CSE 544: Lecture 14

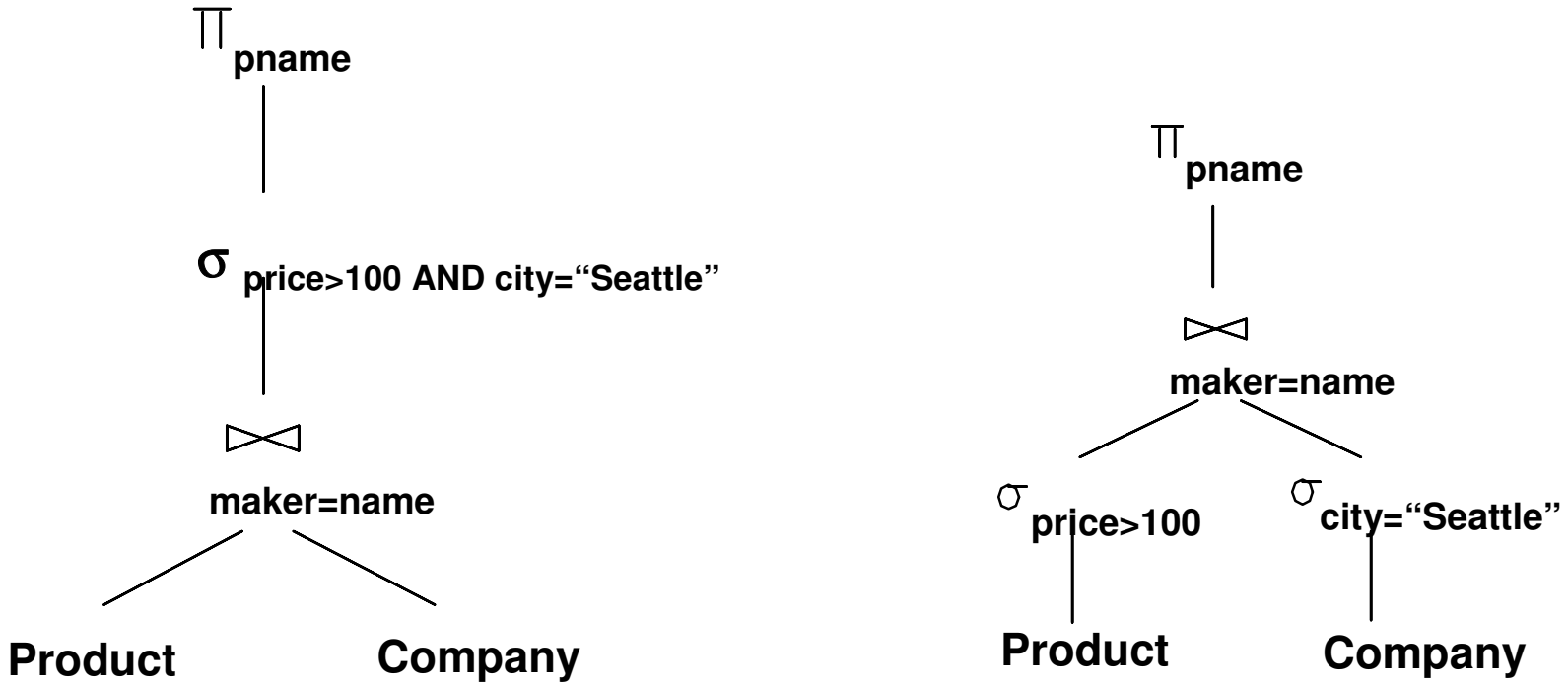
Wednesday, 5/15/2002

Optimization, Size Estimation

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Main heuristics:
  - Push selections down the tree

Heuristic Based Optimizations



The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to indexes).

Heuristic Based Optimizations

Semi-join based optimizations

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

Heuristic Based Optimizations

Semijoins: motivated by distributed databases:

- Product(<u>pid</u>, cid, pname, ...) at site 1
- Company(<u>cid</u>, cname, ...) at site 2
- Query: $\sigma_{price>1000}(\text{Product}) \bowtie_{cid=cid} \text{Company}$
- Compute as follows:

$T1 = \sigma_{price>1000}(\text{Product})$	site 1
$T2 = \Pi_{cid}(T1)$	site 1
send T2 to site 2	(T2 smaller than T1)
$T3 = T2 \ltimes \text{Company}$	site 2 (semijoin)
send T3 to site 1	(T3 smaller than Company)
$\text{Answer} = T3 \ltimes T1$	site 1 (semijoin)

Heuristic Based Optimizations

Semijoins: a bit of theory (see [AHV])

- Given a conjunctive query:

$$Q :- R_1, R_2, \ldots, R_n$$

- A full reducer for Q is a program:

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

- Such that no dangling tuples remain in any relation

Heuristic Based Optimizations

- Example:

```
Q :- R1(A,B), R2(B,C), R3(C,D)
```

- A full reducer is:

```
R2(B,C) := R2(B,C), R1(A,B)
R3(C,D) := R3(C,D), R2(B,C)
R2(B,C) := R2(B,C), R3(C,D)
R1(A,B) := R1(A,B), R2(B,C)
```

- Example:

```
Q :- R1(A,B), R2(B,C), R3(A,C)
```

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

Heuristic Based Optimizations

- Semijoins in [Chaudhuri'98]

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

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
```

Heuristic Based Optimizations

- Semijoins in [Chaudhuri'98]

```
CREATE VIEW partialresult AS
        (SELECT E.id, 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)
```

Heuristic Based Optimizations

- Semijoins in [Chaudhuri'98]

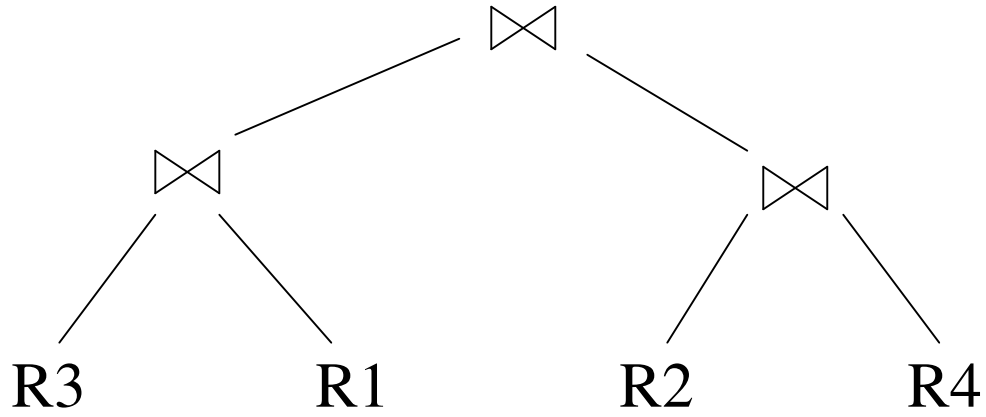
```
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

Cost-Based Optimization

- Main optimization unit:
  - set of joins, i.e. single select-from-where block
  - Hence: *the join reordering problem*
- Optimization methods:
  - Dynamic programming (System R, 1977), for joins:
    - Conceptually cleanest
  - Rule-based optimizations, for arbitrary queries:
    - Volcano → SQL server
    - Starburst → DB2

Join Trees

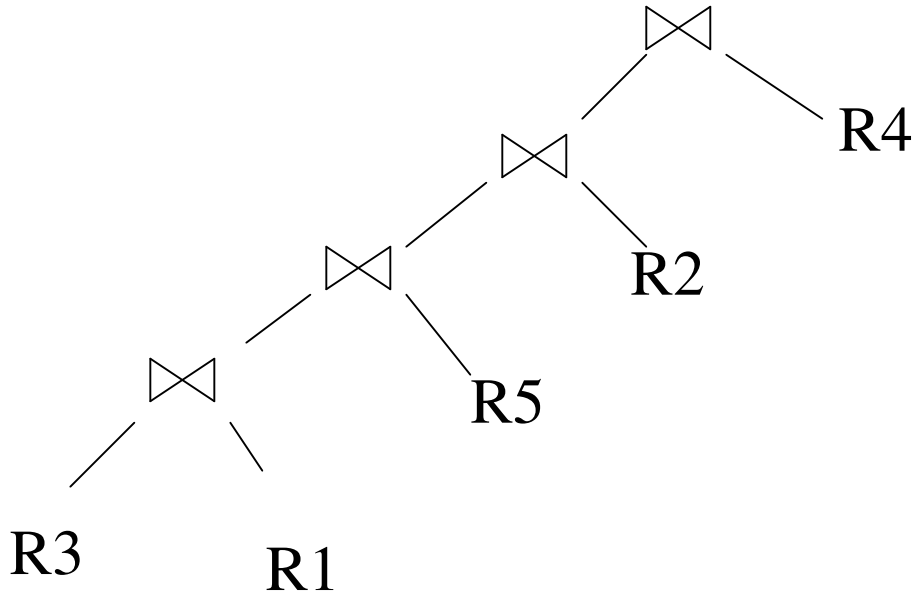
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all ?) join trees

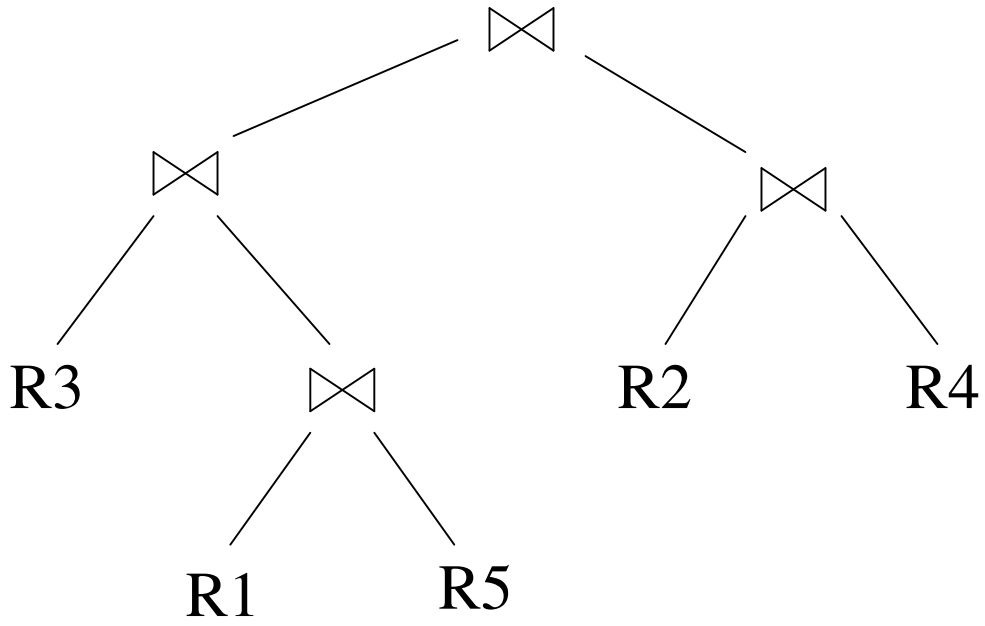
Types of Join Trees

- Left deep:



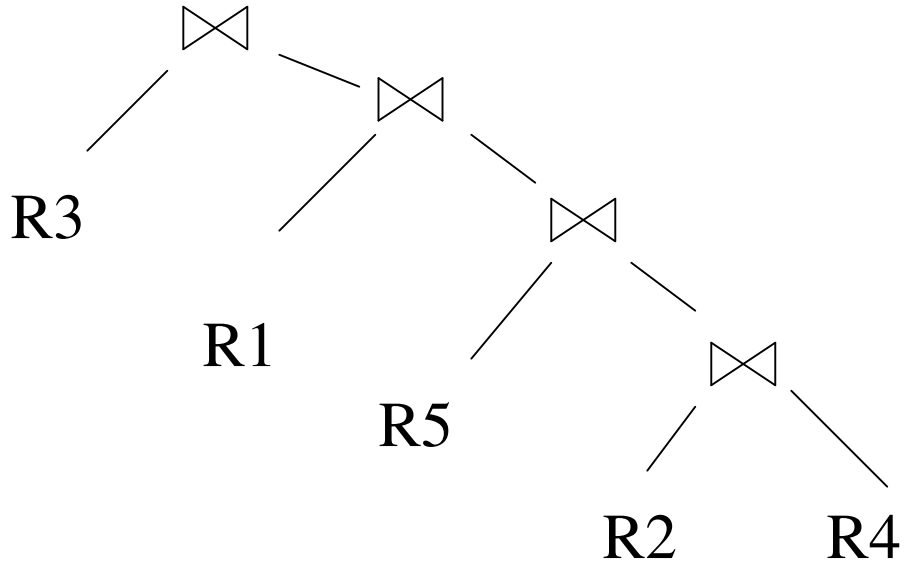
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Problem

- Given: a query $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

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

Dynamic Programming

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

Dynamic Programming

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

Dynamic Programming

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

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

To illustrate, we will make the following simplifications:

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

Dynamic Programming

- Example:
- Cost(R5 ⋈ R7) = 0 (no intermediate results)
- Cost((R2 ⋈ R1) ⋈ R7)
  = Cost(R2 ⋈ R1) + Cost(R7) + size(R2 ⋈ R1)
  = size(R2 ⋈ 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: {R1}, {R2}, …, {Rn}
  - Step 2: {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
  - …
  - Step n: {R1, …, Rn}
- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next)
  - heuristics for Reducing the Search Space
    - Restrict to left linear trees
    - Restrict to trees "without cartesian product"
  - need more than just one plan for each subquery:
    - "interesting orders"

Rule-based Optimizations

- Volcano:
  - Main idea: let programmers define rewrite rules, based on the algebraic laws
  - System searches for “best plan” by applying laws repeatedly
  - Need to avoid cycles, etc.
  - Join-reordering becomes harder, but can handle other operators too
- Starburst:
  - Same, but keep larger nodes, corresponding to one select-from-where block
  - Apply rewrite rules inter-blocks
  - Do dynamic programming inside blocks