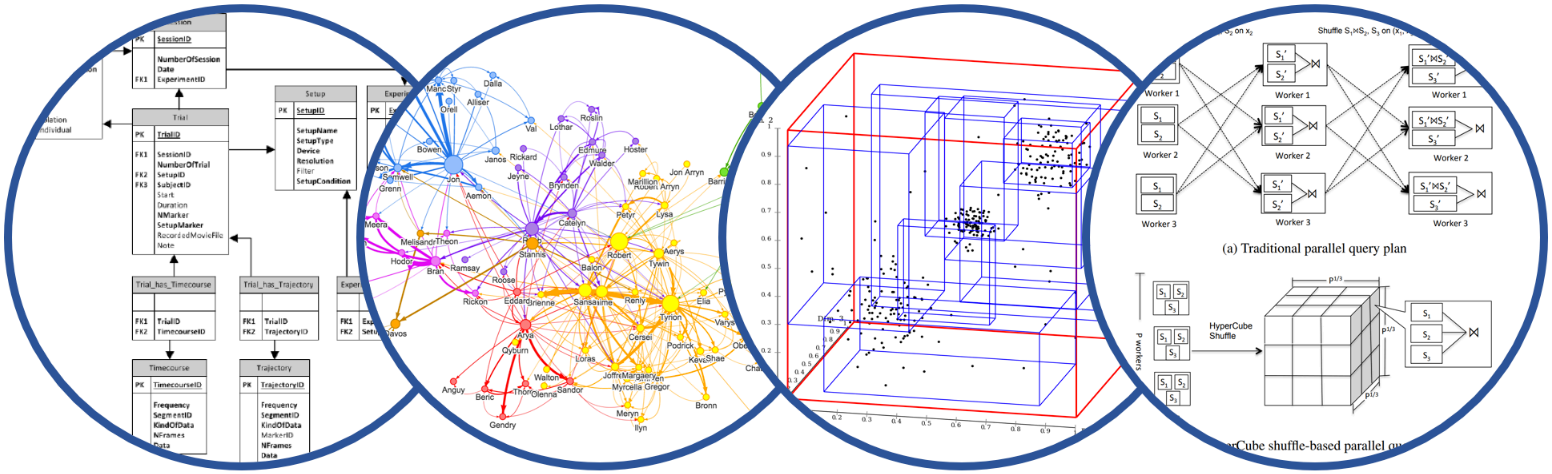


Database System Internals

Query Optimization (part 4)

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

Announcements

- Lab1 is graded and the feedback is pushed
- HW2 due tonight
- Lab2 due on Friday
- Quiz next Wednesday (May 6)

Where We Are

Three components:

- Cost/cardinality estimation
- Search space
  - Algebraic laws ← we are finishing this...
  - Restricting the query plans ← ...and this next
- Search algorithm ← then we'll discuss this

Laws Involving Constraints

- These are laws that hold only under constraints
- Most common: redundant key foreign-key join

Laws Involving Constraints

```
Supply(sid, pno, discount)
Part(pno, pname, category, price)
```

```
select x.sid, x.pno, x.discount
from Supply x, Part y
where x.pno = y.pno
```

Laws Involving Constraints

```
Supply(sid, pno, discount)
Part(pno, pname, category, price)
```

```
select x.sid, x.pno, x.discount
from Supply x, Part y
where x.pno = y.pno
```

Three constraints are needed

```
select x.sid, x.pno, x.disount
from Supply x
```

Laws Involving Constraints

`Supply(sid, pno, discount)`
`Part(pno, pname, category, price)`

```
select x.sid, x.pno, x.discount
from Supply x, Part y
where x.pno = y.pno
```

⬇

```
select x.sid, x.pno, x.disount
from Supply x
```

Three constraints are needed

1. Part.pno is a key
2. Supply.pno is a foreign key
3. Supply.pno IS NOT NULL

Discussion

- When implemented in the optimizer, algebraic laws are called *optimization rules*
- More rules → larger search space → better plan
- Less rules → faster optimization → less good plan
- There is no "complete set" of rules for SQL; Commercial optimizers typically use 5-600 rules, constantly adding rules in response to customer's needs

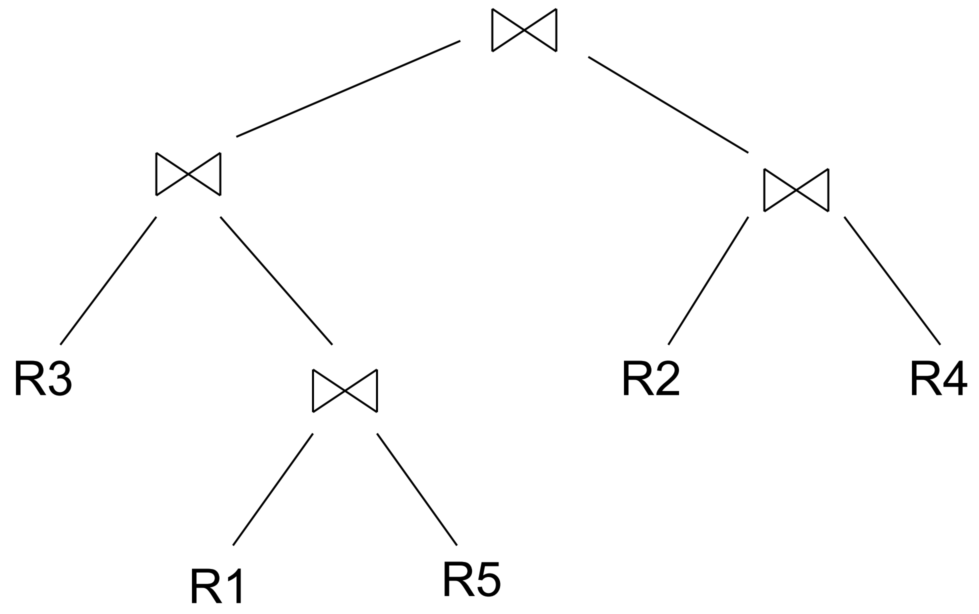
Restricting the Shape of the Query Plans

- The number of query plans is huge

- Optimizers often restrict them:
  - Restrict the types of trees
  - Restrict cartesian products

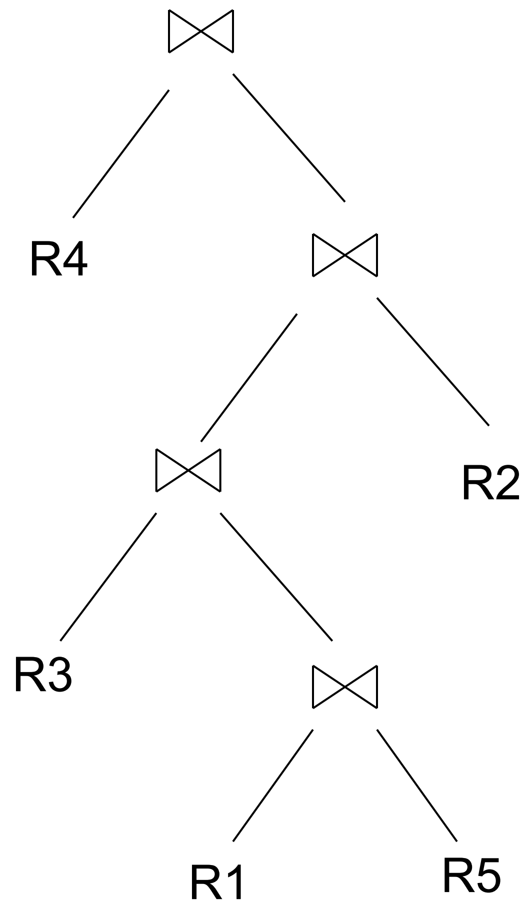
Types of Join Trees

- Bushy:



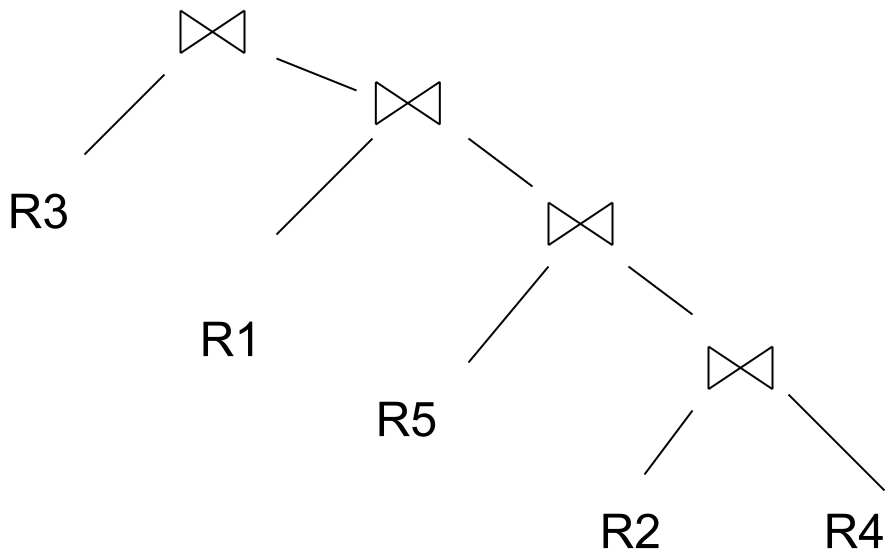
Types of Join Trees

- Linear (aka zig-zag):



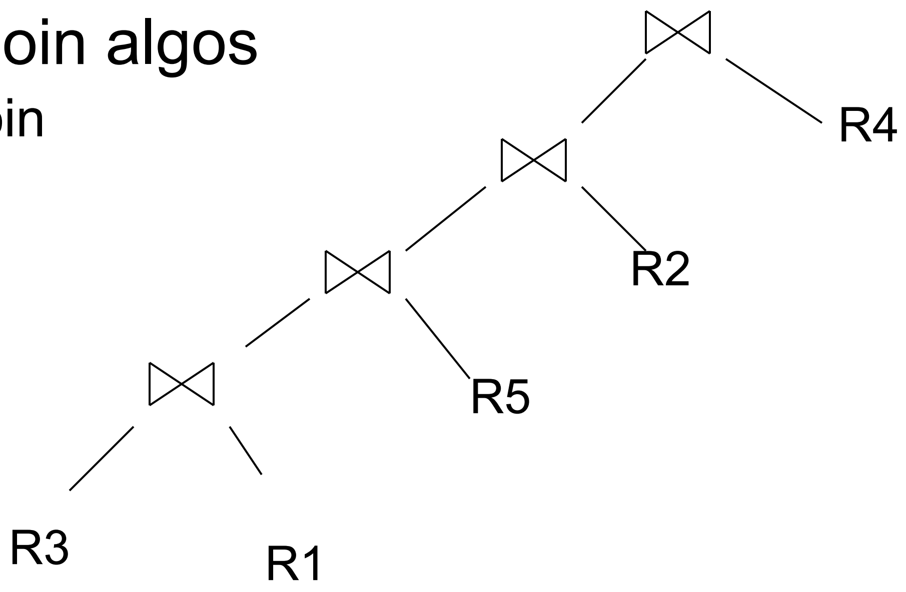
Types of Join Trees

- Right deep:



Types of Join Trees

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



Avoid Cartesian Products

- Cartesian products are usually inefficient
- Most query optimizers avoid them

Avoid Cartesian Products

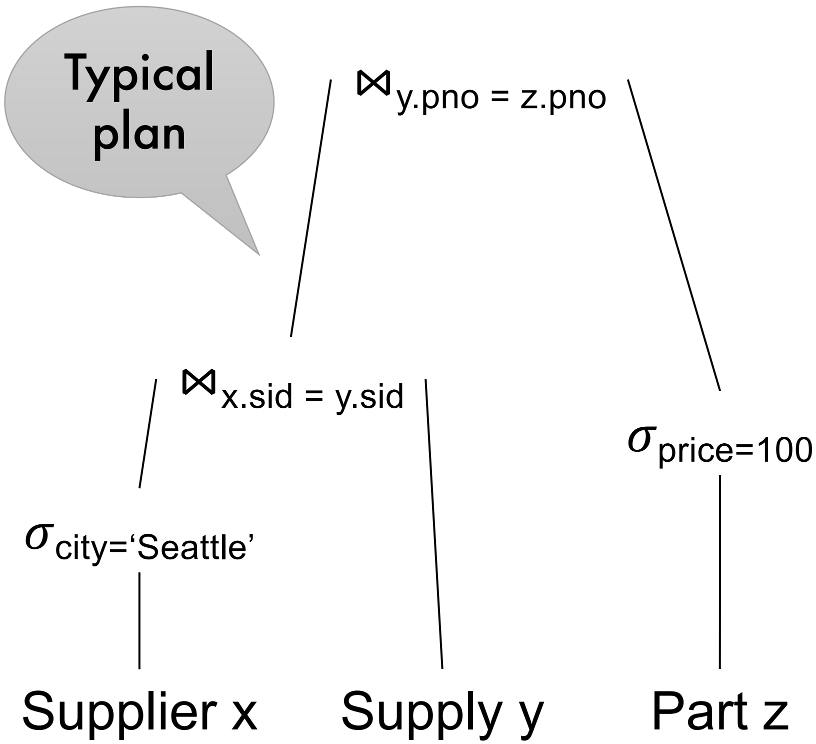
```
Supplier(sid,name,discount,city)
Supply(sid, pno)
Part(pno, pname, price)
```

```
select *
from   Supplier x, Supply y, Part z
where x.sid = y.sid and y.pno = z.pno
   and x.city='Seattle' and z.price=100;
```

Avoid Cartesian Products

```
Supplier(sid,name,discount,city)
Supply(sid, pno)
Part(pno, pname, price)
```

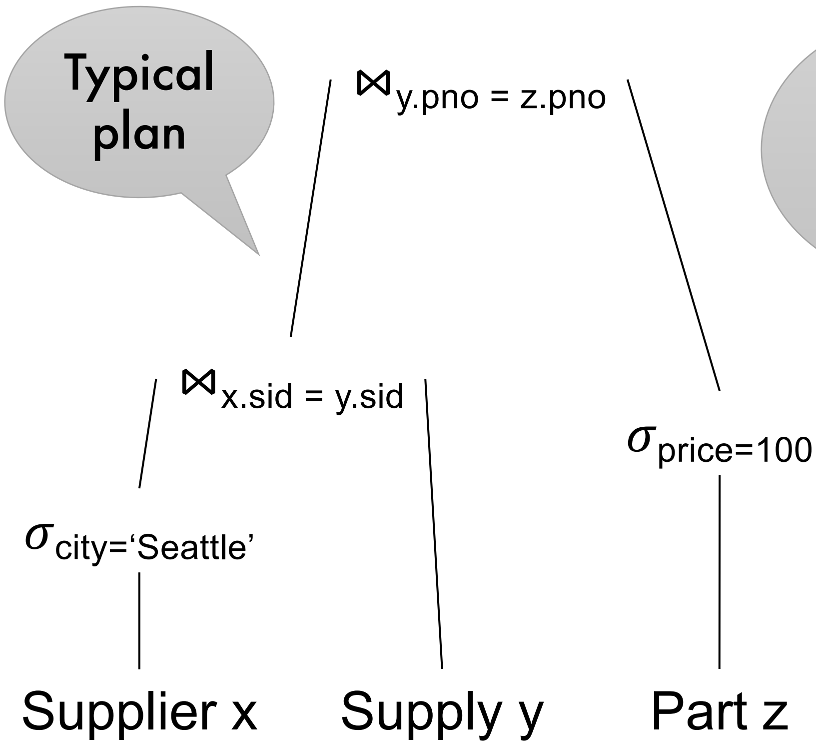
select *
from Supplier x, Supply y, Part z
where x.sid = y.sid and y.pno = z.pno
and x.city='Seattle' and z.price=100;

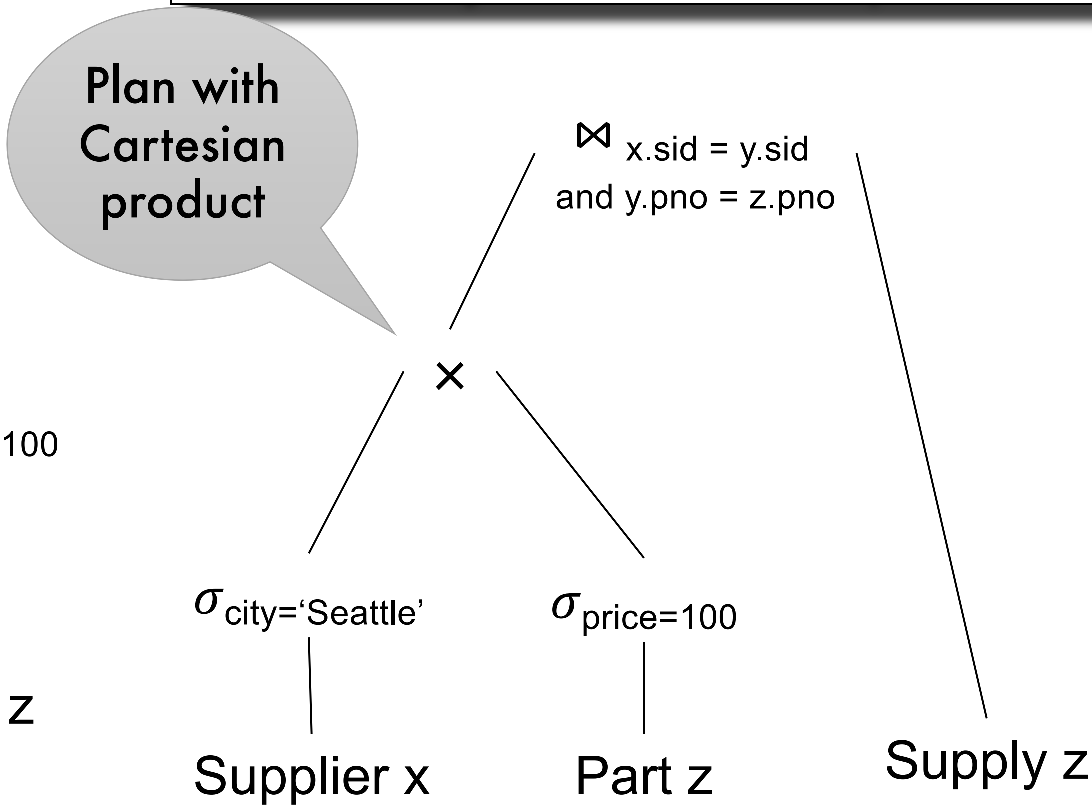


Avoid Cartesian Products

```
Supplier(sid,name,discount,city)
Supply(sid, pno)
Part(pno, pname, price)
```

```
select *
from   Supplier x, Supply y, Part z
where x.sid = y.sid and y.pno = z.pno
   and x.city='Seattle' and z.price=100;
```





Most optimizers will not consider this plan

Query Optimization

Three components:

- Cost/cardinality estimation
- Search space
- Search algorithm ← rest of this lecture

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 discuss these

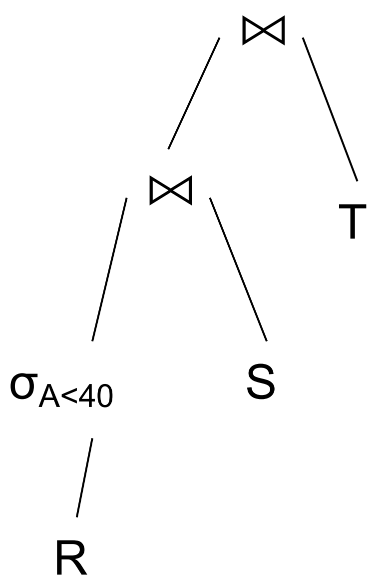
Approaches to Search Space Enumeration

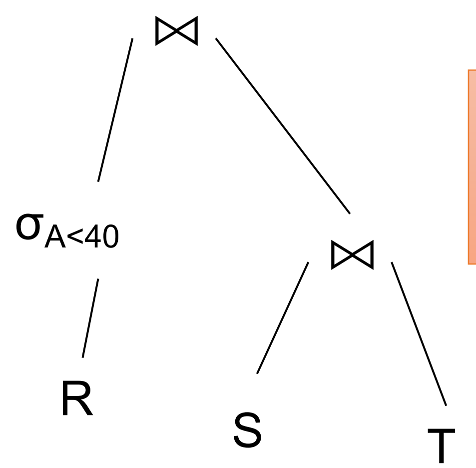
- Complete plans
- Bottom-up plans
- Top-down plans

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
```





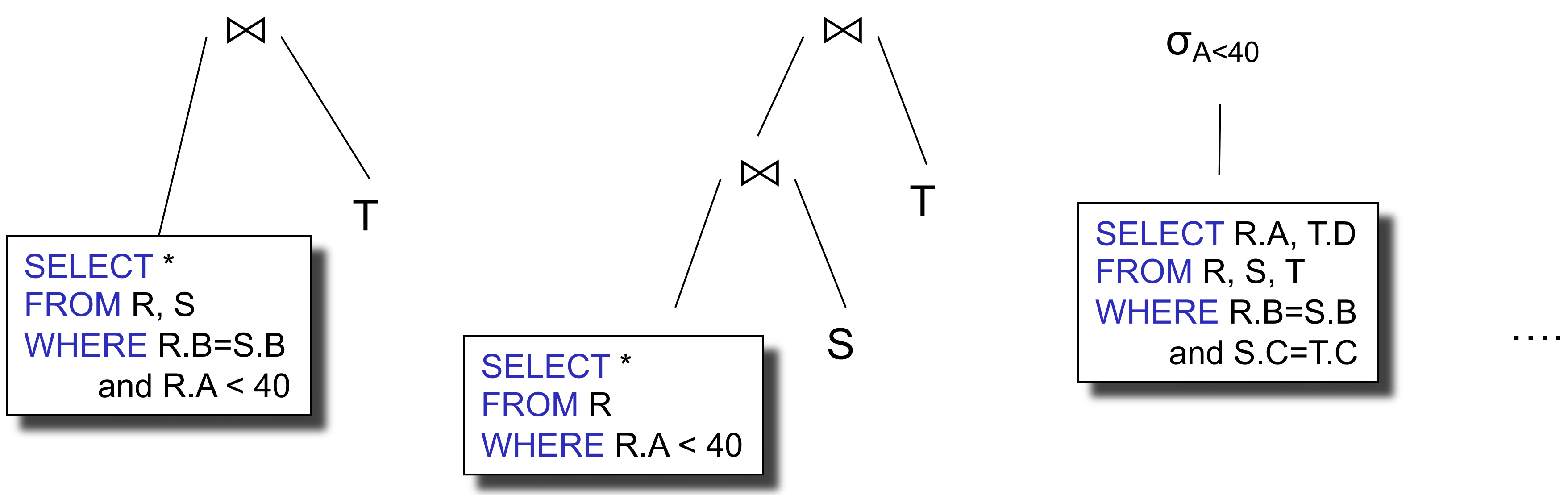
Why is this search space inefficient ?

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
```

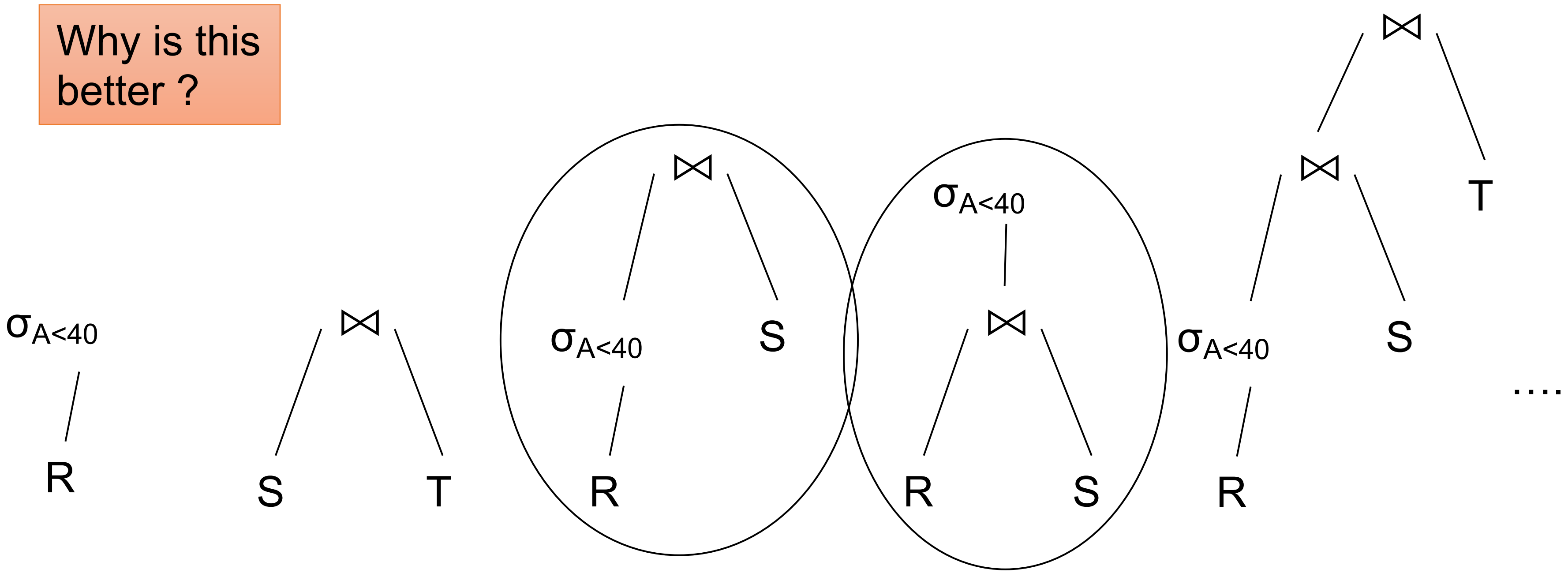


Bottom-up Partial Plans

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

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 $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

For each subquery Q ⊆{R1, …, Rn} compute:

- 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 $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

- **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, …, 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

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		

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

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	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

$$R \bowtie S \text{ and } S \bowtie R$$

  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

How Many Plans Are There?

A bit of math…

- The n'th Catalan number = number of ways to write n pairs of parentheses

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

- n pairs of parentheses go around n+1 items:

How Many Plans Are There?

A bit of math…

- The n'th Catalan number =
  number of ways to write n pairs of parentheses

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

- n pairs of parentheses go around n+1 items:
- 3 items: (AB)C, A(BC) $C_2 = \frac{1}{3}\binom{4}{2} = 2$

How Many Plans Are There?

A bit of math...

- The n'th Catalan number =
  number of ways to write n pairs of parentheses

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

- n pairs of parentheses go around n+1 items:
- 3 items: (AB)C, A(BC) $C_2 = \frac{1}{3}\binom{4}{2} = 2$
- 4 items: ((AB)C)D, (AB)(CD), (A(BC))D, A((BC)D), A(B(CD)) $C_3 = \frac{1}{4}\binom{6}{3} = 5$

How Many Plans Are There?

- The number of plans with n relations $R_1, R_2, \ldots, R_n$ is

$$P_n = n!\, C_{n-1} = \frac{n!}{n}\binom{2(n-1)}{n-1} = \frac{(2(n-1))!}{(n-1)!}$$

- Reason: any parenthesis times any permutation

- E.g. n=4: $P_4 = 6!/3! = 120$
  - $((R_1R_2)R_3)R_4$, $((R_1R_2)R_4)R_3$, $((R_1R_3)R_2)R_4$, $((R_1R_3)R_4)R_2 \ldots$
  - $(R_1R_2)(R_3R_4)$, $(R_1R_2)(R_4R_3), \ldots$
  - $(R_1(R_2R_3))R_4$, $(R_1(R_2R_4))R_3, \ldots$
  - …

Discussion

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

Given a query with n relations R1, ..., Rn

- How many plans are there?
  - A: $(2(n-1))! / (n-1)! = n(n+1)(n+2)\ldots(2n-3)(2n-2)$

- 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 $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

Given a query with n relations R1, ..., Rn

- How many plans are there?
  - A: $(2(n-1))! / (n-1)! = n(n+1)(n+2)\ldots(2n-3)(2n-2)$
- 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

Discussion

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

Given a query with n relations R1, …, Rn
Assume left-linear plans only

- How many plans are there?
- 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 $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

Given a query with n relations R1, …, Rn
Assume left-linear plans only

- How many plans are there?
  - A: $n! = 1*2*3*...*n$

- 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 k plan

Reducing the Search Space

- Left-linear trees
- No cartesian products

Discussion

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

Chain join: $R_1(A_0,A_1) \bowtie R_2(A_1,A_2) \bowtie \ldots \bowtie R_n(A_{n-1},A_n)$

Assume left-linear plans without cartesian product

- How many plans are there?
- 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 $cond_1$ AND $cond_2$ AND . . . AND $cond_k$

Chain join: $R_1(A_0,A_1) \bowtie R_2(A_1,A_2) \bowtie \ldots \bowtie R_n(A_{n-1},A_n)$

Assume left-linear plans without cartesian product

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