CSE544
Data Management

Lectures 10
Query Optimization (part 2)

Announcements

- HW2 was due last night
- HW3 is posted, due on Friday, May 7
- Review 5 was due today
- Review 6 due on Monday, May 3rd

Query Optimization

Three major components:

1. Cardinality and cost estimation
2. Search space
3. Plan enumeration algorithms

Query Optimization

Three major components:

1. Cardinality and cost estimation
2. Search space
3. Plan enumeration algorithms

First, let's discuss the paper

Discuss the paper

- Why do they use the IMDB database instead of TPC-H?
- Do cardinality estimators typically under- or over-estimate?
- From cardinality to cost: how critical is that?

[How good are they]

Single Table Estimation

	median	90th	95th	max
PostgreSQL	1.00	2.08	6.10	207
DBMS A	1.01	1.33	1.98	43.4
DBMS B	1.00	6.03	30.2	104000
DBMS C	1.06	1677	5367	20471
HyPer	1.02	4.47	8.00	2084

Table 1: Q-errors for base table selections

Discuss histograms v.s. samples

Single Table Estimation

- 1d Histograms: accurate for selection on a single equality or range predicate; poor for multiple predicates; useless for LIKE
- Samples: great for correlations, or predicates like LIKE; poor for low selectivity predicates: estimate is 0, then use ”magic constants”

[How good are they]

Joins (0 to 6)

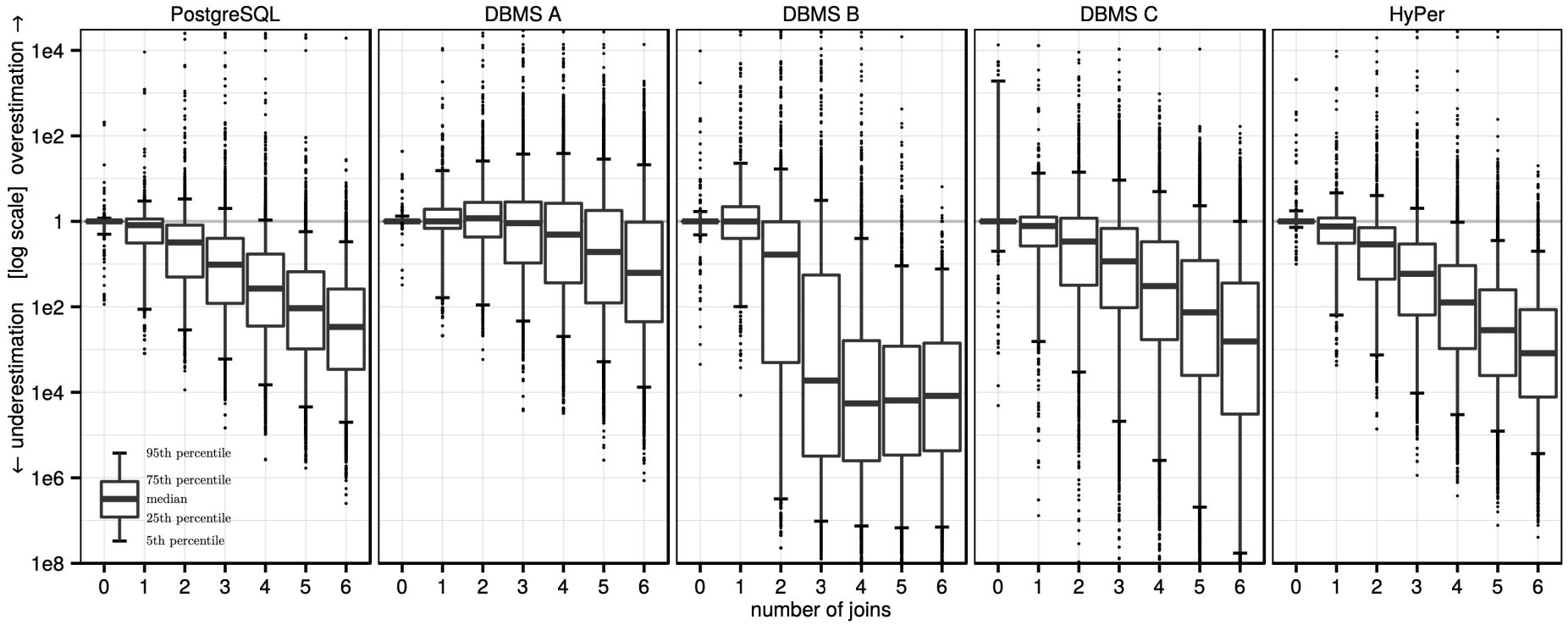


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

[How good are they]

Joins (0 to 6)

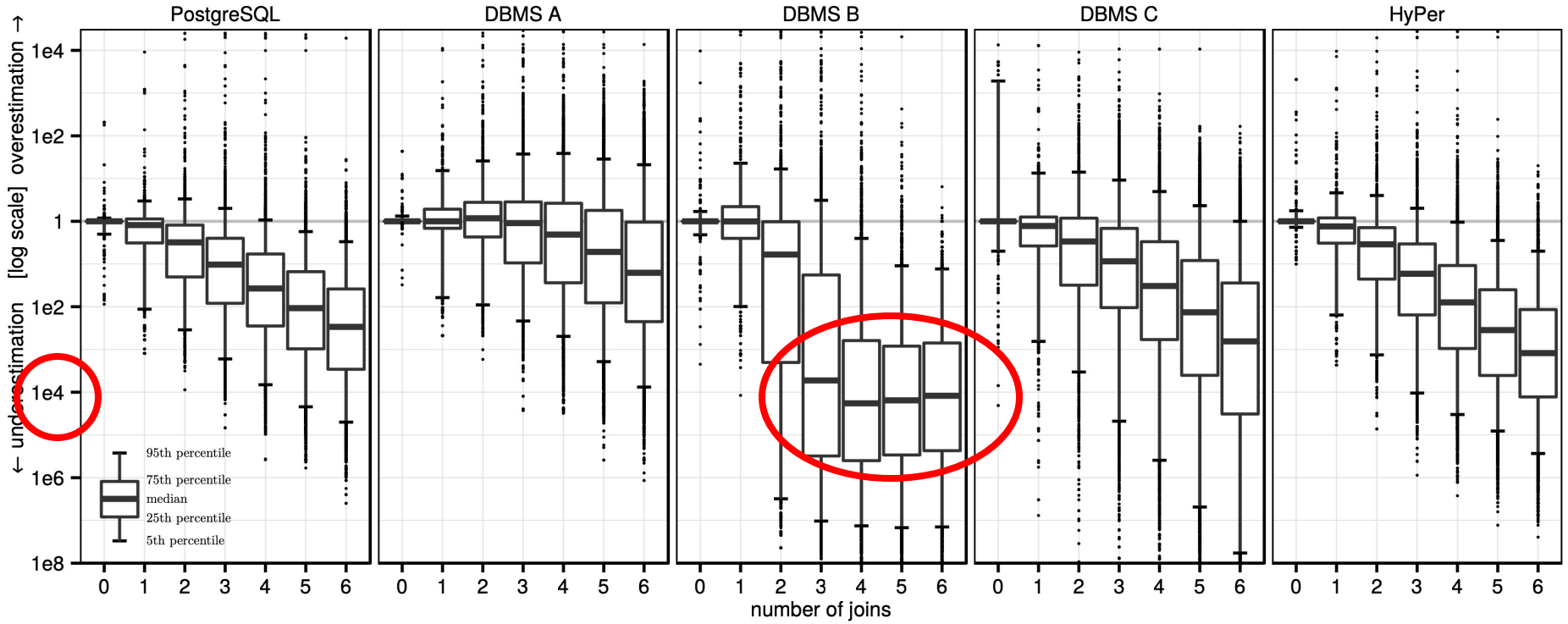
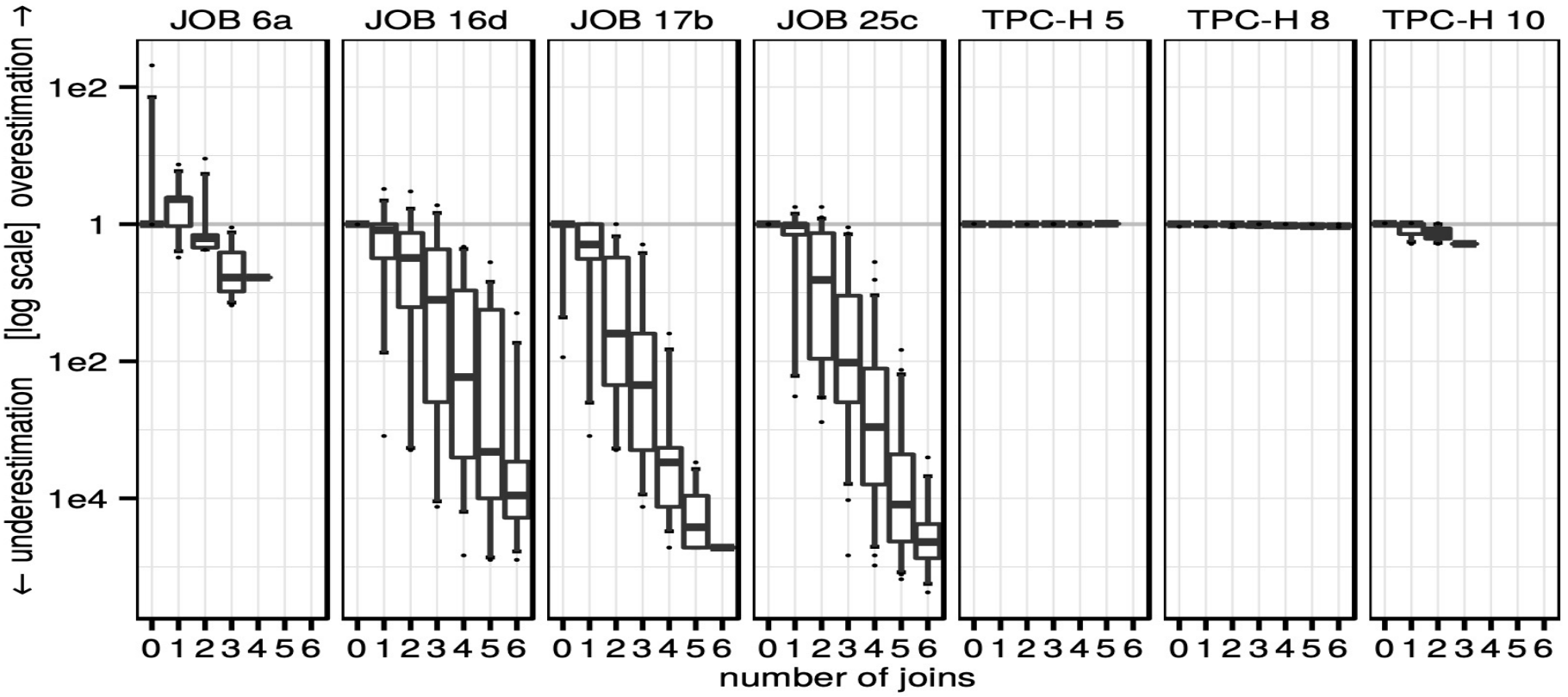


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

[How good are they]

TPC-H v.s. Real Data (IMDB)



[How good are they]

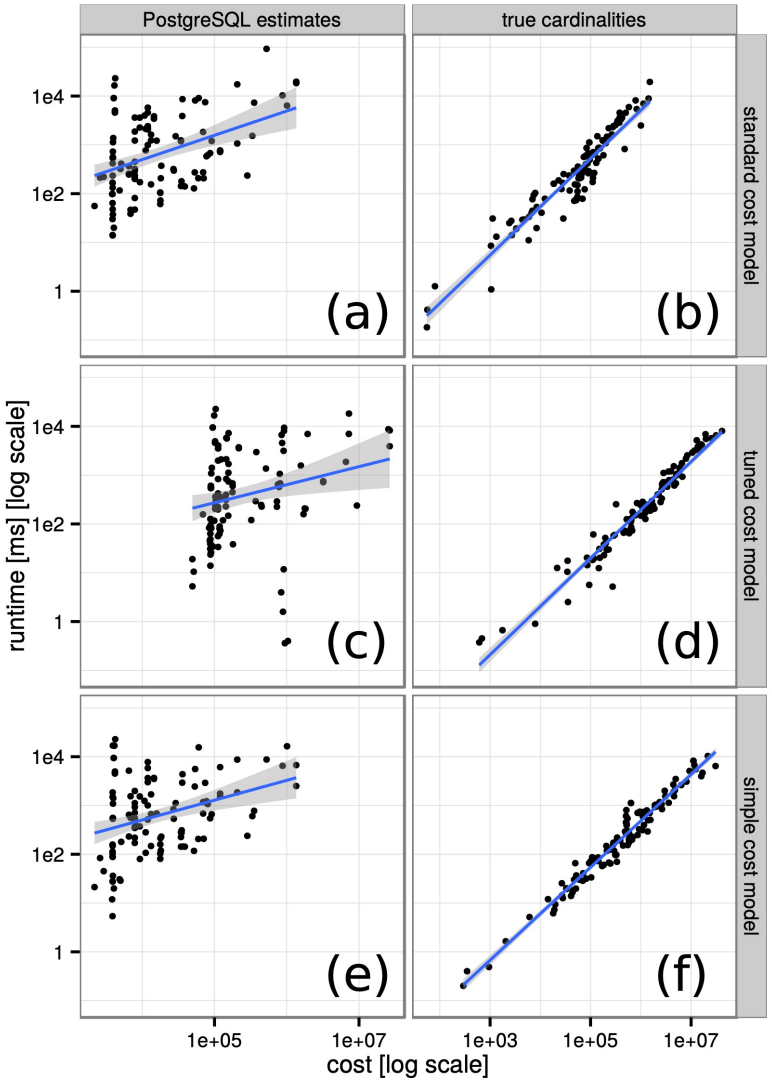
Cardinalities to Cost

- Cardinality estimation creates largest errors
- Complex or simple cost models don't differ much

Postgres cost

No I/O, keep only CPU

Their own simple formula



Yet Another Difficulties

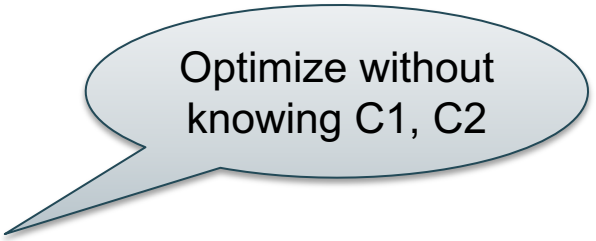
SQL Queries issued from applications:

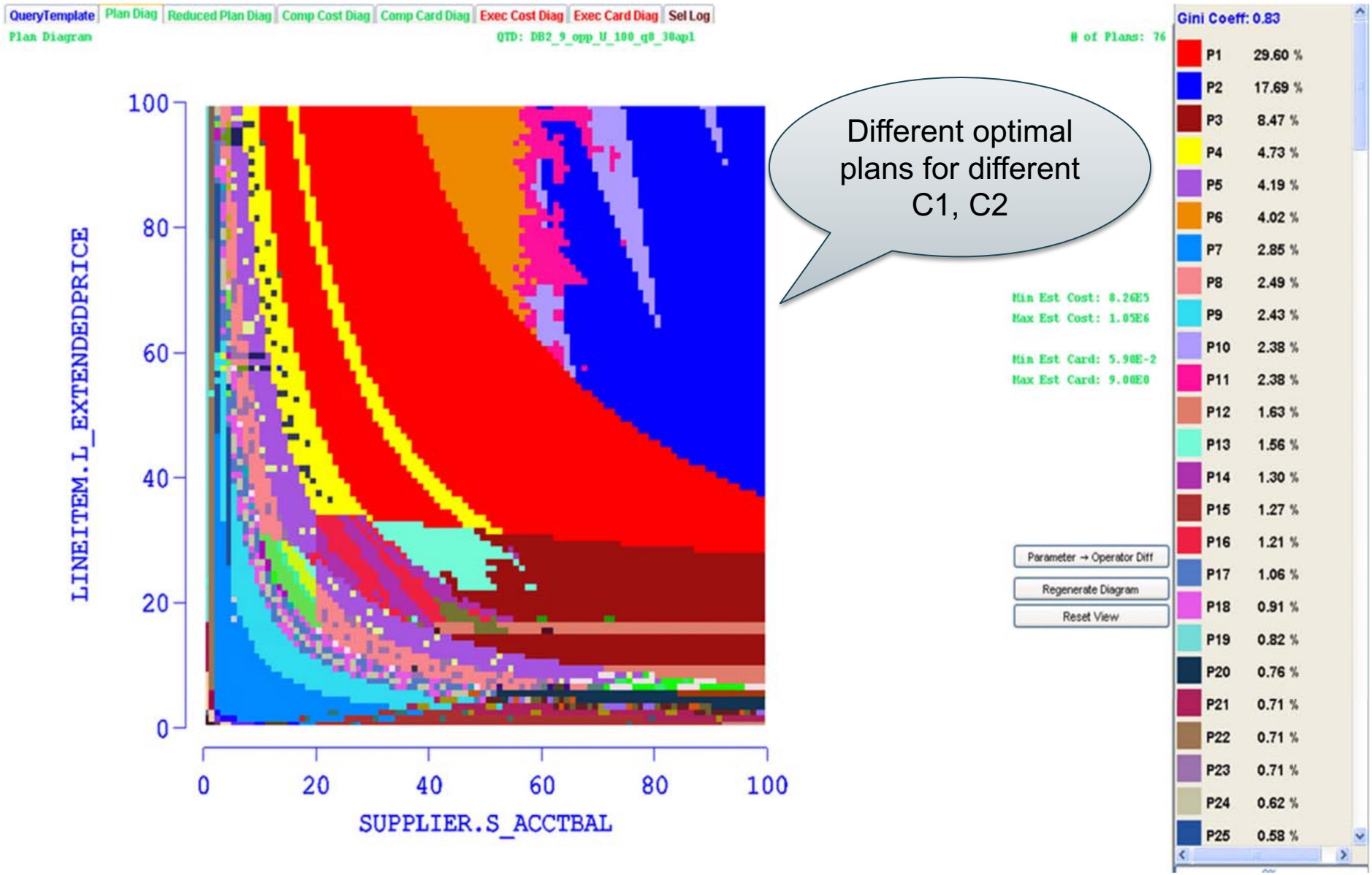
- Query is optimized once: *prepare*
- Then, executed repeatedly

Query constants are unknow until execution: optimized plan is suboptimal

```
select
  o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume)
from
 (select YEAR(o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
                  n2.n_name as nation
  from part, supplier, lineitem, orders,
       customer, nation n1, nation n2, region
  where p_partkey = l_partkey and s_suppkey = l_suppkey
    and l_orderkey = o_orderkey and o_custkey = c_custkey
    and c_nationkey = n1.n_nationkey
    and n1.n_regionkey = r_regionkey
    and r_name = 'AMERICA'
    and s_nationkey = n2.n_nationkey
    and o_orderdate between '1995-01-01'
    and '1996-12-31'
    and p_type = 'ECONOMY ANODIZED STEEL'
   and s_acctbal ≤ C1 and l_extendedprice ≤ C2 ) as all_nations
group by o_year order by o_year
```

```
select
  o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume)
from
 (select YEAR(o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
                  n2.n_name as nation
  from part, supplier, lineitem, orders,
       customer, nation n1, nation n2, region
  where p_partkey = l_partkey and s_suppkey = l_suppkey
    and l_orderkey = o_orderkey and o_custkey = c_custkey
    and c_nationkey = n1.n_nationkey
    and n1.n_regionkey = r_regionkey
    and r_name = 'AMERICA'
    and s_nationkey = n2.n_nationkey
    and o_orderdate between '1995-01-01'
    and '1996-12-31'
    and p_type = 'ECONOMY ANODIZED STEEL'
   and s_acctbal ≤ C1 and l_extendedprice ≤ C2 ) as all_nations
group by o_year order by o_year
```




QueryTemplate
Plan Diag
Reduced Plan Diag
Comp Cost Diag
Comp Card Diag
Exec Cost Diag
Exec Card Diag
Sel Log
Plan Diagram
QTD: DB2_9_opp_U_100_q8_30apl
# of Plans: 76
Different optimal plans for different C1, C2
LINEITEM.L_EXTENDEDPRICE
SUPPLIER.S_ACCTBAL
0
20
40
60
80
100
Min Est Cost: 8.26E5
Max Est Cost: 1.05E6
Min Est Card: 5.90E-2
Max Est Card: 9.00E0
Parameter → Operator Diff
Regenerate Diagram
Reset View
Gini Coeff: 0.83
P1 29.60 %
P2 17.69 %
P3 8.47 %
P4 4.73 %
P5 4.19 %
P6 4.02 %
P7 2.85 %
P8 2.49 %
P9 2.43 %
P10 2.38 %
P11 2.38 %
P12 1.63 %
P13 1.56 %
P14 1.30 %
P15 1.27 %
P16 1.21 %
P17 1.06 %
P18 0.91 %
P19 0.82 %
P20 0.76 %
P21 0.71 %
P22 0.71 %
P23 0.71 %
P24 0.62 %
P25 0.58 %

Query Optimization

Three major components:

1. Cardinality and cost estimation
2. Search space
3. Plan enumeration algorithms

Search Space

Refers to the set of all possible alternate plans that the optimizers may explore

- **Access Path Selection**: how to use indices
- **Rewrite Rules**: what identities the optimizer applies

Access Path

Access path: implements a selection $\sigma_P(R)$

(P is sometimes called *search argument* SARG)

- A file scan, or
- An index *plus* a matching selection condition

Supplier(sid,sname,scity,sstate)

Access Path Selection

```
SELECT * FROM Supplier
WHERE sid > 300 ∧ scity='Seattle'
```

B(Supplier) = 100
T(Supplier) = 1000
V(Supplier,scity) = 20
Max(Supplier, sid) = 1000
Min(Supplier,sid) =1

Indices:

B+-tree on sid; clustered

B+-tree on scity; unclustered

Which access path should we use?

Access Path Selection

```
SELECT * FROM Supplier
WHERE sid > 300 ∧ scity='Seattle'
```

B(Supplier) = 100
T(Supplier) = 1000
V(Supplier,scity) = 20
Max(Supplier, sid) = 1000
Min(Supplier,sid) =1

Indices:

B+-tree on sid; clustered

B+-tree on scity; unclustered

Which access path should we use?

1. Sequential scan: cost = 100

Supplier(sid,sname,scity,sstate)

Access Path Selection

```
SELECT * FROM Supplier
WHERE sid > 300 ∧ scity='Seattle'
```

B(Supplier) = 100
T(Supplier) = 1000
V(Supplier,scity) = 20
Max(Supplier, sid) = 1000
Min(Supplier,sid) =1

Indices:

B+-tree on sid; clustered

B+-tree on scity; unclustered

Which access path should we use?

1. Sequential scan: cost = 100
2. Index scan on sid: cost = 7/10 * 100 = 70

Supplier(sid,sname,scity,sstate)

Access Path Selection

```
SELECT * FROM Supplier
WHERE sid > 300 ∧ scity='Seattle'
```

B(Supplier) = 100
T(Supplier) = 1000
V(Supplier,scity) = 20
Max(Supplier, sid) = 1000
Min(Supplier,sid) =1

Indices:

B+-tree on sid; clustered
B+-tree on scity; unclustered

Which access path should we use?

1. Sequential scan: cost = 100
2. Index scan on sid: cost = 7/10 * 100 = 70
3. Index scan on scity: cost = 1000/20 = 50

Rewrite Rules

Search space is defined by the set of rewrite rules that the optimizer implements

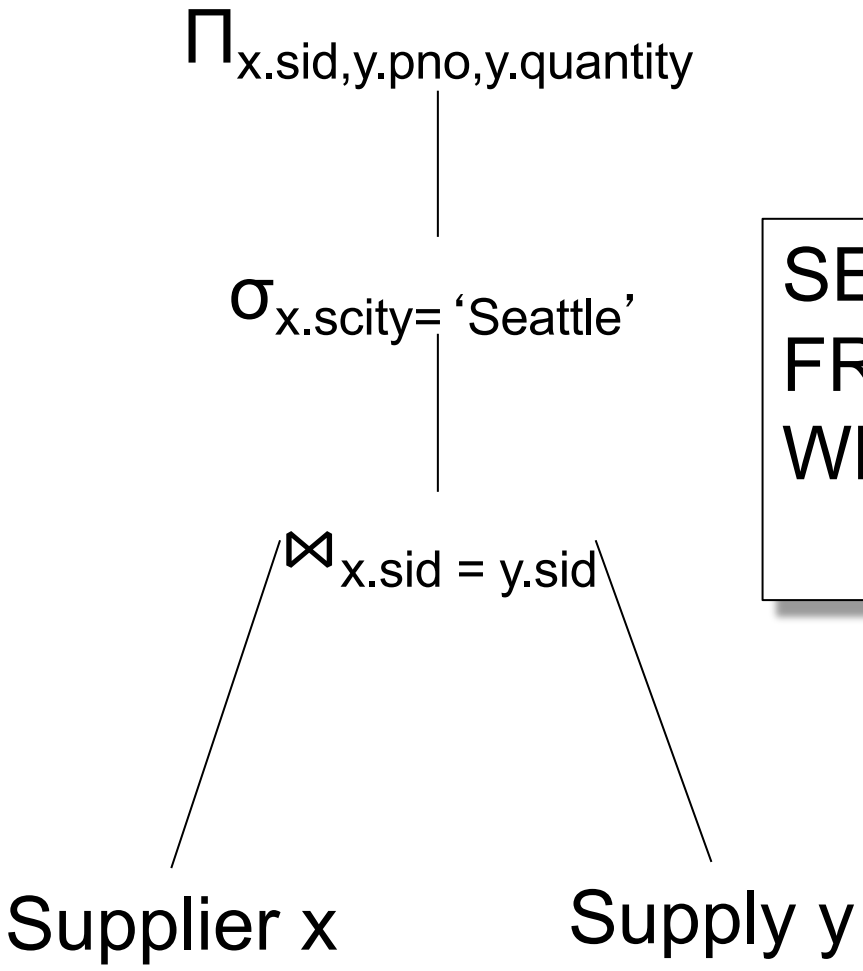
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Example Optimization

```
SELECT x.sid, y.pno, y.quantity
FROM.   Supplier x, Supply y
WHERE x.sid = y.sid
        and x.scity = 'Seattle'
```

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

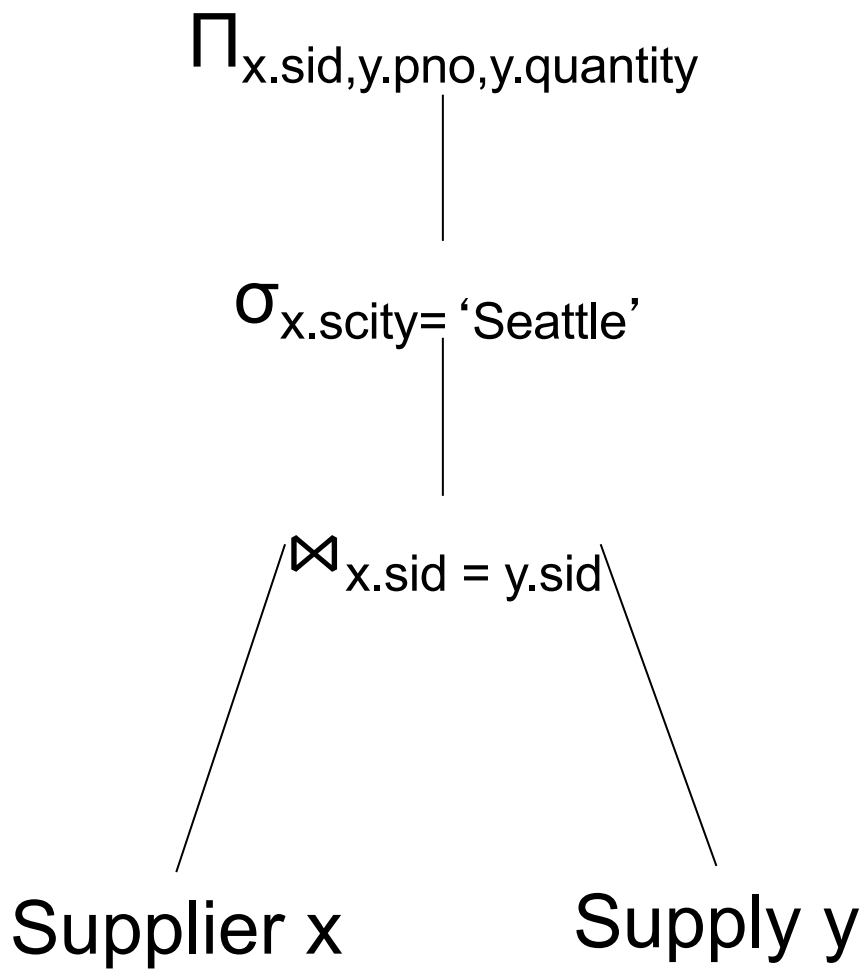
Example Optimization




```
SELECT x.sid, y.pno, y.quantity
FROM.   Supplier x, Supply y
WHERE x.sid = y.sid
       and x.scity = 'Seattle'
```

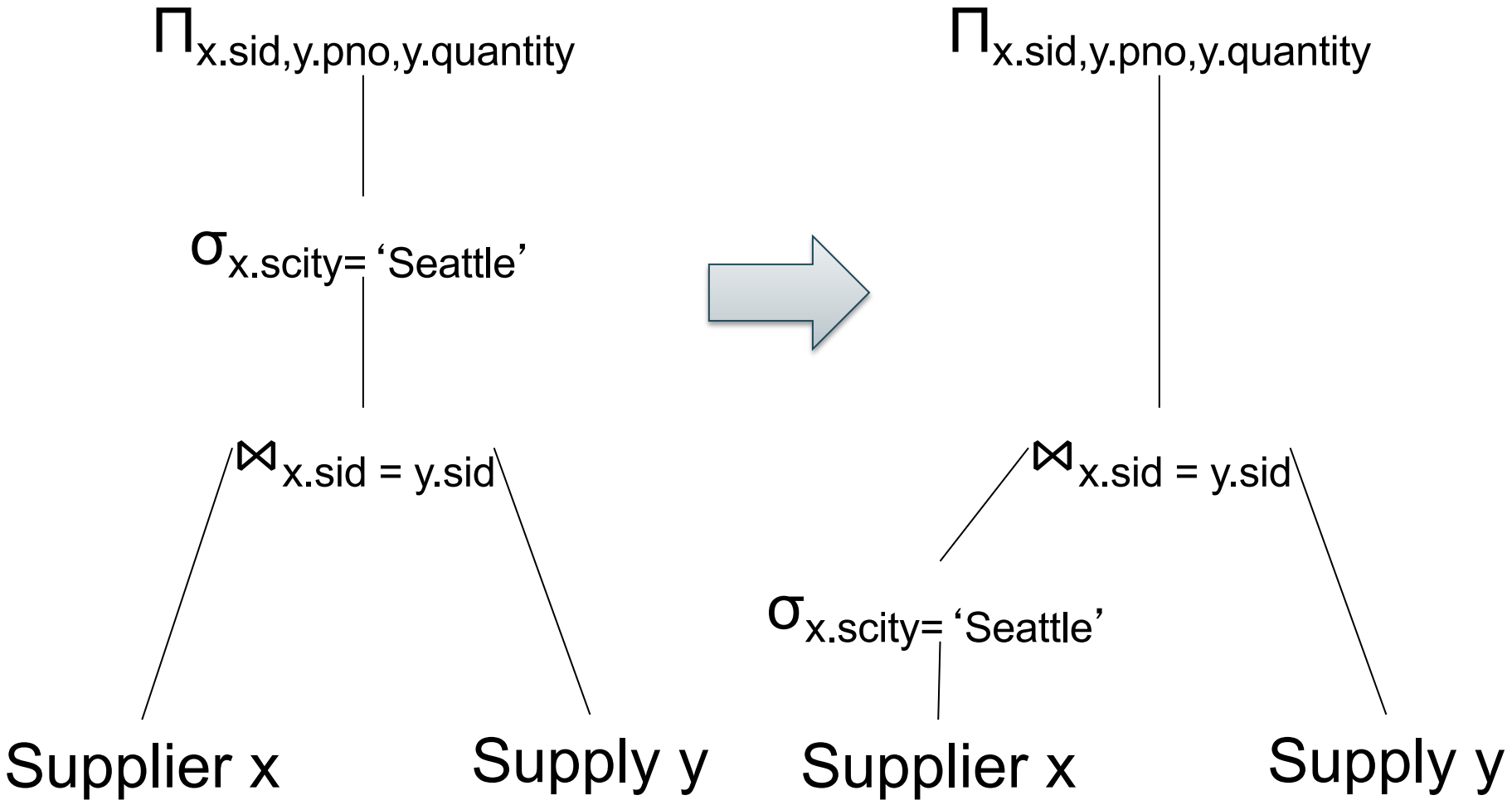
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



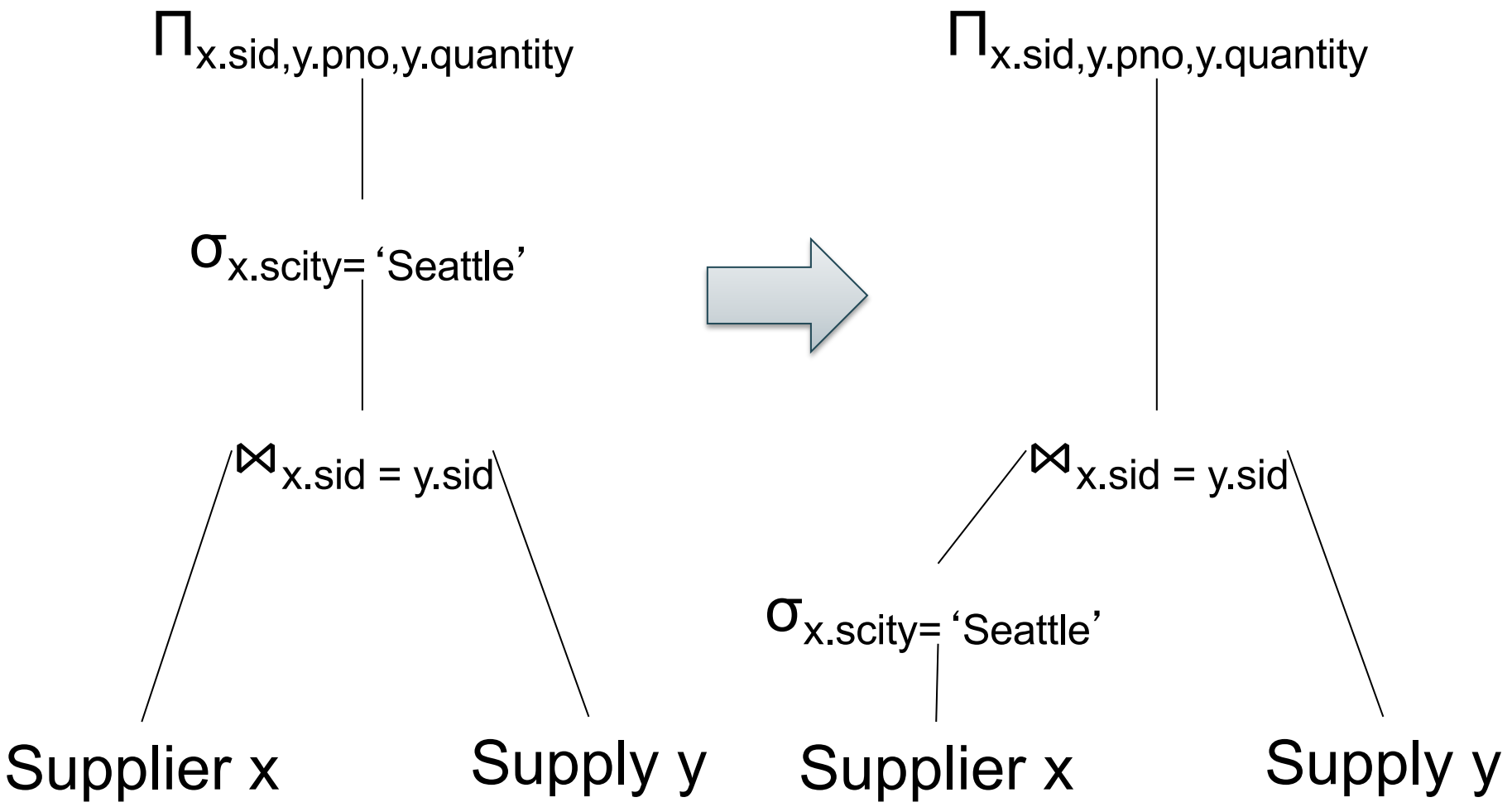
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



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

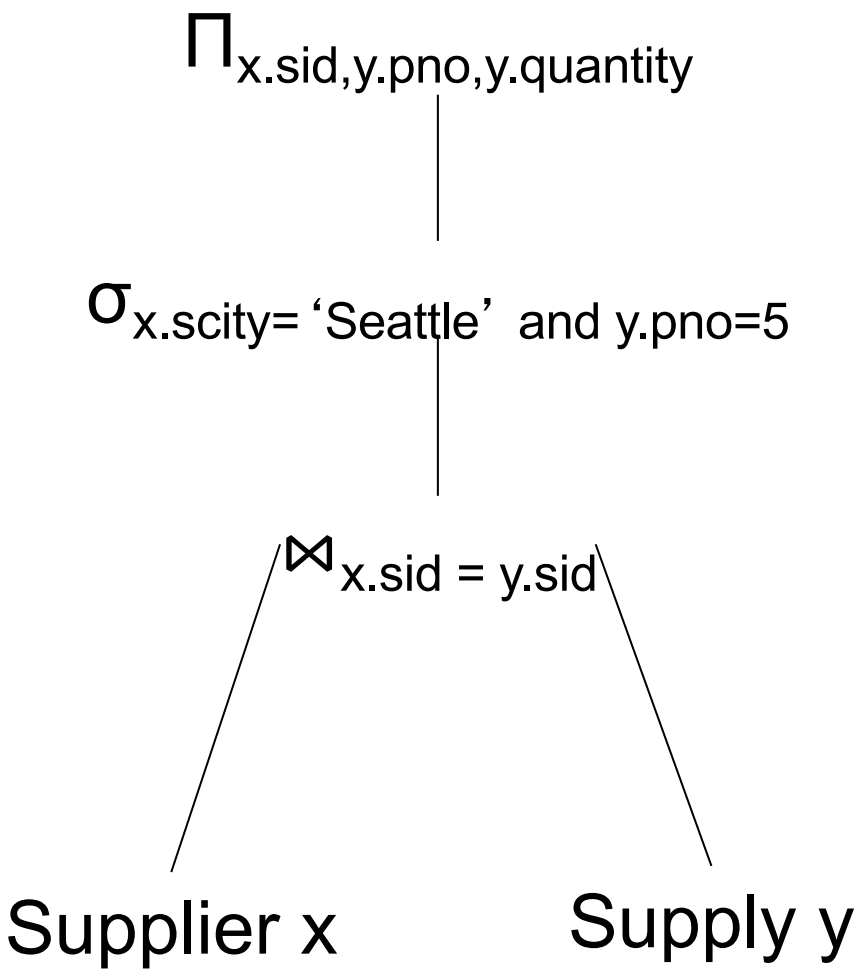
Push Selections Down



$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ when C refers only to R

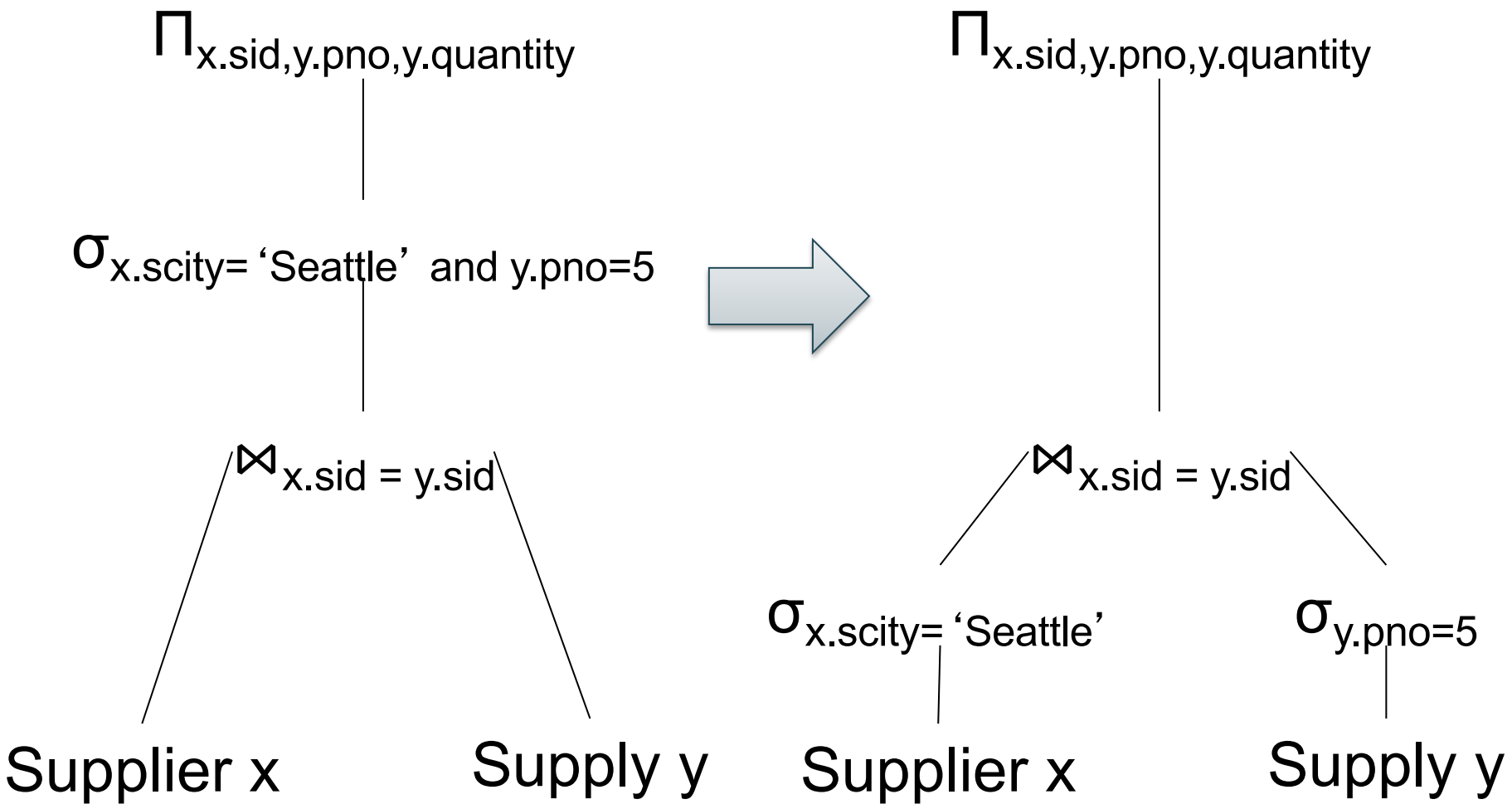
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



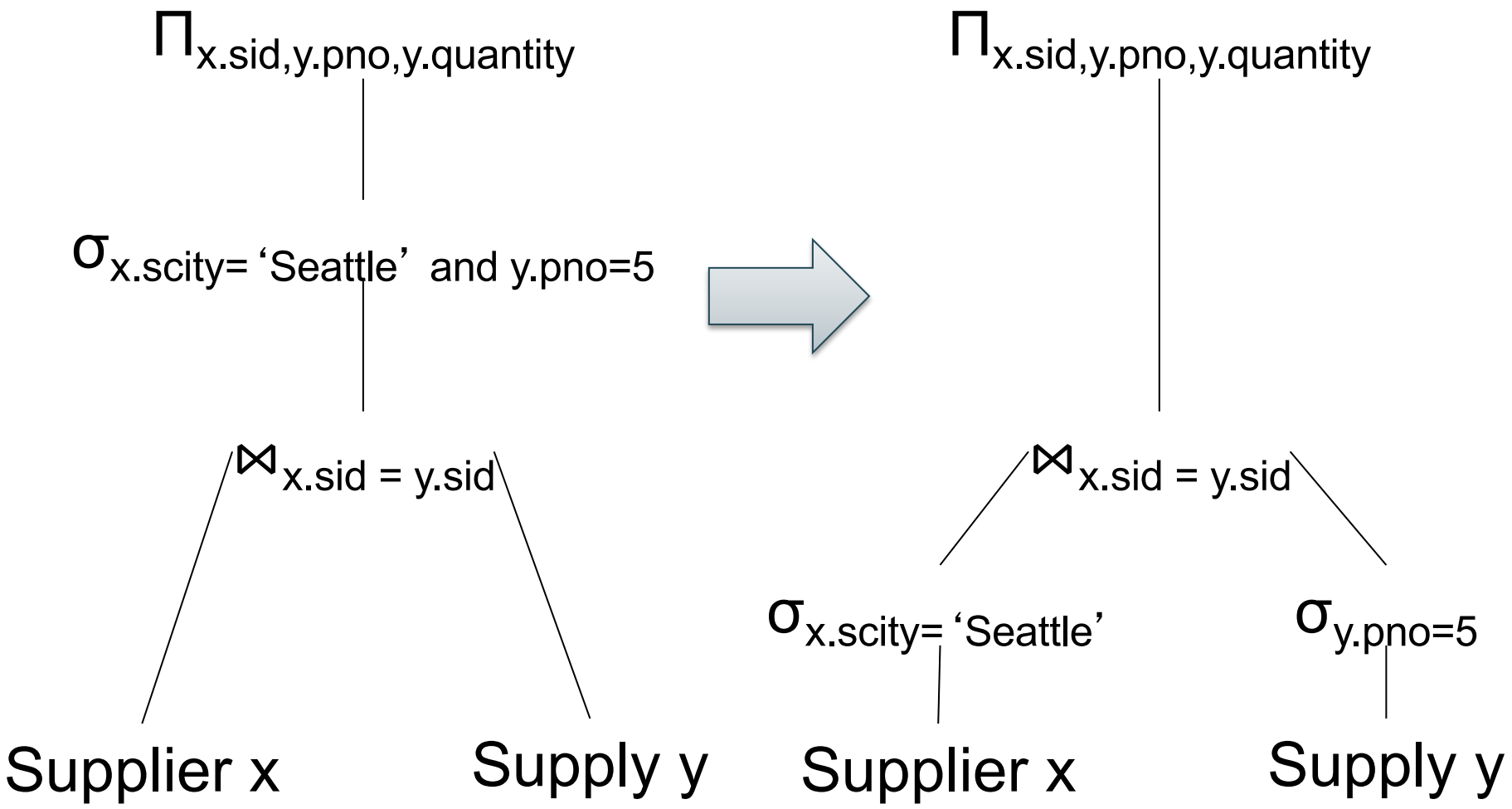
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



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

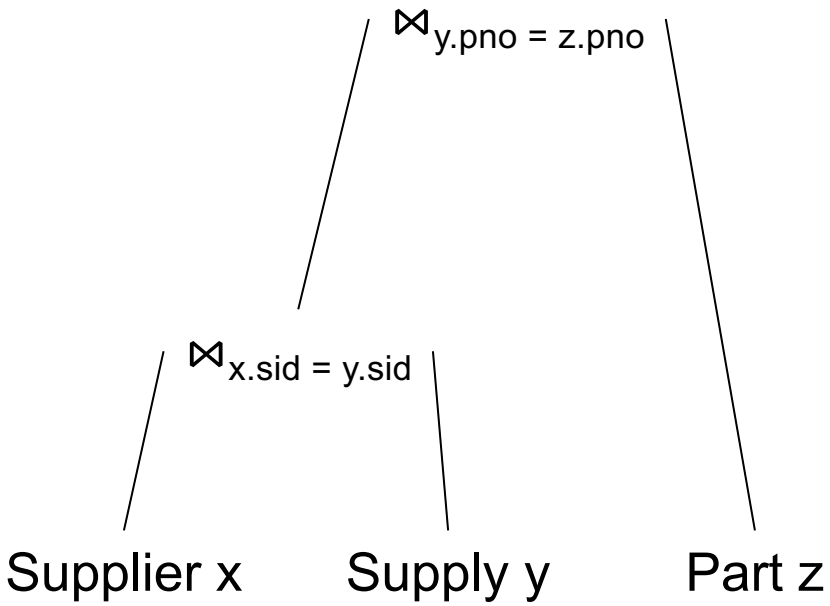
Push Selections Down



$$\sigma_{C1 \text{ and } C2}(R \bowtie S) = \sigma_{C1}(\sigma_{C2}(R \bowtie S)) = \sigma_{C1}(R \bowtie \sigma_{C2}(S)) = \sigma_{C1}(R) \bowtie \sigma_{C2}(S)$$

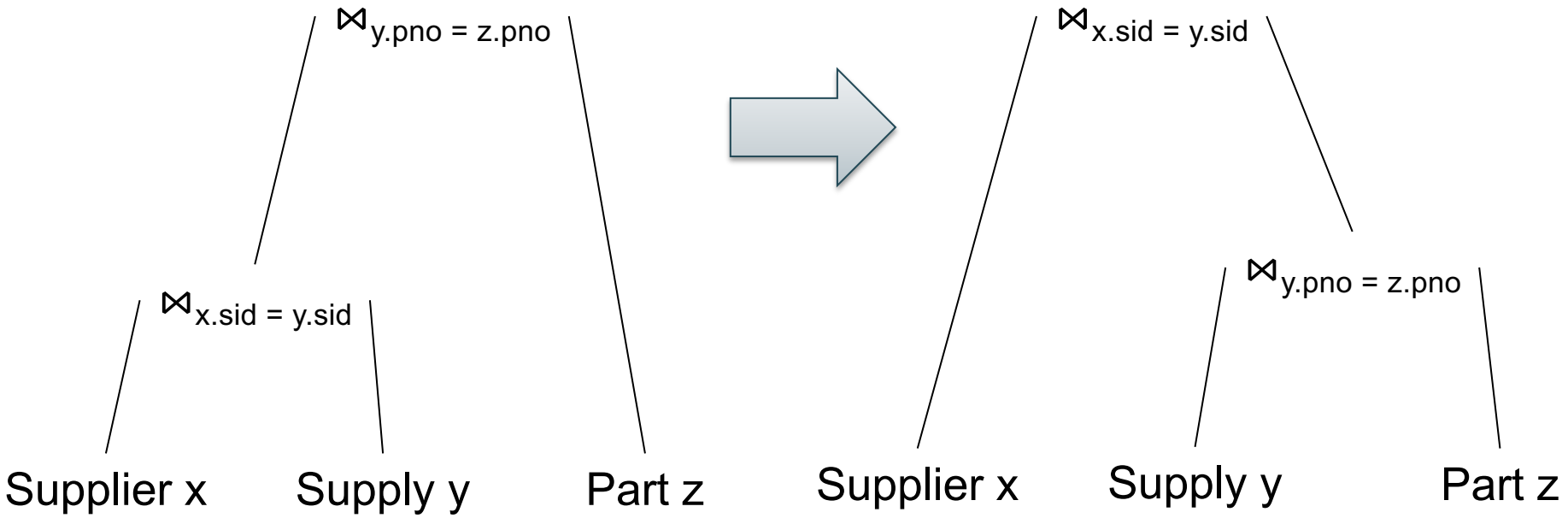
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder



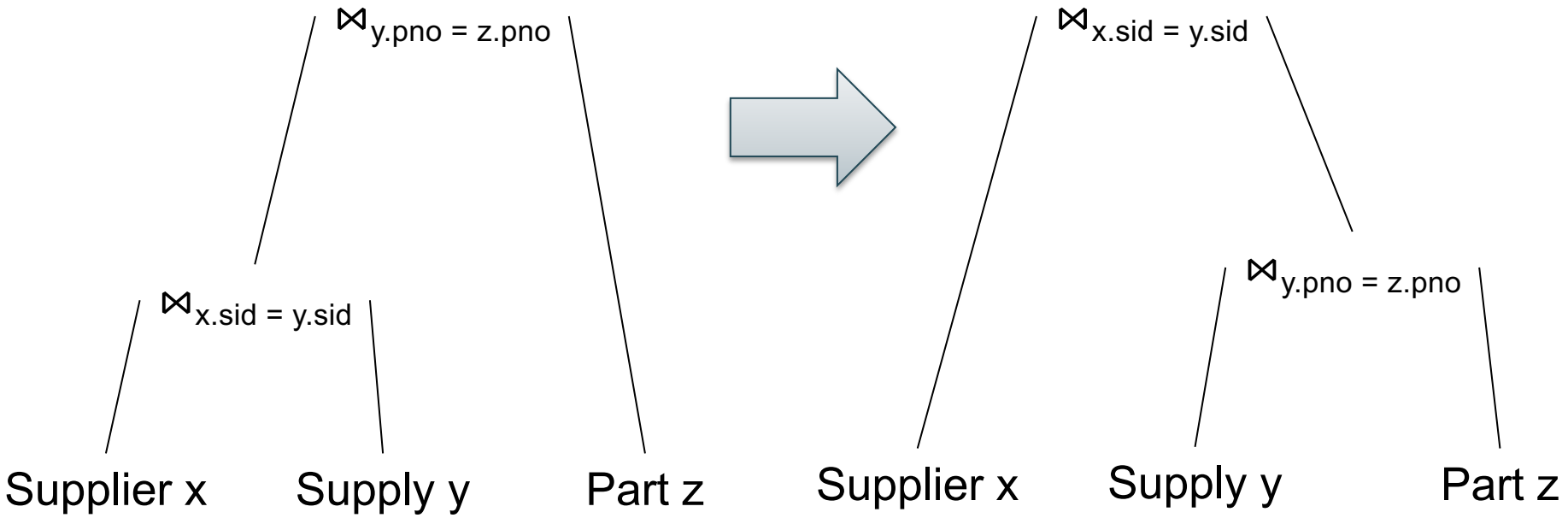
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder



Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder



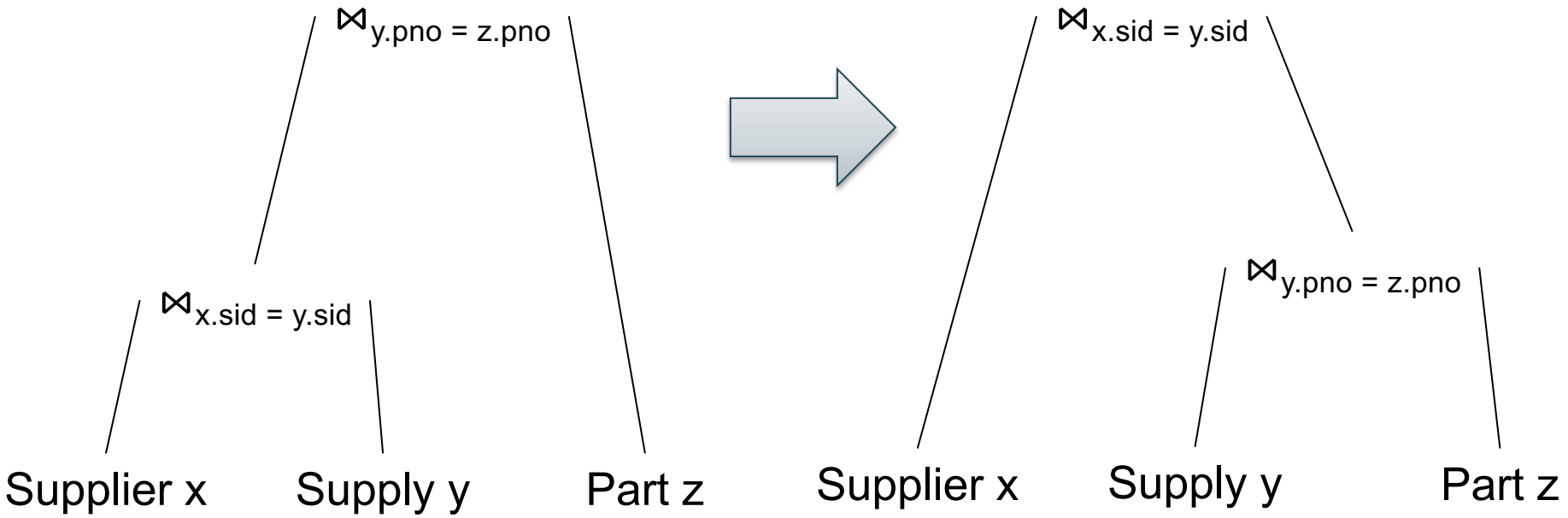
$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$R \bowtie S = S \bowtie R$$

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder

Which plan is better?



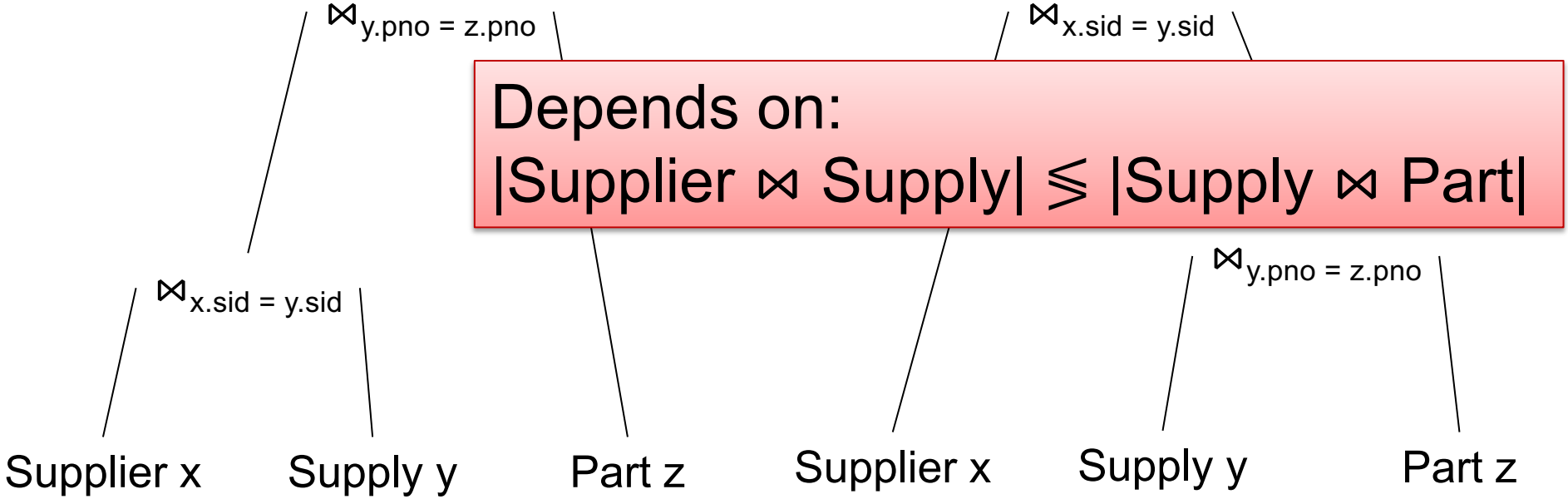
$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

$R \bowtie S = S \bowtie R$

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder

Which plan is better?



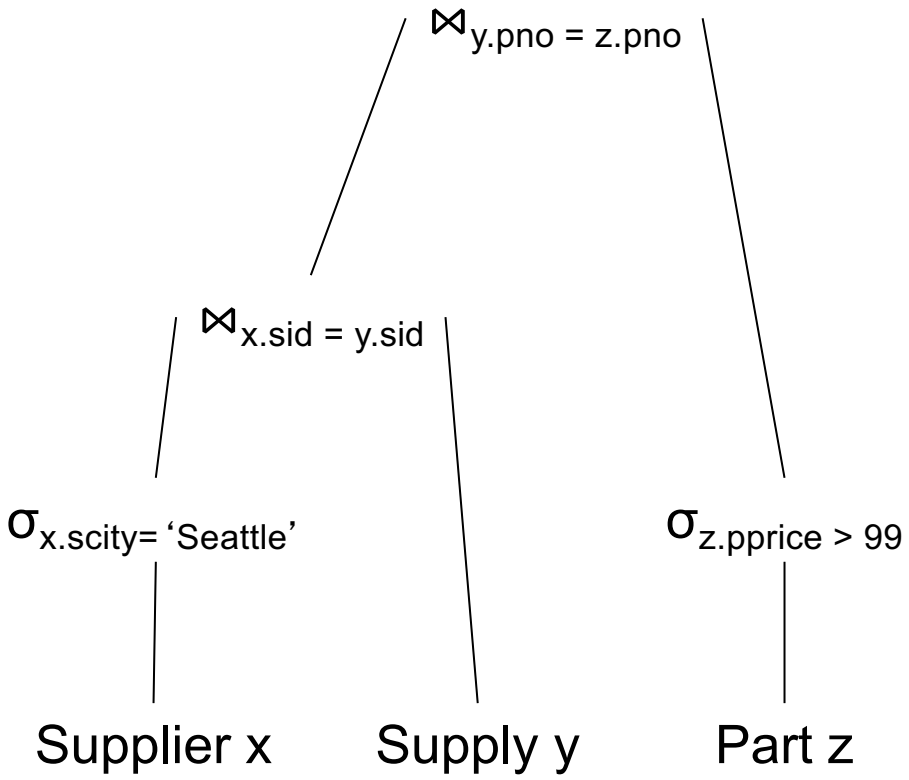
Depends on:
|Supplier ⋈ Supply| ⩽ |Supply ⋈ Part|

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

$R \bowtie S = S \bowtie R$

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

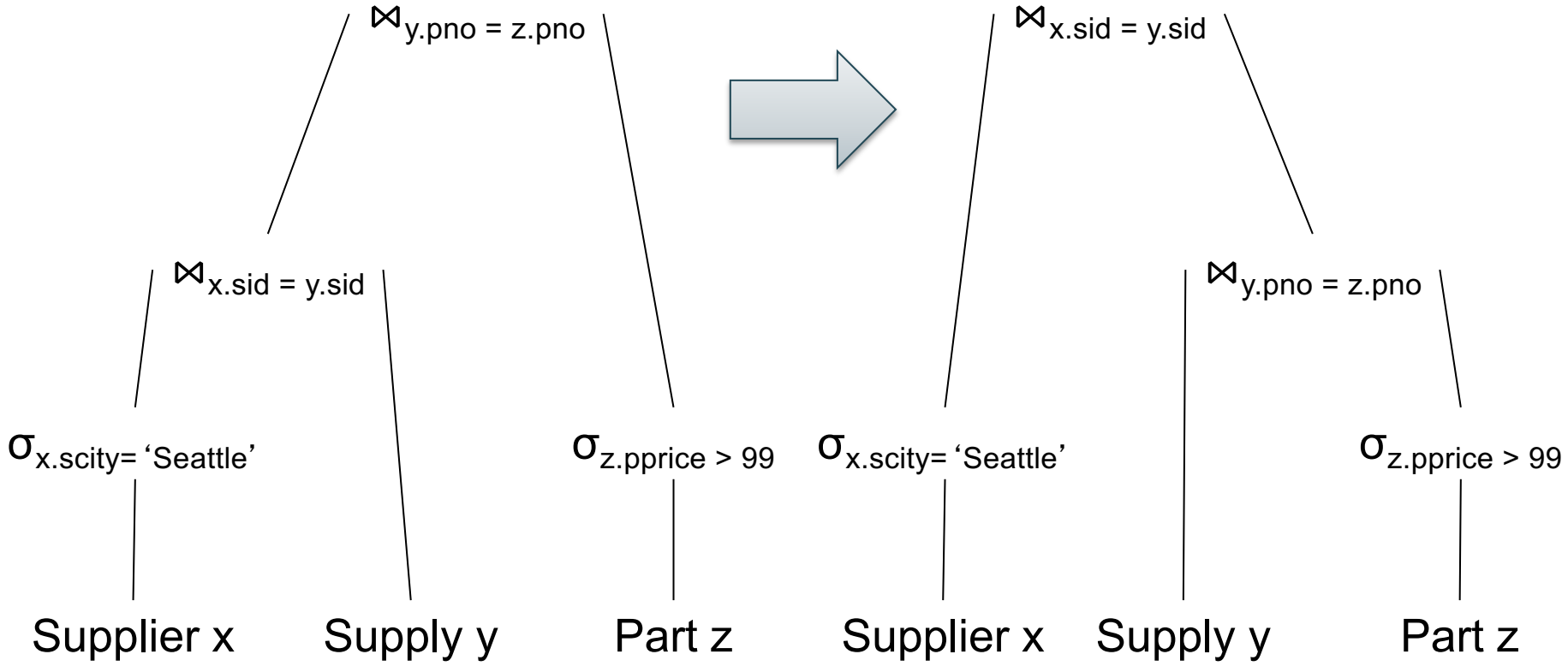
Join Reorder



Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder

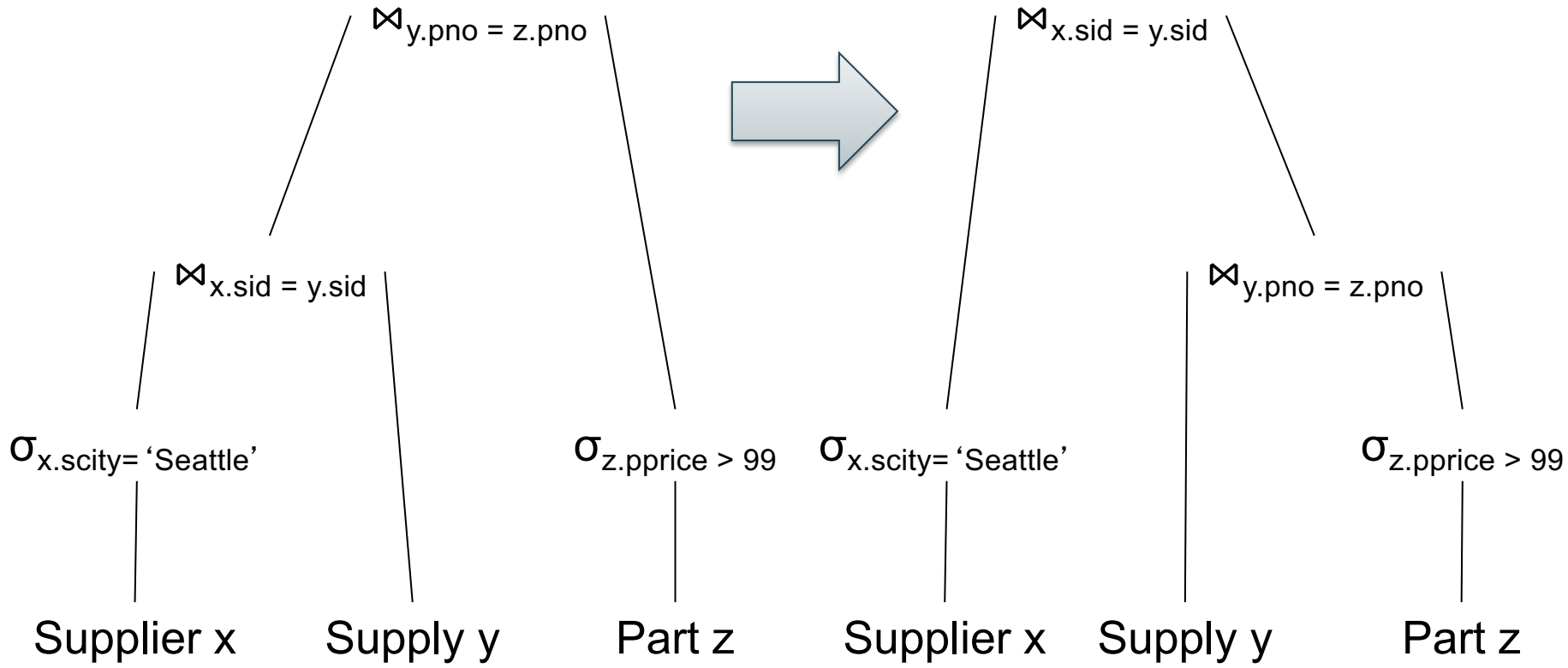
Which plan is better?



Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder

Which plan is better?



Lesson: need sizes of $\sigma_{x.scity= \text{'Seattle'}}$(Supplier), $\sigma_{z.pprice > 99}$(Part)

Discussion

- Both rewrite rules **and** cardinality estimations are needed for optimization
- They can be developed independently

- Next: a powerful optimization rule: aggregate pushdown or better known as generalized distributivity law

Motivation

- Try this in postgres

```
select count(*) from author;
```

Answer: 2652053
Time: 0.058 s

Motivation

- Try this in postgres

```
select count(*) from author;
```

Answer: 2652053
Time: 0.058 s

```
select count(*) from publication;
```

Answer: 5120896
Time: 0.062 s

Motivation

- Try this in postgres

```
select count(*) from author;
```

Answer: 2652053
Time: 0.058 s

```
select count(*) from publication;
```

Answer: 5120896
Time: 0.062 s

```
select count(*) from author, publication;
```

Timeout!!!

Motivation

- Try this in postgres

```
select count(*) from author;
```

Answer: 2652053
Time: 0.058 s

```
select count(*) from publication;
```

Answer: 5120896
Time: 0.062 s

```
select count(*) from author, publication;
```

Timeout!!!

But 3rd query is simply the **product** of the first two!

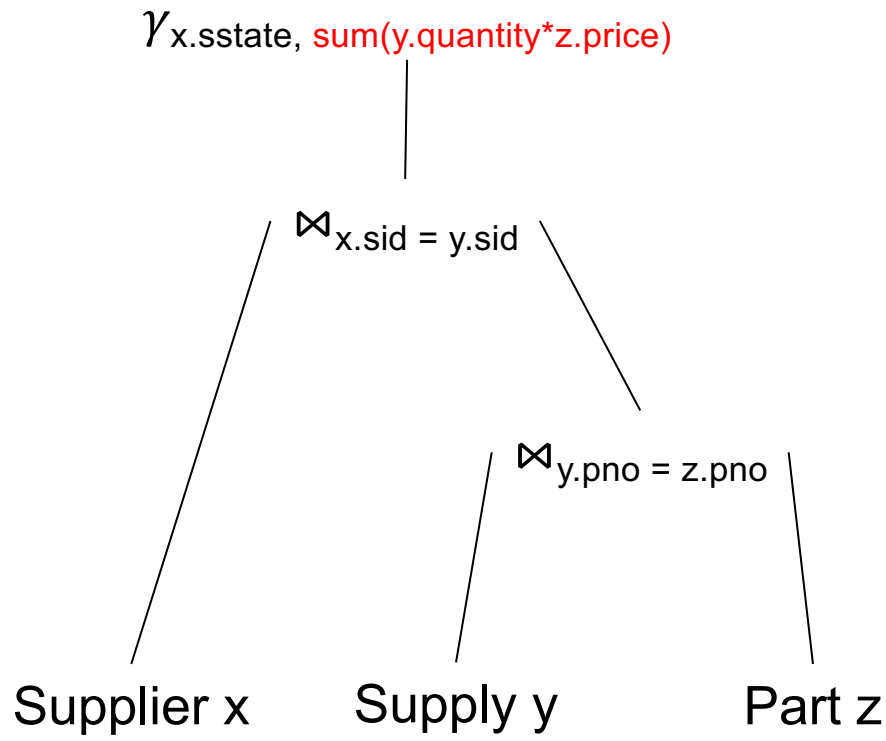
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Aggregate Push-down

```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

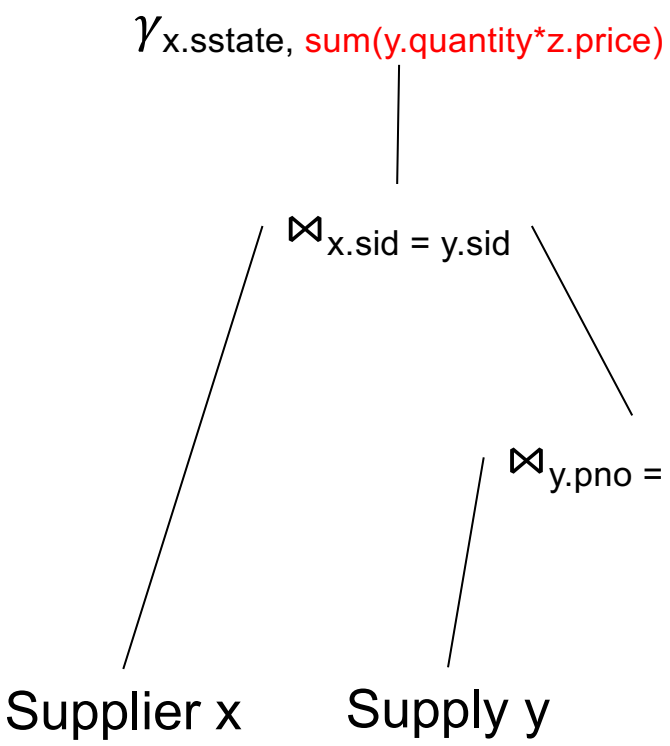
Aggregate Push-down

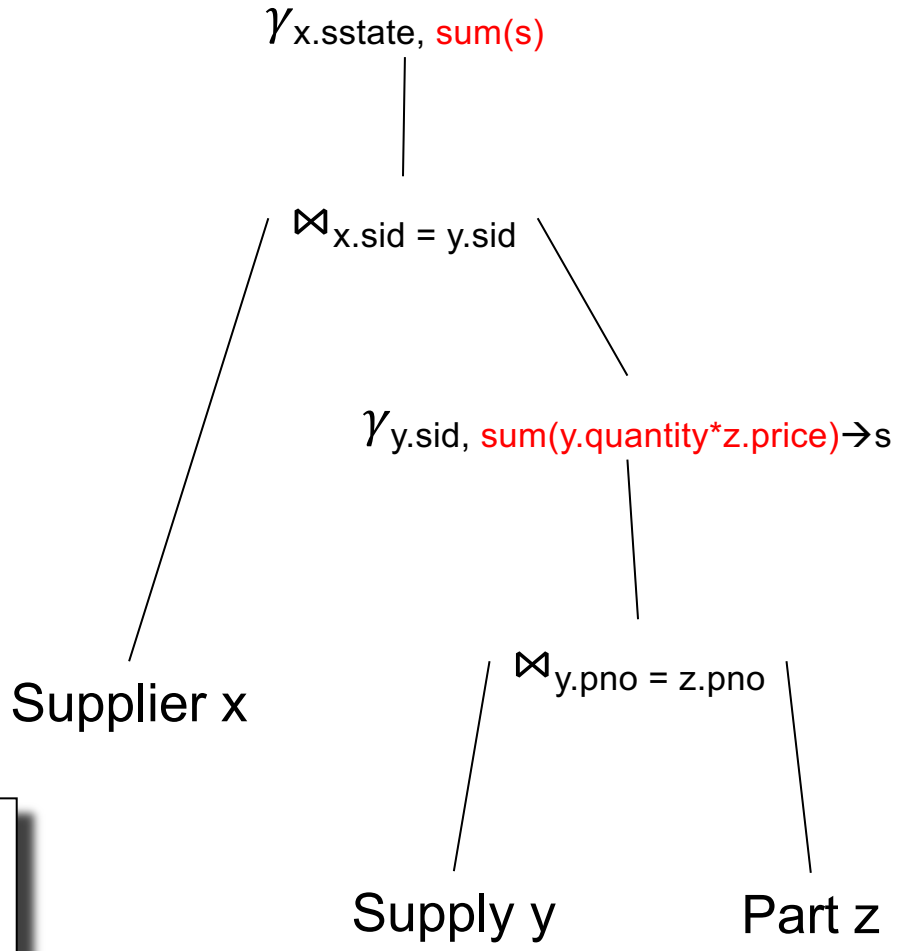



```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Aggregate Push-down

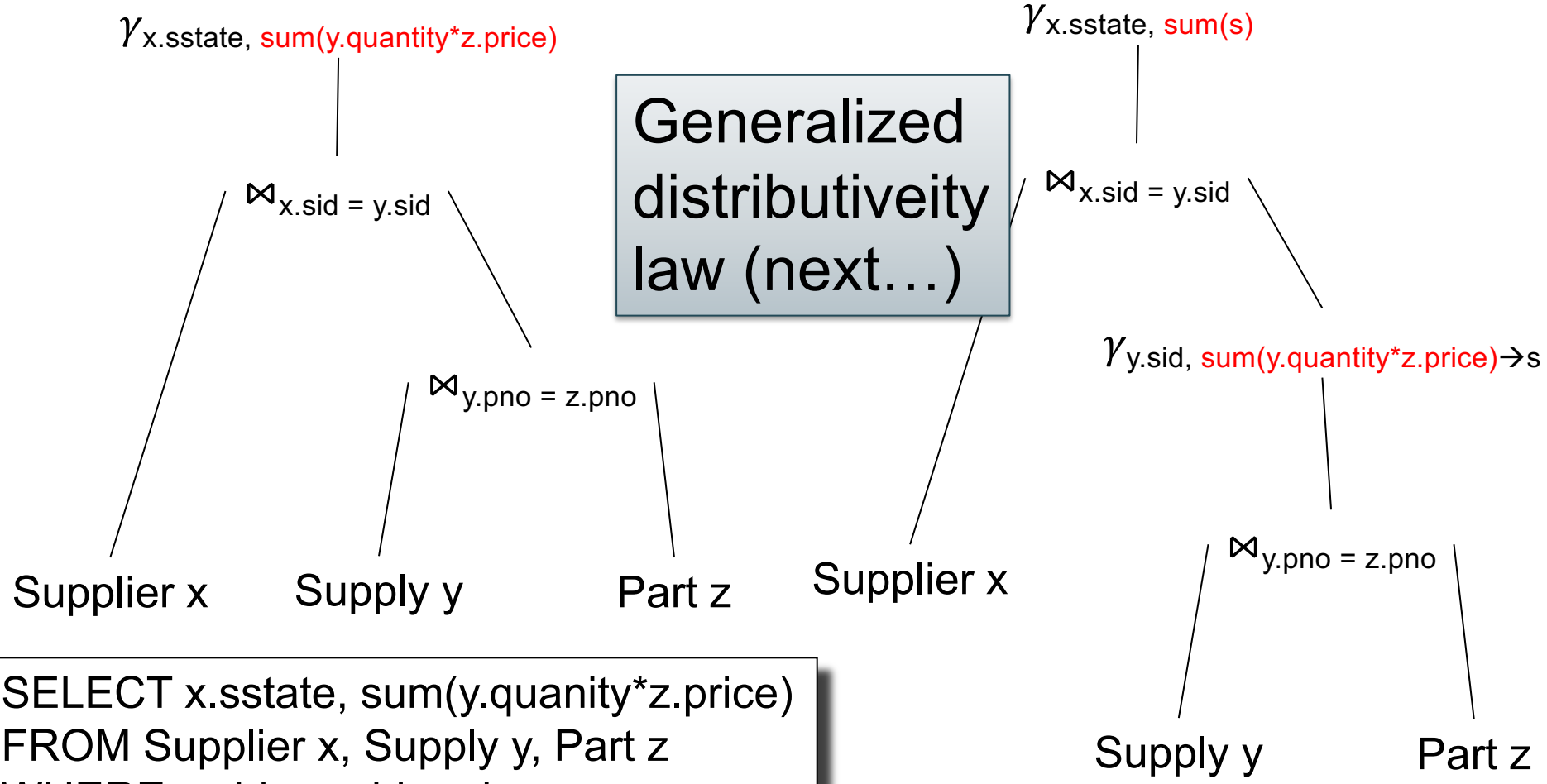





```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

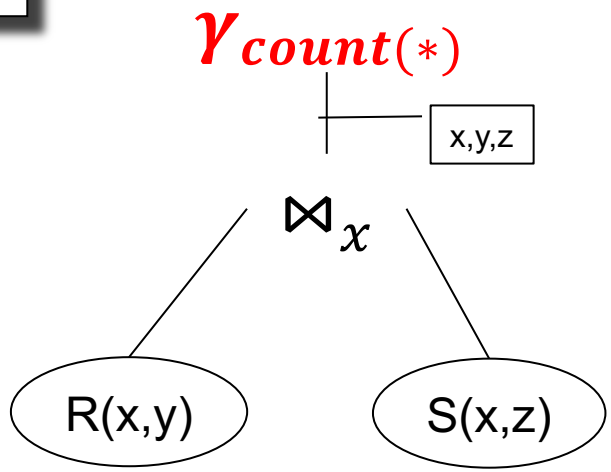
Aggregate Push-down




```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Generalized Distributivity Law

SELECT count(*) from R, S where R.x=S.x



Generalized Distributivity Law

SELECT count(*) from R, S where R.x=S.x

R:

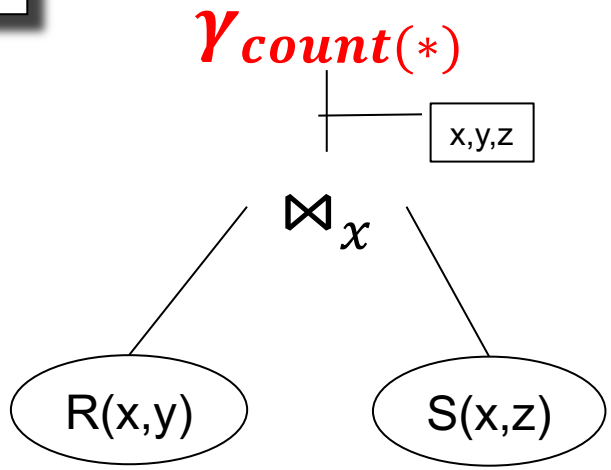
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



Generalized Distributivity Law

SELECT count(*) from R, S where R.x=S.x

R:

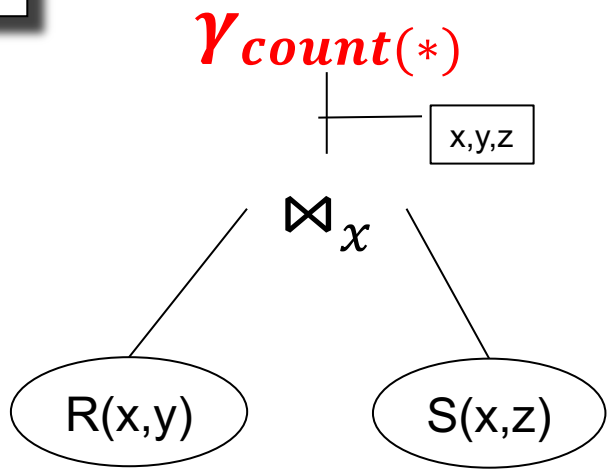
x	y
b	a
b	c
f	d
h	g

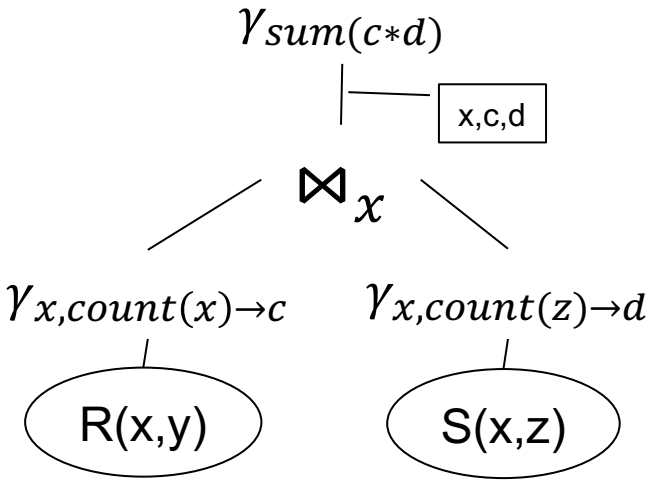
S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$





Generalized Distributivity Law

SELECT count(*) from R, S where R.x=S.x

R:

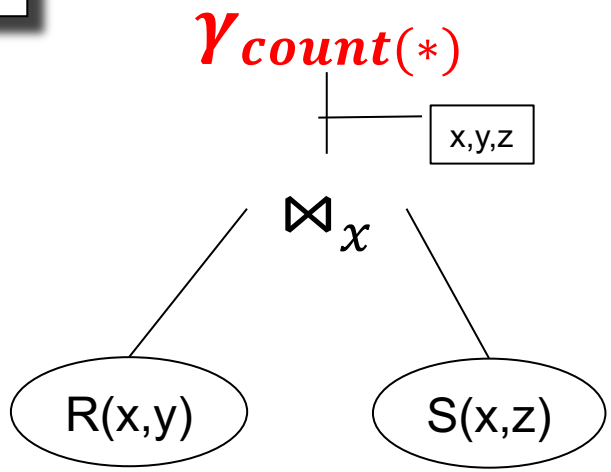
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A:

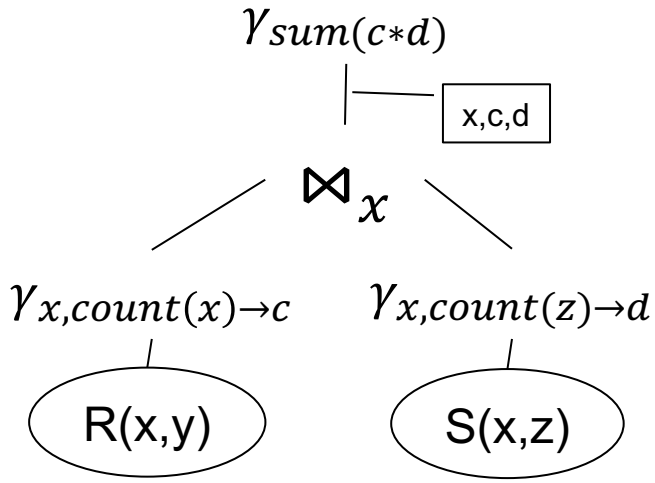
x	c
b	2
f	1
h	1

B:

x	d
b	2
h	1

A⋈B

x	c	d
b	2	2
h	1	1



Generalized Distributivity Law

SELECT count(*) from R, S where R.x=S.x

R:

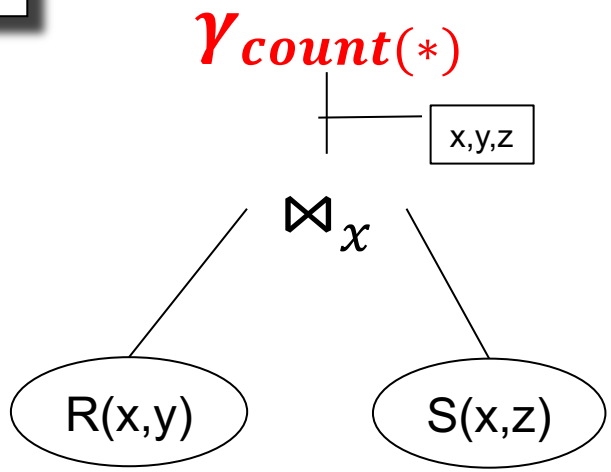
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



Runtime = O(N)

A:

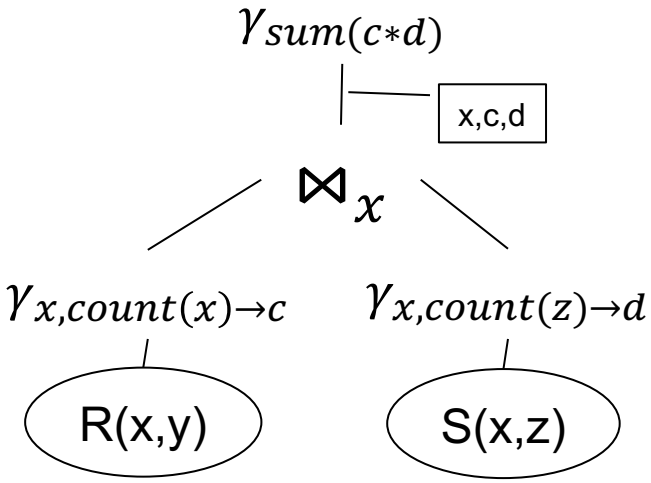
x	c
b	2
f	1
h	1

B:

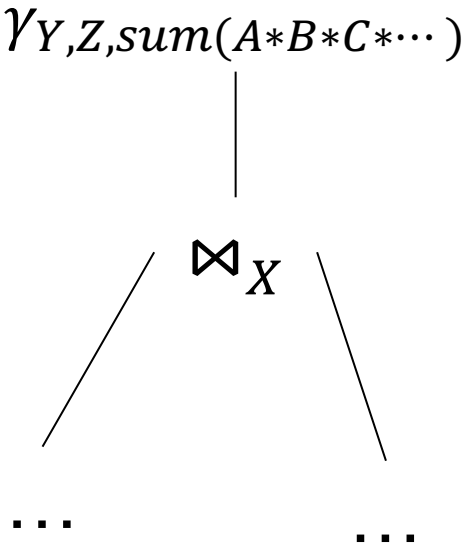
x	d
b	2
h	1

A⋈B

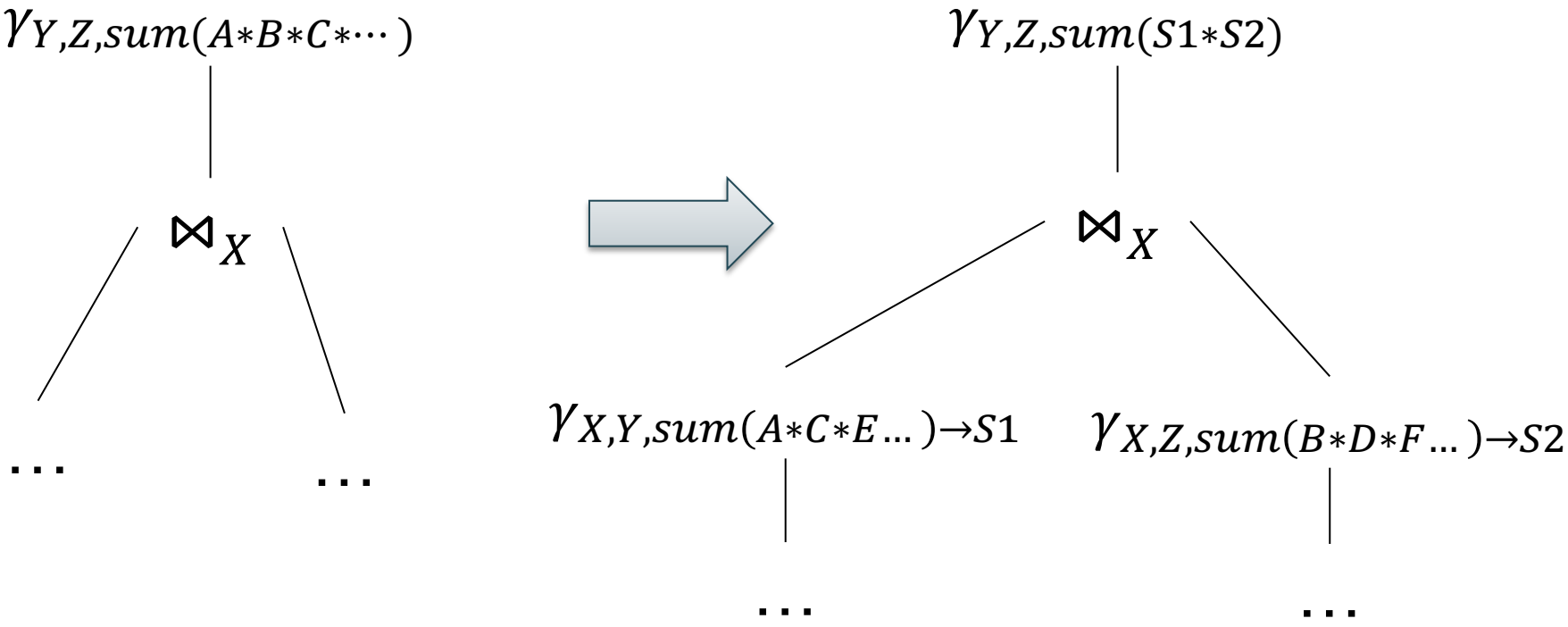
x	c	d
b	2	2
h	1	1



Generalized Distributivity Law



Generalized Distributivity Law



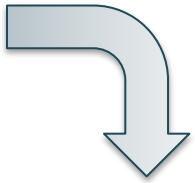
Discussion

- Most DBMS support only some subset of the generalized distributivity law
- Fun activity: check if your favorite DBMS supports it
- Next: a rewrite rule that most DBMS support

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

Key / Foreign-Key

```
Select x.pno, x.quantity
From Supply x, Supplier y
Where x.sid = y.sid
```

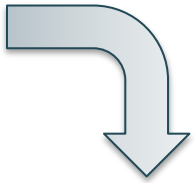


?

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

Key / Foreign-Key

```
Select x.pno, x.quantity
From Supply x, Supplier y
Where x.sid = y.sid
```

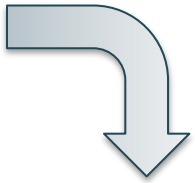


```
Select x.pno, x.quantity
From Supply x
```

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

Key / Foreign-Key

```
Select x.pno, x.quantity
From Supply x, Supplier y
Where x.sid = y.sid
```



```
Select x.pno, x.quantity
From Supply x
```

Only if these constraints hold:

1. Supplier.sid = key
2. Supply.sid = foreign key
3. Supply.sid NOT NULL

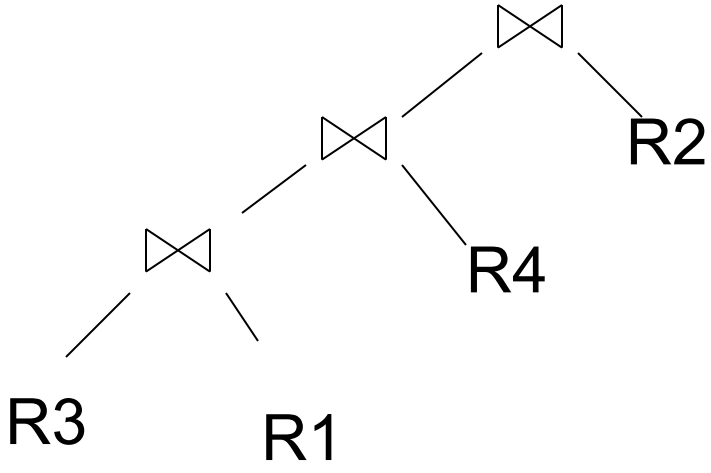
Practice

- Database optimizers typically have a database of rewrite rules
- E.g. SQL Server: about 500 rules
- Rules become complex as they need to serve specialized types of queries

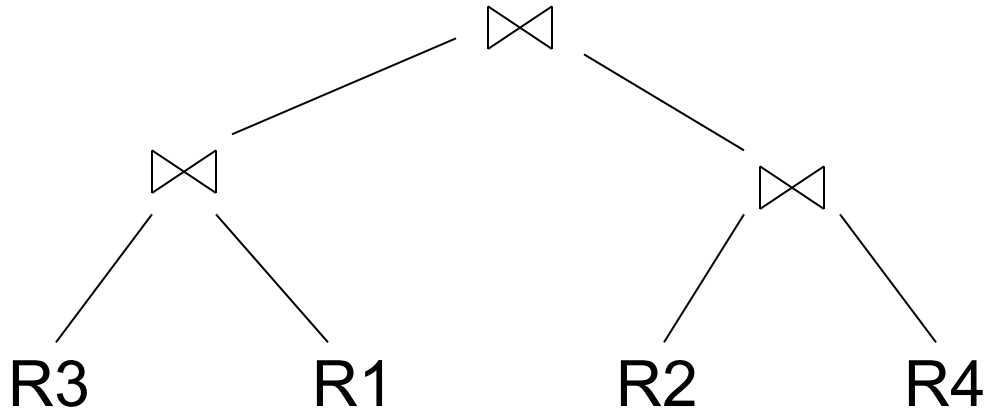
Search Space Challenges

- Search space is huge
- Typical compromises:
  - Left-deep plans
  - Plans without cartesian products

Left-Deep Plans and Bushy Plans



Left-deep plan



Bushy plan

[How good are they]

The need for a rich search space

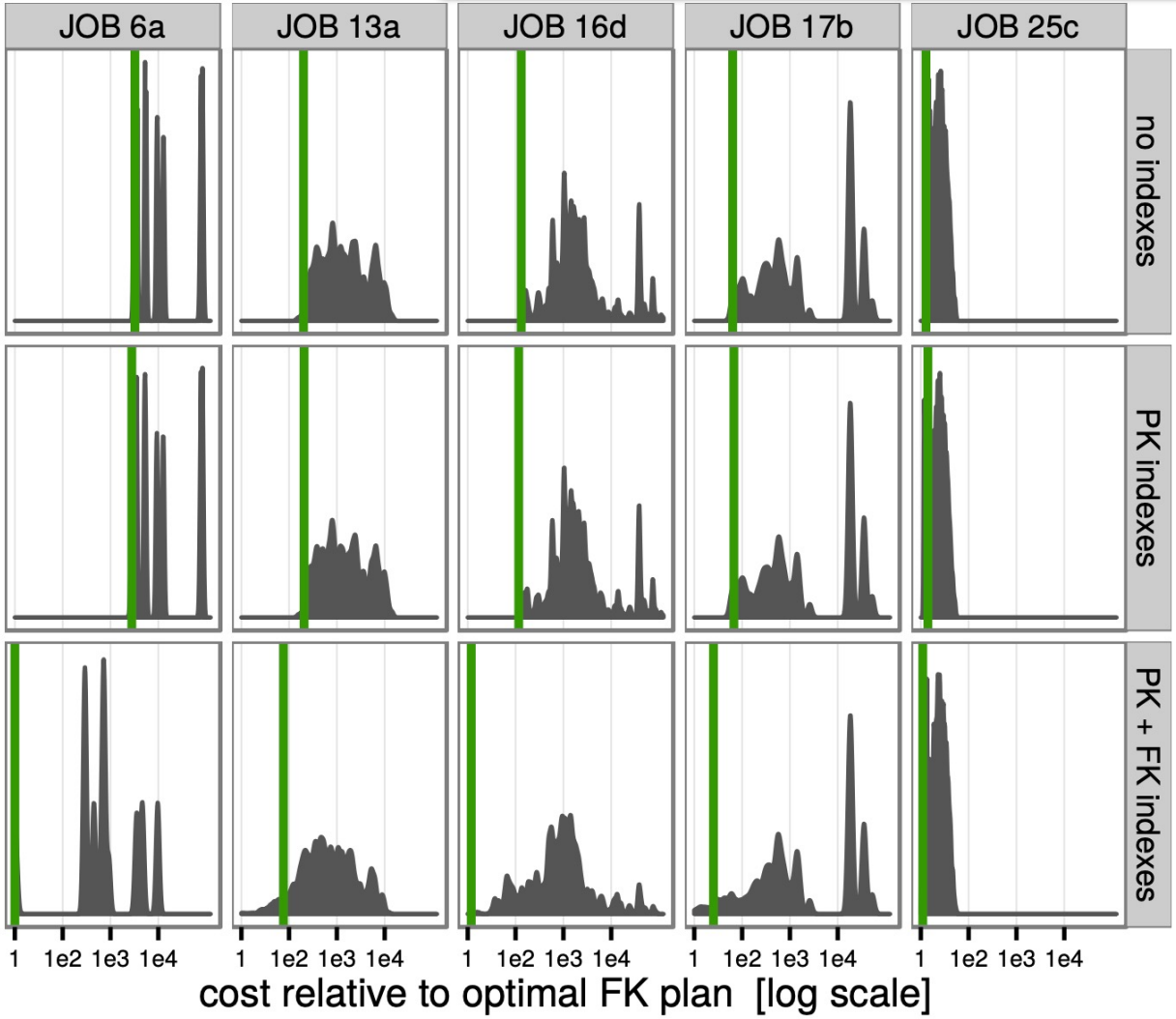


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

[How good are they]

The effect of restricting the search space

	PK indexes			PK + FK indexes		
	median	95%	max	median	95%	max
zig-zag	1.00	1.06	1.33	1.00	1.60	2.54
left-deep	1.00	1.14	1.63	1.06	2.49	4.50
right-deep	1.87	4.97	6.80	47.2	30931	738349

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

Query Optimization

Three major components:

1. Cardinality and cost estimation
2. Search space
3. Plan enumeration algorithms

Two Types of Optimizers

- Heuristic-based optimizers
- Cost-based optimizers (next)

Two Types of Plan Enumeration Algorithms

- Dynamic programming (in class)
  - Based on System R [Selinger 1979]
  - *Join reordering algorithm*

- Rule-based algorithm (will not discuss)
  - Database of rules (=algebraic laws)
  - Usually: dynamic programming

System R Optimizer

For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$, compute best plan:

- Step 1: $Q = \{R_1\}, \{R_2\}, \ldots, \{R_n\}$
- Step 2: $Q = \{R_1,R_2\}, \{R_1,R_3\}, \ldots, \{R_{n-1}, R_n\}$
- …
- Step n: $Q = \{R_1, \ldots, R_n\}$

Details

For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ store:

- Estimated Size(Q)
- A best plan for Q: Plan(Q)
- The cost of that plan: Cost(Q)

One plan for each "interesting order"

Details

Step 1: single relations $\{R_1\}$, $\{R_2\}$, …, $\{R_n\}$

- Consider all possible access paths:
  - Sequential scan, or
  - Index 1, or
  - Index 2, or
  - …
- Keep optimal plan for each "interesting order"

Details

Step k = 2…n:

For each $Q = \{R_{i_1}, \ldots, R_{i_k}\}$

- For each j=1,…,k:
  - Let: $Q' = Q - \{R_{i_j}\}$
  - Let: $Plan(Q') \bowtie R_{i_j}$ $Cost(Q') + CostOf(\bowtie)$
- $Plan(Q), Cost(Q) =$ cheapest of the above
  - Keep separate optimal for “interesting orders”

[How good are they]

Is Dynamic Programming needed?

	PK indexes						PK + FK indexes					
	PostgreSQL estimates			true cardinalities			PostgreSQL estimates			true cardinalities		
	median	95%	max	median	95%	max	median	95%	max	median	95%	max
Dynamic Programming	1.03	1.85	4.79	1.00	1.00	1.00	1.66	169	186367	1.00	1.00	1.00
Quickpick-1000	1.05	2.19	7.29	1.00	1.07	1.14	2.52	365	186367	1.02	4.72	32.3
Greedy Operator Ordering	1.19	2.29	2.36	1.19	1.64	1.97	2.35	169	186367	1.20	5.77	21.0

Table 3: Comparison of exhaustive dynamic programming with the Quickpick-1000 (best of 1000 random plans) and the Greedy Operator Ordering heuristics. All costs are normalized by the optimal plan of that index configuration

Discussion

- All database systems implement Selinger's algorithm for join reorder
- For other operators (group-by, aggregates, difference): rule-based
- Many search strategies beyond dynamic programming

Final Discussion

- Query optimizer realizes the *physical data independence* principle:
  - SQL = what we want to get
  - Optimized plan = how to get it

- Optimizer =
  Search space + Size est + Algorithm
- Typically, good at avoiding “worst plans”, less so at finding “best plan”