Lecture 9
Query Optimization

November 24, 2010

Outline

- Chapter 15 in the textbook
- Paper: *Query Optimizers: Time to Rethink the Contract ?*, by Surajit Chaudhuri
- Next time: parallel databases, Bloom filters

Query Optimization Algorithm

- Enumerate alternative plans
- Compute estimated cost of each plan
  - Compute number of I/Os
  - Compute CPU cost
- Choose plan with lowest cost
  - This is called cost-based optimization

Example

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

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

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

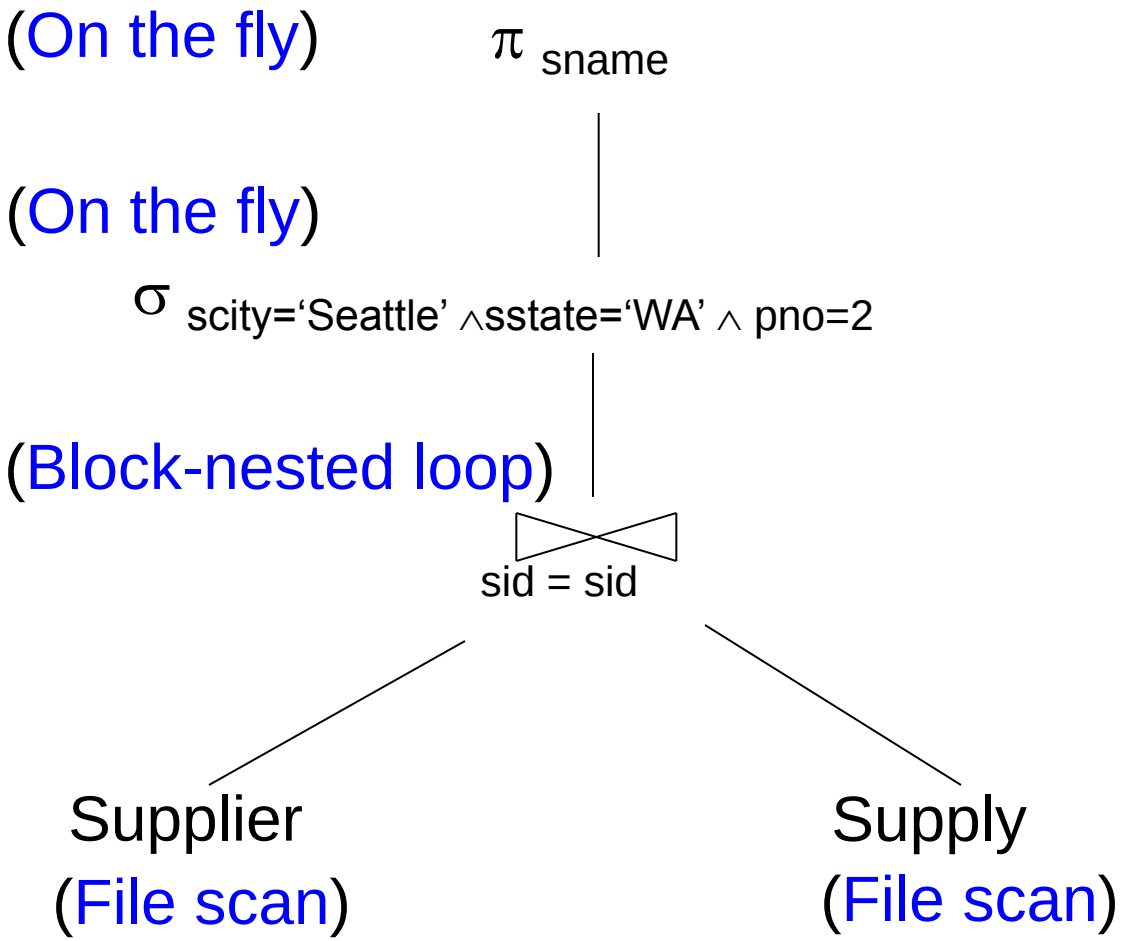
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 1



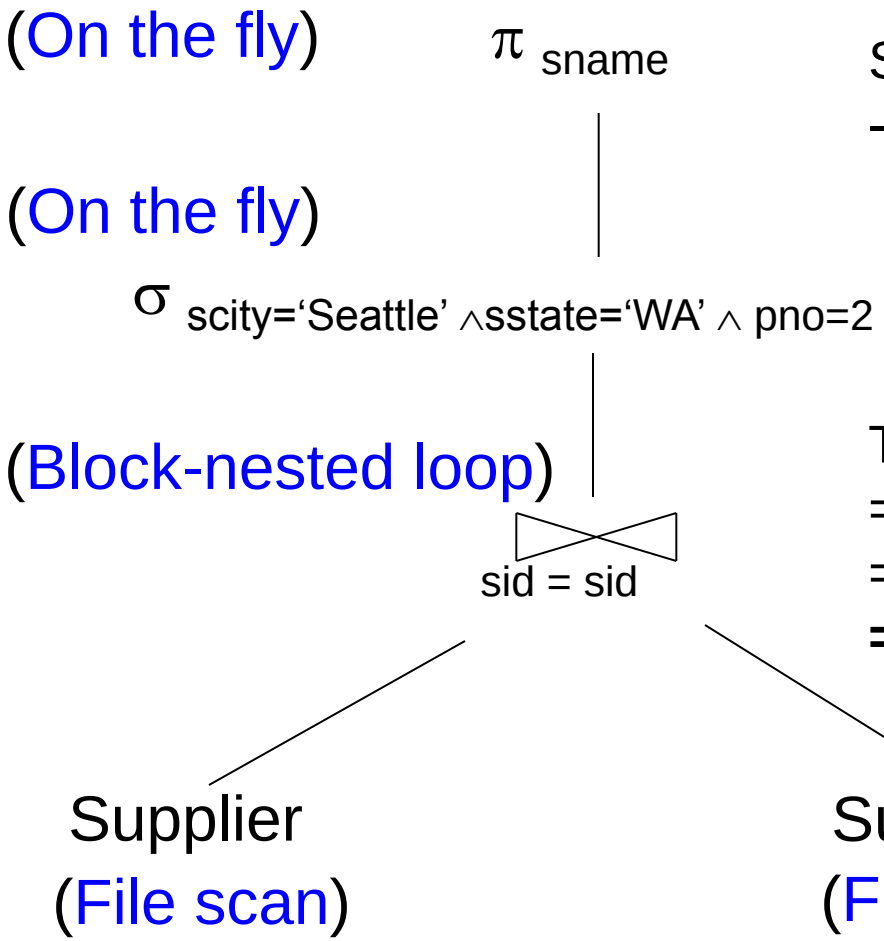
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 1



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

Total cost of plan is thus cost of join:
= B(Supplier)+B(Supplier)*B(Supply)/M
= 100 + 10 * 100
= 1,100 I/Os

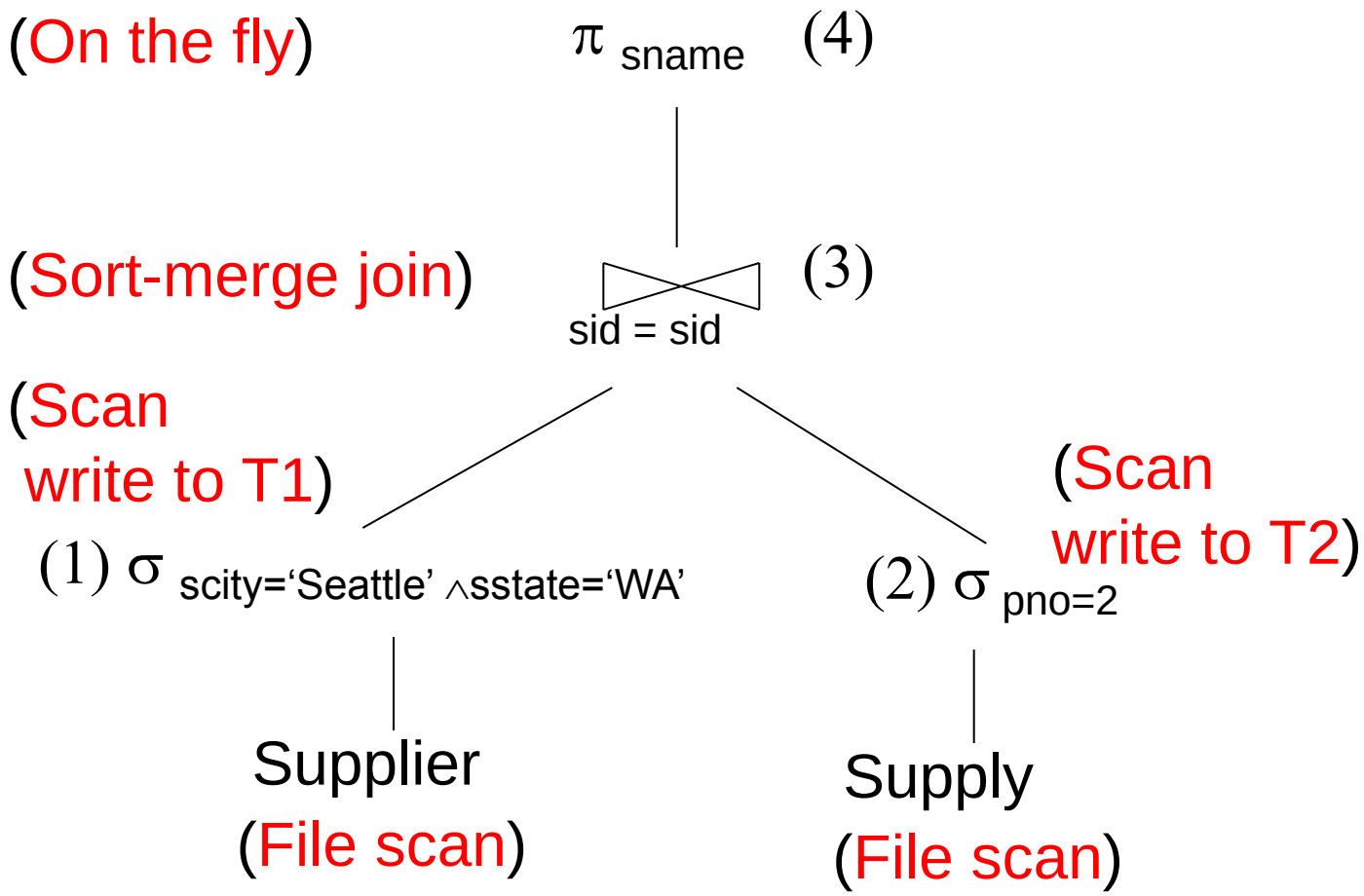
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 2



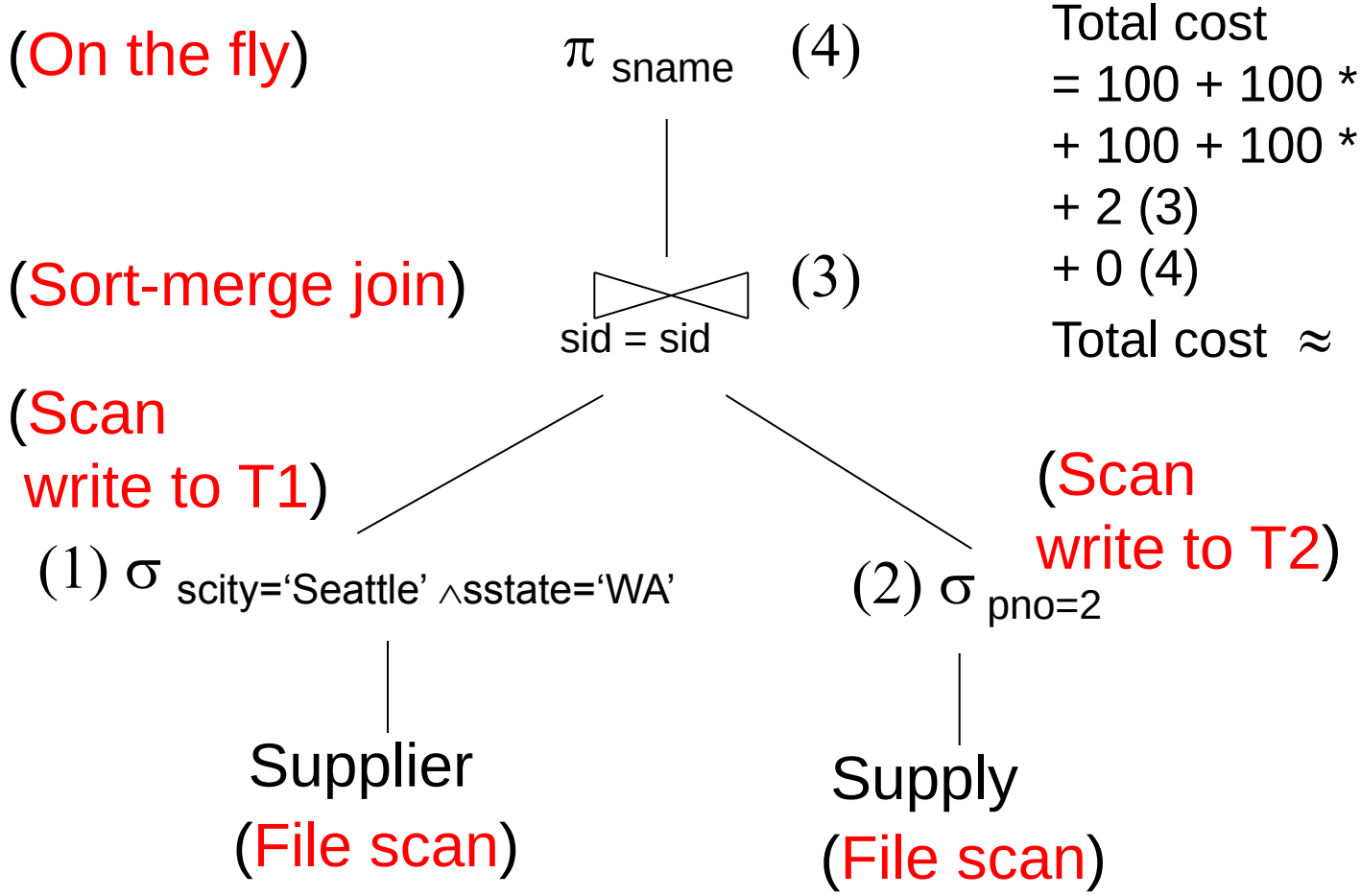
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 2



Total cost
= 100 + 100 * 1/20 * 1/10 (1)
+ 100 + 100 * 1/2500 (2)
+ 2 (3)
+ 0 (4)
Total cost ≈ **204 I/Os**

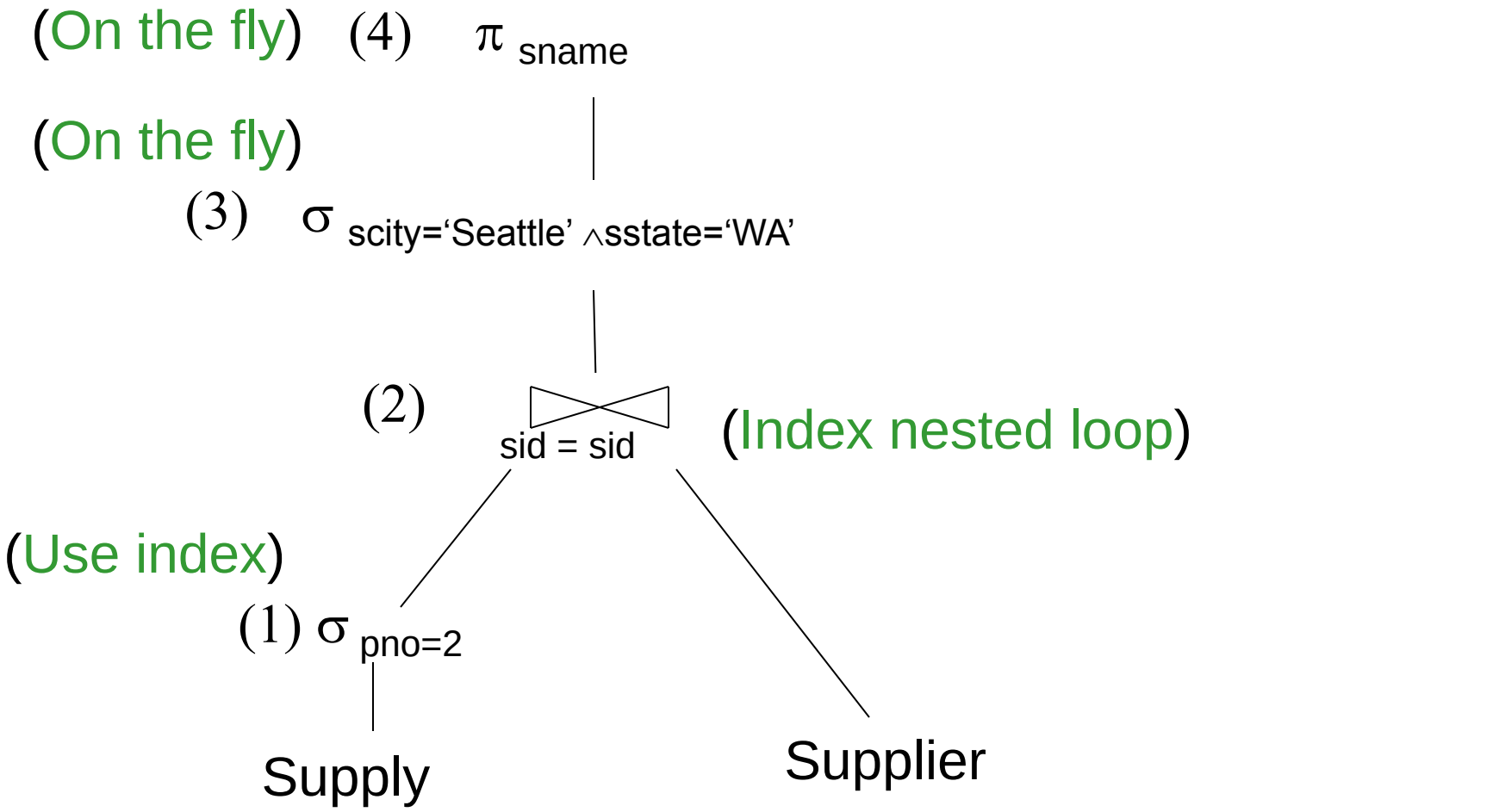
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 3



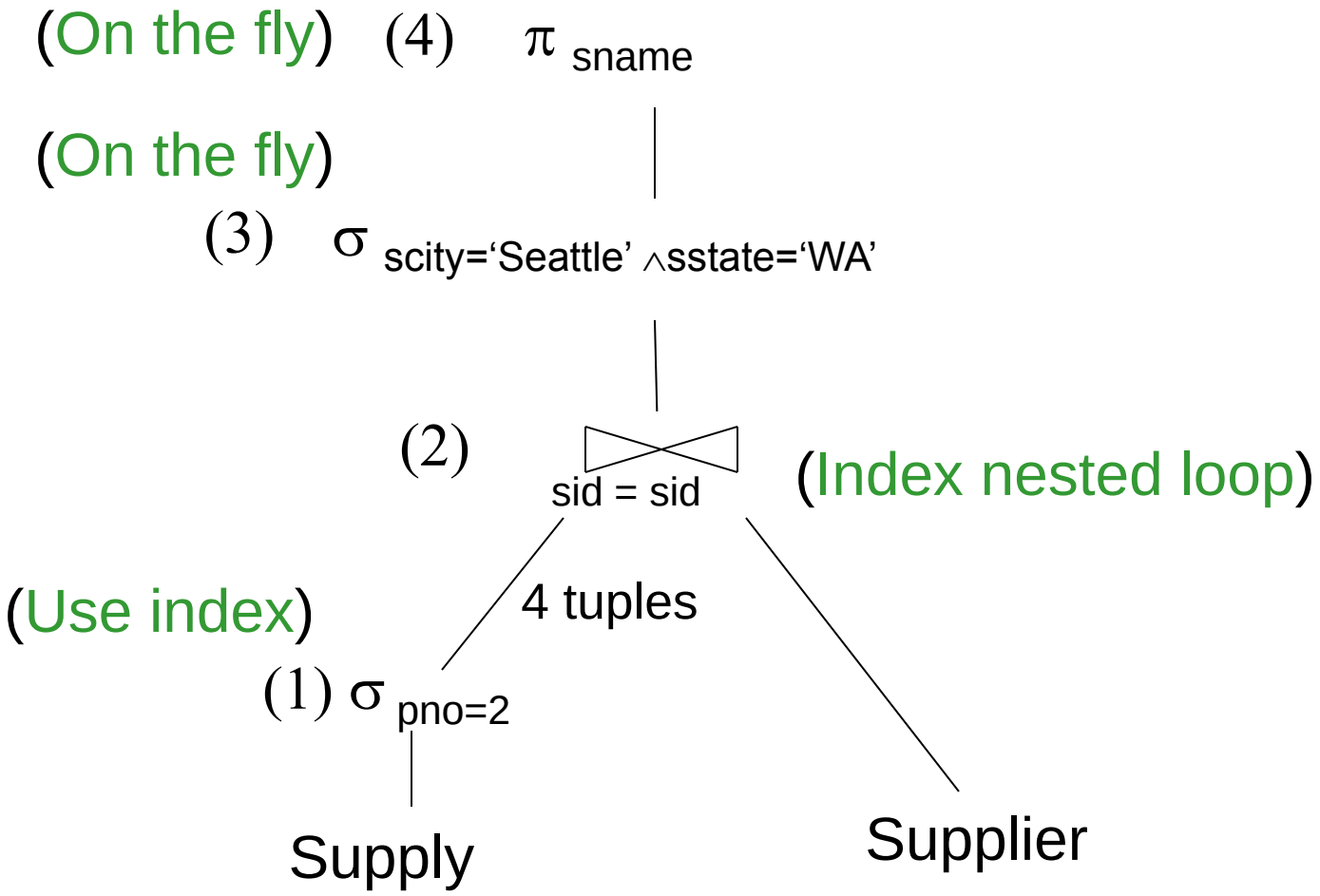
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 3



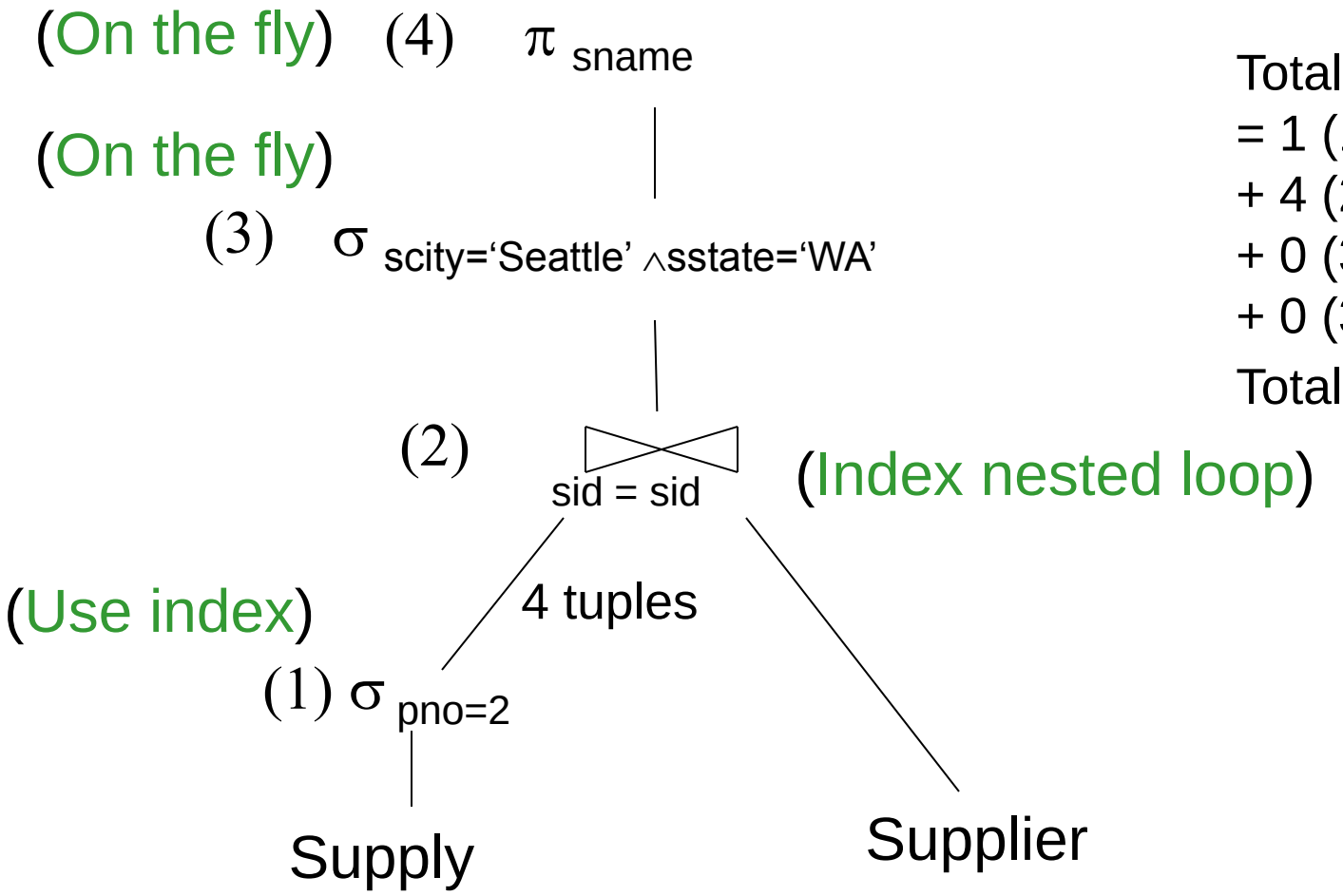
T(Supplier) = 1000
T(Supply) = 10,000

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

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

M = 10

Physical Query Plan 3



Total cost
= 1 (1)
+ 4 (2)
+ 0 (3)
+ 0 (3)
Total cost ≈ **5 I/Os**

Simplifications

- In the previous examples, we assumed that all index pages were in memory
- When this is not the case, we need to add the cost of fetching index pages from disk

Lessons

1. Need to consider several physical plan
   - even for one, simple logical plan

2. No plan is best in general
   - need to have ***statistics*** over the data
   - the B's, the T's, the V's

The Contract of the Optimizer

[Chaudhuri]

- high-quality execution plans for all queries,
- while taking relatively small optimization time, and
- with limited additional input such as histograms.

Query Optimization

Three major components:

1. Search space
2. Algorithm for enumerating query plans
3. Cardinality and cost estimation

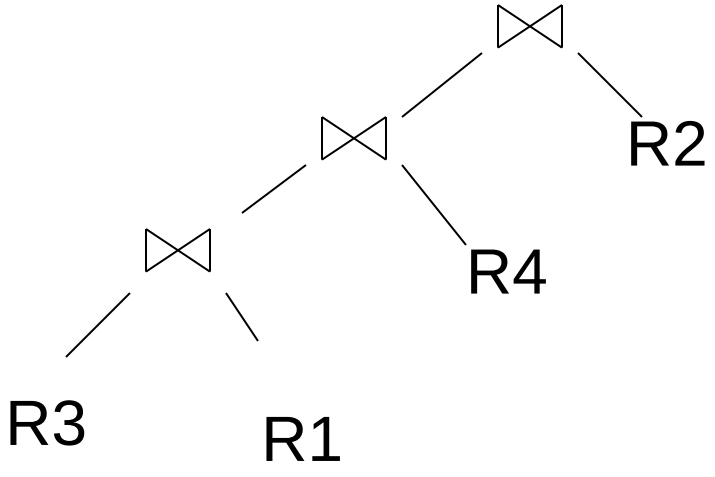
History of Query Optimization

- First query optimizer was for System R, from IBM, in 1979
- It had all three components in place, and defined the architecture of query optimizers for years to come
- You will see often references to System R
- Read Section 15.6 in the book

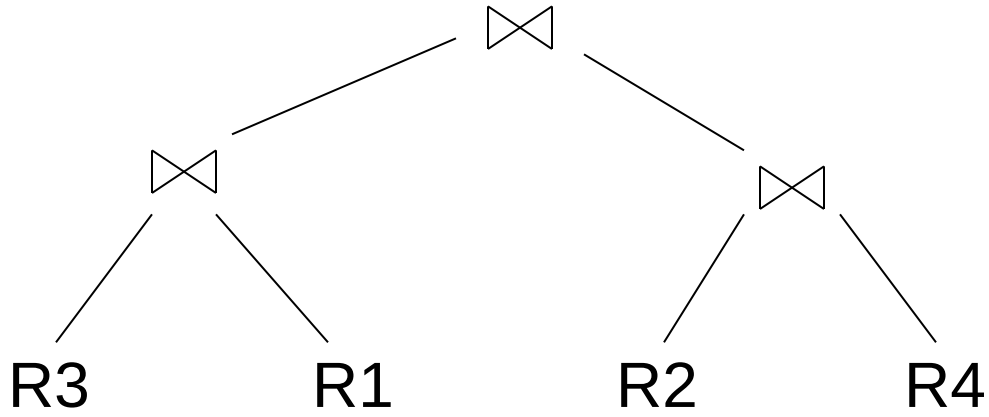
1. Search Space

- This is the set of all alternative plans that are considered by the optimizer
- Defined by the set of *algebraic laws* and the *set of plans* used by the optimizer
- Will discuss these laws next

Left-Deep Plans and Bushy Plans



Left-deep plan



Bushy plan

System R considered only left deep plans, and so do some optimizers today

Relational Algebra Laws

- Selections
  - Commutative: $\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$
  - Cascading: $\sigma_{c1 \wedge c2}(R) = \sigma_{c2}(\sigma_{c1}(R))$
- Projections
- Joins
  - Commutativity : $R \bowtie S = S \bowtie R$
  - Associativity: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
  - Distributivity: $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$
  - Outer joins get more complicated

Example

Which plan is more efficient ?
$R \bowtie (S \bowtie T)$ or $(R \bowtie S) \bowtie T$?

- Assumptions:
  - Every join selectivity is 10%
    - That is: $T(R \bowtie S) = 0.1 * T(R) * T(S)$ etc.
  - B(R)=100, B(S) = 50, B(T)=500
  - All joins are main memory joins
  - All intermediate results are materialized

Simple Laws

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

$$\sigma_{C}(R - S) = \sigma_{C}(R) - S$$
$$\sigma_{C}(R \cup S) = \sigma_{C}(R) \cup \sigma_{C}(S)$$
$$\sigma_{C}(R \bowtie S) = \sigma_{C}(R) \bowtie S$$

When C involves only attributes of R

Example

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

Simple Laws

$\Pi_M(R \bowtie S) = \Pi_M(\Pi_P(R) \bowtie \Pi_Q(S))$
$\Pi_M(\Pi_N(R)) = \Pi_M(R)$ /* 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 for Group-by and Join

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

if agg is "duplicate insensitive"

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

Example

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

These are very powerful laws.
They were introduced only in the 90's.

Laws Involving Constraints

Foreign key

Product(<u>pid</u>, pname, price, cid)
Company(<u>cid</u>, cname, city, state)

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

Need a second constraint for this law to hold. Which ?

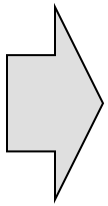
Example

Foreign key

Product(pid, pname, price, cid)
Company(cid, cname, city, state)

```
CREATE VIEW CheapProductCompany
    SELECT *
    FROM Product x, Company y
    WHERE x.cid = y.cid and x.price < 100
```

```
SELECT pname, price
FROM CheapProductCompany
```



```
SELECT pname, price
FROM Product
```

Law of Semijoins

Recall the definition of a semijoin:

- $R \ltimes S = \Pi_{A1,\ldots,An} (R \bowtie S)$
- Where the schemas are:
  - Input: R(A1,…An), S(B1,…,Bm)
  - Output: T(A1,…,An)
- The law of semijoins is:

$$R \bowtie S = (R \ltimes S) \bowtie S$$

Laws with Semijoins

- Very important in parallel databases
- Often combined with Bloom Filters (next lecture)
- Read pp. 747 in the textbook

Semijoin Reducer

- Given a query: $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
- A *semijoin reducer* for Q is

$$R_{i1} = R_{i1} \ltimes R_{j1}$$
$$R_{i2} = R_{i2} \ltimes R_{j2}$$
$$\ldots\ldots$$
$$R_{ip} = R_{ip} \ltimes R_{jp}$$

such that the query is equivalent to:

$$Q = R_{k1} \bowtie R_{k2} \bowtie \ldots \bowtie R_{kn}$$

- A *full reducer* is such that no dangling tuples remain

Example

- Example:

$$Q = R(A,B) \bowtie S(B,C)$$

- A semijoin reducer is:

$$R_1(A,B) = R(A,B) \ltimes S(B,C)$$

- The rewritten query is:

$$Q = R_1(A,B) \bowtie S(B,C)$$

Why Would We Do This ?

- Large attributes:

$$Q = R(A, B, D, E, F, \ldots) \bowtie S(B, C, M, K, L, \ldots)$$

- Expensive side computations

$$Q = \gamma_{A,B,count(*)} R(A,B,D) \bowtie \sigma_{C=value}(S(B,C))$$

$$R_1(A,B,D) = R(A,B,D) \ltimes \sigma_{C=value}(S(B,C))$$
$$Q = \gamma_{A,B,count(*)} R_1(A,B,D) \bowtie \sigma_{C=value}(S(B,C))$$

Semijoin Reducer

- Example:

$$Q = R(A,B) \bowtie S(B,C)$$

- A semijoin reducer is:

$$R_1(A,B) = R(A,B) \ltimes S(B,C)$$

- The rewritten query is:

$$Q = R_1(A,B) \bowtie S(B,C)$$

Are there dangling tuples ?

Semijoin Reducer

- Example:

$$Q = R(A,B) \bowtie S(B,C)$$

- A full semijoin reducer is:

$$R_1(A,B) = R(A,B) \ltimes S(B,C)$$
$$S_1(B,C) = S(B,C) \ltimes R_1(A,B)$$

- The rewritten query is:

$$Q :- R_1(A,B) \bowtie S_1(B,C)$$

No more dangling tuples

Semijoin Reducer

- More complex example:

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

- A full reducer is:

$$S'(B,C) := S(B,C) \ltimes R(A,B)$$
$$T'(C,D,E) := T(C,D,E) \ltimes S(B,C)$$
$$S''(B,C) := S'(B,C) \ltimes T'(C,D,E)$$
$$R'(A,B) := R(A,B) \ltimes S''(B,C)$$

$$Q = R'(A,B) \bowtie S''(B,C) \bowtie T'(C,D,E)$$

Semijoin Reducer

- Example:

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

- Doesn't have a full reducer (we can reduce forever)

Theorem a query has a full reducer iff it is "acyclic"
[*Database Theory*, by Abiteboul, Hull, Vianu]

Example with Semijoins

Emp(eid, ename, sal, did)
Dept(did, dname, budget)
DeptAvgSal(did, avgsal) /* view */

[Chaudhuri'98]

View:

```
CREATE VIEW DepAvgSal As (
        SELECT E.did, Avg(E.Sal) AS avgsal
        FROM Emp E
        GROUP BY E.did)
```

Query:

```
SELECT E.eid, E.sal
FROM Emp E, Dept D, DepAvgSal V
WHERE E.did = D.did AND E.did = V.did
        AND E.age < 30 AND D.budget > 100k
        AND E.sal > V.avgsal
```

Goal: compute only the necessary part of the view

Example with Semijoins

Emp(eid, ename, sal, did)
Dept(did, dname, budget)
DeptAvgSal(did, avgsal) /* view */

[Chaudhuri'98]

New view uses a reducer:

```
CREATE VIEW LimitedAvgSal As (
        SELECT E.did, Avg(E.Sal) AS avgsal
        FROM Emp E, Dept D
        WHERE E.did = D.did AND D.buget > 100k
        GROUP BY E.did)
```

New query:

```
SELECT E.eid, E.sal
FROM Emp E, Dept D, LimitedAvgSal V
WHERE E.did = D.did AND E.did = V.did
        AND E.age < 30 AND D.budget > 100k
        AND E.sal > V.avgsal
```

Example with Semijoins

Emp(eid, ename, sal, did)
Dept(did, dname, budget)
DeptAvgSal(did, avgsal) /* view */

[Chaudhuri'98]

Full reducer:

```
CREATE VIEW PartialResult AS
        (SELECT E.eid, E.sal, E.did
        FROM Emp E, Dept D
        WHERE E.did=D.did AND E.age < 30
        AND D.budget > 100k)

CREATE VIEW Filter AS
        (SELECT DISTINCT P.did FROM PartialResult P)

CREATE VIEW LimitedAvgSal AS
        (SELECT E.did, Avg(E.Sal) AS avgsal
        FROM Emp E, Filter F
        WHERE E.did = F.did GROUP BY E.did)
```

Example with Semijoins

New query:

```
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

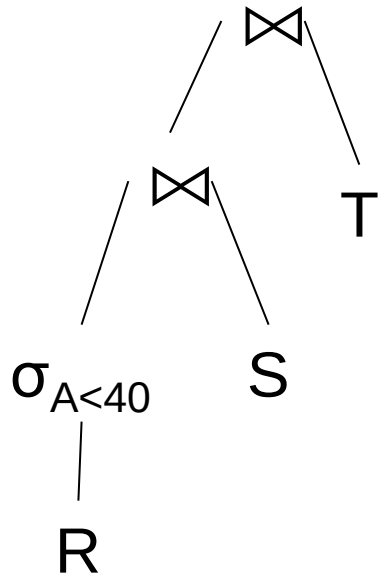
Pruning the Search Space

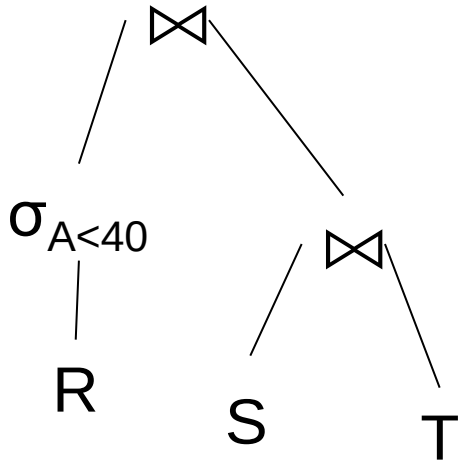
- Prune entire sets of plans that are unpromising
- The choice of *partial plans* influences how effective we can prune

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
```





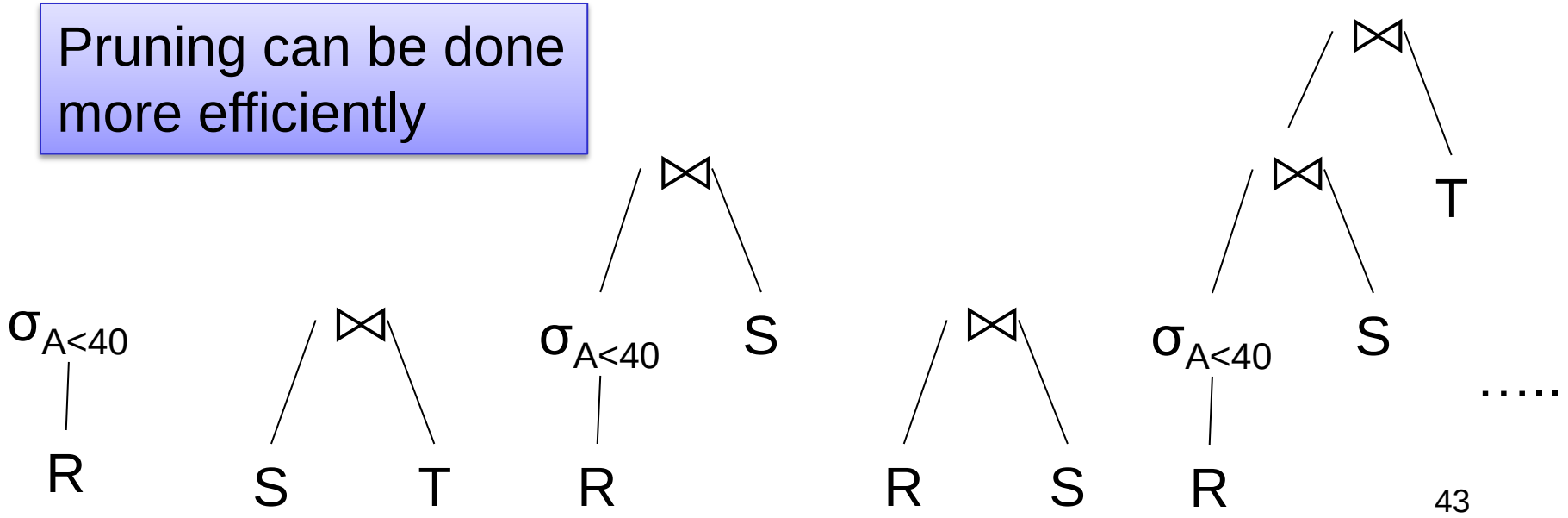
Pruning is difficult here.

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
```

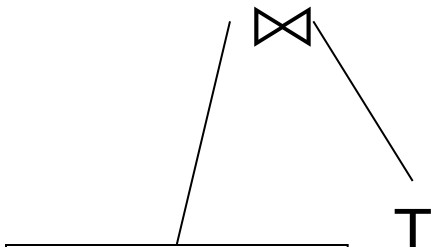
Pruning can be done more efficiently

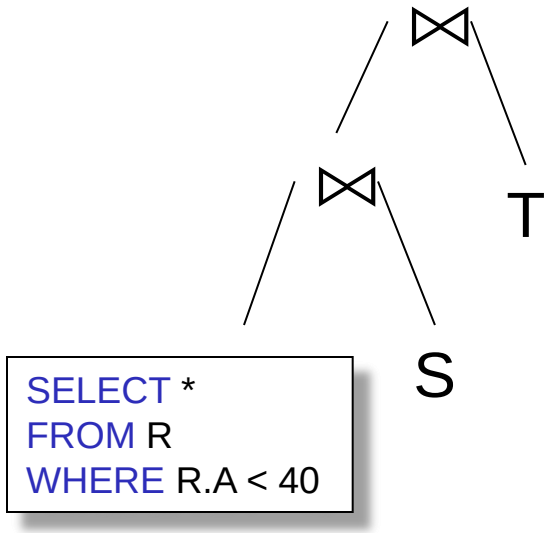


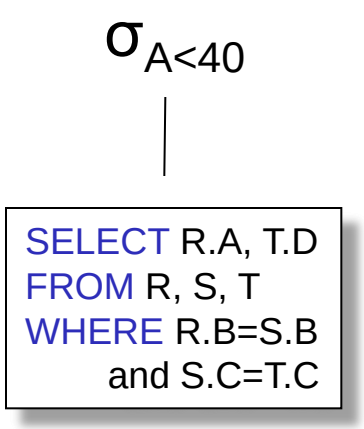
Top-down Partial Plans

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 Optimization

Three major components:

1. Search space
2. Algorithm for enumerating query plans
3. Cardinality and cost estimation

2. Plan Enumeration Algorithms

- System R (in class)
  - *Join reordering* – dynamic programming
  - *Access path selection*
  - Bottom-up; simple; limited
- Modern database optimizers (will not discuss)
  - Rule-based: database of rules (x 100s)
  - Dynamic programming
  - Top-down; complex; extensible

Join Reordering

System R [1979]

- Push all selections down (=early) in the query plan
- Pull all projections up (=late) in the query plan
- What remains are joins:

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

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

Join Reordering

Dynamic programming

- For each subquery Q $\subseteq${R1, …, Rn}, compute the optimal join order for Q
- Store results in a table: 2^n-1 entries
  - Often much fewer entries

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

Join Reordering

Step 1: For each $\{R_i\}$ do:

- Initialize the table entry for $\{R_i\}$ with the cheapest access path for R_i

Step 2: For each subset $Q \subseteq \{R_1, \ldots, R_n\}$ do:

- For every partition $Q = Q' \cup Q''$
- Lookup optimal plan for Q' and for Q'' in the table
- Compute the cost of the plan $Q' \bowtie Q''$
- Store the cheapest plan $Q' \bowtie Q''$ in table entry for Q

Reducing the Search Space

Restriction 1: only left linear trees (no bushy)

Restriction 2: no trees with cartesian product

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

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

Access Path Selection

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

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

Access Path Selection

- Supplier(sid,sname,scity,sstate)
- Selection condition: sid > 300 $\wedge$ scity='Seattle'
- Indexes: B+-tree on sid and B+-tree on scity

Access Path Selection

- Supplier(sid,sname,scity,sstate)
- Selection condition: sid > 300 $\wedge$ scity='Seattle'
- Indexes: B+-tree on sid and B+-tree on scity
- Which access path should we use?

Access Path Selection

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

Access Path Selectivity

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

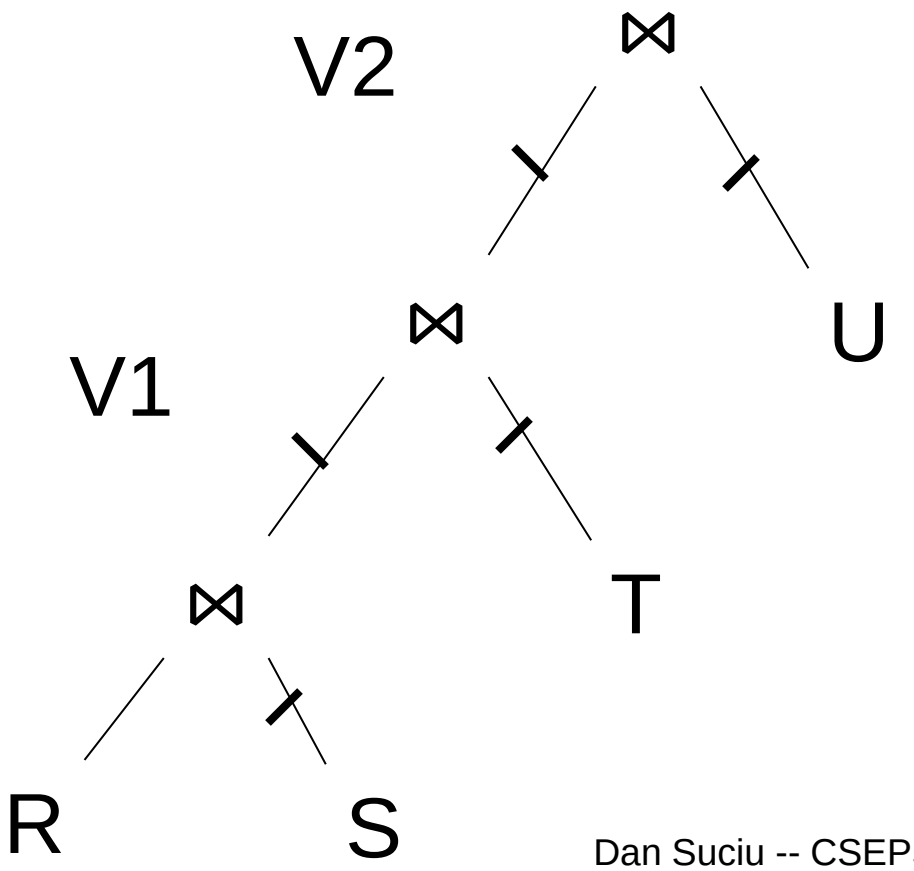
- As we saw earlier, **for equality predicates**
  - Selection on equality: $\sigma_{a=v}(R)$
  - V(R, a) = # of distinct values of attribute a
  - 1/V(R,a) is thus the reduction factor
  - Clustered index on a: cost B(R)/V(R,a)
  - Unclustered index on a: cost T(R)/V(R,a)
  - (we are ignoring I/O cost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

Other Decisions for the Optimization Algorithm

- How much memory to allocate to each operator

- Pipeline or materialize (next)

Materialize Intermediate Results Between Operators




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

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

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

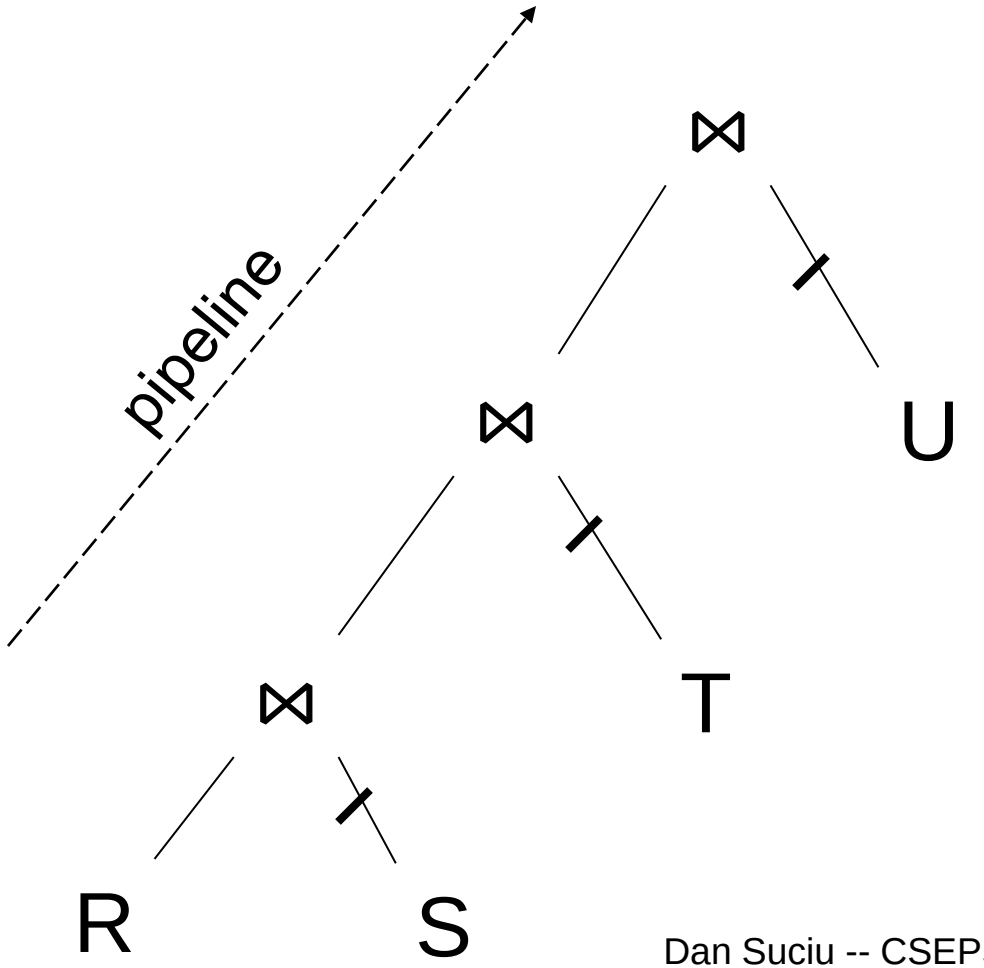
Materialize Intermediate Results Between Operators

Question in class

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

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

Pipeline Between Operators




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

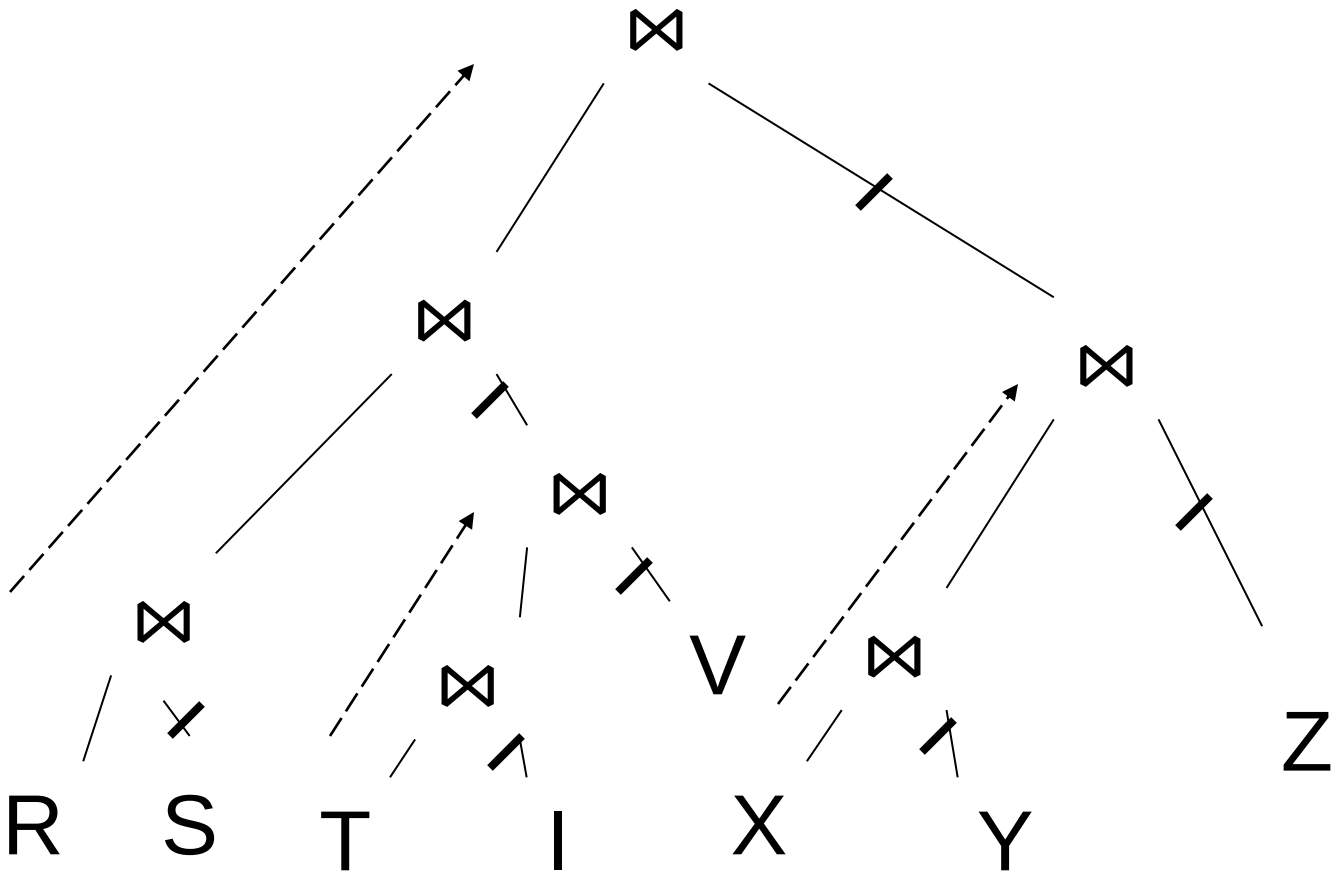
Pipeline Between Operators

Question in class

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

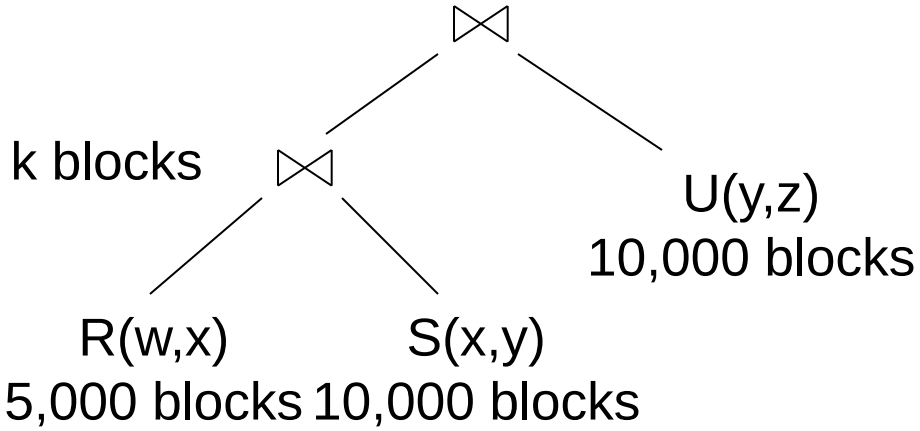
- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline in Bushy Trees



Example (will skip in class)

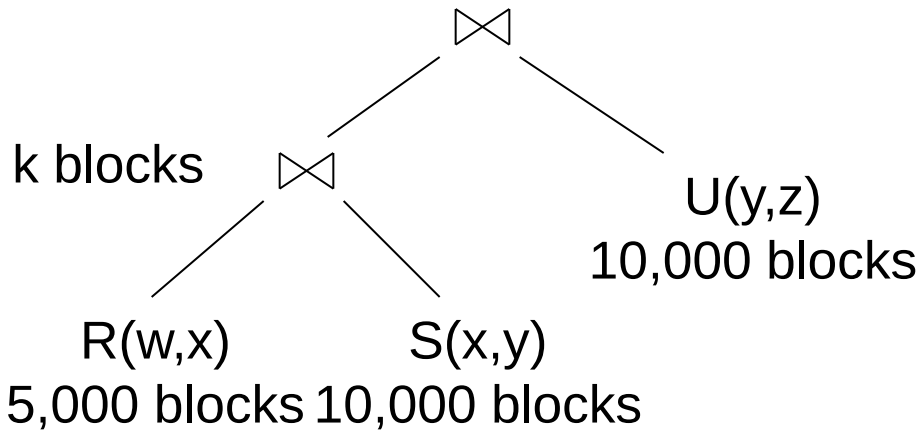
- Logical plan is:



- Main memory M = 101 buffers

Example (will skip in class)

M = 101

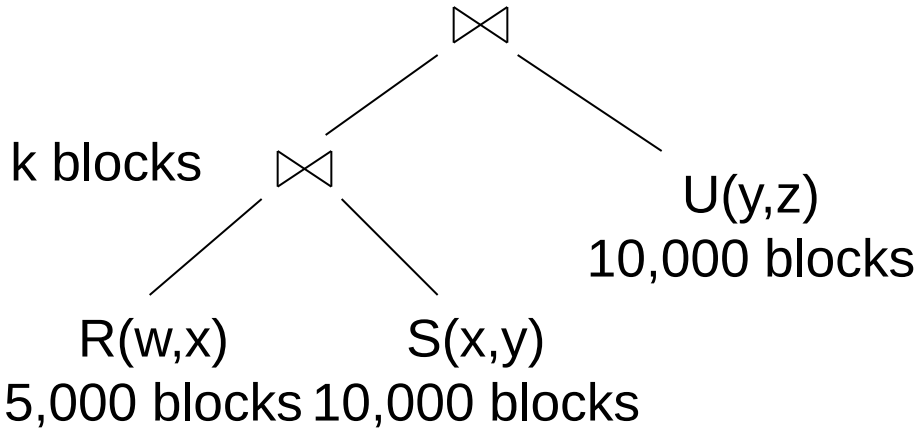


Naïve evaluation:

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

Example (will skip in class)

M = 101

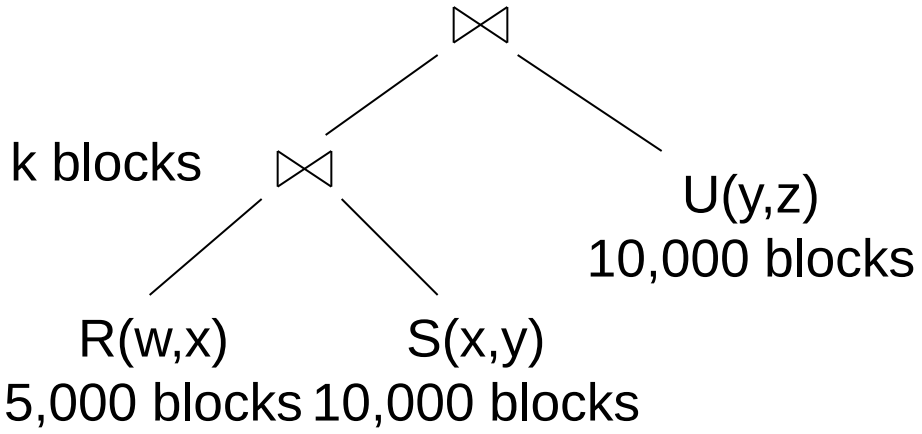


Smarter:

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

Example (will skip in class)

M = 101

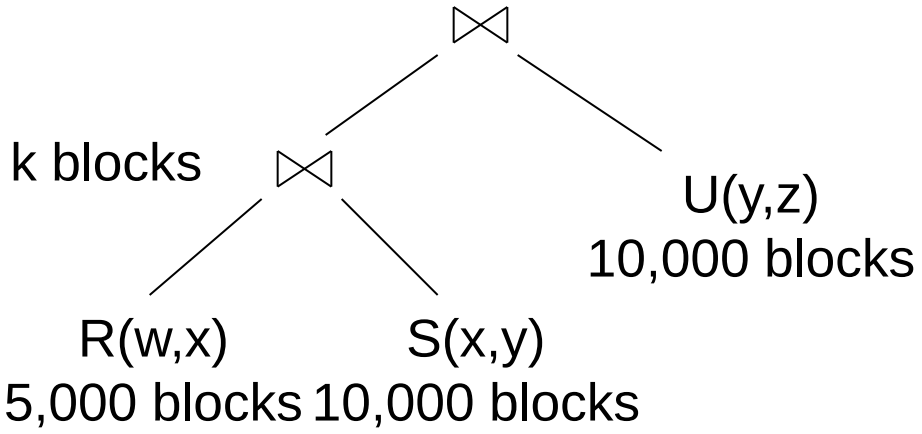


Continuing:

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

Example (will skip in class)

M = 101

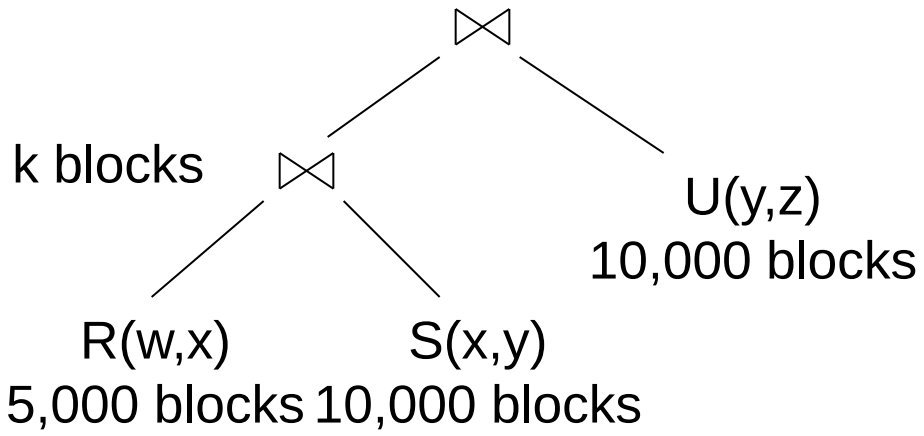


Continuing:

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

Example (will skip in class)

M = 101



Continuing:

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

Query Optimization

Three major components:

1. Search space
2. Algorithm for enumerating query plans
3. Cardinality and cost estimation

3. Cardinality and Cost Estimation

- 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

Statistics on Base Data

- Collected information for each relation
  - Number of tuples (cardinality)
  - Indexes, number of keys in the index
  - Number of physical pages, clustering info
  - Statistical information on attributes
    - Min value, max value, number distinct values
    - Histograms
  - Correlations between columns (hard)
- Collection approach: periodic, using sampling

Size Estimation Problem

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

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

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

Size Estimation Problem

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

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

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)

```
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

What is the estimated size of the query output ?

Rule of Thumb

- If selectivities are unknown, then:
  selectivity factor = 1/10
  [System R, 1979]

Using Data Statistics

- Condition is A = c /* value selection on R */
  - Selectivity = 1/V(R,A)
- Condition is A < c /* range selection on R */
  - Selectivity = (c - Low(R, A))/(High(R,A) - Low(R,A))T(R)
- Condition is 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 the set of A values of R is included in the set of B values of S
  - 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))

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

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

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

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

Size Estimation for Join

Example:

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

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

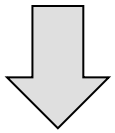
$\sigma_{age=48}(Empolyee) = ?$ $\sigma_{age>28 \text{ and } age<35}(Empolyee) = ?$

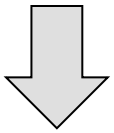
Histograms

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 / 60 = 2500

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 = 2*80 + 5*500 = 2660

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

Employee(ssn, name, age)

Histograms

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..20	20..29	30-39	40-49	50-59	> 60
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 ?

Summary of Query Optimization

- Three parts:
  - search space, algorithms, size/cost estimation
- Ideal goal: find optimal plan. But
  - Impossible to estimate accurately
  - Impossible to search the entire space
- Goal of today's optimizers:
  - Avoid very bad plans