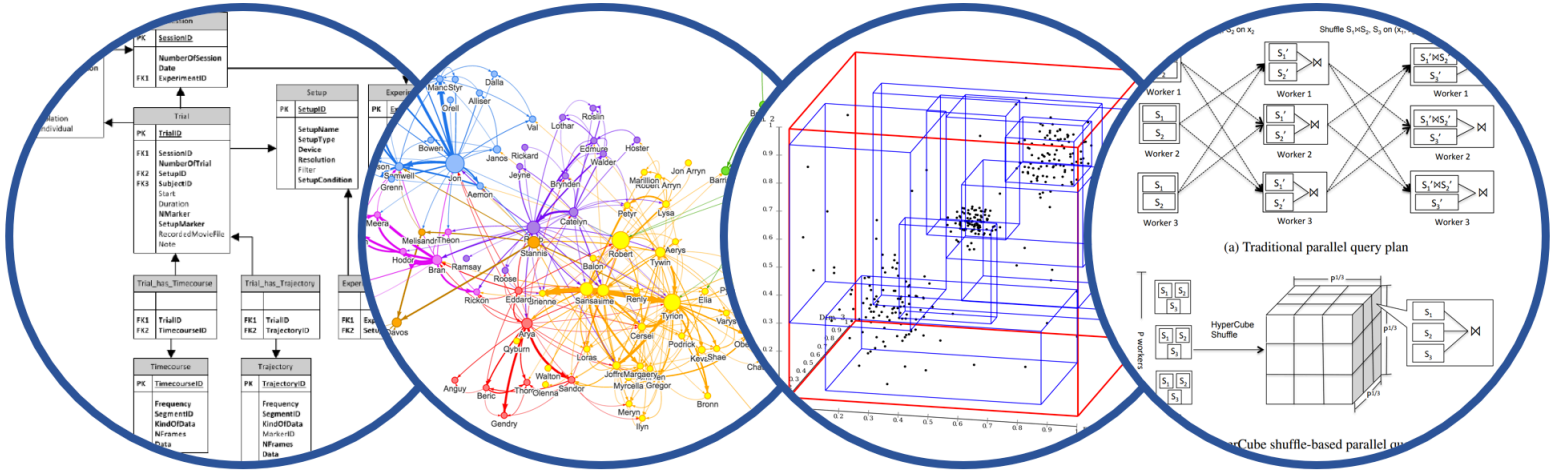


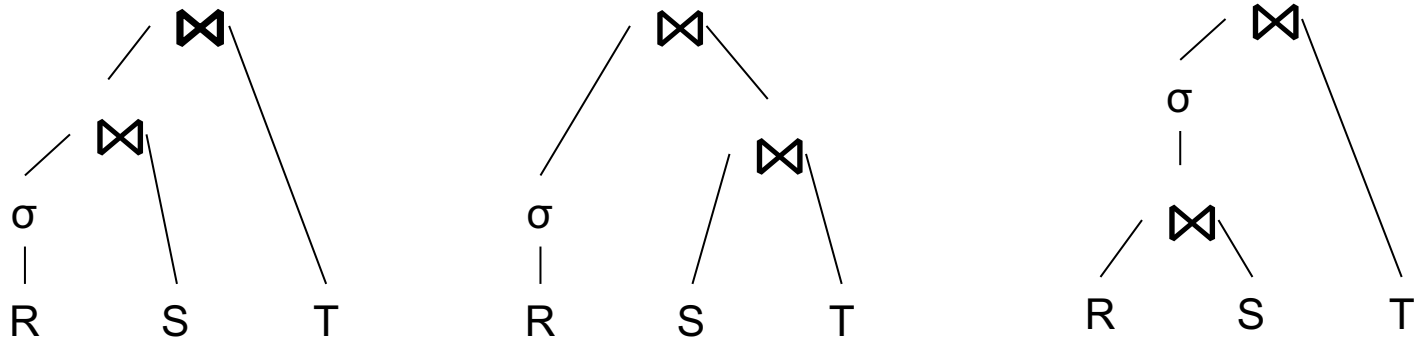
Database System Internals

Query Plan Costs

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Query Optimization Summary

Goal: find a physical plan that has minimal cost



What is the cost of a plan?

For each operator, cost is function of CPU, IO, network bw

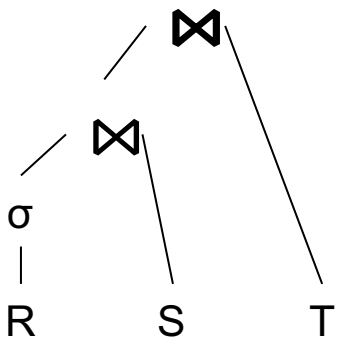
Total_Cost = CPUCost + w_{IO} IOCost+ w_{BW} BWCost

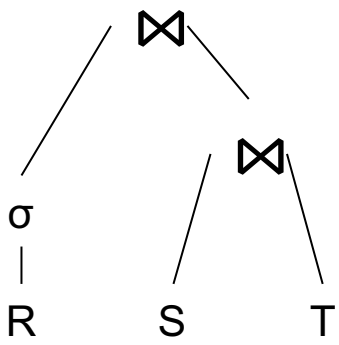
Cost of plan is total for all operators

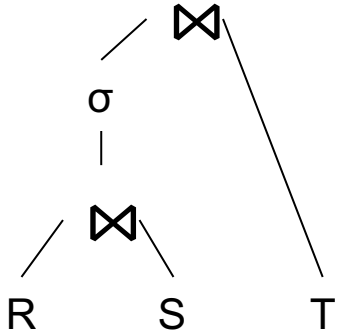
In this class, we look only at IO

Query Optimization Summary

Goal: find a physical plan that has minimal cost

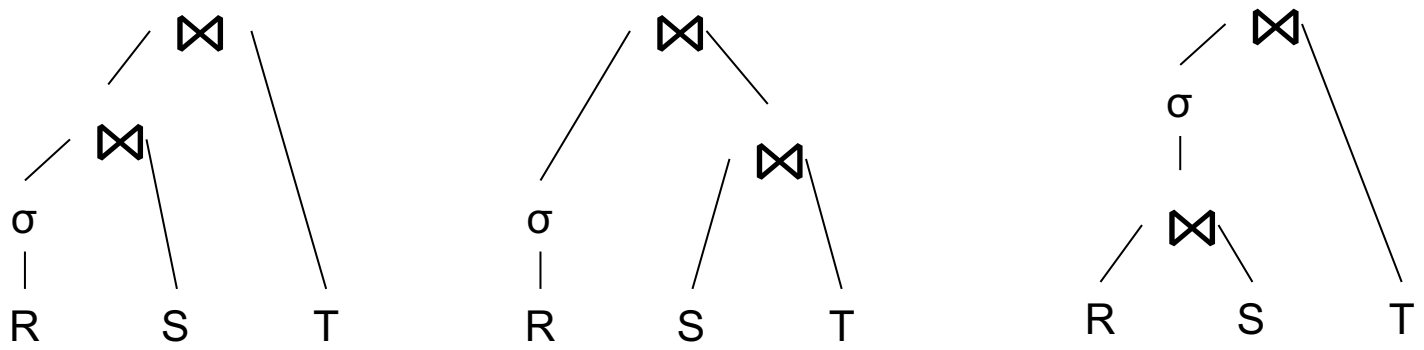






Query Optimization Summary

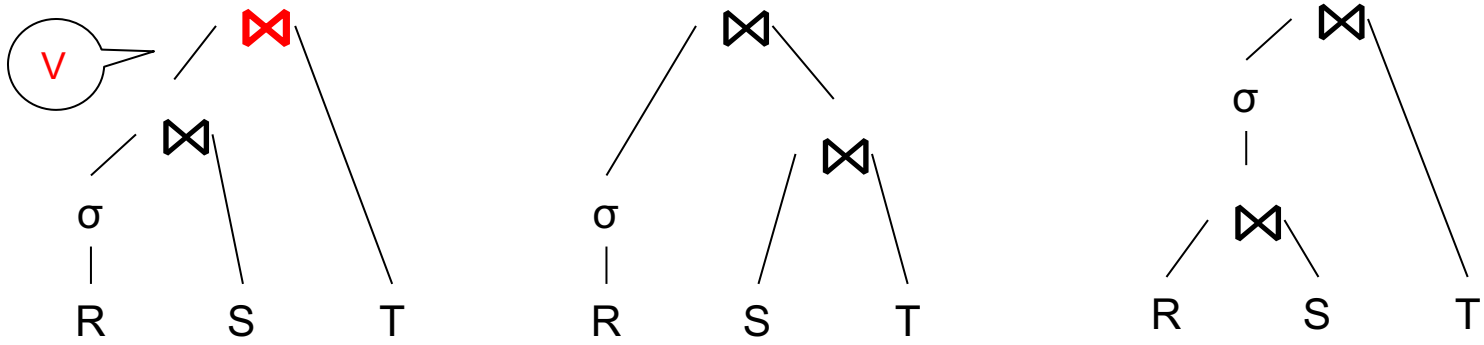
Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

Query Optimization Summary

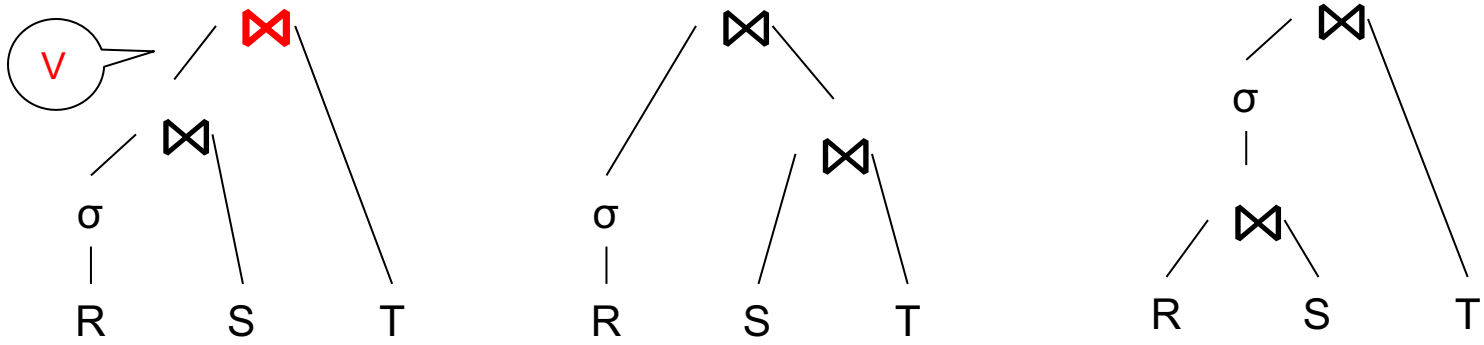
Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

Query Optimization Summary

Goal: find a physical plan that has minimal cost

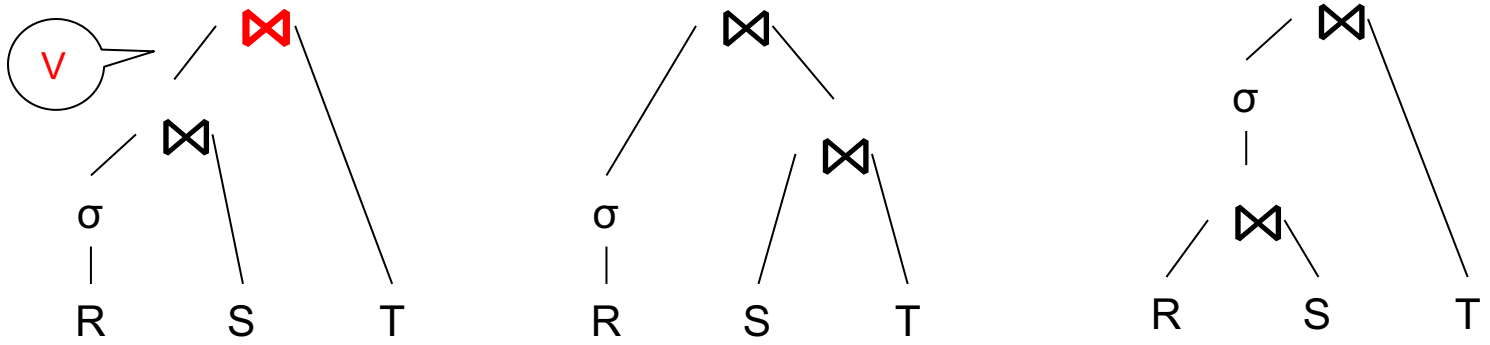


Know how to compute cost if know cardinalities

- Eg. Cost(V ⋈ T) = 3B(V) + 3B(T)
- B(V) = T(V) / PageSize
- T(V) = T(σ(R) ⋈ S)

Query Optimization Summary

Goal: find a physical plan that has minimal cost



Know how to compute cost if know cardinalities

- Eg. Cost(V ⋈ T) = 3B(V) + 3B(T)
- B(V) = T(V) / PageSize
- T(V) = T(σ(R) ⋈ S)

Cardinality estimation problem: e.g. estimate T(σ(R) ⋈ S)

Database Statistics

- Collect statistical summaries of stored data

- Estimate size (=cardinality) in a bottom-up fashion
  - This is the most difficult part, and still inadequate in today's query optimizers
- Estimate cost by using the estimated size
  - Hand-written formulas, similar to those we used for computing the cost of each physical operator

Database Statistics

- Number of tuples (cardinality) T(R)
- Indexes, number of keys in the index V(R,a)
- Number of physical pages B(R)
- Statistical information on attributes
  - Min value, Max value, V(R,a)
- Histograms

- Collection approach: periodic, using sampling

Size Estimation Problem

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

Given T(R1), T(R2), …, T(Rn)
Estimate T(Q)

How can we do this ? Note: doesn't have to be exact.

Size Estimation Problem

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

Remark: $T(Q) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Size Estimation Problem

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

Remark: $T(Q) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Key idea: each condition reduces the size of T(Q) by some factor, called selectivity factor

Selectivity Factor

- Each condition cond reduces the size by some factor called selectivity factor
- Assuming independence, multiply the selectivity factors

Example

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

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

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

Q: What is the estimated size of the query output T(Q) ?

Example

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

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

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

Q: What is the estimated size of the query output T(Q) ?

A: T(Q) = 30k * 200k * 10k * 1/3 * 1/10 * ½ = 10^{12}

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- A < c /* $\sigma_{A<c}(R)$*/
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- A < c /* $\sigma_{A<c}(R)$*/
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))
- A = B /* $R \bowtie_{A=B} S$ */
  - Selectivity = 1 / max(V(R,A),V(S,A))
  - (will explain next)

Assumptions

- *Containment of values*: if V(R,A) <= V(S,B), then all values R.A occur in S.B
  - Note: this indeed holds when A is a foreign key in R, and B is a key in S

- *Preservation of values*: for any other attribute C, $V(R \bowtie_{A=B} S, C) = V(R, C)$ (or V(S, C))
  - Note: we don't need this to estimate the size of the join, but we need it in estimating the next operator

Selectivity of $R \bowtie_{A=B} S$

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

- A tuple t in R joins with T(S)/V(S,B) tuple(s) in S
- Hence $T(R \bowtie_{A=B} S)$ = T(R) T(S) / V(S,B)

$$T(R \bowtie_{A=B} S) = T(R)\,T(S) / \max(V(R,A),V(S,B))$$

Complete Example

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sno = y.sno
    and  y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

Suppy.sno references Supplier.sno

- Some statistics
  - T(Supplier) = 1000 records
  - T(Supply) = 10,000 records
  - B(Supplier) = 100 pages
  - B(Supply) = 100 pages
  - V(Supplier,scity) = 20, V(Suppliers,state) = 10
  - V(Supply,pno) = 2,500
  - Both relations are clustered
- M = 11

Physical Query Plan 1

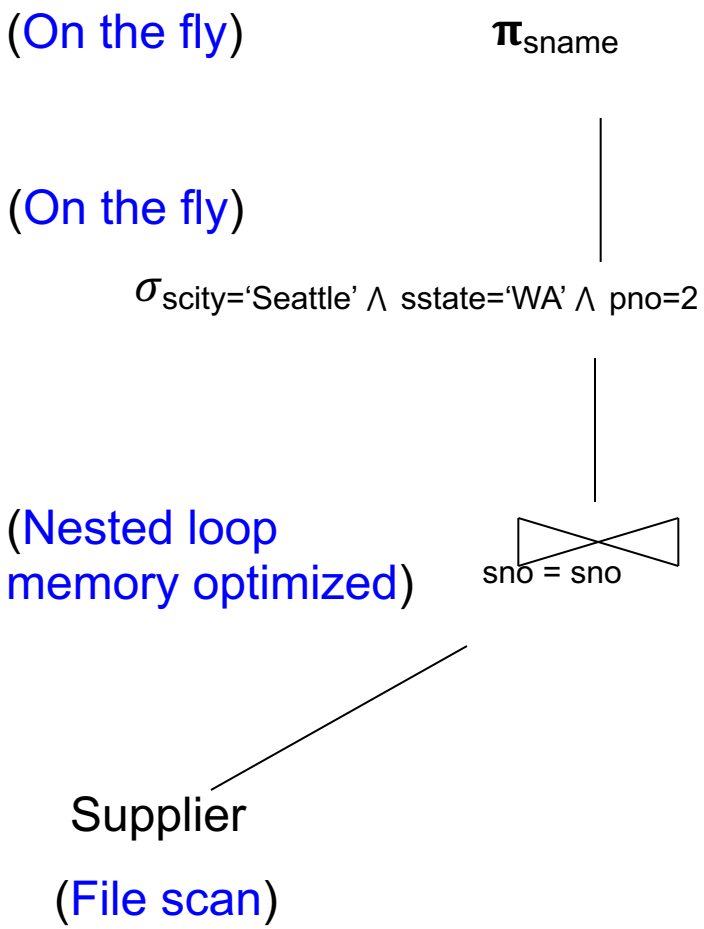
T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno



Selection and project on-the-fly
-> No additional cost.

Total cost of plan is thus cost of join:
= B(Supplier)+B(Supplier)*B(Supply)
= 100 + 100 * 100 / (11-1)
= 1,100 I/Os

Physical Query Plan 2

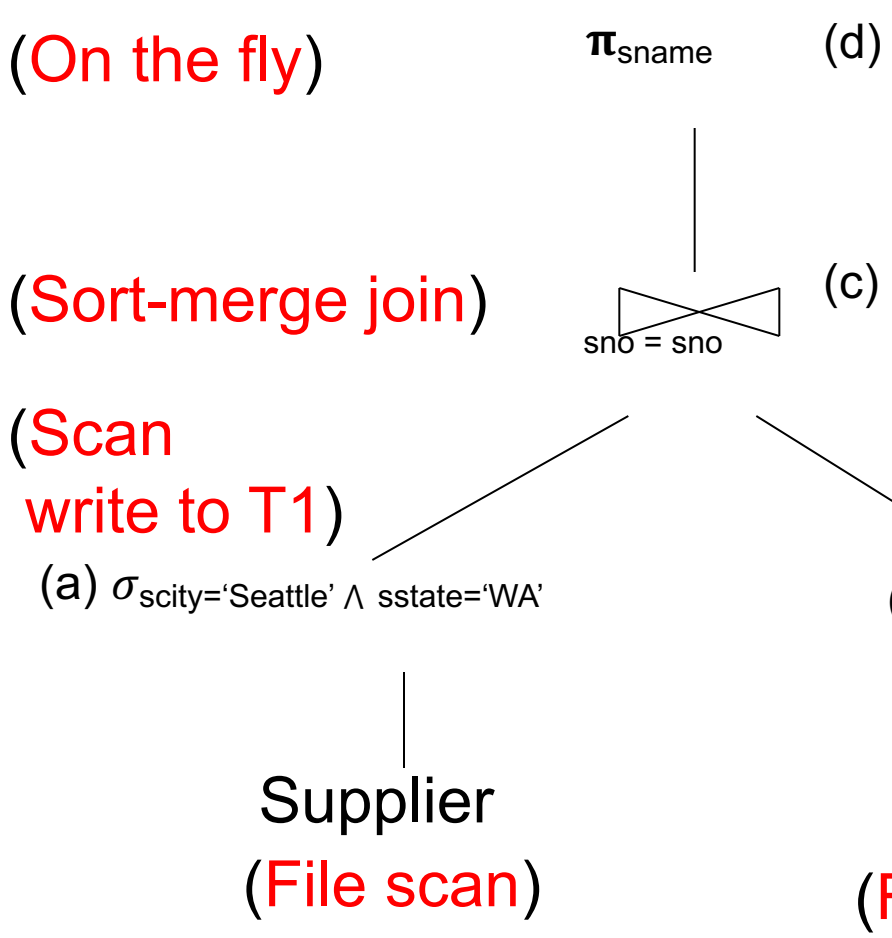
T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno



Total cost
= 100 + 100 * 1/20 * 1/10 (a)
+ 100 + 100 * 1/2500 (b)
+ 1 + 1 (c)
+ 0 (d)

Total cost ≈ **204 I/Os**

Plan 2 with Different Numbers

V(Supplier,scity) = 20 V(Supplier,state) = 10 V(Supply,pno) = 2,500

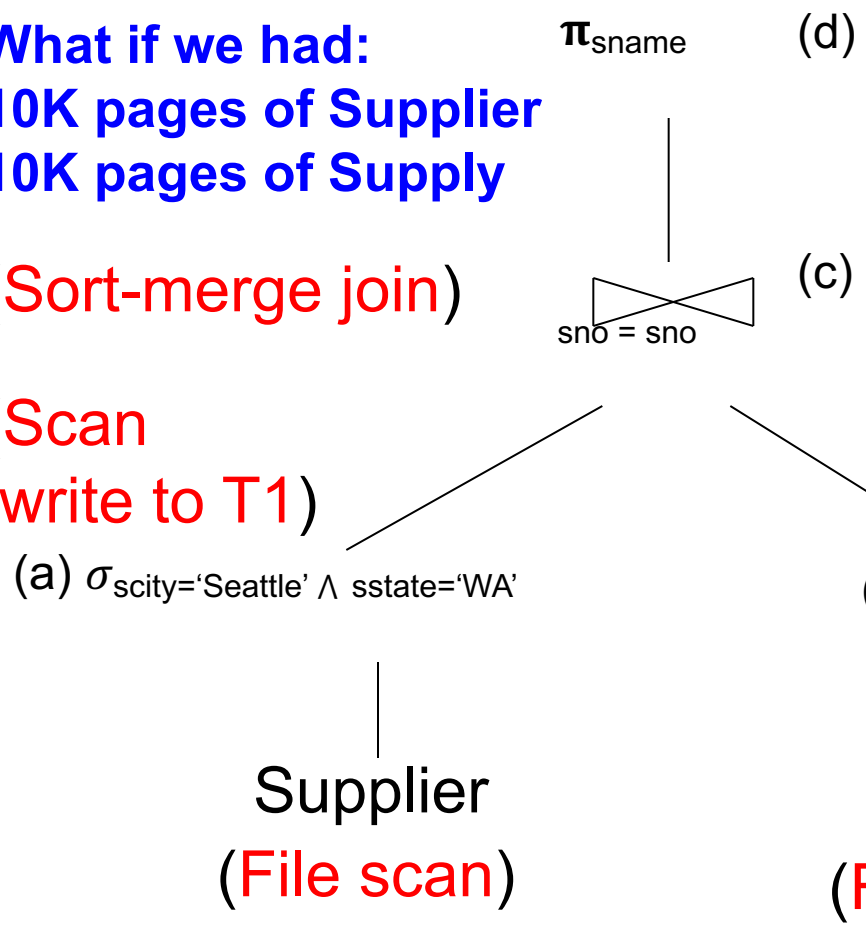
M = 11
Suppy.sno references Supplier.sno

What if we had:
10K pages of Supplier
10K pages of Supply

(Sort-merge join)

(Scan write to T1)

(Scan write to T2)



Total cost
= 10000 + 50 (a)
+ 10000 + 4 (b)
+ 3*50 + 4 (c)
+ 0 (d)
Total cost ≈ 20,208 I/Os

Need to do a two-pass sort algorithm

Physical Query Plan 3

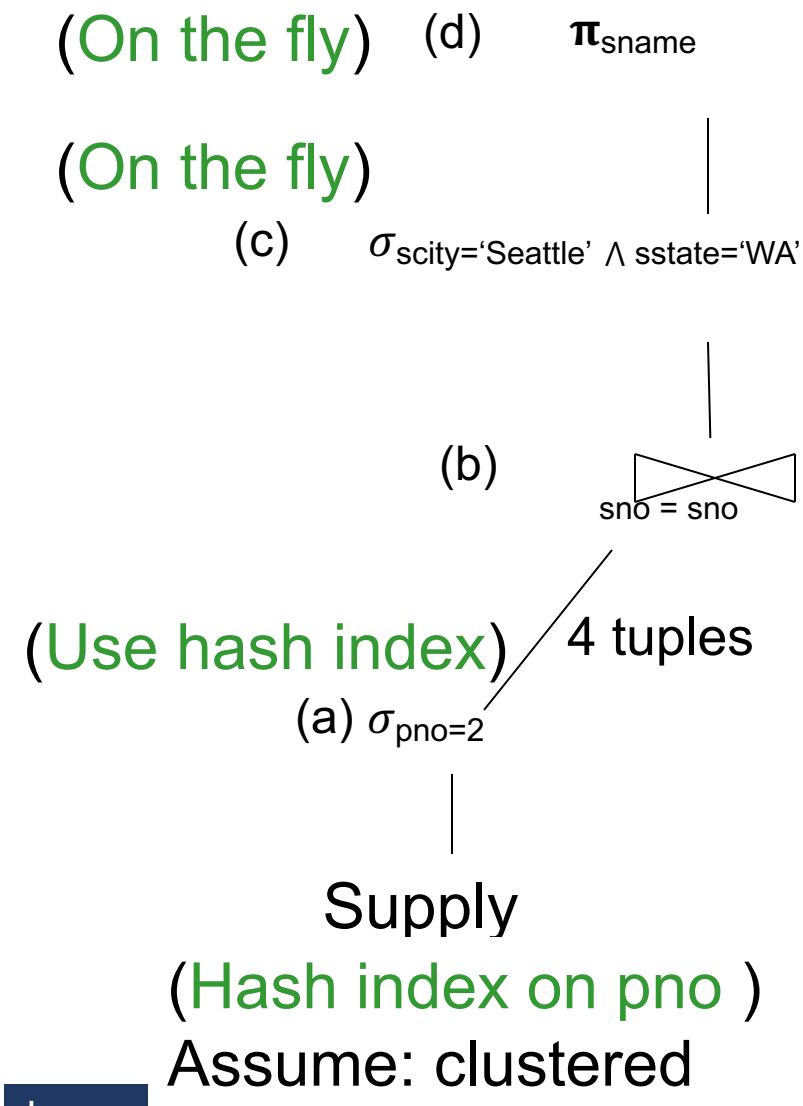
T(Supplier) = 1000
T(Supply) = 10,000

B(Supplier) = 100
B(Supply) = 100

V(Supplier,scity) = 20
V(Supplier,state) = 10
V(Supply,pno) = 2,500

M = 11

Suppy.sno references Supplier.sno



(Index nested loop)

Remember: Suppy.sno references Supplier.sno

Total cost
= 1 (a)
+ 4 (b)
+ 0 (c)
+ 0 (d)
Total cost ≈ **5 I/Os**

Histograms

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

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

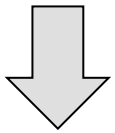
$\sigma_{age=48}(Empolyee) = ?$ $\sigma_{age>28 \text{ and } age<35}(Empolyee) = ?$

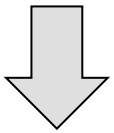
Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}(Empolyee) = ?$ $\sigma_{age>28 \text{ and } age<35}(Empolyee) = ?$





Estimate = 25000 / 50 = 500

Estimate = 25000 * 6 / 50 = 3000

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}(Empolyee) = ?$ $\sigma_{age>28 \text{ and } age<35}(Empolyee) = ?$

Age:	0-20	20-29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}(Empolyee) = ?$ $\sigma_{age>28 \text{ and } age<35}(Empolyee) = ?$

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Estimate = 1200

Estimate = 1*80 + 5*500 = 2580

Types of Histograms

- How should we determine the bucket boundaries in a histogram?

Types of Histograms

- How should we determine the bucket boundaries in a histogram?

- Eq-Width
- Eq-Depth
- Compressed
- V-Optimal histograms

Histograms

Employee(ssn, name, age)

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Eq-depth:

Age:	0-33	33-38	38-43	43-45	45-54	> 54
Tuples	1800	2000	2100	2200	1900	1800

Compressed: store separately highly frequent values: (48,1900)

V-Optimal Histograms

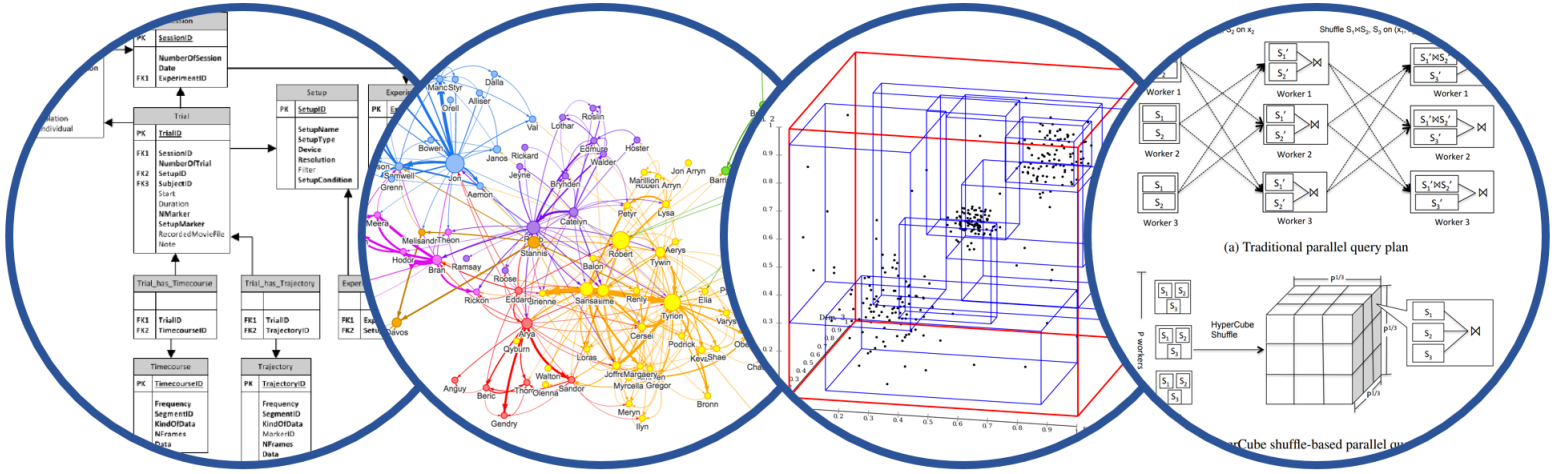
- Defines bucket boundaries in an optimal way, to minimize the error over all point queries
- Computed rather expensively, using dynamic programming
- Modern databases systems use V-optimal histograms or some variations

Difficult Questions on Histograms

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY ?
- *Not* updated during database update, but recomputed periodically
  - WHY ?
- Multidimensional histograms rarely used
  - WHY ?

Difficult Questions on Histograms

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY? All histograms are kept in main memory during query optimization; plus need fast access
- *Not* updated during database update, but recomputed periodically
  - WHY? Histogram update creates a write conflict; would dramatically slow down transaction throughput
- Multidimensional histograms rarely used
  - WHY? Too many possible multidimensional histograms, unclear which ones to choose



Database System Internals

Query Optimization (part 1)

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- Lab 2 part 1 due Today
- Homework 2 due Monday

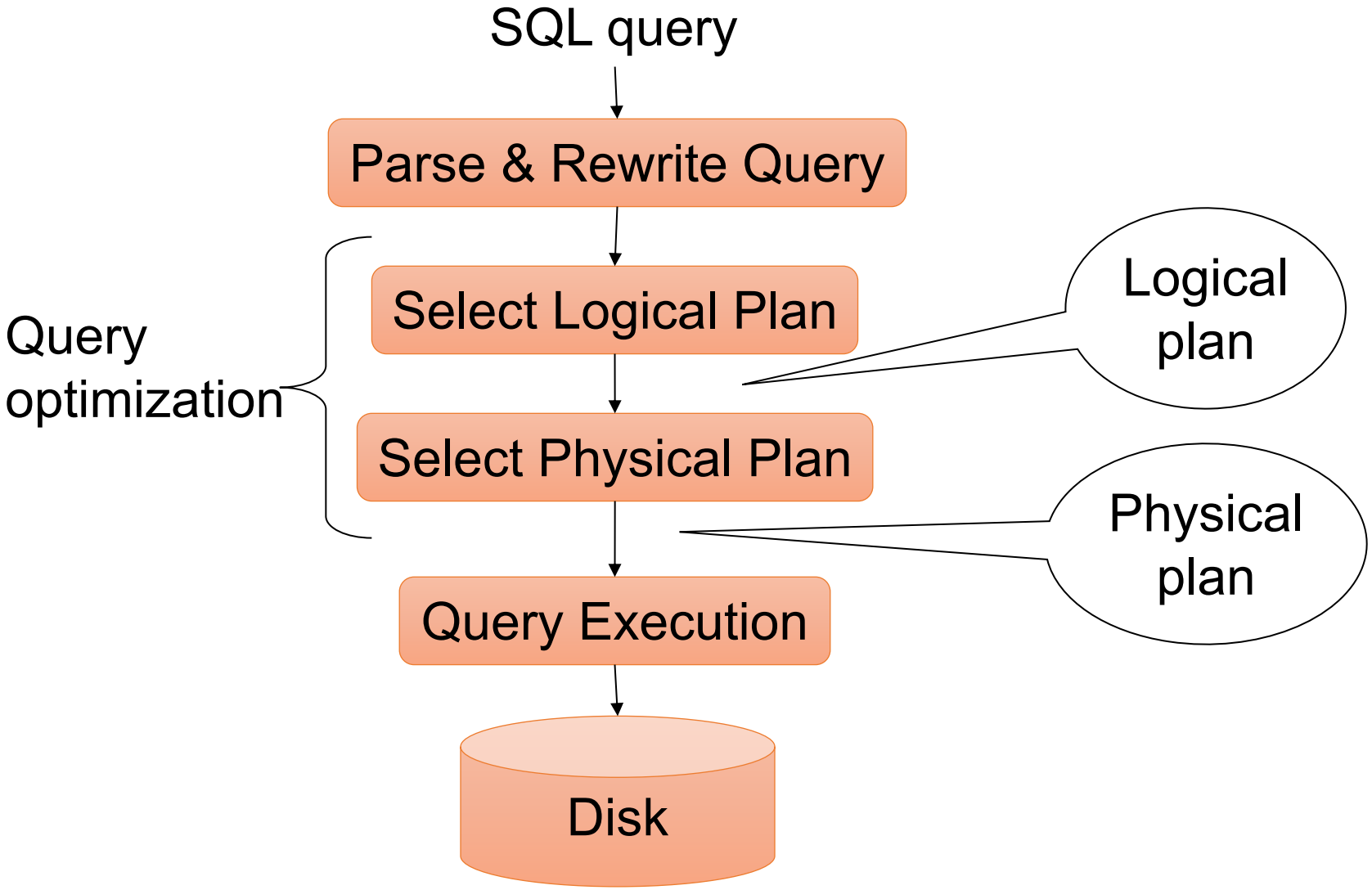
Query Optimization Overview

We know how to compute the cost of a plan

Next: Find a good plan automatically?

This is the role of the query optimizer

Query Optimization Overview



What We Already Know…

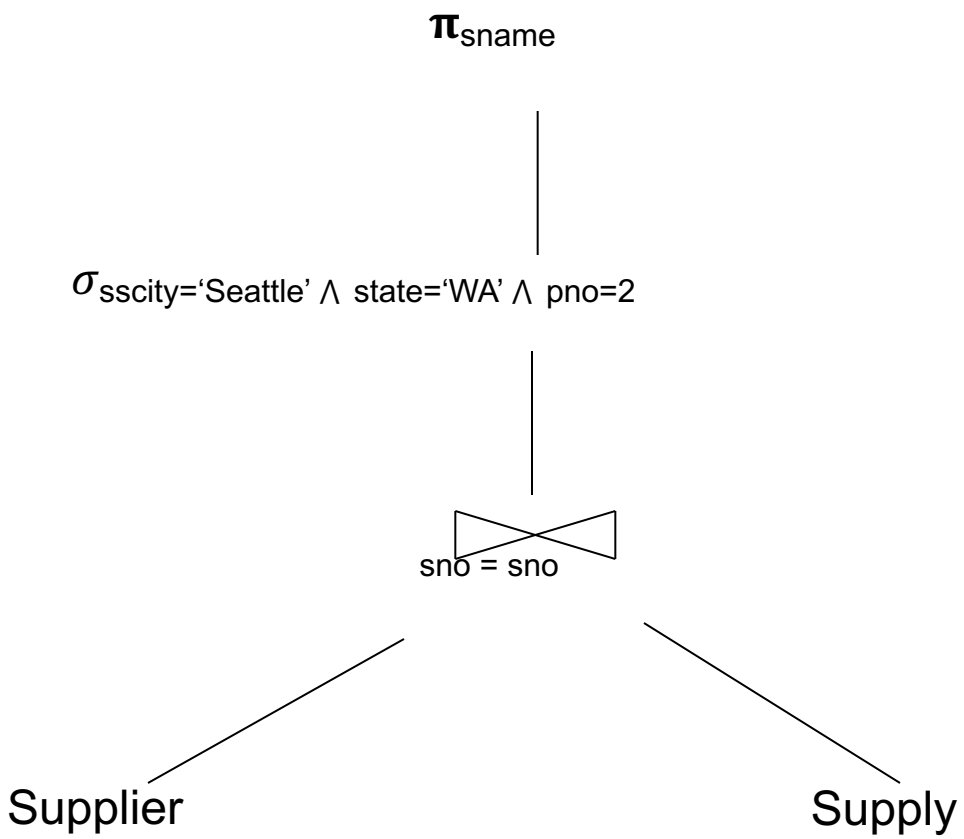
```
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
```

For each SQL query….

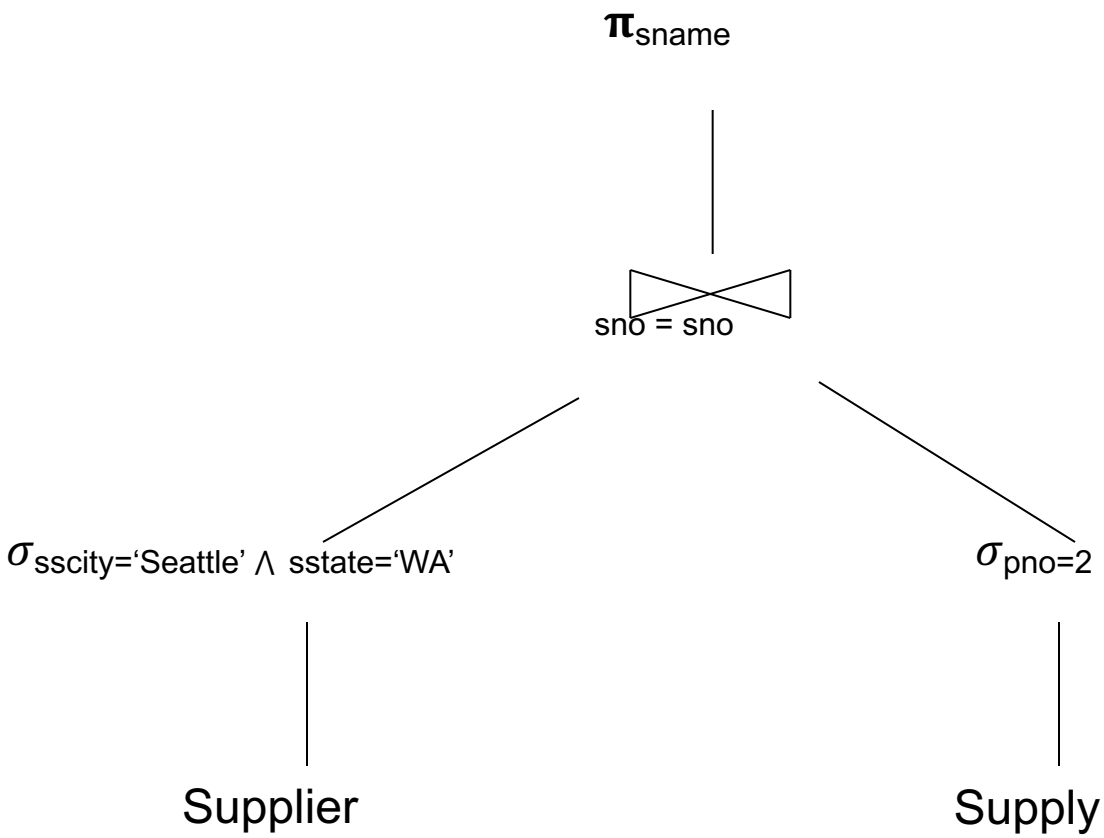
```
SELECT S.sname
FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno
AND U.pno = 2
```

There exist many logical query plans…

Example Query: Logical Plan 1



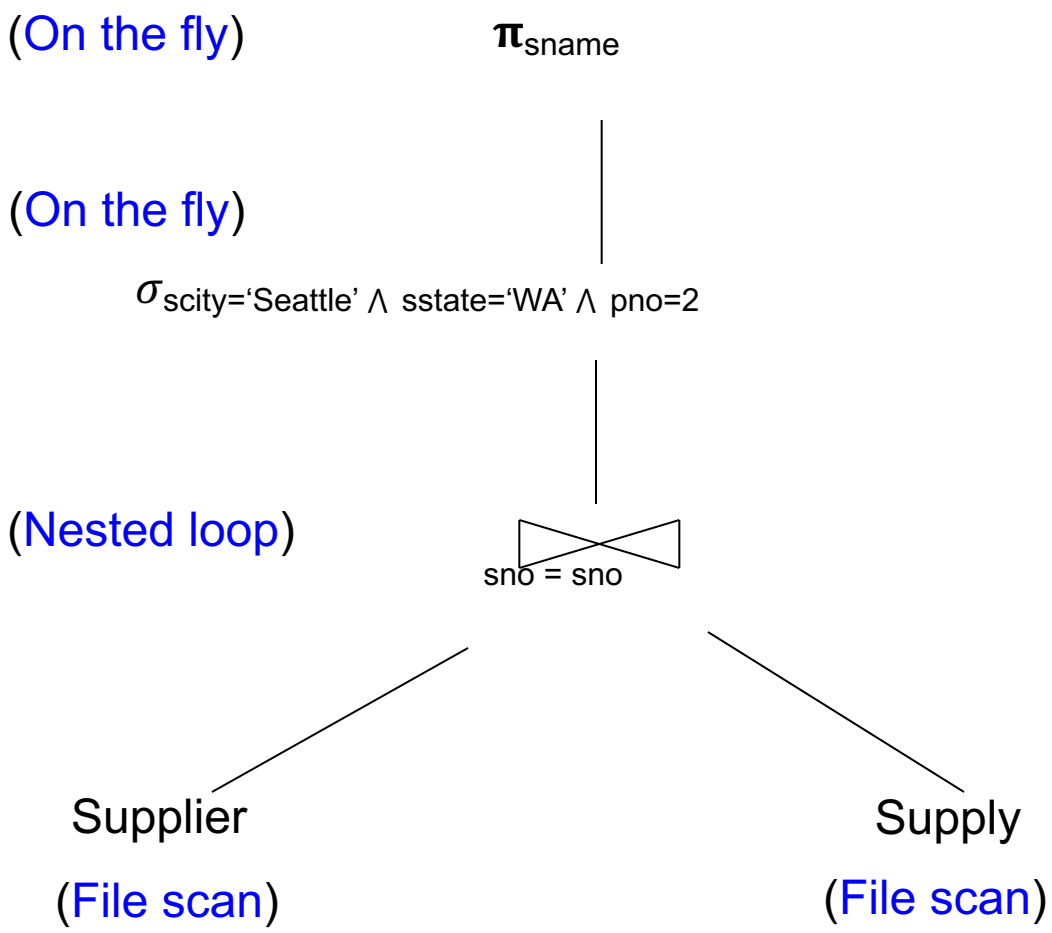
Example Query: Logical Plan 2



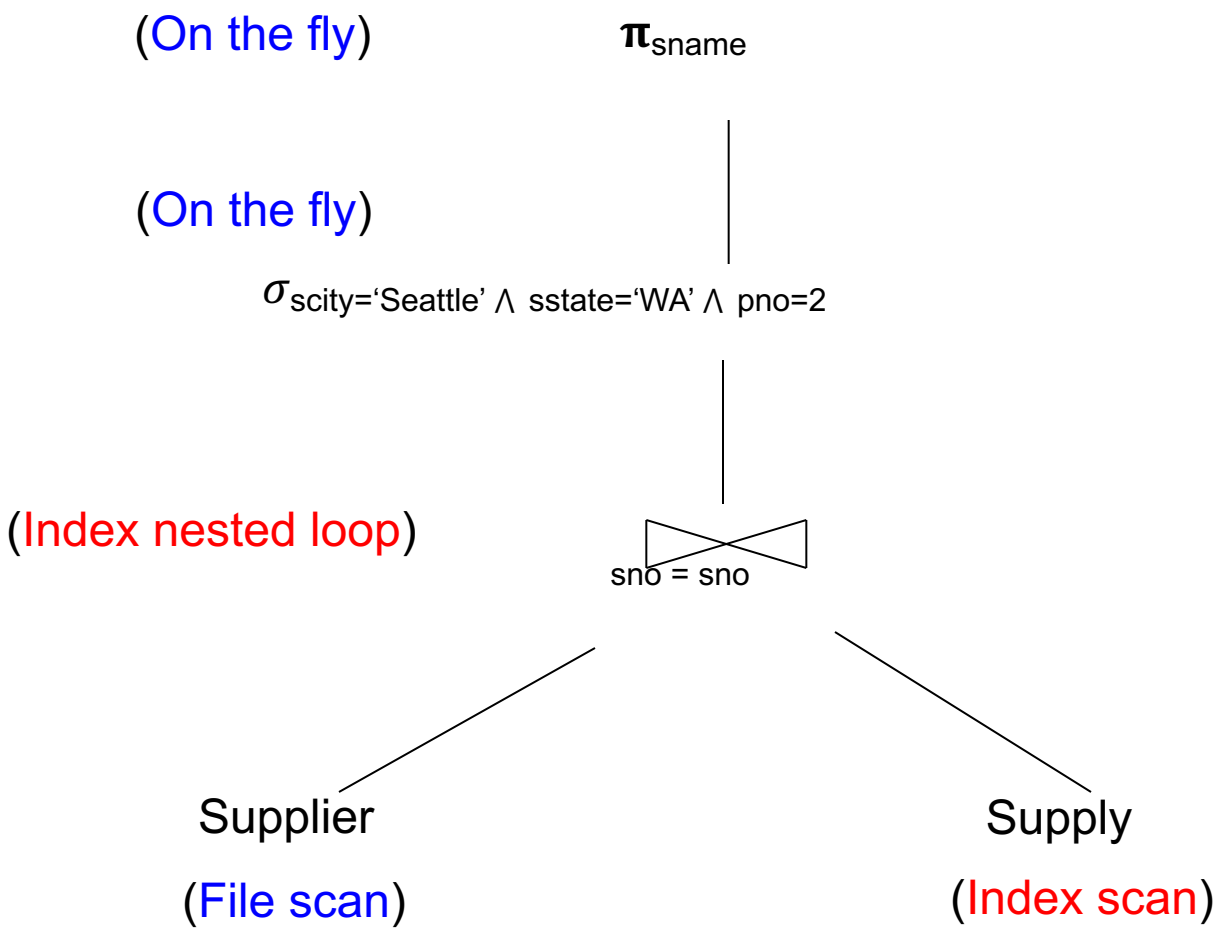
What We Also Know

- For each logical plan…
- There exist many physical plans

Example Query: Physical Plan 1



Example Query: Physical Plan 2



Query Optimizer Overview

- **Input**: A logical query plan
- **Output**: A good physical query plan

Query Optimizer Overview

- **Input**: A logical query plan
- **Output**: A good physical query plan
- **Basic query optimization algorithm**
  - Enumerate alternative plans (logical and physical)
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

Two Types of Optimizers

- Rule-based (heuristic) optimizers:
  - Apply greedily rules that always improve plan
    - 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
  - We focus on cost-based optimizers

Observations

- No magic "best" plan: depends on the data

- In order to make the right choice
  - Need to have ***statistics*** over the data
  - The B's, the T's, the V's
  - Commonly: histograms over base data
    - In SimpleDB as well… lab 5.

Key Decisions for Implementation

Search Space

Optimization rules

Optimization algorithm

Key Decisions for Implementation

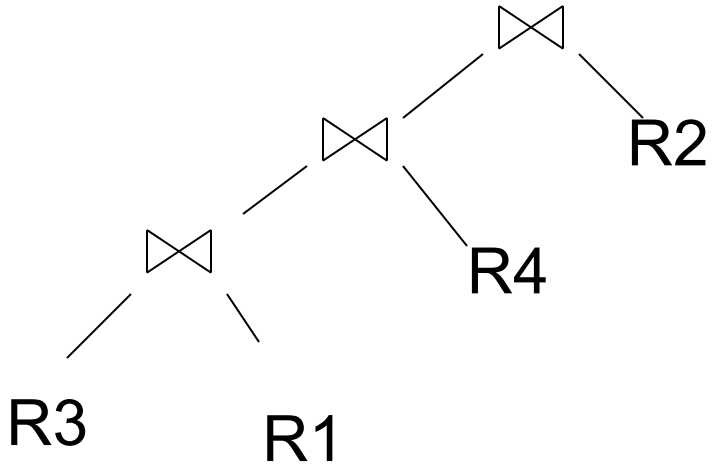
Search Space

What form of plans do we consider?

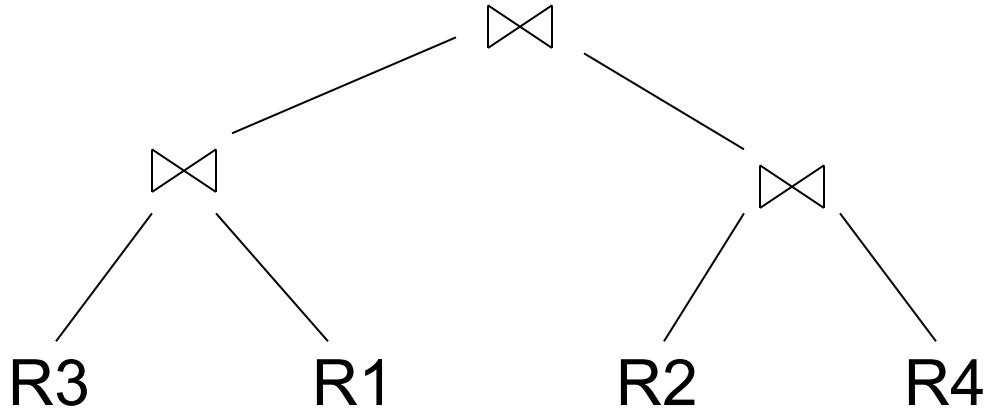
Optimization rules

Optimization algorithm

Search Space – Type of Plan



Left-deep plan



Bushy plan

Linear plan: One input to each join is a relation from disk
Can be either left or right input

Key Decisions for Implementation

Search Space

Optimization rules

Which algebraic laws do we apply?

Optimization algorithm

Optimization Rules – RA equivalencies

- Selections
  - Commutative: $\sigma_{c1}(\sigma_{c2}(R))$ same as $\sigma_{c2}(\sigma_{c1}(R))$
  - Cascading: $\sigma_{c1 \wedge c2}(R)$ same as $\sigma_{c2}(\sigma_{c1}(R))$
- Projections
  - Cascading
- Joins
  - Commutative : $R \bowtie S$ same as $S \bowtie R$
  - Associative: $R \bowtie (S \bowtie T)$ same as $(R \bowtie S) \bowtie T$

Example: Simple Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)

$\sigma_{F=3}(R \bowtie_{D=E} S) =$

$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$

Example: Simple Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)

$\sigma_{F=3}(R \bowtie_{D=E} S) = R \bowtie_{D=E} \sigma_{F=3}(S)$

$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$

Example: Simple Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)

$$\sigma_{F=3}(R \bowtie_{D=E} S) = R \bowtie_{D=E} \sigma_{F=3}(S)$$

$$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) = \sigma_{A=5}(R) \bowtie_{D=E} \sigma_{G=9}(S)$$

Commutativity, Associativity, Distributivity

$$R \cup S = S \cup R, \quad R \cup (S \cup T) = (R \cup S) \cup T$$
$$R \bowtie S = S \bowtie R, \quad R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$

$$R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$$

Laws Involving Selection

$$\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$$
$$\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$$
$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$$

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

Assuming C on attributes of R

Laws Involving Projections

$$\Pi_M(R \bowtie S) = \Pi_M(\Pi_P(R) \bowtie \Pi_Q(S))$$

$$\Pi_M(\Pi_N(R)) = \Pi_M(R)$$

/* note that $M \subseteq N$ */

- Example R(A,B,C,D), S(E, F, G)

$$\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_{?}(R) \bowtie_{D=E} \Pi_{?}(S))$$

Laws Involving Projections

$$\Pi_M(R \bowtie S) = \Pi_M(\Pi_P(R) \bowtie \Pi_Q(S))$$

$$\Pi_M(\Pi_N(R)) = \Pi_M(R)$$

/* note that $M \subseteq N$ */

- Example R(A,B,C,D), S(E, F, G)

$$\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{A,B,G}(\Pi_{A,B,D}(R) \bowtie_{D=E} \Pi_{E,G}(S))$$

Laws for grouping and aggregation

$$\gamma_{A,\,agg(D)}(R(A,B) \bowtie_{B=C} S(C,D)) =$$
$$\gamma_{A,\,agg(D)}(R(A,B) \bowtie_{B=C} (\gamma_{C,\,agg(D)} S(C,D)))$$

Laws for grouping and aggregation

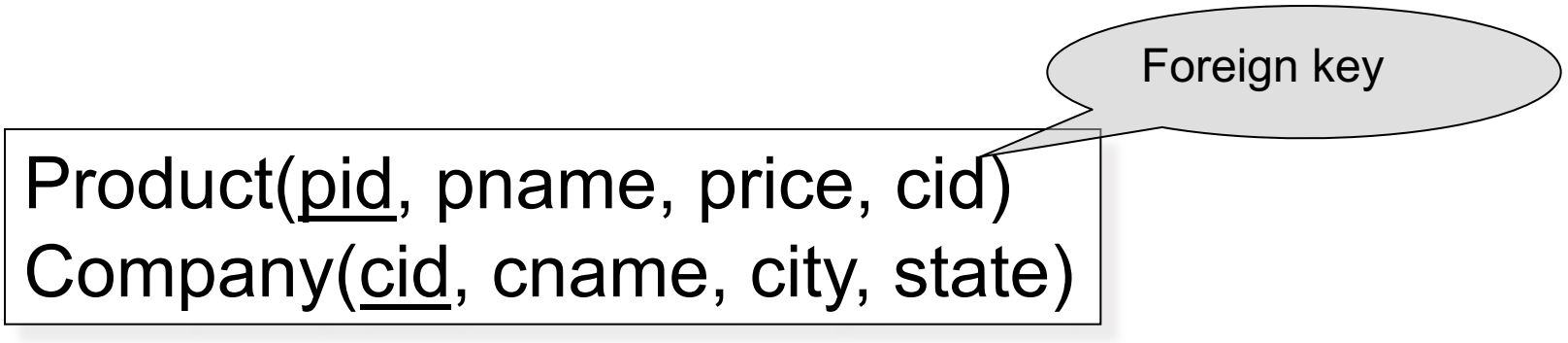
$$\delta(\gamma_{A,\ agg(B)}(R)) = \gamma_{A,\ agg(B)}(R)$$

$$\gamma_{A,\ agg(B)}(\delta(R)) = \gamma_{A,\ agg(B)}(R)$$

if agg is "duplicate insensitive"

Which of the following are "duplicate insensitive" ?
sum, count, avg, min, max

Laws Involving Constraints



$$\Pi_{\text{pid, price}}(\text{Product} \bowtie_{\text{cid=cid}} \text{Company}) = \Pi_{\text{pid, price}}(\text{Product})$$

Search Space Challenges

- Search space is huge!
  - Many possible equivalent trees
  - Many implementations for each operator
  - Many access paths for each relation
    - File scan or index + matching selection condition
- Cannot consider ALL plans
  - Heuristics: only partial plans with "low" cost

Key Decisions

Search Space

Optimization rules

Optimization algorithm

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*

Even More Key Decisions!

Physical plan

- What physical operators to use?
- What access paths to use (file scan or index)?
- Pipeline or materialize intermediate results?

These decisions also affect the *search space*

Two Types of Optimizers

- Heuristic-based optimizers:
  - Apply greedily rules that always improve plan
    - 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
  - We focus on cost-based optimizers

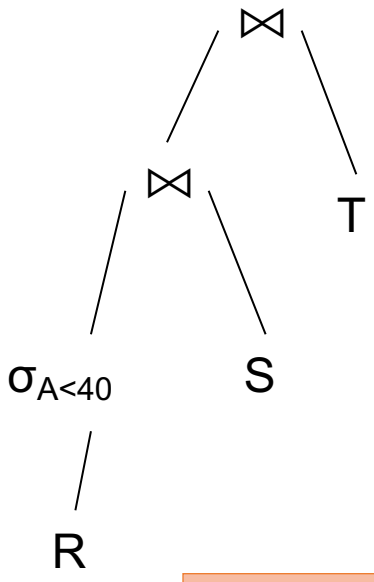
Approaches to Search Space Enumeration

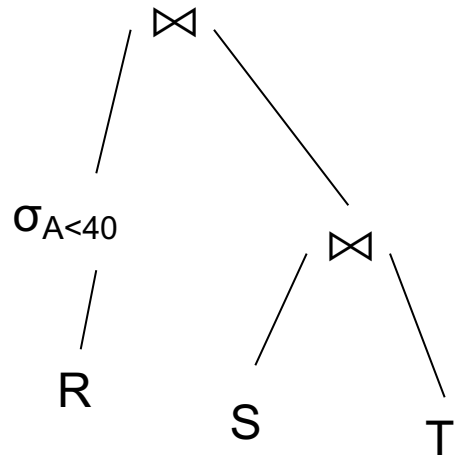
- Complete plans
- Bottom-up plans
- Top-down plans

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 ?

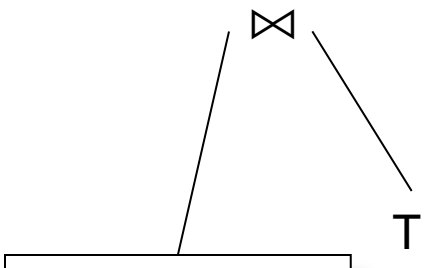
Answer: No way to do early pruning

Top-down Partial Plans

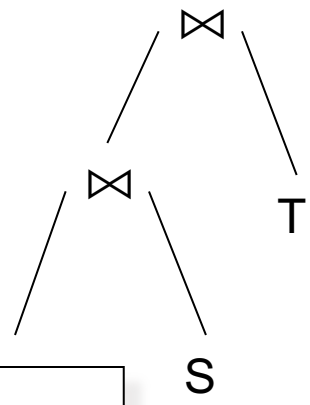
R(A,B)
S(B,C)
T(C,D)

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

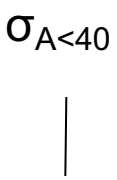
Why is this search space inefficient ?




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




```
SELECT *
FROM R
WHERE R.A < 40
```




```
SELECT R.A, T.D
FROM R, S, T
WHERE R.B=S.B
      and S.C=T.C
```

.....

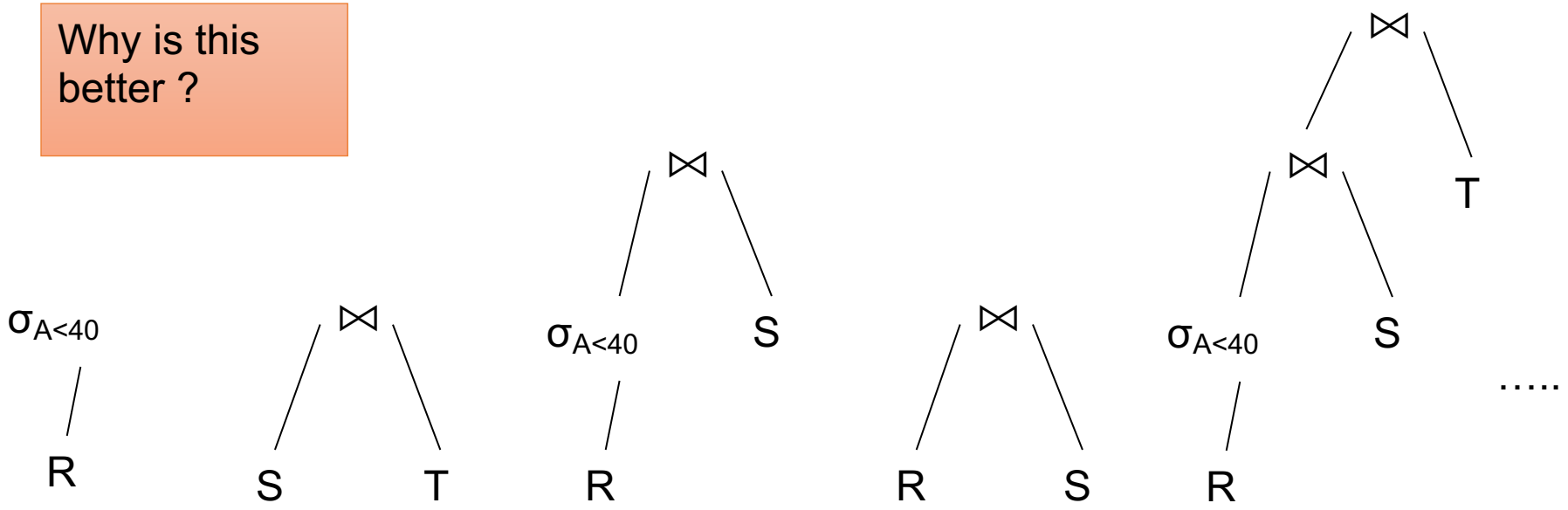
Answer: Can't compute costs of a plan on SQL text alone

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 ?

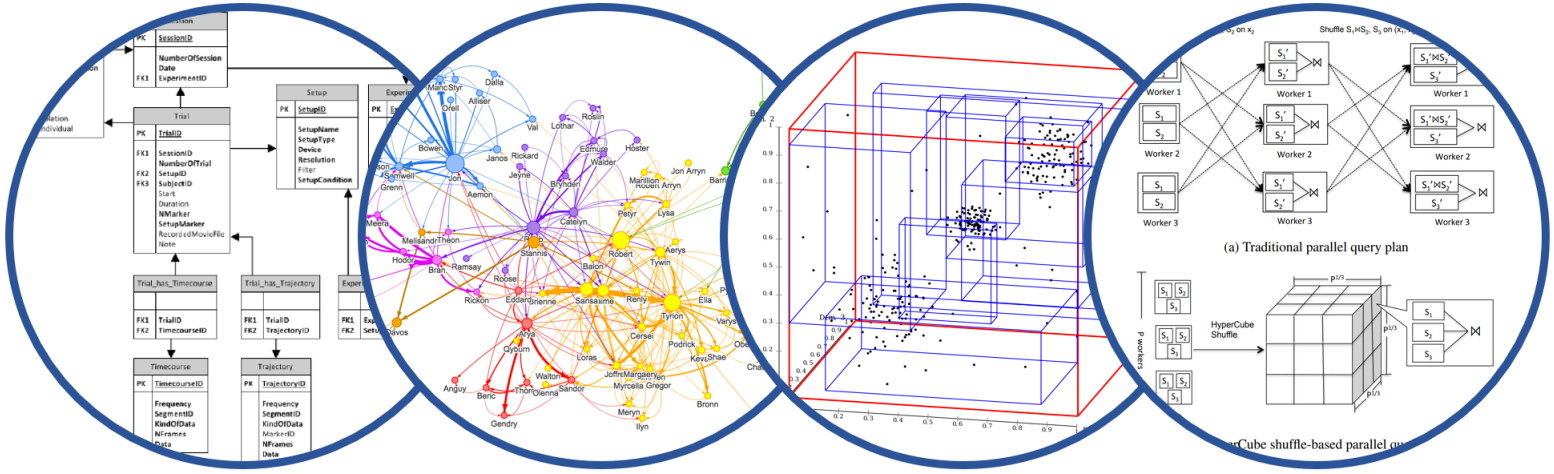


We will prune bad plans for sub-expressions

Two Types of Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Based on System R (aka Selinger) style optimizer[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



Database System Internals

Query Optimization (part 2)

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

- HW 2 due tonight
- Lab 1 grades and feedback back later this week
- Quiz 1+2 on Feb. 12th
  - Not as long as a midterm
  - Examples will be posted on webpage

Two Types of Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Based on System R (aka Selinger) style optimizer[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

Two Types of Optimizers

- Rule-based (heuristic) optimizers:
  - Apply greedily rules that always improve plan
    - 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
  - We focus on cost-based optimizers

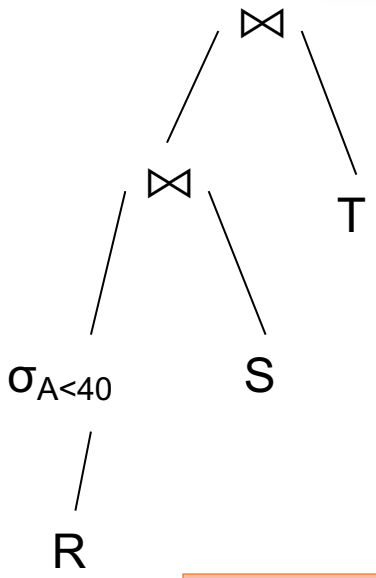
The Three Parts of an Optimizer

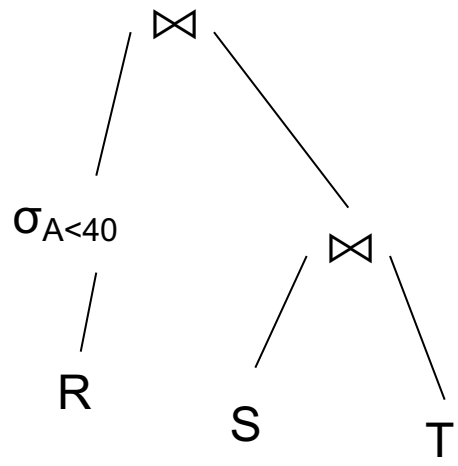
- Cost estimation
  - Based on cardinality estimation
- Search space
- Search algorithm

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 ?

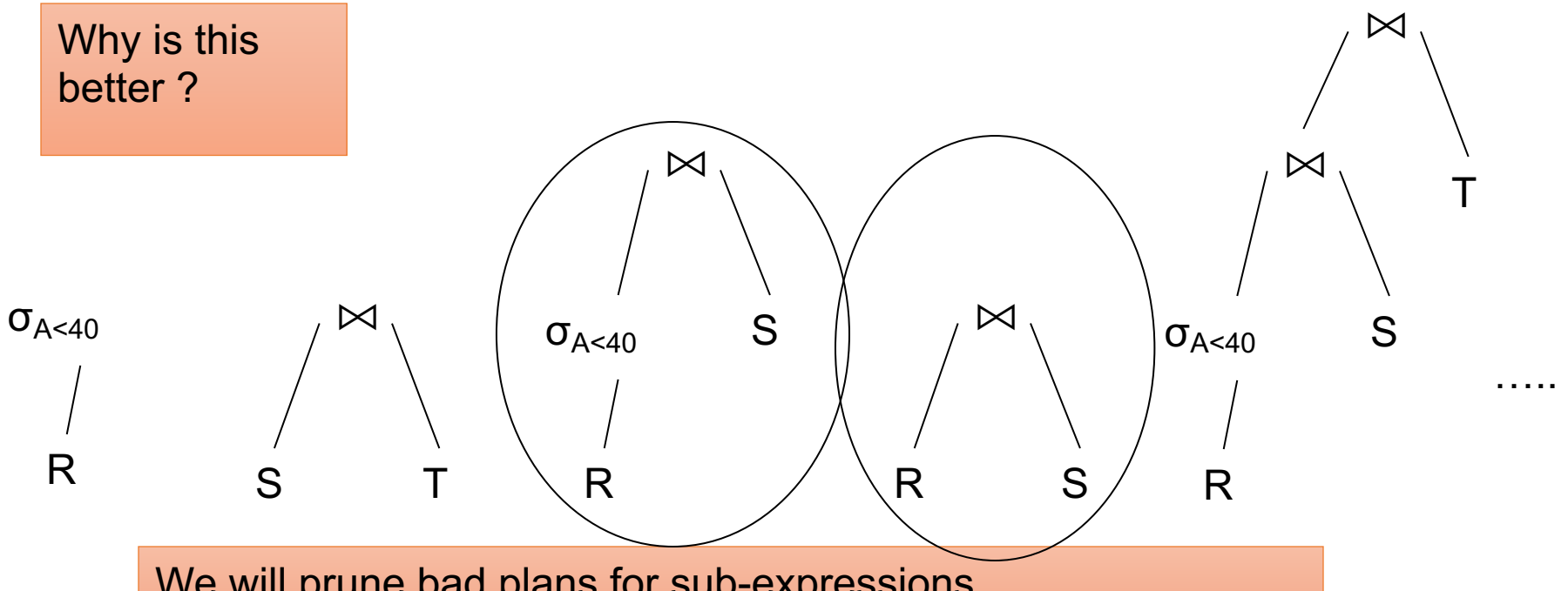
Answer: No way to do early pruning

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 ?

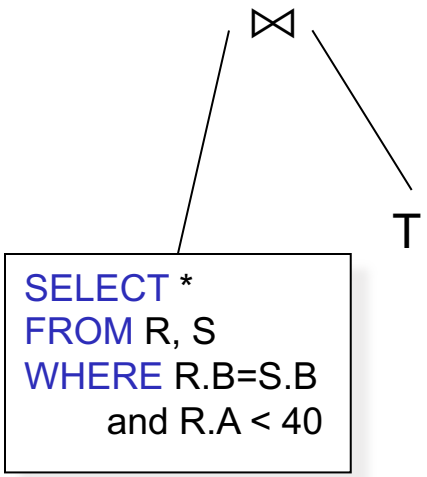


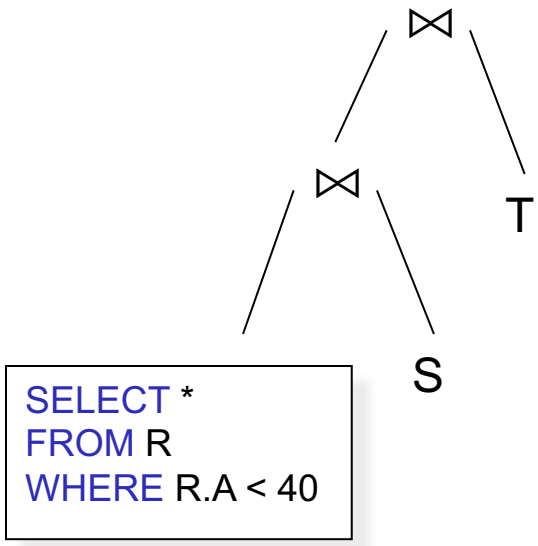
We will prune bad plans for sub-expressions

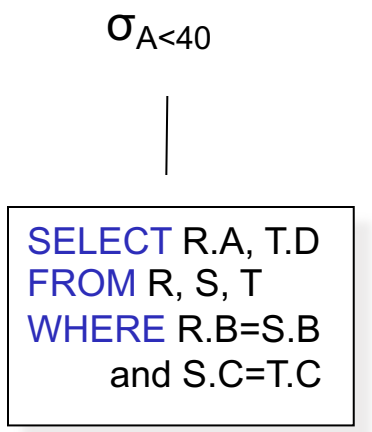
Top-down Partial Plans

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
```







.....

Query Optimizer Overview

- **Input**: A logical query plan
- **Output**: A good physical query plan
- **Basic query optimization algorithm**
  - Enumerate alternative plans (logical and physical)
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

The Three Parts of an Optimizer

- Cost estimation
  - Based on cardinality estimation
- Search space
  - Algebraic laws, restricted types of join trees
- Search algorithm
  - Will discuss next

Search Algorithm

- Dynamic programming (in class)
  - Based on System R (aka Selinger) style optimizer[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
```

- Some heuristics for search space enumeration:
  - Selections down
  - Projections up
  - Avoid cartesian products

Dynamic Programming

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

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

Dynamic Programming

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

- **Step 1**: For each $\{R_i\}$ do:
  - $T(\{R_i\}) = T(R_i)$
  - $Plan(\{R_i\})$ = access method for R_i
  - $Cost(\{R_i\})$ = cost of access method for R_i

Dynamic Programming

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

- **Step 2**: For each Q $\subseteq \{R_1, \ldots, R_n\}$ of size k do:
  - T(Q) = use estimator
  - Consider all partitions Q = Q' ∪ Q'' compute cost(Plan(Q') ⋈ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

- Note
  - If we restrict to left-linear trees: Q'' = single relation
  - May want to avoid cartesian products

Dynamic Programming

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

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

Example

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

- $R \bowtie S \bowtie T \bowtie U$
- Assumptions:

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

- Every join selectivity is 0.001

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000		
S	5000		
T	3000		
U	1000		
RS			
RT			
RU			
ST			
SU			
TU			
RST			
RSU			
RTU			
STU			
RSTU			

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000		
S	5000		
T	3000		
U	1000		
RS	10000		
RT	6000		
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000		
T	3000		
U	1000		
RS	10000		
RT	6000		
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000		
U	1000		
RS	10000		
RT	6000		
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	Unclustered index scan U.F	1000
RS	10000		
RT	6000		
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	Unclustered index scan U.F	1000
RS	10000	R ⋈ S nested loop join	…
RT	6000		
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	**Unclustered index scan U.F**	1000
RS	10000	R ⋈ S nested loop join	…
RT	6000	**R ⋈ T index** join	…
RU	2000		
ST	15000		
SU	5000		
TU	3000		
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	Unclustered index scan U.F	1000
RS	10000	R ⋈ S nested loop join	…
RT	6000	R ⋈ T index join	…
RU	2000	R ⋈ U index join	
ST	15000	S ⋈ T hash join	
SU	5000	. . .	
TU	3000	. . .	
RST	30000		
RSU	10000		
RTU	6000		
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	Unclustered index scan U.F	1000
RS	10000	R ⋈ S nested loop join	…
RT	6000	R ⋈ T index join	…
RU	2000	R ⋈ U index join	
ST	15000	S ⋈ T hash join	
SU	5000	. . .	
TU	3000	. . .	
RST	30000	(RT) ⋈ S hash join	…
RSU	10000	(SU) ⋈ R merge join	
RTU	6000	…	
STU	15000		
RSTU	30000		

Example

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

Assume
B(..) = T(..)/10

Join selectivity is 0.001

Subquery	T	Plan	Cost
R	2000	Clustered index scan R.A	200
S	5000	Table scan	500
T	3000	Table scan	300
U	1000	Unclustered index scan U.F	1000
RS	10000	R ⋈ S nested loop join	…
RT	6000	R ⋈ T index join	…
RU	2000	R ⋈ U index join	
ST	15000	S ⋈ T hash join	
SU	5000	. . .	
TU	3000	. . .	
RST	30000	(RT) ⋈ S hash join	…
RSU	10000	(SU) ⋈ R merge join	
RTU	6000	…	
STU	15000		
RSTU	30000	(RT) ⋈ (SU) hash join	…

Discussion

- For the subset {RS}, need to consider both

  $R \bowtie S$ and $S \bowtie R$

  - Because the cost may be different!

- When computing the cheapest plan for

  $(Q) \bowtie R$, we may consider new access methods for R, e.g. an index look-up that makes sense only in the context of the join

Discussion

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

Given a query with n relations R1, …, Rn

- How many entries do we have in the dynamic programming table?

- For each entry, how many alternative plans do we need to inspect?

Discussion

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

Given a query with n relations R1, …, Rn

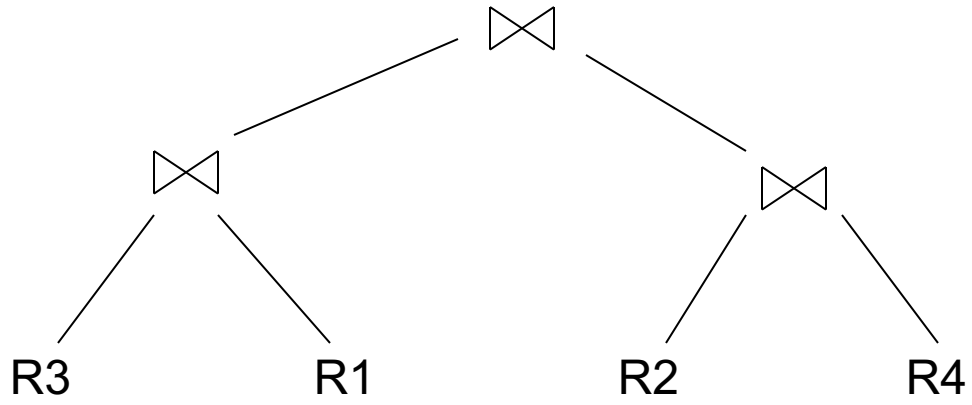
- How many entries do we have in the dynamic programming table?
  - A: $2^n - 1$
- For each entry, how many alternative plans do we need to inspect?
  - A: for each entry with k tables, examine $2^k - 2$ plans

Reducing the Search Space

- Left-linear trees
- No cartesian products

Join Trees

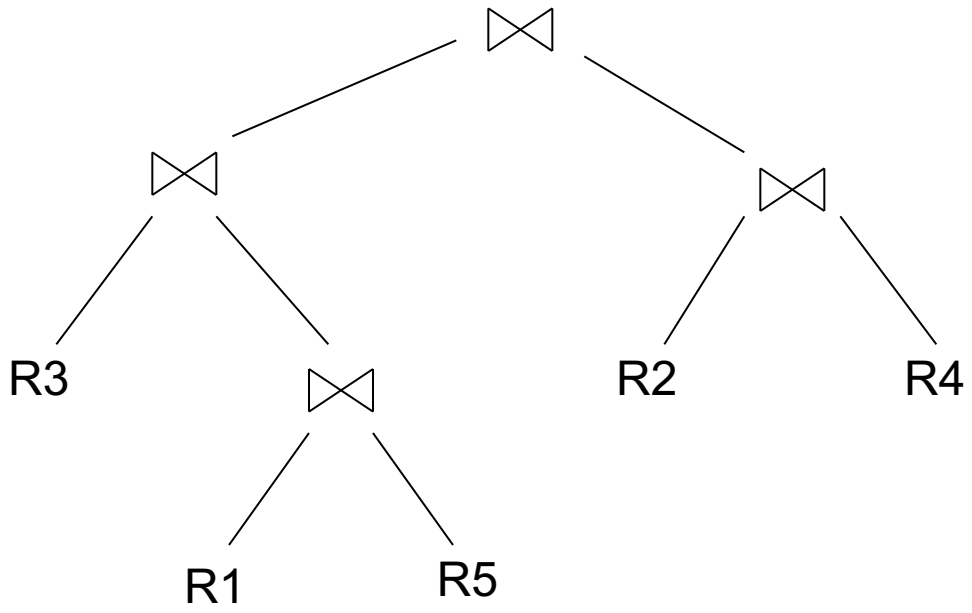
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



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

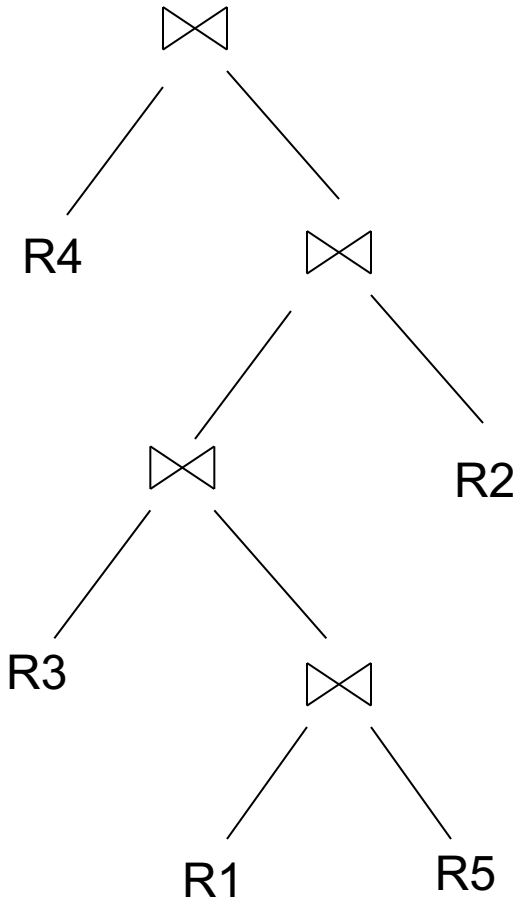
Types of Join Trees

- Bushy:



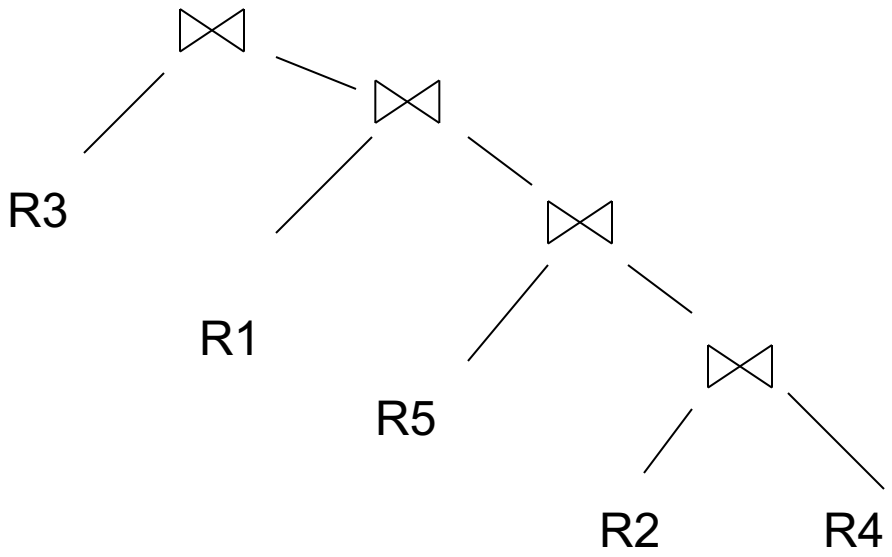
Types of Join Trees

- Linear :



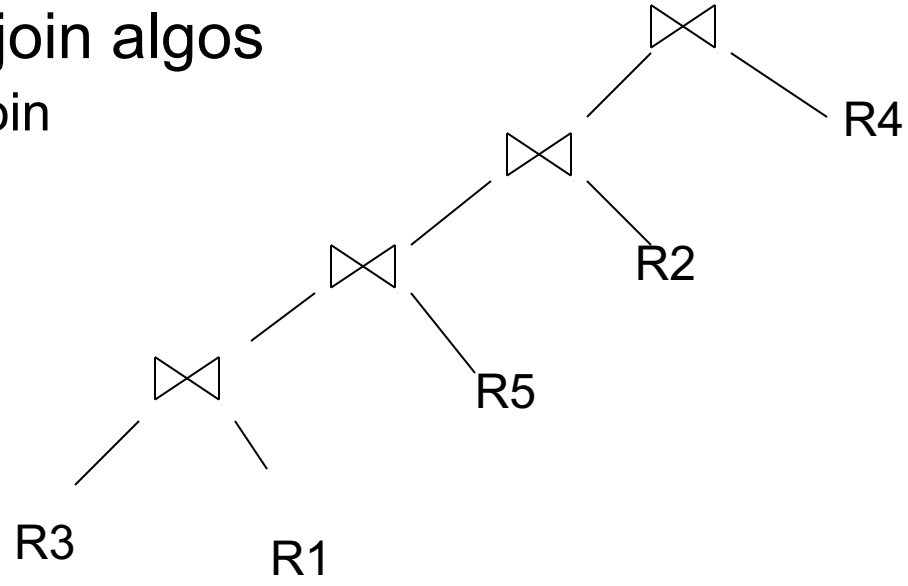
Types of Join Trees

- Right deep:



Types of Join Trees

- Left deep:
  - Work well with existing join algos
    - Nested-loop and hash-join
  - Facilitate pipelining

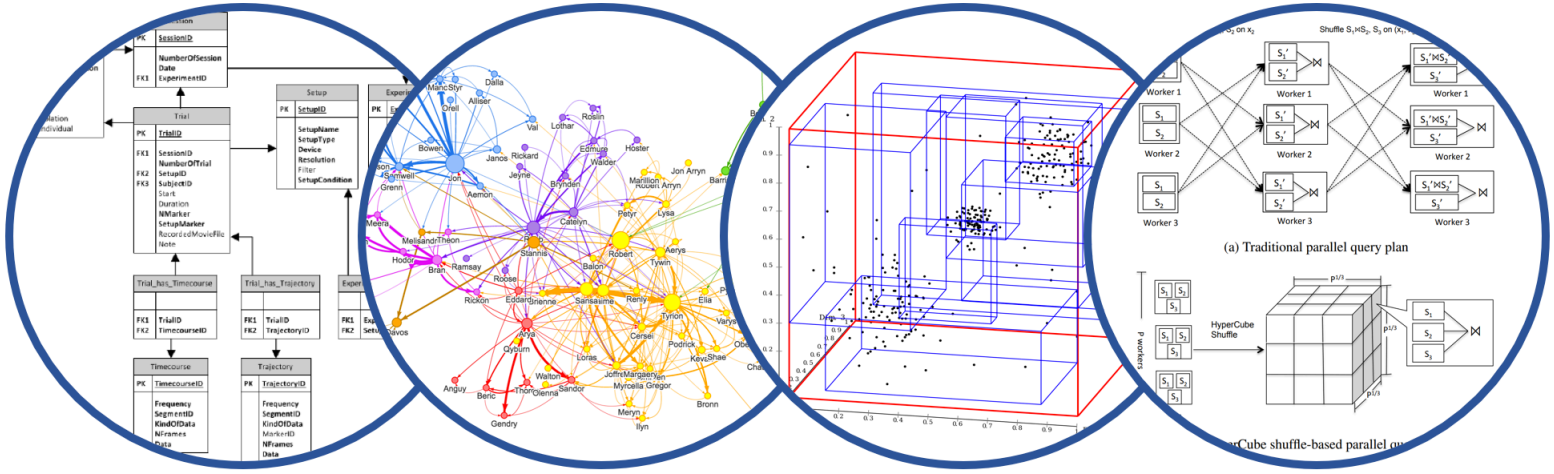


  - Dynamic programming can be used with all trees

No Cartesian Products

$$R(A,B) \bowtie S(B,C) \bowtie T(C,D)$$

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



Database System Internals

Query Optimization (part 3)

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Selinger Optimizer History

- 1960's: first database systems
  - Use tree and graph data models
- 1970: Ted Codd proposes relational model
  - E.F. Codd. A relational model of data for large shared data banks. Communications of the ACM, 1970
- 1974: System R from IBM Research
  - One of first systems to implement relational model
- 1979: Seminal query optimizer paper by P. Selinger et. al.
  - Invented cost-based query optimization
  - Dynamic programming algorithm for join order computation

References

- P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. Proceedings of ACM SIGMOD, **1979**. Pages 22-34.

Selinger Algorithm

Selinger enumeration algorithm considers

- Different logical and physical plans *at the same time*
- Cost of a plan is IO + CPU
- Concept of *interesting order* during plan enumeration
  - A sorted order as that requested by ORDER BY or GROUP GY
  - Or order on attributes that appear in equi-join predicates
    - Because they may enable cheaper sort-merge joins later

More about the Selinger Algorithm

- Step 1: Enumerate all access paths for a single relation
  - File scan or index scan
  - Keep the cheapest for each *interesting order*
- Step 2: Consider all ways to join two relations
  - Use result from step 1 as the outer relation
  - Consider every other possible relation as inner relation
  - Estimate cost when using sort-merge or nested-loop join
  - Keep the cheapest for each *interesting order*
- Steps 3 and later: Repeat for three relations, etc.

Example From Selinger Paper

EMP

NAME	DNO	JOB	SAL
SMITH	50	12	8500
JONES	50	5	15000
DOE	51	5	9500

DEPT

DNO	DNAME	LOC
50	MFG	DENVER
51	BILLING	BOULDER
52	SHIPPING	DENVER

JOB

JOB	TITLE
5	CLERK
6	TYPIST
8	SALES
12	MECHANIC

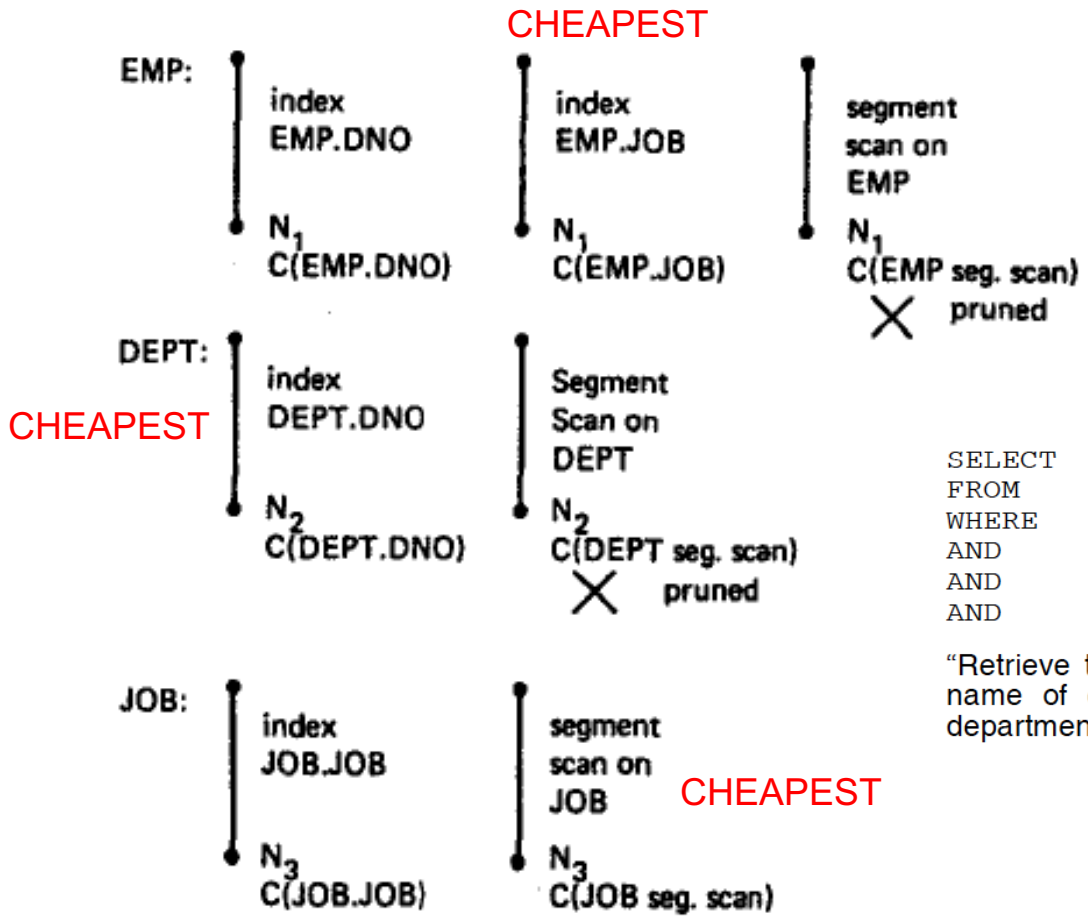
```
SELECT   NAME,TITLE,SAL,DNAME
FROM     EMP,DEPT,JOB
WHERE    TITLE='CLERK'
AND      LOC='DENVER'
AND      EMP.DNO=DEPT.DNO
AND      EMP.JOB=JOB.JOB
```

"Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver."

Figure 1. JOIN example

Step1: Access Path Selection for Single Relations

- Eligible Predicates: Local Predicates Only
- "Interesting" Orderings: DNO, JOB



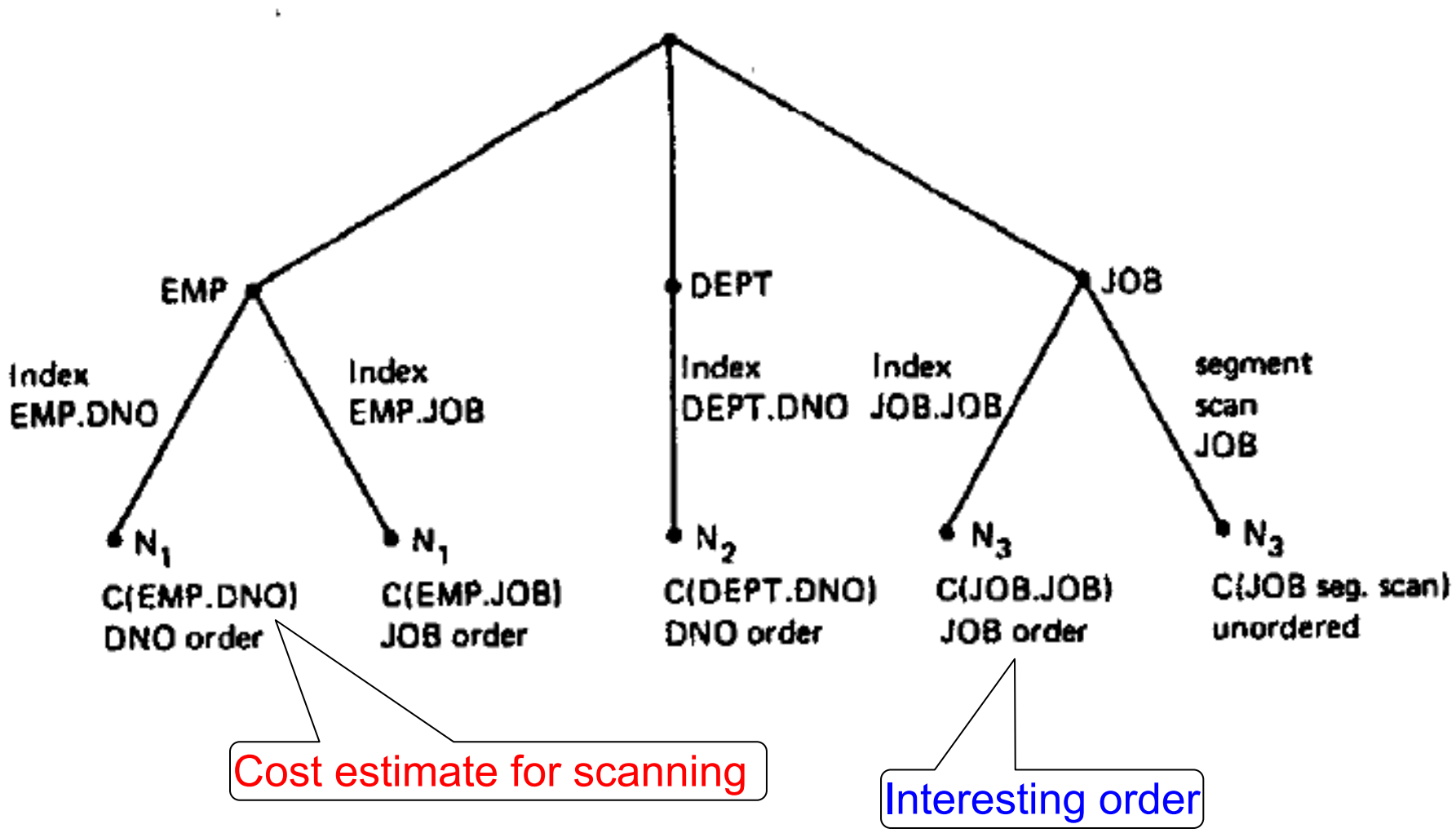

```
SELECT   NAME,TITLE,SAL,DNAME
FROM     EMP,DEPT,JOB
WHERE    TITLE='CLERK'
AND      LOC='DENVER'
AND      EMP.DNO=DEPT.DNO
AND      EMP.JOB=JOB.JOB
```

"Retrieve the name, salary, job title, and department name of employees who are clerks and work for departments in Denver."

Figure 1. JOIN example

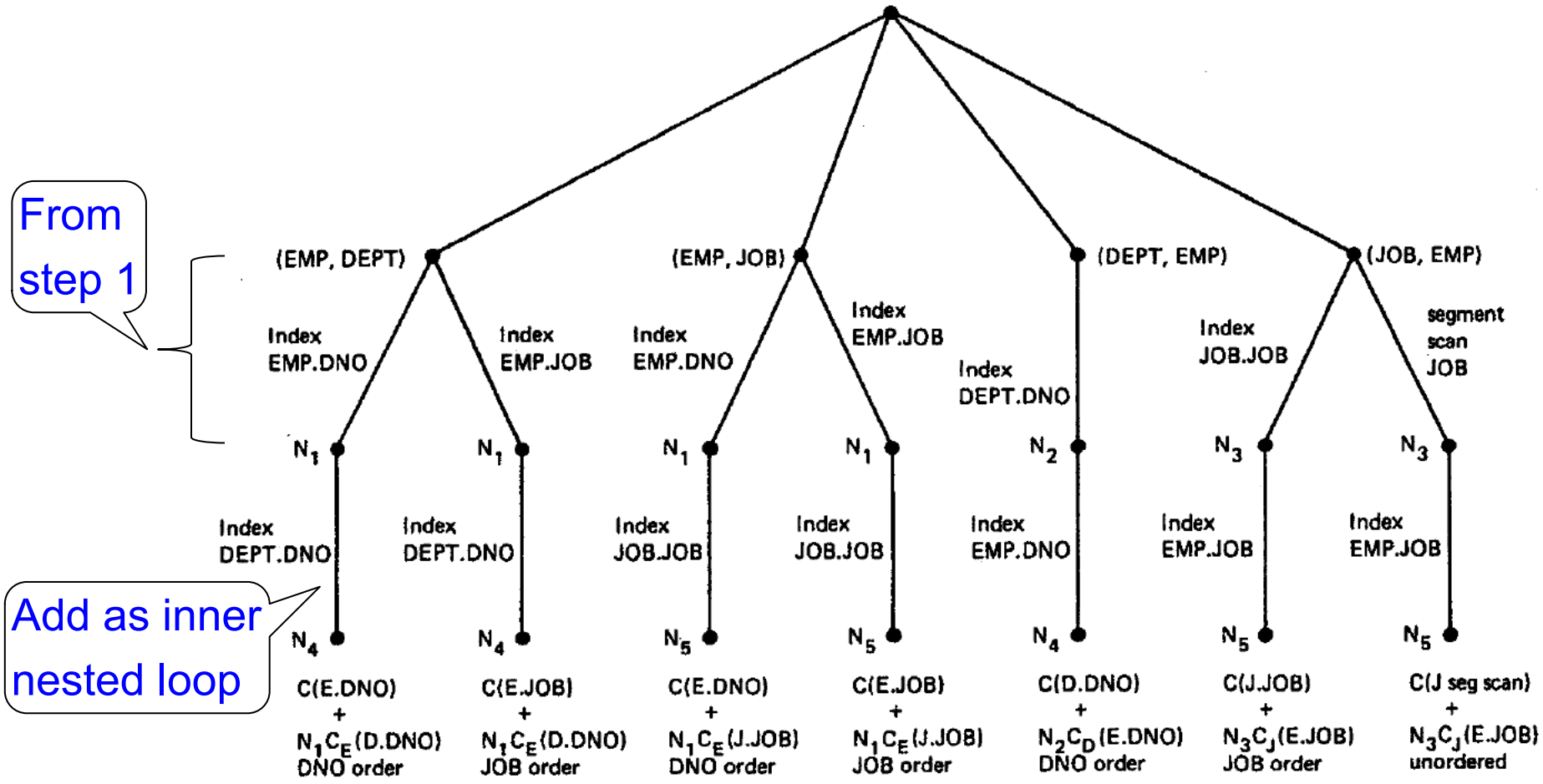
SELECT NAME, TITLE, SAL, DNAME
FROM EMP, DEPT, JOB
WHERE TITLE='CLERK' AND LOC='DENVER' AND EMP.DNO=DEPT.DNO AND EMP.JOB=JOB.JOB

Step1: Resulting Plan Search Tree for Single Relations



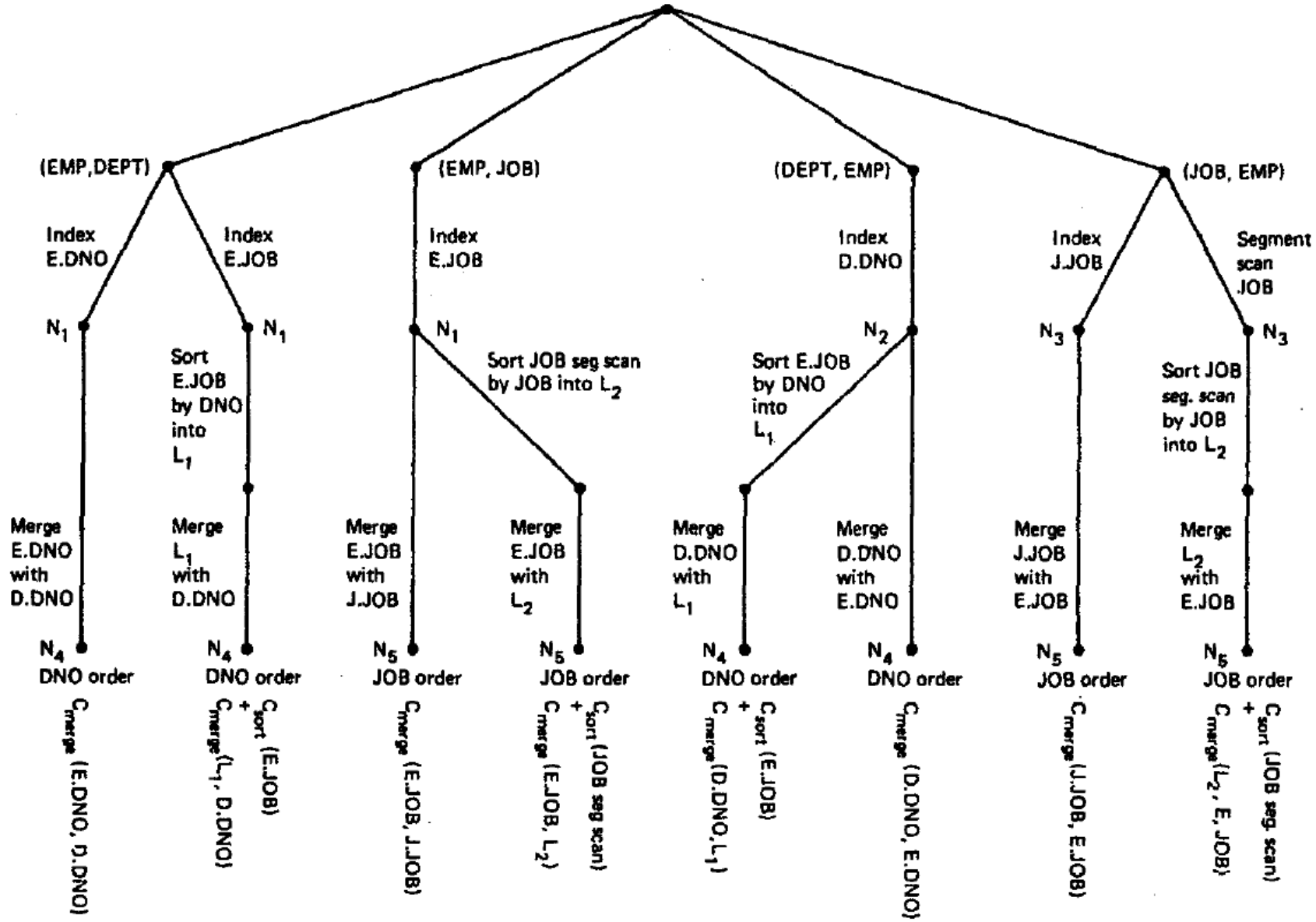
SELECT NAME, TITLE, SAL, DNAME
FROM EMP, DEPT, JOB
WHERE TITLE='CLERK' AND LOC='DENVER' AND EMP.DNO=DEPT.DNO AND EMP.JOB=JOB.JOB

Step2: Pairs of Relations (nested loop joins)

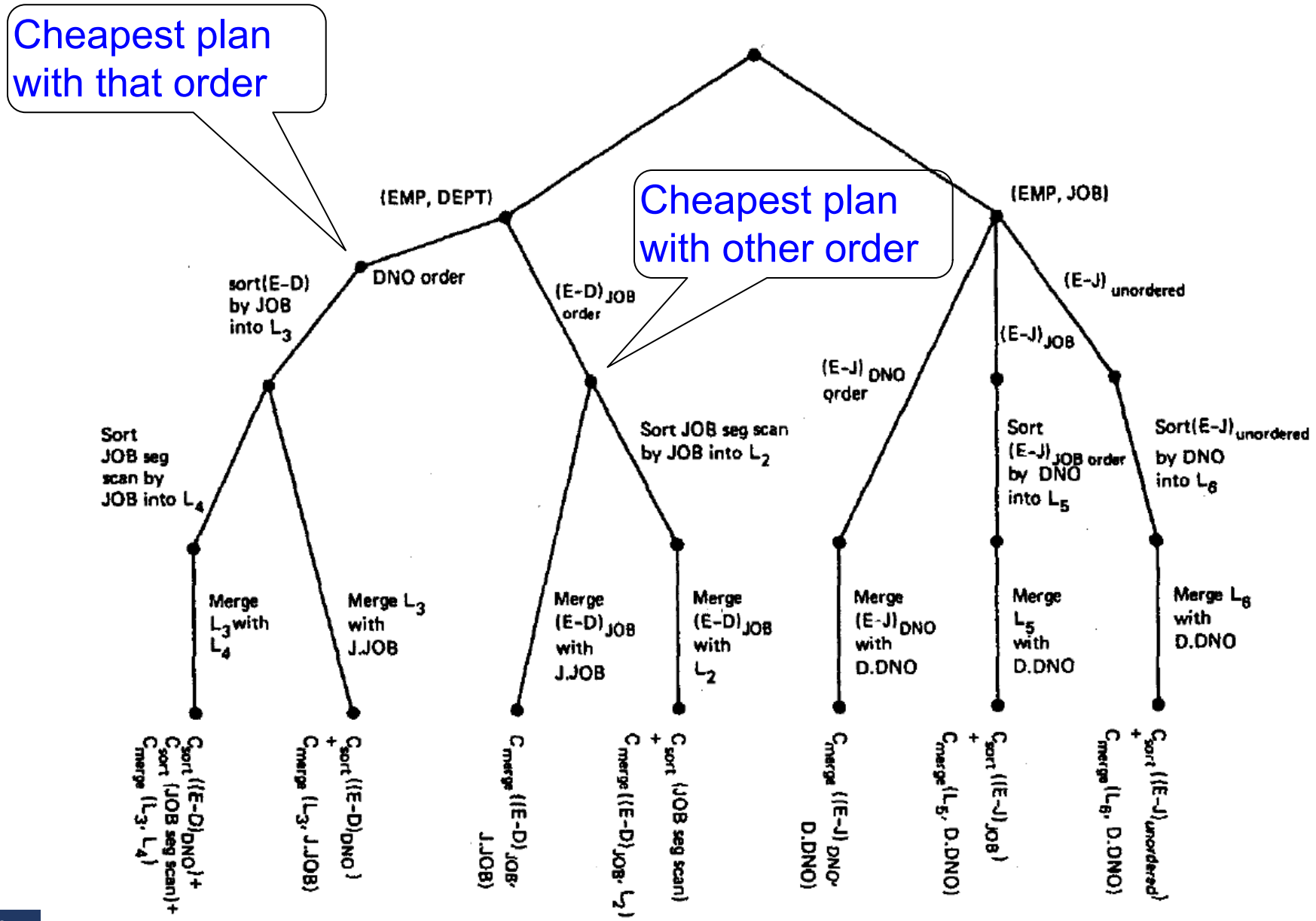


SELECT NAME, TITLE, SAL, DNAME
FROM EMP, DEPT, JOB
WHERE TITLE='CLERK' AND LOC='DENVER' AND EMP.DNO=DEPT.DNO AND EMP.JOB=JOB.JOB

Step2: Pairs of Relations (sort-merge joins)



Step3:Add Third Relation (sort-merge join)



Next Example Acks

Implement variant of Selinger optimizer in SimpleDB

Designed to help you understand how this would work in SimpleDB

Many following slides from Sam Madden at MIT

Selinger Optimizer

Problem:

- How to order a series of joins over N tables A,B,C,...
  E.g. A.a = B.b AND A.c = D.d AND B.e = C.f

- N! ways to order joins; e.g. ABCD, ACBD, ...

- $C_{N-1} = \frac{1}{N}\binom{2(N-1)}{N-1}$ plans/ordering; e.g. (((AB)C)D),((AB)(CD)))

- Multiple implementations (hash, nested loops)

- Naïve approach does not scale
  - E.g. N = 20, #join orders 20! = 2.4 x 10^{18} ; many more plans

Selinger Optimizer

- Only left-deep plan: (((AB)C)D) – eliminate C_{N-1}.
- Push down selections
- Don't consider cartesian products
- Dynamic programming algorithm

Dynamic Programming

OrderJoins(…):
R = set of relations to join
For d = 1 to N: /* where N = |R| */
  For S in {all size-d subsets of R}:
    optjoin(S) = (S – a) join a,
      where a is the single relation that minimizes:
        cost(optjoin(S – a)) +
        min.cost to join (S – a) with a +
        min.access cost for a

Note: optjoin(S-a) is cached from previous iterations

SimpleDB Lab5:
you implement orderJoins

Use: enumerateSubsets

Use:
computeCostAndCardOfSubplan

Example

- **orderJoins(A, B, C, D)**
- Assume all joins are Nested Loop

Subplan S	optJoin(S)	Cost(OptJoin(S))
A		

Example

- **orderJoins(A, B, C, D)**
- Assume all joins are NL
- d = 1
  - A = best way to access A (sequential scan, predicate-pushdown on index, etc)
  - B = best way to access B
  - C = best way to access C
  - D = best way to access D
- Total number of steps: **choose(N, 1)**

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
C	Seq scan	120
D	B+tree scan	400

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156
{B, C}	BC	98

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156
{B, C}	BC	98

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB
  - {C,D} = CD or DC
  - {A,C} = AC or CA
  - {B,D} = BD or DB
  - {A,D} = AD or DA

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156
{B, C}	BC	98
……..		

Example

- **orderJoins(A, B, C, D)**

- d = 2
  - {A,B} = AB or BA
    use previously computed
    best way to access A and B
  - {B,C} = BC or CB
  - {C,D} = CD or DC
  - {A,C} = AC or CA
  - {B,D} = BD or DB
  - {A,D} = AD or DA
- Total number of steps: choose(N, 2) × 2

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
…		
{A, B}	BA	156
{B, C}	BC	98
……..		

Example

- **orderJoins(A, B, C, D)**

- d = 3

  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
……..		

Example

- **orderJoins(A, B, C, D)**

- d = 3

  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
……..		

optJoin(B,C) and its cost are already cached in table

Example

- **orderJoins(A, B, C, D)**

- d = 3

  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
……..		

optJoin(B,C) and its cost are already cached in table

Example

- **orderJoins(A, B, C, D)**

- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
……..		

optJoin(B,C) and its cost are already cached in table

Example

- **orderJoins(A, B, C, D)**

- d = 3
  - {A,B,C} =
    Remove A: compare A({B,C}) to ({B,C})A
    Remove B: compare B({A,C}) to ({A,C})B
    Remove C: compare C({A,B}) to ({A,B})C
  - {A,B,D} =
    Remove A: compare A({B,D}) to ({B,D})A
    …
  - {A,C,D} =…
  - {B,C,D} =…
- Total number of steps: choose(N, 3) × 3 × 2

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
….		
{A, B}	BA	156
{B, C}	BC	98
….		
{A, B, C}	BAC	500
……..		

optJoin(B,C) and its cost are already cached in table

Example

- **orderJoins(A, B, C, D)**

Subplan S	optJoin(S)	Cost(OptJoin(S))
A	Index scan	100
B	Seq. scan	50
{A, B}	BA	156
{B, C}	BC	98
{A, B, C}	BAC	500
{B, C, D}	DBC	150
……		

optJoin(B, C, D) and its cost are already cached in table

- d = 4
  - {A,B,C,D} =

Remove A: compare A({B,C,D}) to ({B,C,D})A
Remove B: compare B({A,C,D}) to ({A,C,D})B
Remove C: compare C({A,B,D}) to ({A,B,D})C
Remove D: compare D({A,B,C}) to ({A,B,C})D

- Total number of steps: choose(N, 4) × 4 × 2

Interesting Orders

- Some query plans produce data in sorted order
  - E.g scan over a primary index, merge-join
  - Called *interesting order*
- Next operator may use this order
  - E.g. can be another merge-join
- For each subset of relations, compute multiple optimal plans, one for each interesting order
- Increases complexity by factor k+1, where k=number of interesting orders

Why Left-Deep

Asymmetric, cost depends on the order

- Left: Outer relation Right: Inner relation

- For nested-loop-join, we try to load the outer (typically smaller) relation in memory, then read the inner relation one page at a time

  B(R) + B(R)*B(S) or B(R) + B(R)/M * B(S)

- For index-join,

  we assume right (inner) relation has index

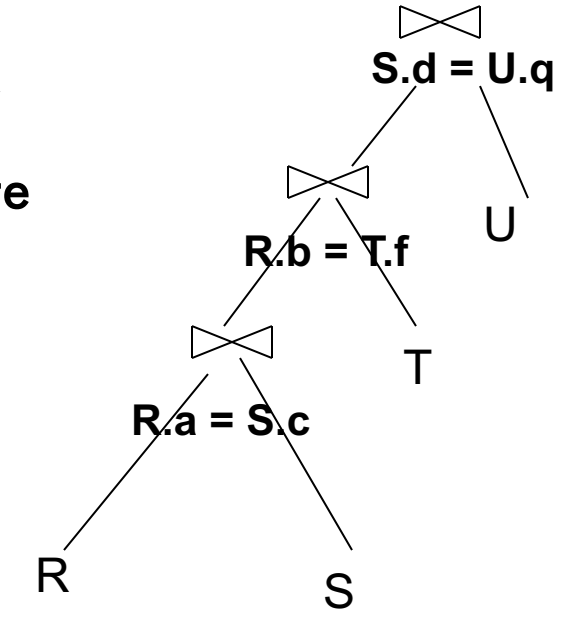
Why Left-Deep

- Advantages of left-deep trees?
  1. Fits well with standard join algorithms (nested loop, one-pass), more efficient
  2. One pass join: Uses smaller memory
     1. ((R, S), T), can reuse the space for R while joining (R, S) with T
     2. (R, (S, T)): Need to hold R, compute (S, T), then join with R, worse if more relations
  3. Nested loop join, consider top-down iterator next()
     1. ((R, S), T), Reads the chunks of (R, S) once, reads stored base relation T multiple times
     2. (R, (S, T)): Reads the chunks of R once, reads computed relation (S, T) multiple times, either more time or more space

Implementation in SimpleDB (lab5)

1. JoinOptimizer.java (and the classes used there)

2. Returns vector of "LogicalJoinNode"
   Two base tables, two join attributes, predicate
   e.g. R(a, b), S(c, d), T(a, f), U(p, q)
   (R, S, R.a, S.c, =)
   Recall that SimpleDB keeps all attributes of R, S after their join R.a, R.b, S.c, S.d



3. Output vector looks like:
   <(R, S, R.a, S.c), (R, T, R.b, T.f), (S, U, S.d, U.q)>

Implementation in SimpleDB (lab5)

Any advantage of returning pairs?

- Flexibility to consider all linear plans
  <(R, S, R.a,S.c), (R, T, R.b, T.f), (U, S, U.q, S.d)>

More Details:

1. You mainly need to implement "orderJoins(..)"
2. "CostCard" data structure stores a plan, its cost and cardinality: you would need to estimate them
3. "PlanCache" stores the table in dyn. Prog:

   Maps a set of LJN to
   a vector of LJN (best plan for the vector),
   its cost, and its cardinality
   LJN = LogicalJoinNode