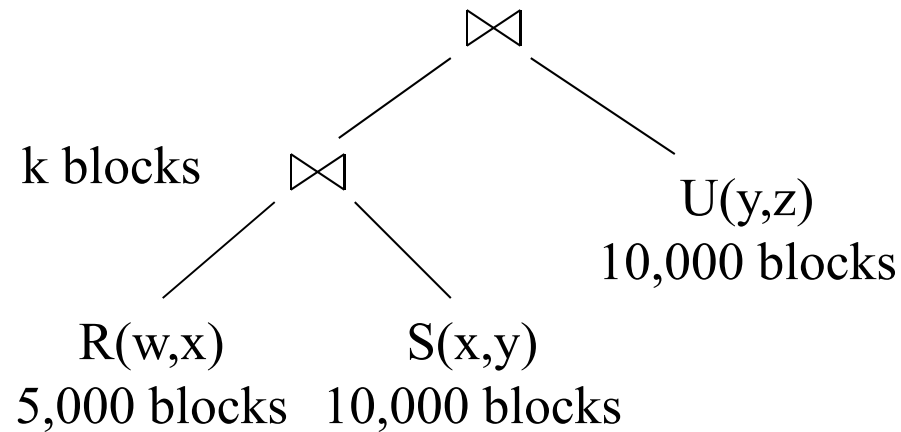
Example



Naïve evaluation:

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

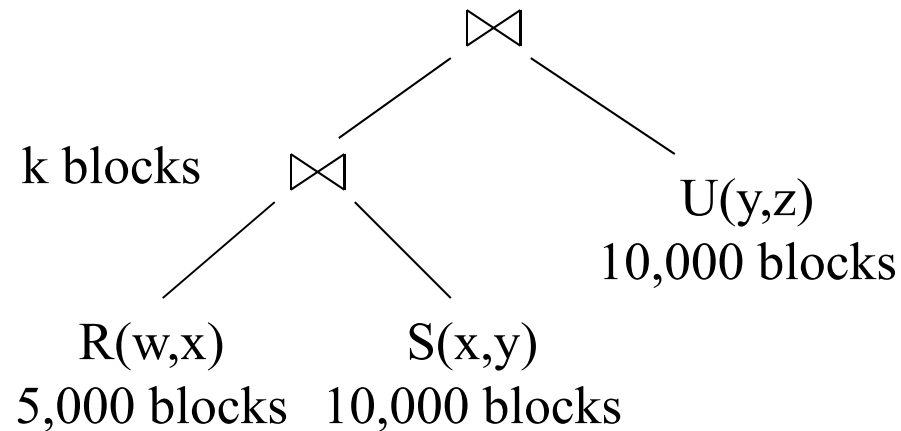
Example



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 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: $3B(R) + 3B(S)$

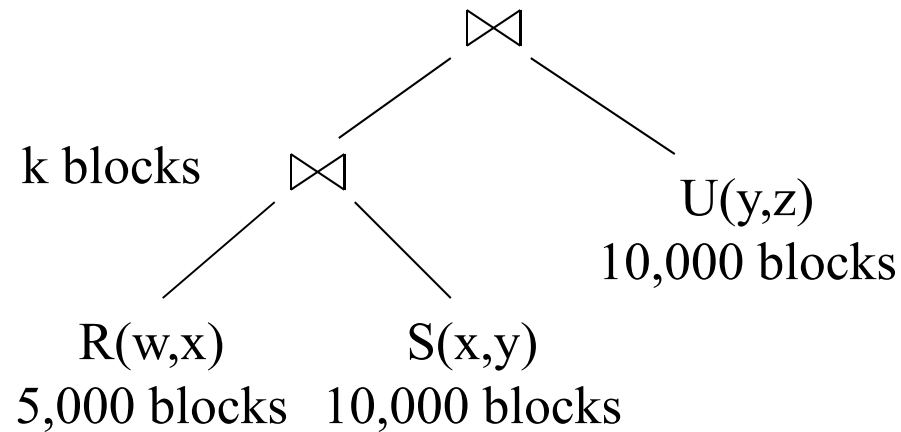
Example



Continuing:

- How large are the 50 buckets on y ? Answer: $k/50$.
- If $k \leq 50$ 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: $3B(R) + 3B(S) + B(U) = 55,000$

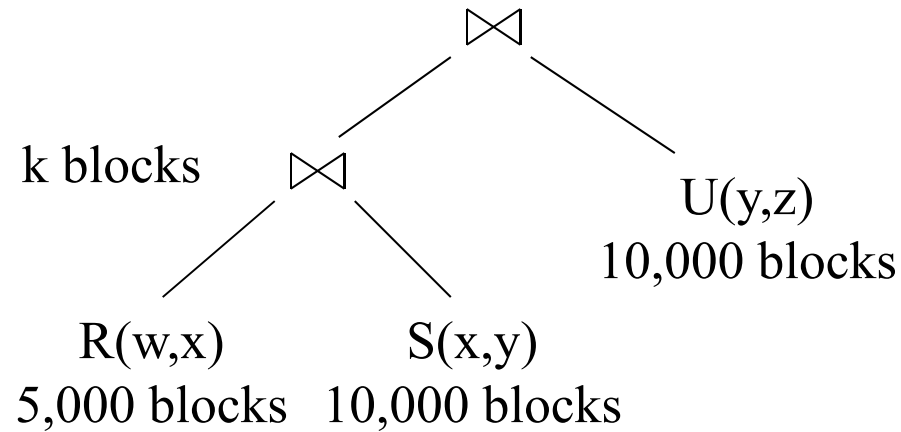
Example



Continuing:

- If $50 < k \leq 5000$ then send the 50 buckets in Step 3 to disk
 - Each bucket has size $k/50 \leq 100$
- 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



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 $k \leq 50$, $\text{cost} = 55,000$
- If $50 < k \leq 5000$, $\text{cost} = 75,000 + 2k$
- If $k > 5000$, $\text{cost} = 75,000 + 4k$

Estimating Sizes

Estimating Sizes

- Need size in order to estimate cost
- Example:
 - Cost of partitioned hash-join $E1 \bowtie E2$ is $3B(E1) + 3B(E2)$
 - $B(E1) = T(E1) / \text{block size}$
 - $B(E2) = T(E2) / \text{block size}$
 - 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 $T(R)$
 - Heuristics: $T(S) = T(R)/3$

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
$$T(R \bowtie_A S) = T(R)$$
- When A has a unique value, the same in R and S , then
$$T(R \bowtie_A S) = \min(T(R), T(S))$$

Estimating Sizes

Assumptions:

- Containment of values: if $V(R,A) \leq V(S,A)$, 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,
 $V(R \bowtie_A S, B) = V(R, B)$ (or $V(S, B)$)

Estimating Sizes

Assume $V(R,A) \leq V(S,A)$

- 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(ssn, 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

- $T(\text{Employee}) = 25000$, 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:
 - $V(\text{Employee}, \text{Salary}) = 200$
 - $V(\text{Ranks}, \text{Salary}) = 250$
- Then $T(\text{Employee} \bowtie_{\text{Salary}} \text{Ranks}) =$
 - $= \sum_{i=1,6} T_i T_i' / 250$
 - $= (200 \times 8 + 800 \times 20 + 5000 \times 40 +$
 $12000 \times 80 + 6500 \times 100 + 500 \times 2) / 250$
 - $= \dots$