DATA516/CSED516
Scalable Data Systems and Algorithms

Lecture 3
Query Optimization, Spark

Administrivia

- Email if there might have a runaway cluster/instance
  - Even if you haven’t received an email, it is worth checking (pause clusters + stop labs)
- Don’t fear Late Day Tokens
- Project Sign Ups

Announcements

- HW2 is posted *(pull upstream)*
  and due on Oct. 31st
- Project proposals due on Oct. 28th
- Review was due today (*How good…?*)
  Review of three papers due next week
- Jack's OH: Thursday 10/27 => Monday 10/24

Outline for Today

- Query Optimization
  - *How good are they?*
- Spark

Recap

- Optimizer has three components:
  - Search space
  - Cardinality and cost estimation
  - Plan enumeration algorithms

Recap

- Optimizer has three components:
  - Search space
  - Cardinality and cost estimation
  - Plan enumeration algorithms
- Paper addresses three questions:
  - How good are the cardinality estimators?
  - How important is the cost model?
  - How large does the search space need to be?

Paper Outline

- How good are the cardinality estimators?
- How important is the cost model?
- How large does the search space need to be?

The Job Benchmark

- Why do they use the IMDB database instead of TPC-H?
- IMDB – popular data on the web, can be imported into any RDBMS with moderate effort

Lesson: you can always import your dataset into RDBMS!

The Job Benchmark

JOB Benchmark: 33 templates, 113 queries

Discuss the difference in class:

- SQL query
- SQL query template (or structure)

Group-by Queries

- None in JOB!
- Important in DS; we'll discuss them later

Review: Cardinality Estimation

Problem: given statistics on base tables and a query, estimate size of the answer

What are the statistics on base tables?

Review: Cardinality Estimation

Problem: given statistics on base tables and a query, estimate size of the answer

What are the statistics on base tables?

- Number of tuples (cardinality) T(R)
- Number of values in R.a: V(R,a)
- Histograms (later today)

Review: Cardinality Estimation

What are the four assumptions that database systems do?

Review: Cardinality Estimation

What are the four assumptions that database systems do?

- Uniformity
- Independence
- Containment of values
- Preservation of values

Single Table Estimation

$\boxed{\sigma_{A=c}(R) = T(R)/V(R,A)}$

What assumption does this make?

Single Table Estimation

$\boxed{\sigma_{A=c}(R) = T(R)/V(R,A)}$

What assumption
does this make?

Uniformity

Single Table Estimation

$\sigma_{A=c}(R) = T(R)/V(R,A)$

What assumption does this make?

Uniformity

	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

Histograms

- T(R), V(R,A) too coarse
- Histogram: separate stats per bucket

- In each bucket store:
  - T(bucket)
  - V(bucket,A)

Employee(ssn, name, age)

Histograms

T(Employee) = 25000, V(Empolyee, age) = 50

Estimate $\sigma_{age=48}$(Empolyee) = ?

Employee(ssn, name, age)

Histograms

T(Employee) = 25000, V(Empolyee, age) = 50

Estimate $\sigma_{age=48}$(Employee) = ?   = 25000/50 = 500

Employee(ssn, name, age)

Histograms

T(Employee) = 25000, V(Empolyee, age) = 50

Estimate $\sigma_{age=48}$(Employee) = ? = 25000/50 = 500

Age:	0..20	20..29	30-39	40-49	50-59	> 60
T =	200	800	5000	12000	6500	500
V =	3	10	7	6	5	4

Estimate $\sigma_{age=48}$(Employee) = ?

Employee(ssn, name, age)

Histograms

T(Employee) = 25000, V(Empolyee, age) = 50

Estimate $\sigma_{age=48}$(Employee) = ? = 25000/50 = 500

Age:	0..20	20..29	30-39	40-49	50-59	> 60
T =	200	800	5000	**12000**	6500	500
V =	3	10	7	**6**	5	4

Estimate $\sigma_{age=48}$(Employee) = ? = 12000/6 = 2000

Types of Histograms

- Eq-Width
- Eq-Depth
- Compressed: store outliers separately
- “Special”: V-Optimal histograms

Employee(ssn, name, age)

Histograms

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
T	200	800	5000	12000	6500	500
V	2	8	10	10	8	3

Employee(ssn, name, age)

Histograms

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
T	200	800	5000	12000	6500	500
V	2	8	10	10	8	3

Eq-depth:

Age:	0..32	33..41	42-46	47-52	53-58	> 60
T	1800	2000	2100	2200	1900	1800
V	8	10	9	10	8	6

Employee(ssn, name, age)

Histograms

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
T	200	800	5000	12000	6500	500
V	2	8	10	10	8	3

Eq-depth:

Age:	0..32	33..41	42-46	47-52	53-58	> 60
T	1800	2000	2100	2200	1900	1800
V	8	10	9	10	8	6

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

V-Optimal Histograms

"Weighed Variance of the source values is minimized"

-Improved Histograms for Selectivity Estimation of Range Predicates

- Pick boundaries that minimize the variance of frequencies within buckets

- Dynamic programming
- Modern databases systems use V-optimal histograms or some variations

Multiple Predicates

- Independence assumption:
  - Simple
  - But often leads to major underestimates

- Modeling correlations:
  - Solution 1: 2d Histograms
  - Solution 2: use sample from the data

Modeling Correlations

1. Multi-dimensional histograms
   – Also called column-group statistics

2. Sample from the data

Supplier(<u>sid</u>, sname, scity, sstate)

2d-Histogram

T(Supplier) = 250,000

1d Histograms

scity:	A..E	F..I	J..M	N..Q	R..U	V..Z
T	2000	8000	50000	120000	65000	5000
V	50	40	250	300	130	100

sstate:	A..J	K..S	T..Z
T	125000	80000	45000
V	20	10	20

Estimate $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Supplier}) = ?$

Supplier(sid, sname, scity, sstate)

2d-Histogram

T(Supplier) = 250,000

1d Histograms

scity:	A..E	F..I	J..M	N..Q	R..U	V..Z
T	2000	8000	50000	120000	65000	5000
V	50	40	250	300	130	100

sstate:	A..J	K..S	T..Z
T	125000	80000	45000
V	20	10	20

Estimate $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Supplier}) = ?$

2d Histogram

Sstate \ scity	A..E	F..I	J..M	N..Q	R..U	V..Z
A..J	…		T,V=…			
K..S						
T..Z						

Supplier(sid, sname, scity, sstate)

2d-Histogram

T(Supplier) = 250,000

1d Histograms

scity:	A..E	F..I	J..M	N..Q	R..U	V..Z
T	2000	8000	50000	120000	65000	5000
V	50	40	250	300	130	100

sstate:	A..J	K..S	T..Z
T	125000	80000	45000
V	20	10	20

Estimate $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Supplier}) = ?$

2d Histogram

Sstate \ scity	A..E	F..I	J..M	N..Q	R..U	V..Z
A..J	…		T,V=…			
K..S						
T..Z						

Answer: T_{bucket} / V_{bucket}

Supplier(sid, sname, scity, sstate)

Sample

- Compute a small, uniform sample from Supplier

Estimate $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Supplier}) = ?$

Supplier(<u>sid</u>, sname, scity, sstate)

Sample

- Compute a small, uniform sample from Supplier
- Use Thomson's estimator:

Estimate $\sigma_{sscity=\text{'Mtv'} \wedge sstate=\text{'CA'}}(\text{Supplier})$ = ?

Supplier(sid, sname, scity, sstate)

Sample

- Compute a small, uniform sample from Supplier

Estimate $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Supplier})$ = ?

- Use Thomson's estimator:

Answer: $\sigma_{\text{sscity='Mtv'} \wedge \text{sstate='CA'}}(\text{Sample})$ * T(Supplier) / T(Sample)

Correlations

- Solution 1: 2d histograms
  - Plus: can be accurate for 2 predicates
  - Minus: unclear how to use for 3 or more preds
  - Minus: too many 2d histogram candidates
- Solution 2: sampling
  - Plus: can be accurate for >2 predicates
  - Plus: work for complex preds, e.g. “like”
  - Minus: fail for low selectivity predicates

Correlations

- Solution 1: 2d histograms
  - Plus: can be accurate for 2 predicates
  - Minus: unclear how to use for 3 or more preds
  - Minus: too many 2d histogram candidates
- Solution 2: sampling
  - Plus: can be accurate for >2 predicates
  - Plus: work for complex preds, e.g. "like"
  - Minus: fail for low selectivity predicates

Correlations

- Solution 1: 2d histograms
  - Plus: can be accurate for 2 predicates
  - Minus: unclear how to use for 3 or more preds
  - Minus: too many 2d histogram candidates
- Solution 2: sampling
  - Plus: can be accurate for >2 predicates
  - Plus: work for complex preds, e.g. “like”
  - Minus: fail for low selectivity predicates

Correlations

- Solution 1: 2d histograms
  - Plus: can be accurate for 2 predicates
  - Minus: unclear how to use for 3 or more preds
  - Minus: too many 2d histogram candidates
- Solution 2: sampling
  - Plus: can be accurate for >2 predicates
  - Plus: work for complex preds, e.g. “like”
  - Minus: fail for low selectivity predicates

Correlations

- Solution 1: 2d histograms
  - Plus: can be accurate for 2 predicates
  - Minus: unclear how to use for 3 or more preds
  - Minus: too many 2d histogram candidates
- Solution 2: sampling
  - Plus: can be accurate for >2 predicates
  - Plus: work for complex preds, e.g. “like”
  - Minus: fail for low selectivity predicates

Discussion

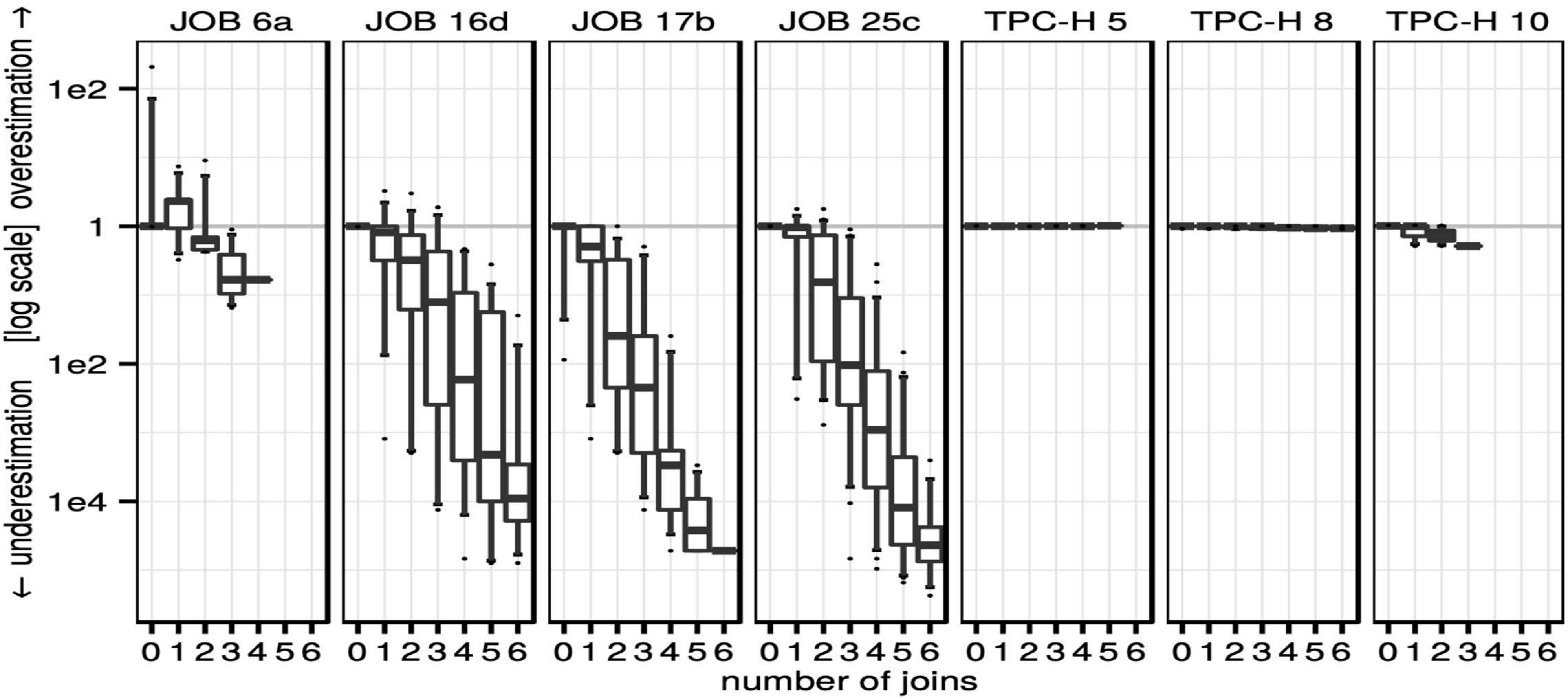
- Paper explains the need for real data

Discussion

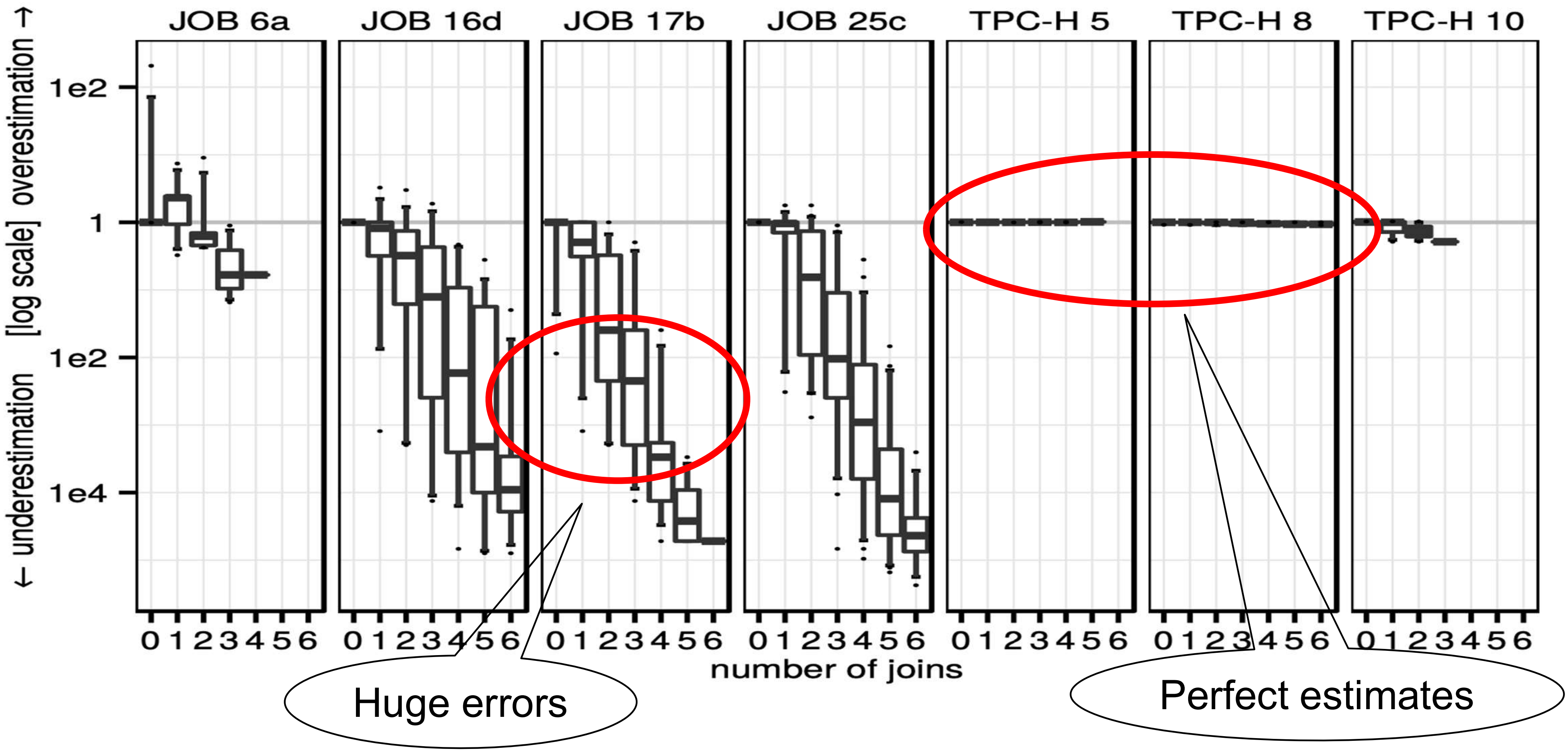
- Paper explains the need for real data
- Synthetic data used in benchmarks is often generated using uniform, independent distributions; formulas for cardinality estimation are perfect

[How good are they]

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



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



Paper Outline

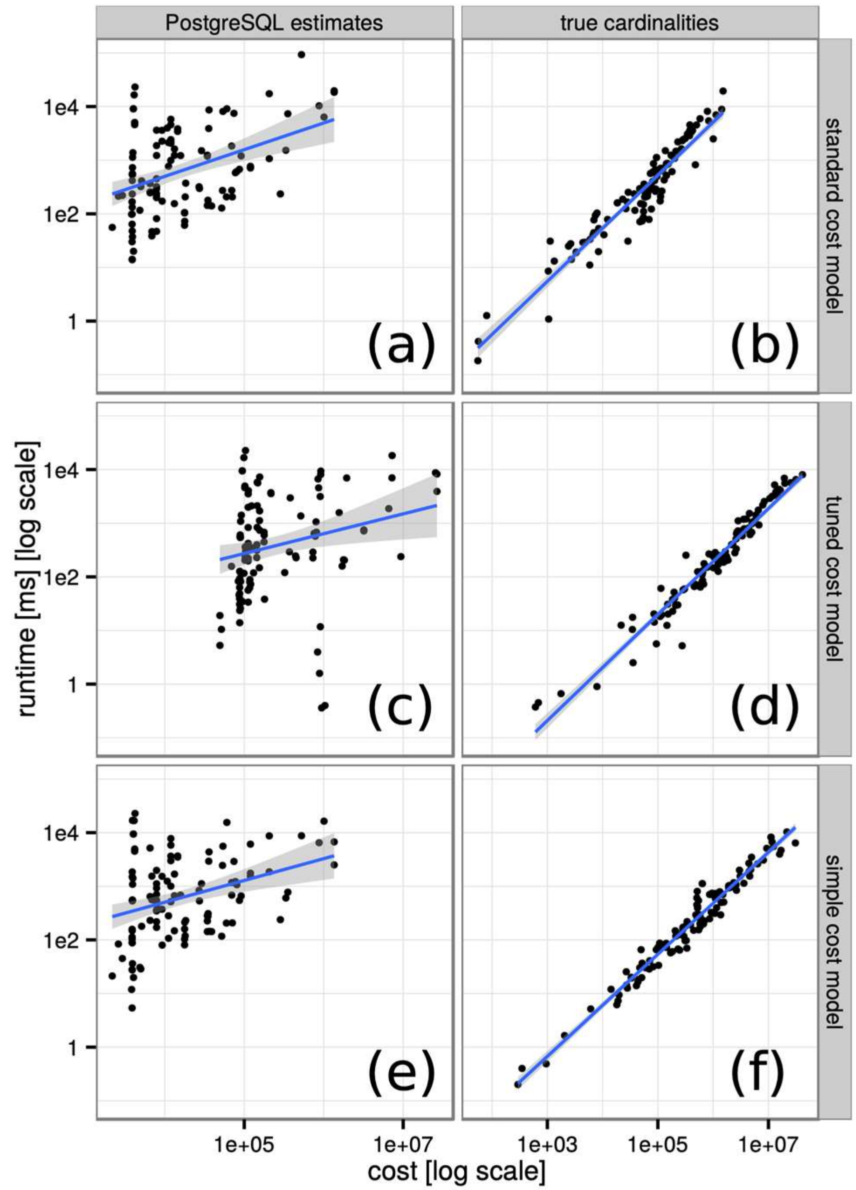
- How good are the cardinality estimators?

- How important is the cost model?

- How large does the search space need to be?

[How good are they]

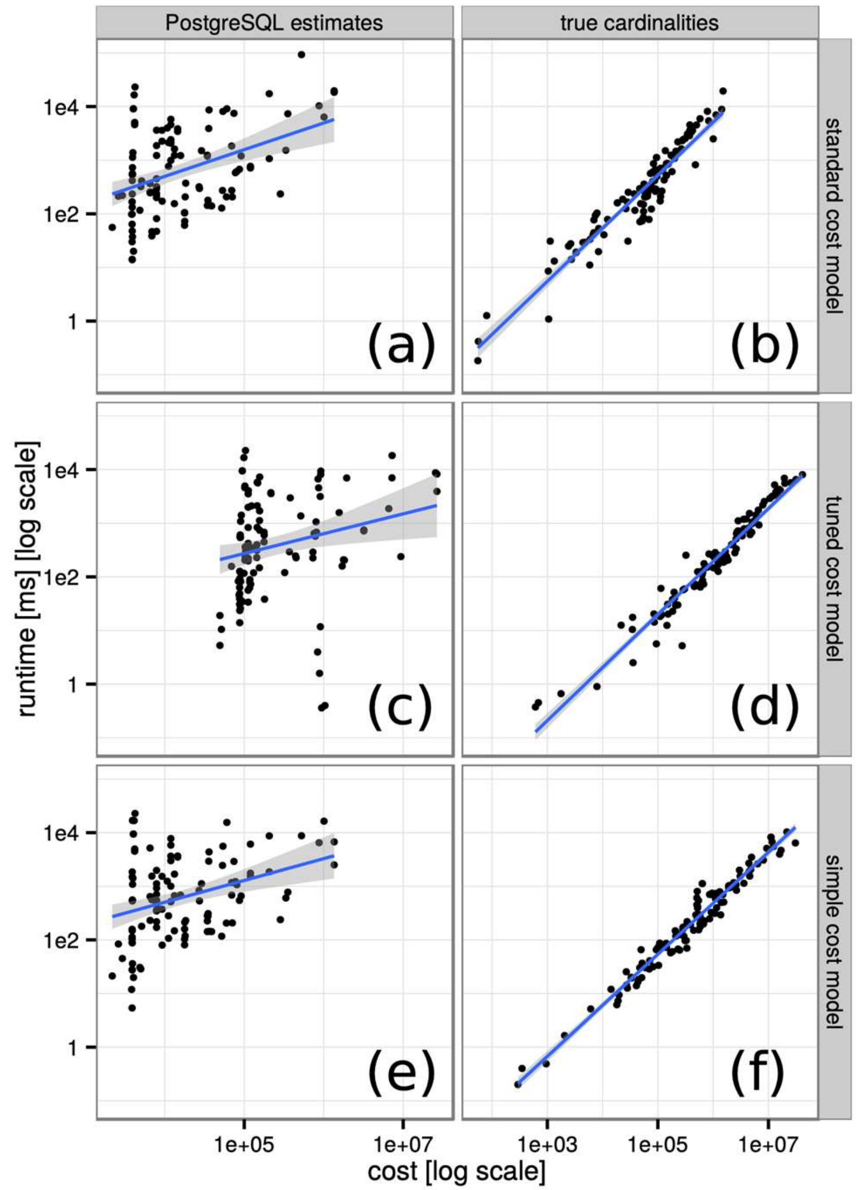
Cardinalities to Cost



[How good are they]

Cardinalities to Cost

Postgres cost

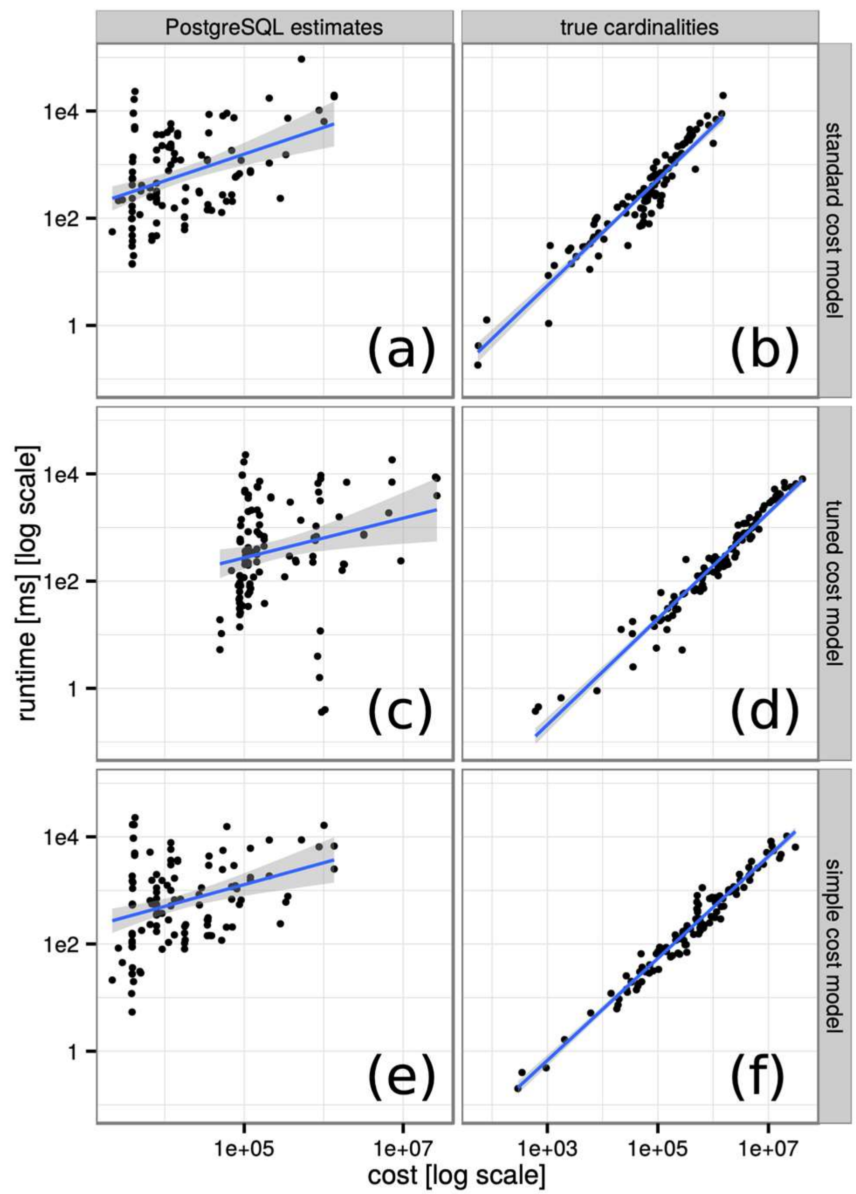


[How good are they]

Cardinalities to Cost

Postgres cost

No I/O, keep only CPU



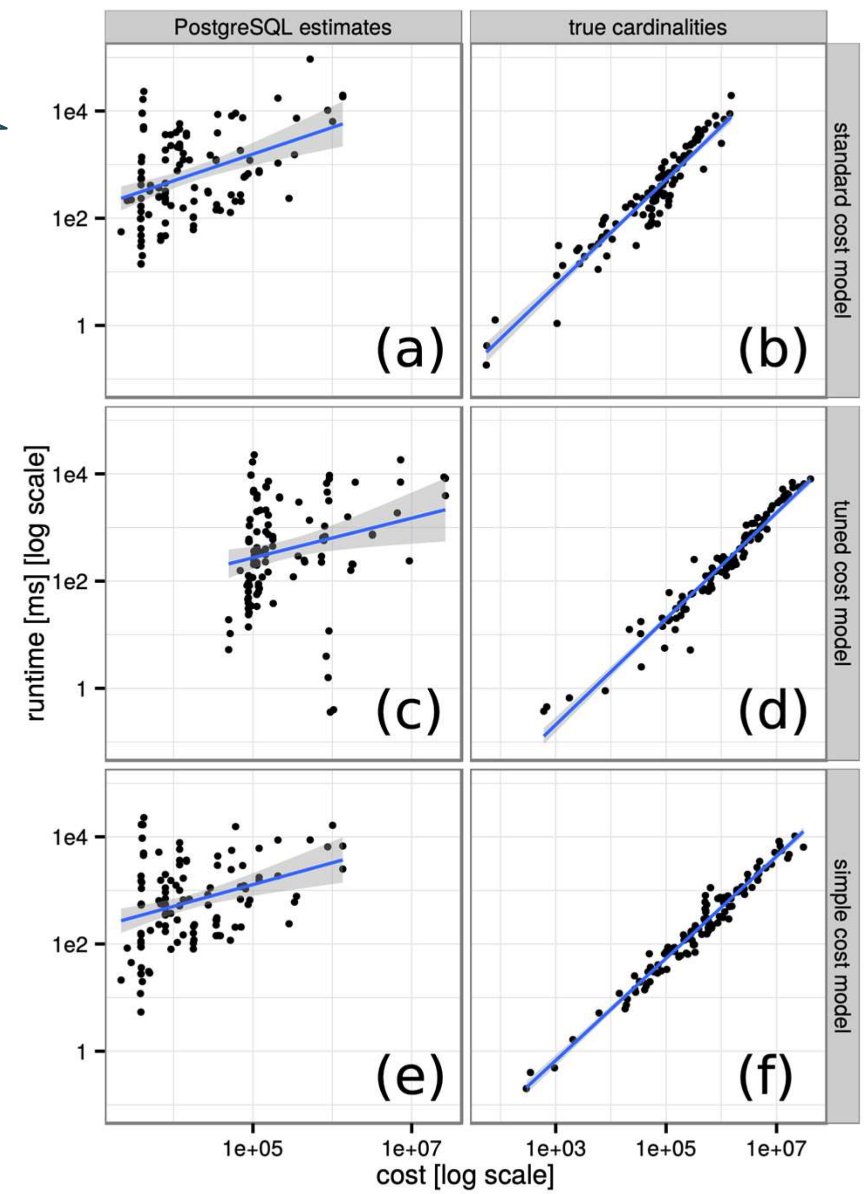
[How good are they]

Cardinalities to Cost

Postgres cost

No I/O, keep only CPU

Their own simple formula



[How good are they]

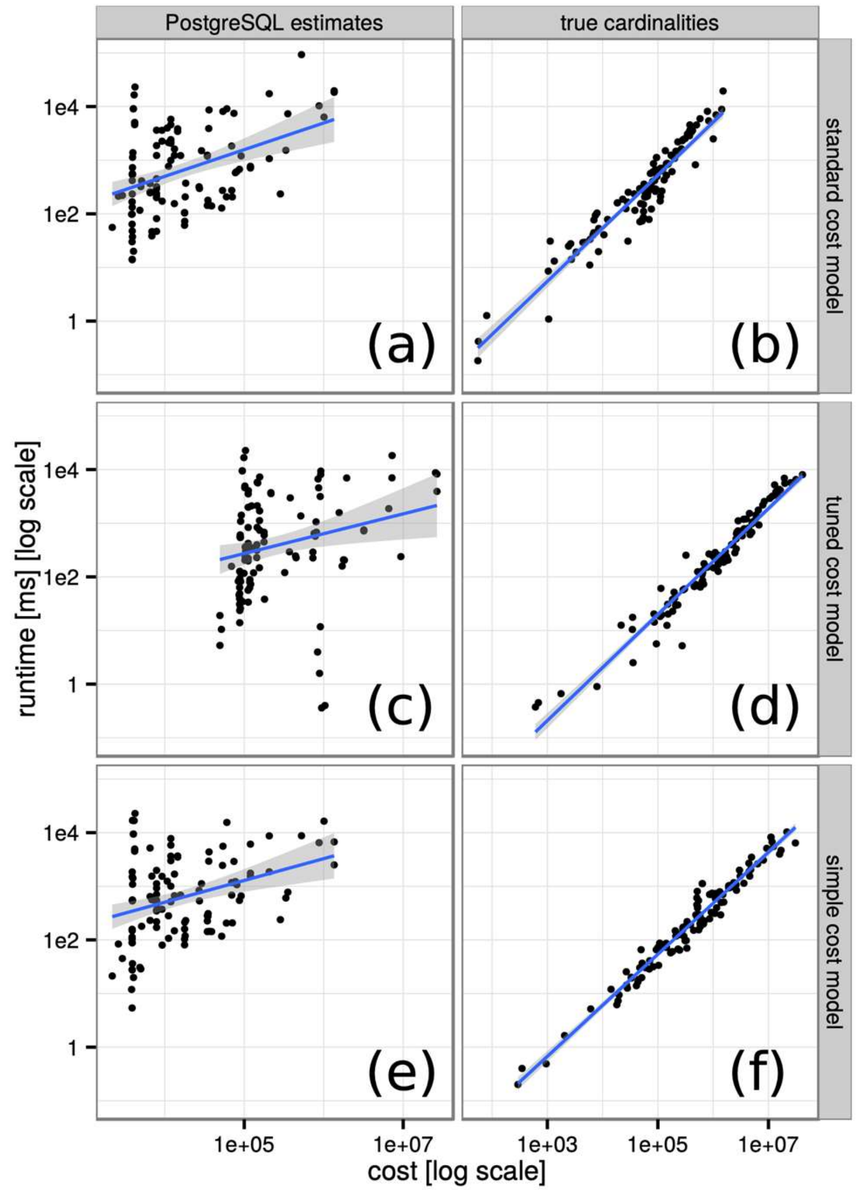
Cardinalities to Cost

- 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



Not in the paper!

Digression: Yet Another Difficulty

SQL Queries issued from applications:

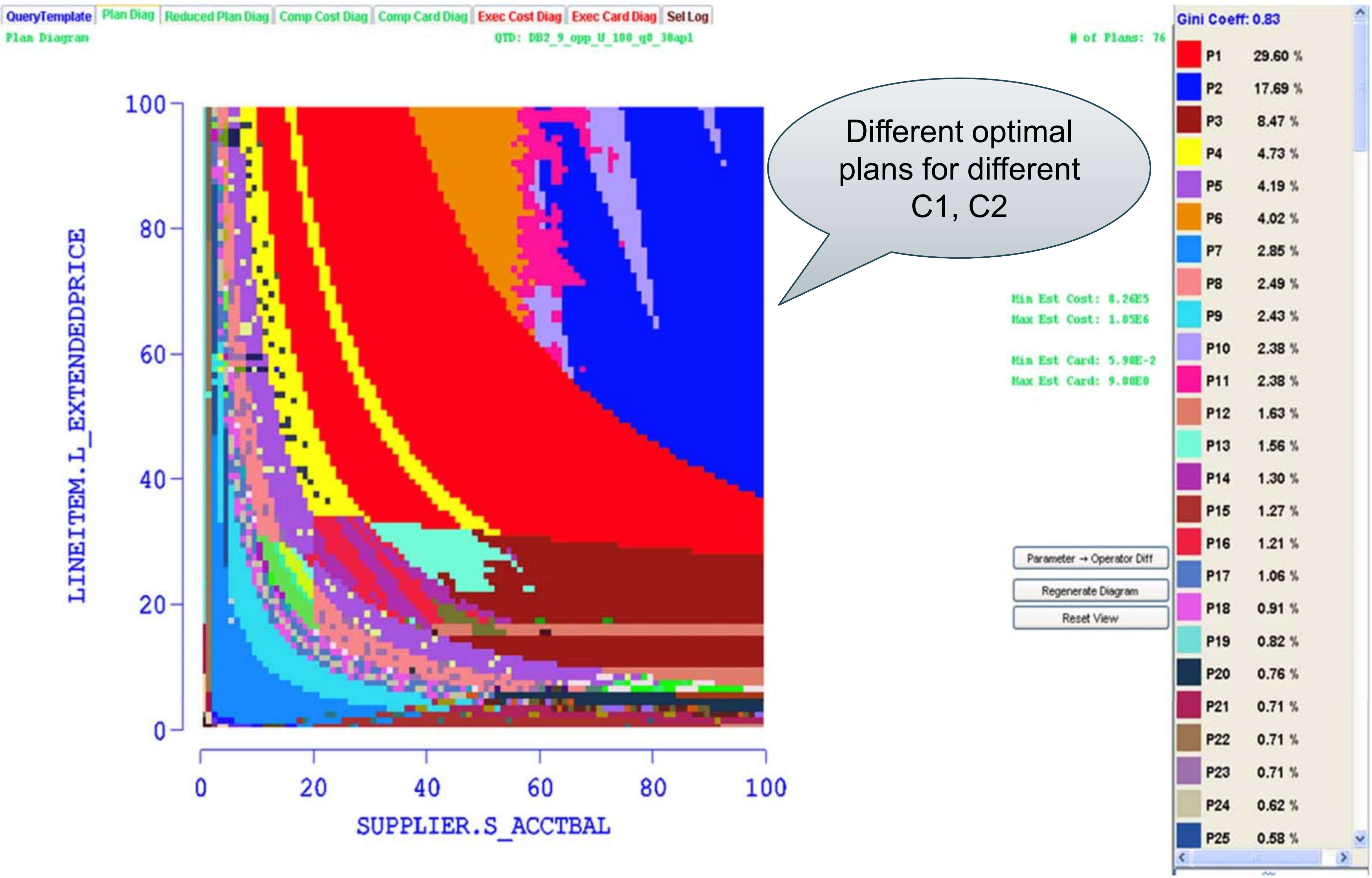
- Query is optimized once: *prepare*
- Then, executed repeatedly

Query constants are unknown 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
```

Optimize without knowing C1, C2



Different optimal plans for different C1, C2

Paper Outline

- How good are the cardinality estimators?
- How important is the cost model?
- How large does the search space need to be?

Search Space

- The set of alternative plans

- Rewrite rules; examples:
  - Push selections down: $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
  - Join reorder: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
  - Push aggregates down (later today)

- Types of join trees (next)

[How good are they]

The need for a rich search space

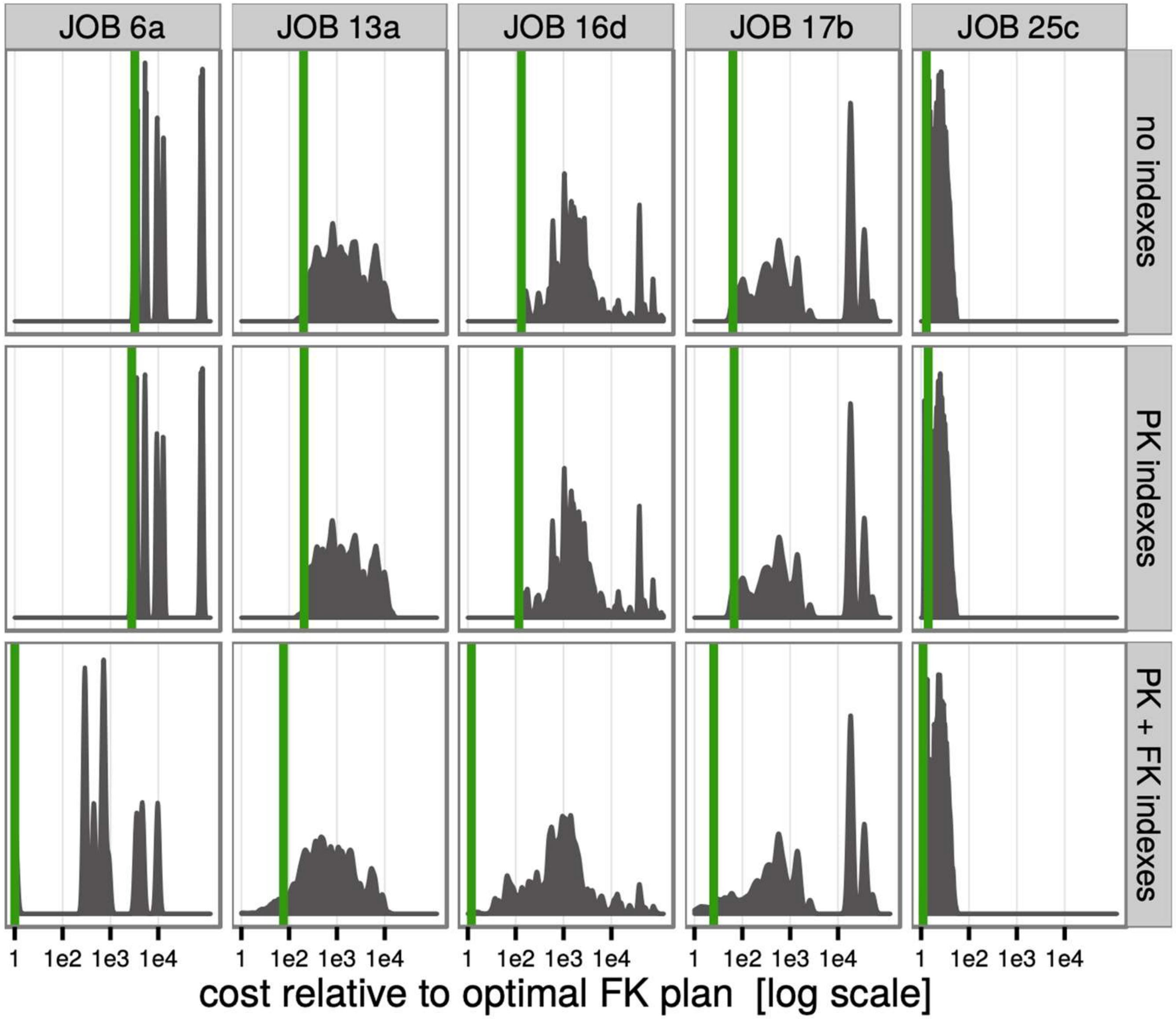


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

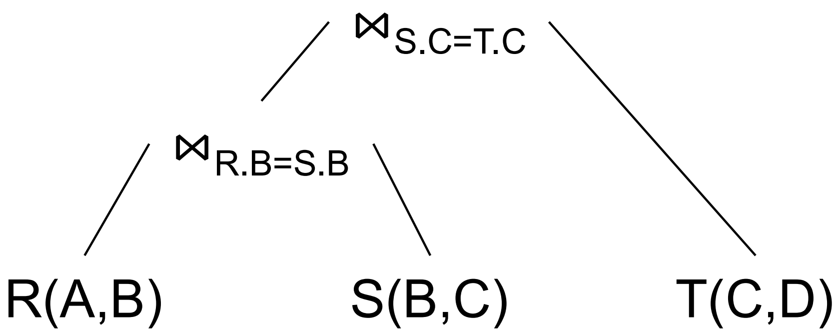
Types of Join Trees

- Based on the join condition:
  – With cartesian products
  – Without cartesian products

- Based on the shape:
  – Left deep
  – Right deep
  – Zig-zag
  – Bushy

Cartesian Product: with or without

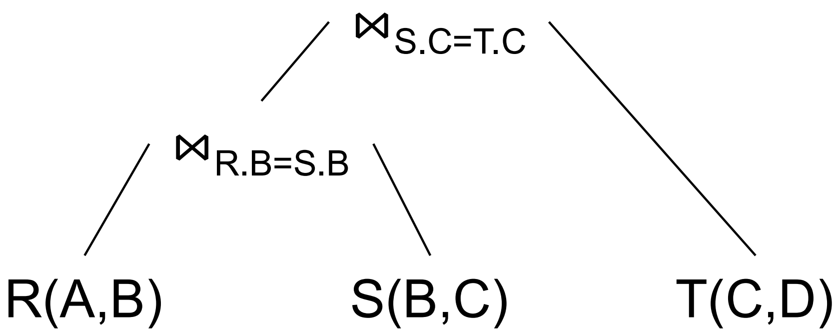
R(A,B) $\bowtie_{R.B=S.B}$ S(B,C) $\bowtie_{S.C=T.C}$ T(C,D)



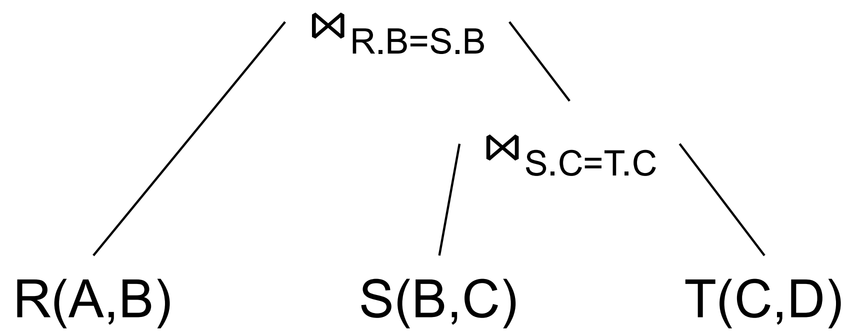
Without
cartesian
product

Cartesian Product: with or without

$R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D)$

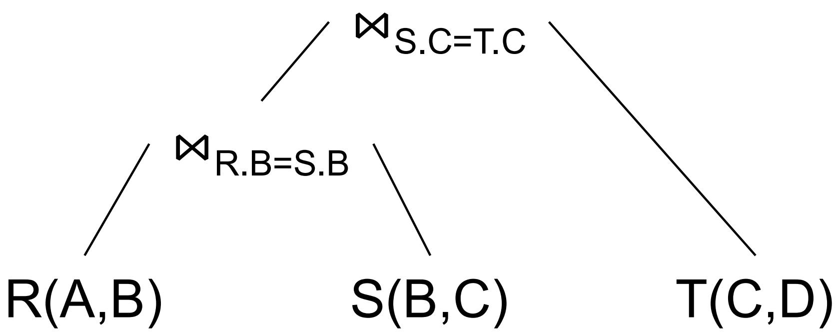


Without
cartesian
product

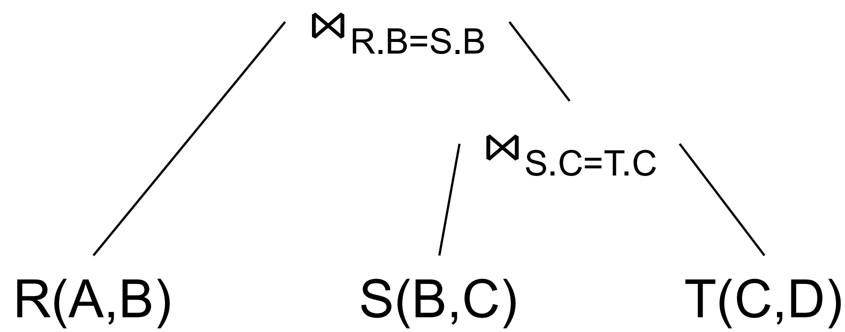


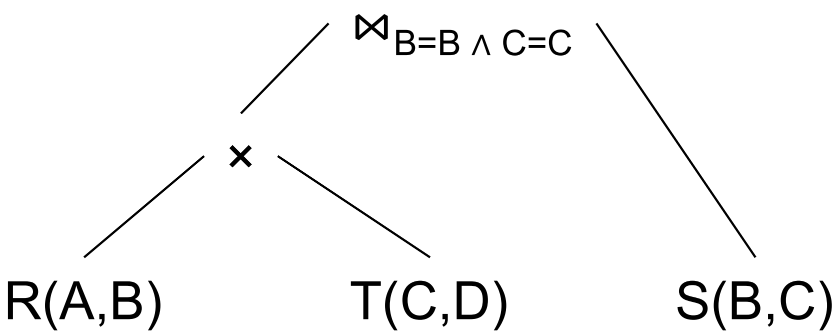
Cartesian Product: with or without

$R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D)$



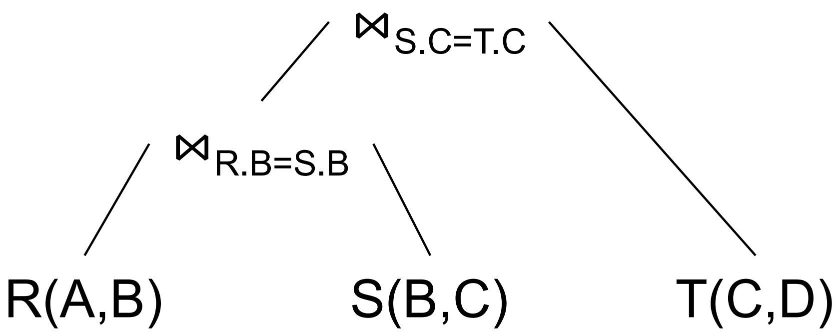
Without cartesian product



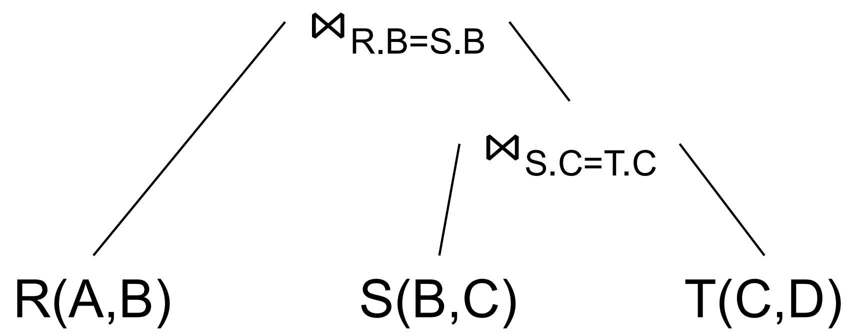


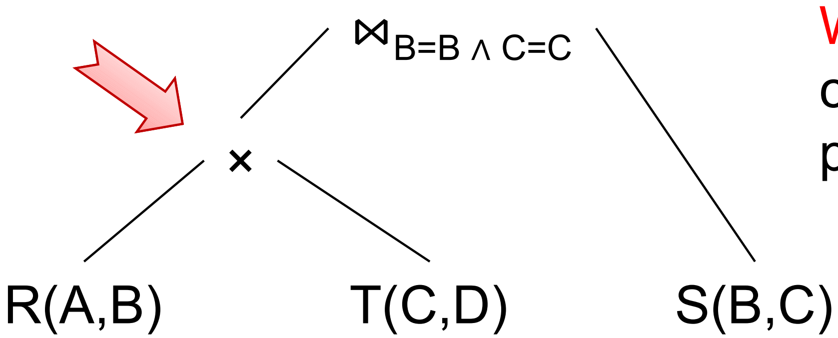
Cartesian Product: with or without

$R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D)$



Without
cartesian
product

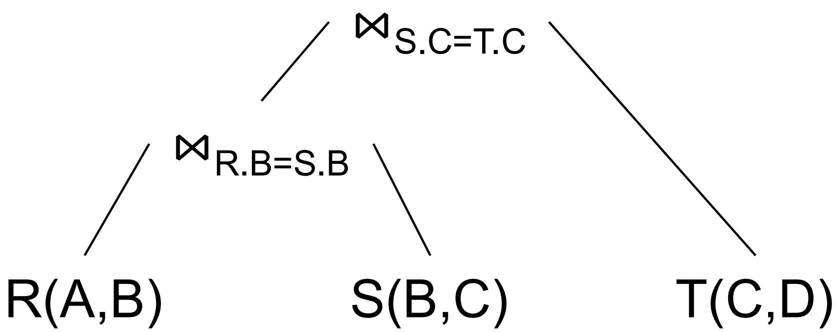




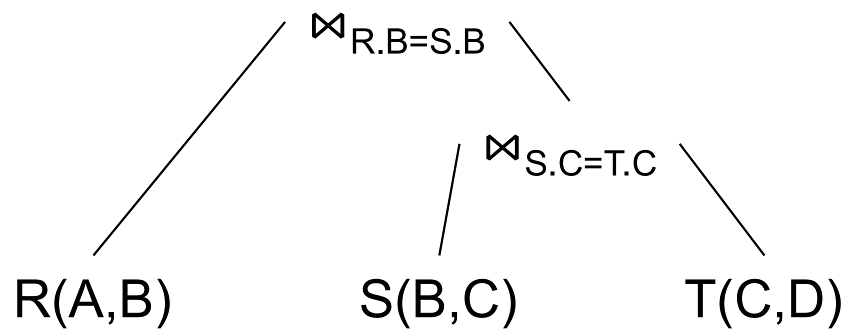
With
cartesian
product

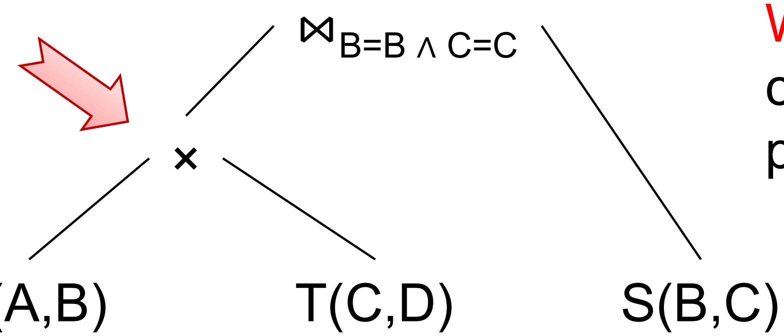
Cartesian Product: with or without

R(A,B) $\bowtie_{R.B=S.B}$ S(B,C) $\bowtie_{S.C=T.C}$ T(C,D)



Without
cartesian
product



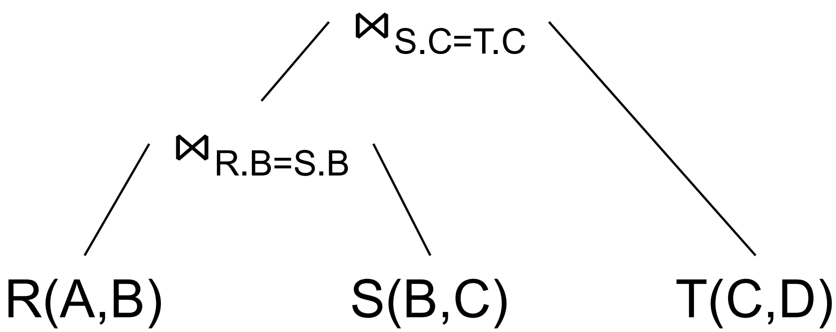


With
cartesian
product

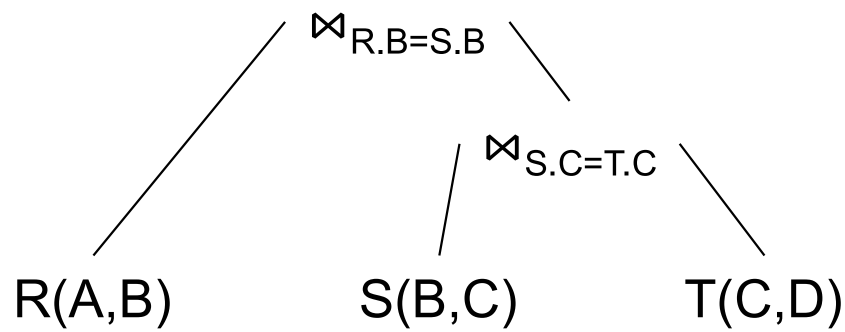
When could this plan be better?

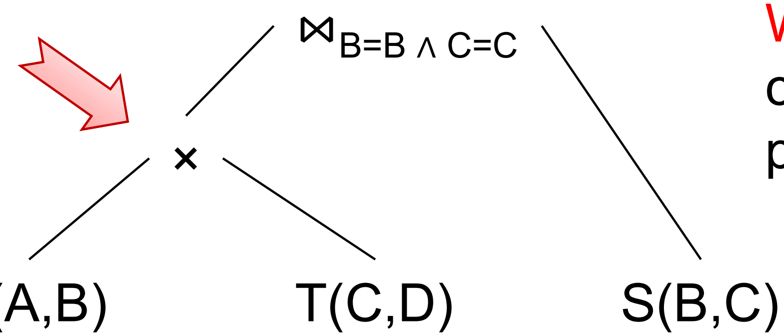
Cartesian Product: with or without

$R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D)$



Without cartesian product



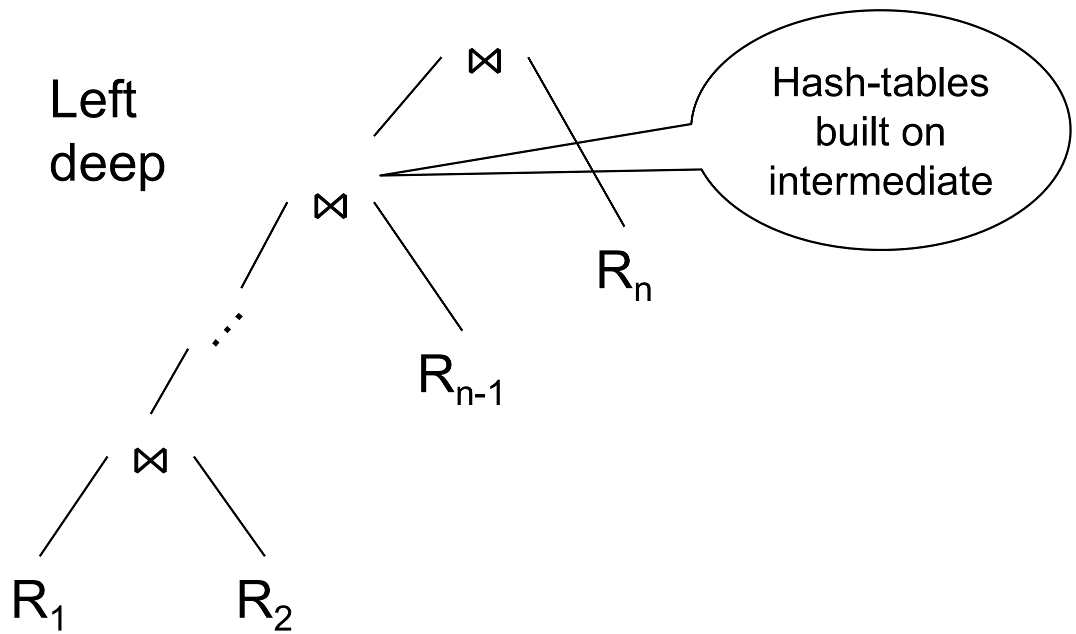


With cartesian product

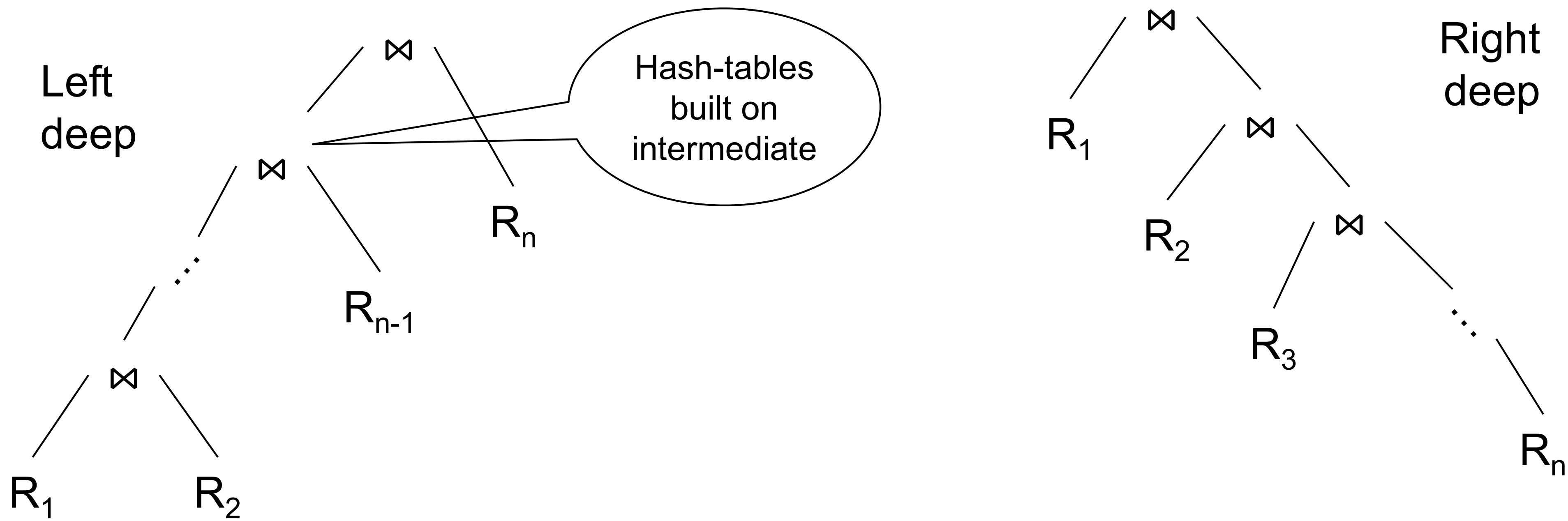
When could this plan be better?

When R, T are very small, and S is very large

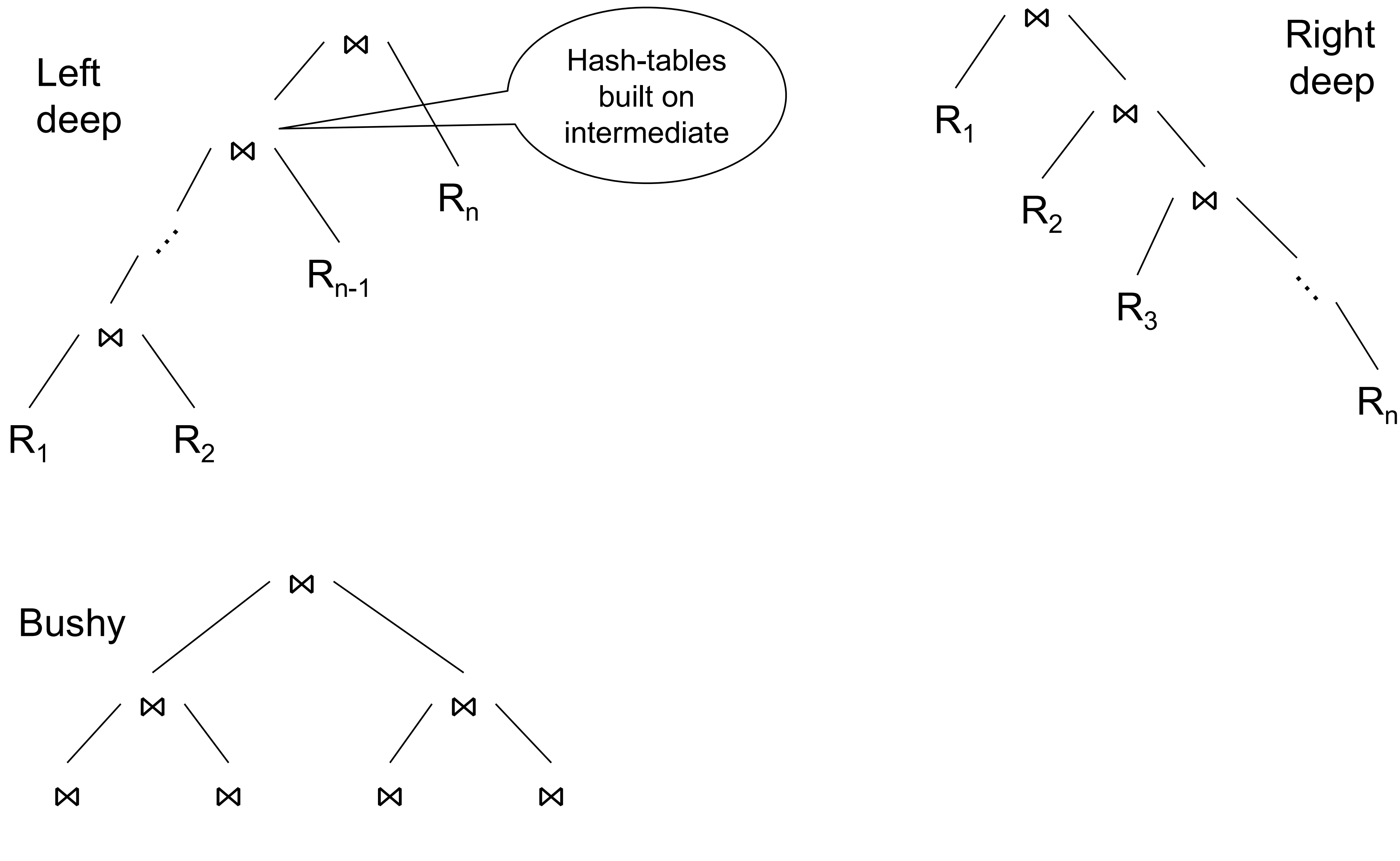
Shapes of Join Trees



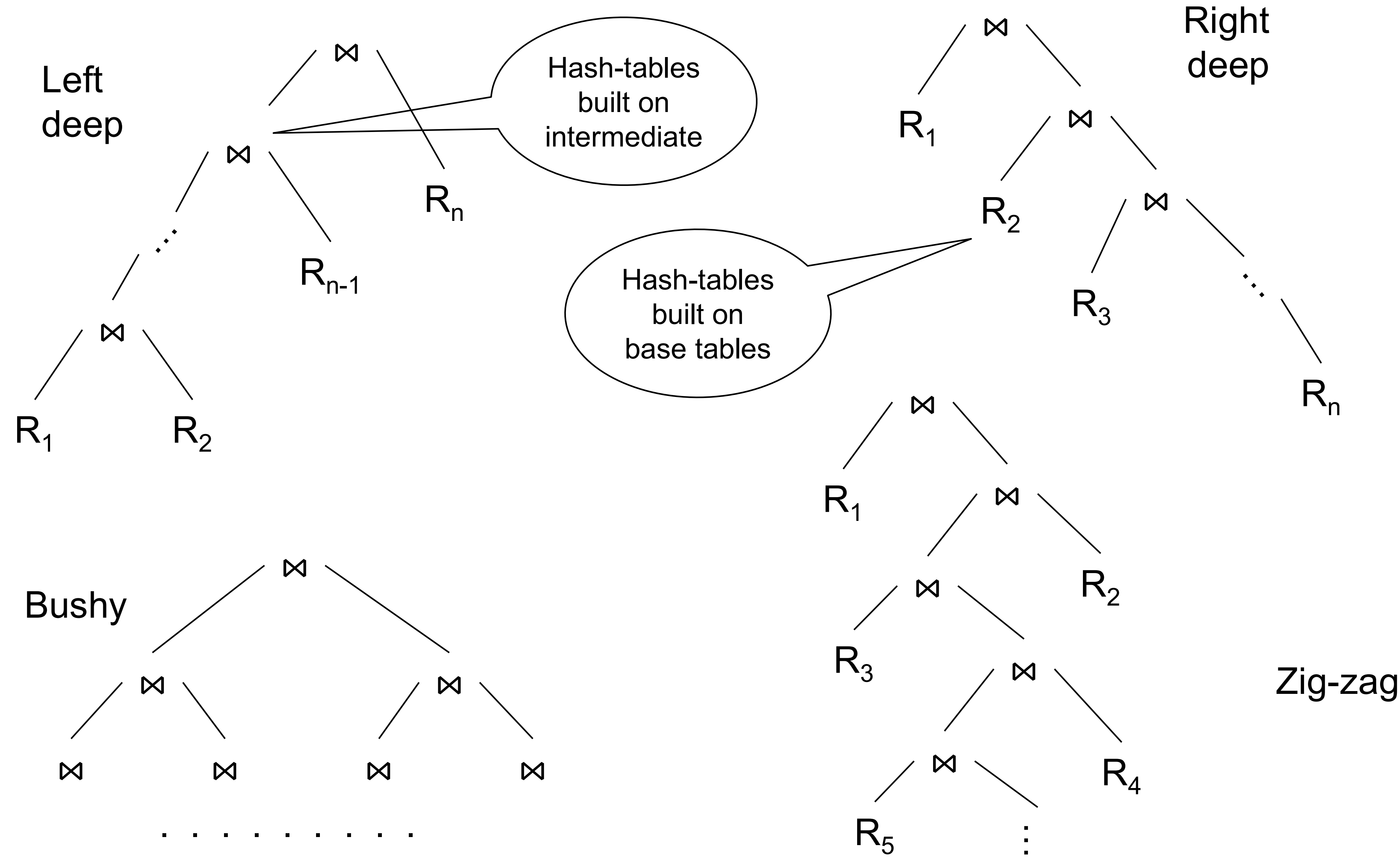
Shapes of Join Trees



Shapes of Join Trees



Shapes of Join Trees



[How good are they]

Left/right convention switches: Depending on Author/Convention

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)

Search Space: Discussion

- Search space can be huge

- Database systems often reduce it by applying heuristics:
  - No cartesian products
  - Restrict to left-deep trees (or other restriction)

Rewrite Rules

- We have seen last time:
  - Push selection down: $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
  - AND: $\sigma_{C1 \text{ and } C2}(R \bowtie S) = \sigma_{C1}(\sigma_{C2}(R \bowtie S))$
  - Join associativity: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
  - Join commutativity: $R \bowtie S = S \bowtie R$
- Two more rules
  - Push aggregates down
  - Remove redundant joins

Very important for Data Science!

Motivation

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

Answer: 1500000
Time: 2 s

Motivation

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

Answer: 1500000
Time: 2 s

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

Answer: 59986052
Time: 1 s

Motivation

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

Answer: 1500000
Time: 2 s

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

Answer: 59986052
Time: 1 s

```
select count(*) from customer, lineitem;
```

Motivation

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

Answer: 1500000
Time: 2 s

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

Answer: 59986052
Time: 1 s

```
select count(*) from customer, lineitem;
```

Timeout!!!

Motivation

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

Answer: 1500000
Time: 2 s

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

Answer: 59986052
Time: 1 s

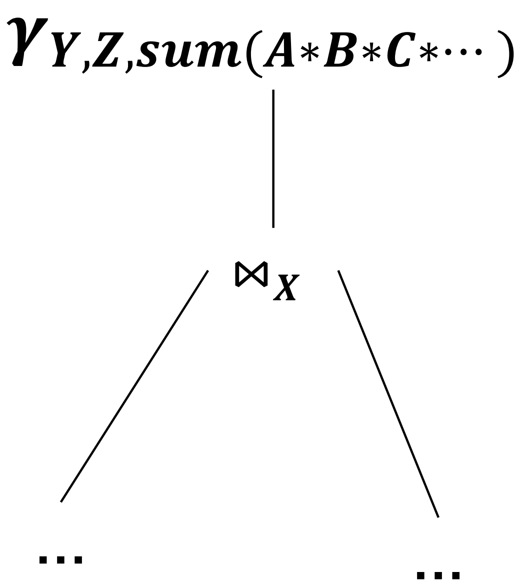
```
select count(*) from customer, lineitem;
```

Timeout!!!

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

Pushing Aggregates Down

select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z



Pushing Aggregates Down

```
select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z
```

$$\gamma_{Y,Z,sum(A*B*C*\cdots)}$$

$$\bowtie_X$$

…　　　　…

As data scientists, you may really need this optimization; do it manually, if needed!

Pushing Aggregates Down

select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z

$$\gamma_{Y,Z,sum(A*B*C*\cdots)}$$

$$\bowtie_X$$

… …

$$\gamma_{Y,Z,sum(S1*S2)}$$

$$\bowtie_X$$

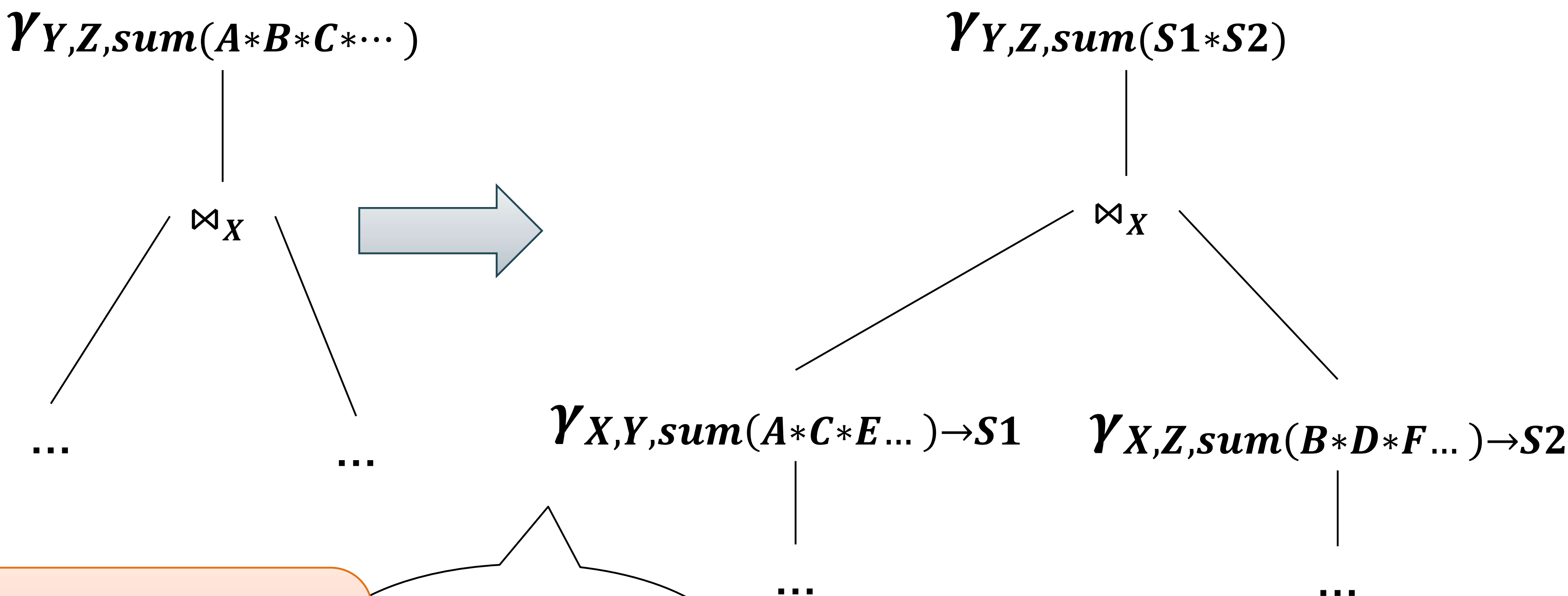
$$\gamma_{X,Y,sum(A*C*E\ldots)\rightarrow S1} \qquad \gamma_{X,Z,sum(B*D*F\ldots)\rightarrow S2}$$

… …

As data scientists, you may really need this optimization; do it manually, if needed!

Pushing Aggregates Down

select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z



As data scientists, you may really need this optimization; do it manually, if needed!

Group by the attrs from the left Y, plus join attrs X

Pushing Aggregates Down

select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z

$$\gamma_{Y,Z,sum(A*B*C*\cdots)}$$

$$\bowtie_X$$

… …

$$\gamma_{Y,Z,sum(S1*S2)}$$

$$\bowtie_X$$

$$\gamma_{X,Y,sum(A*C*E\ldots)\to S1} \qquad \gamma_{X,Z,sum(B*D*F\ldots)\to S2}$$

… …

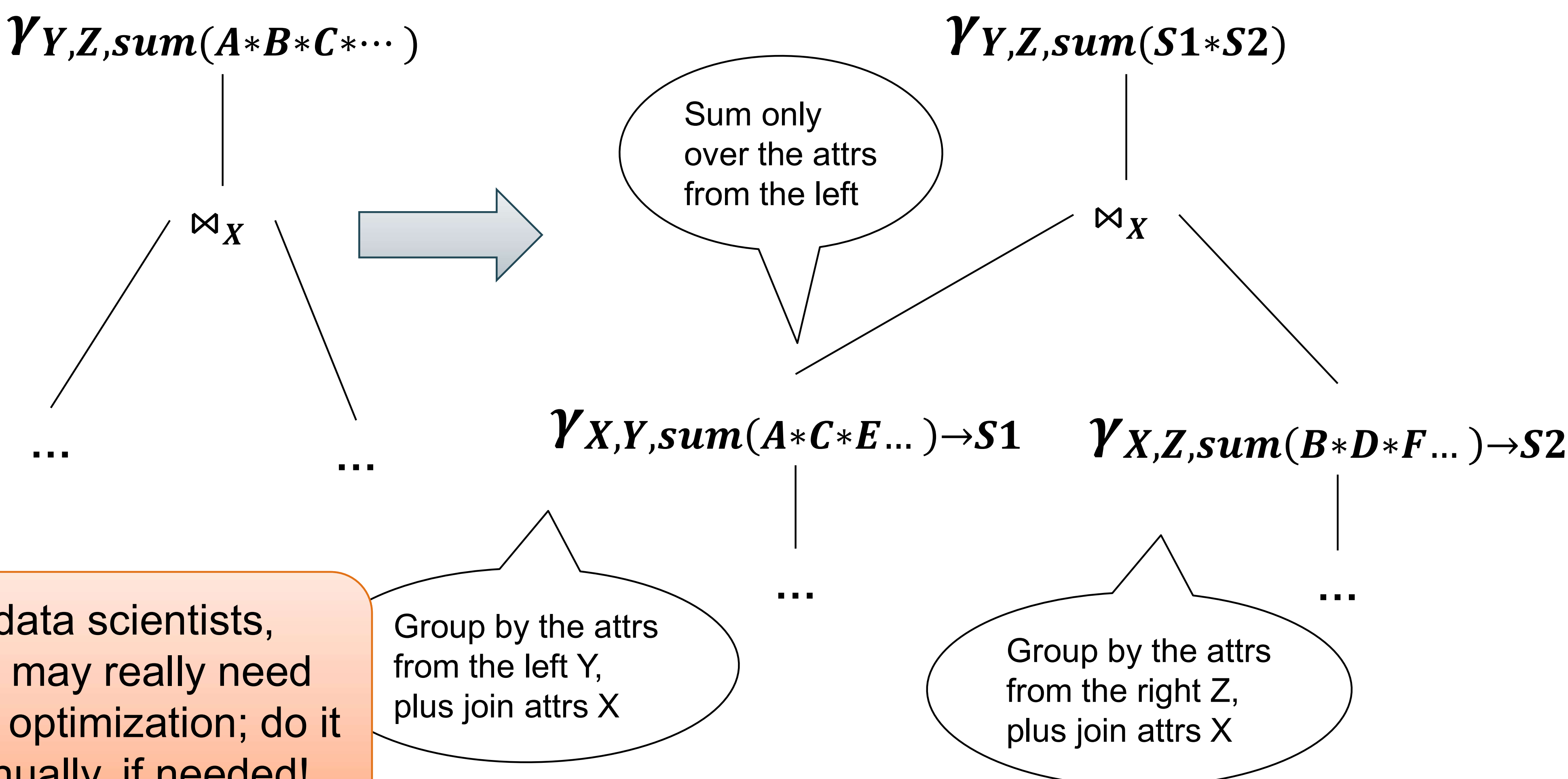
Group by the attrs from the left Y, plus join attrs X

Group by the attrs from the right Z, plus join attrs X

As data scientists, you may really need this optimization; do it manually, if needed!

Pushing Aggregates Down

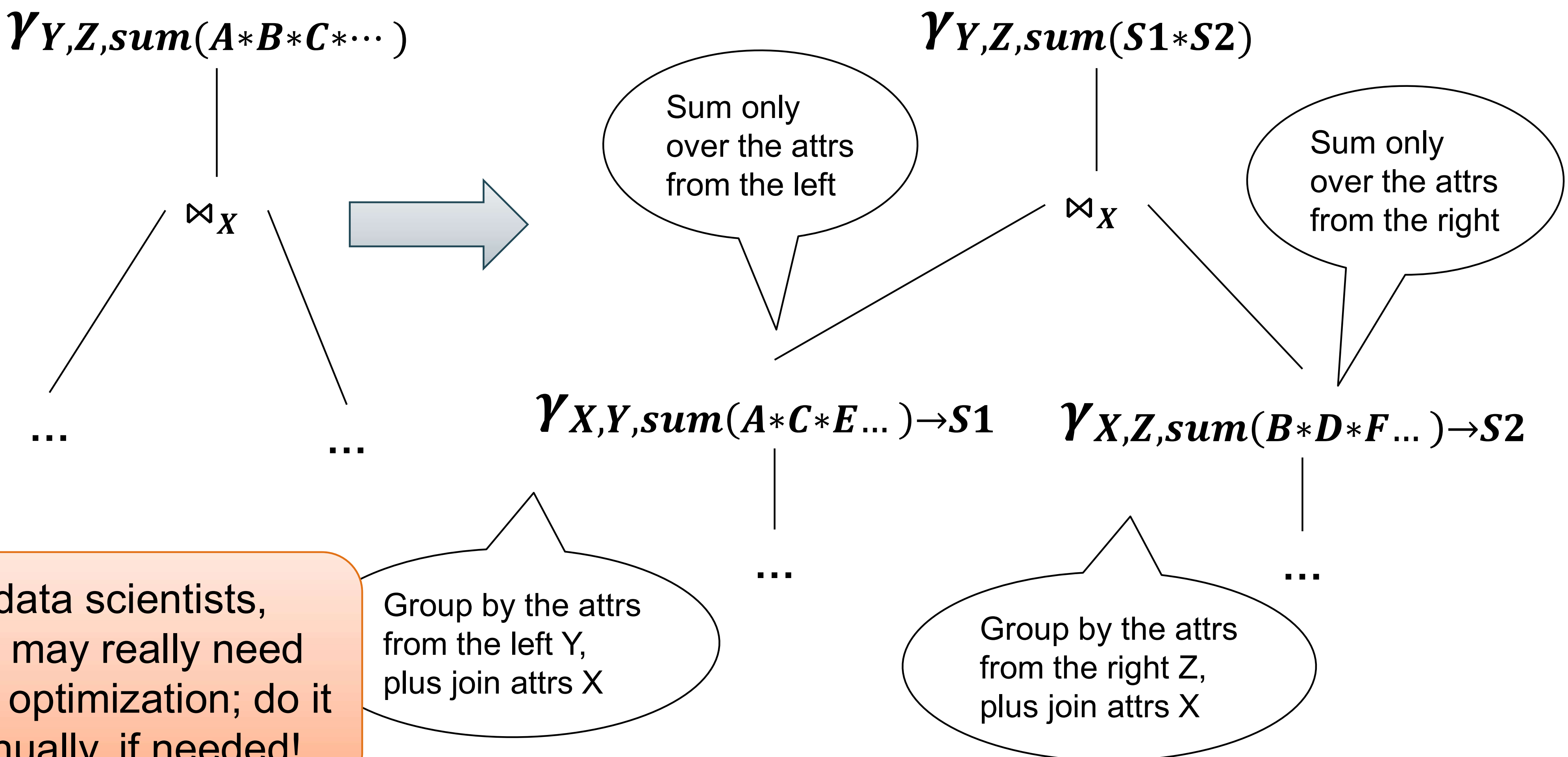
select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z



As data scientists, you may really need this optimization; do it manually, if needed!

Pushing Aggregates Down

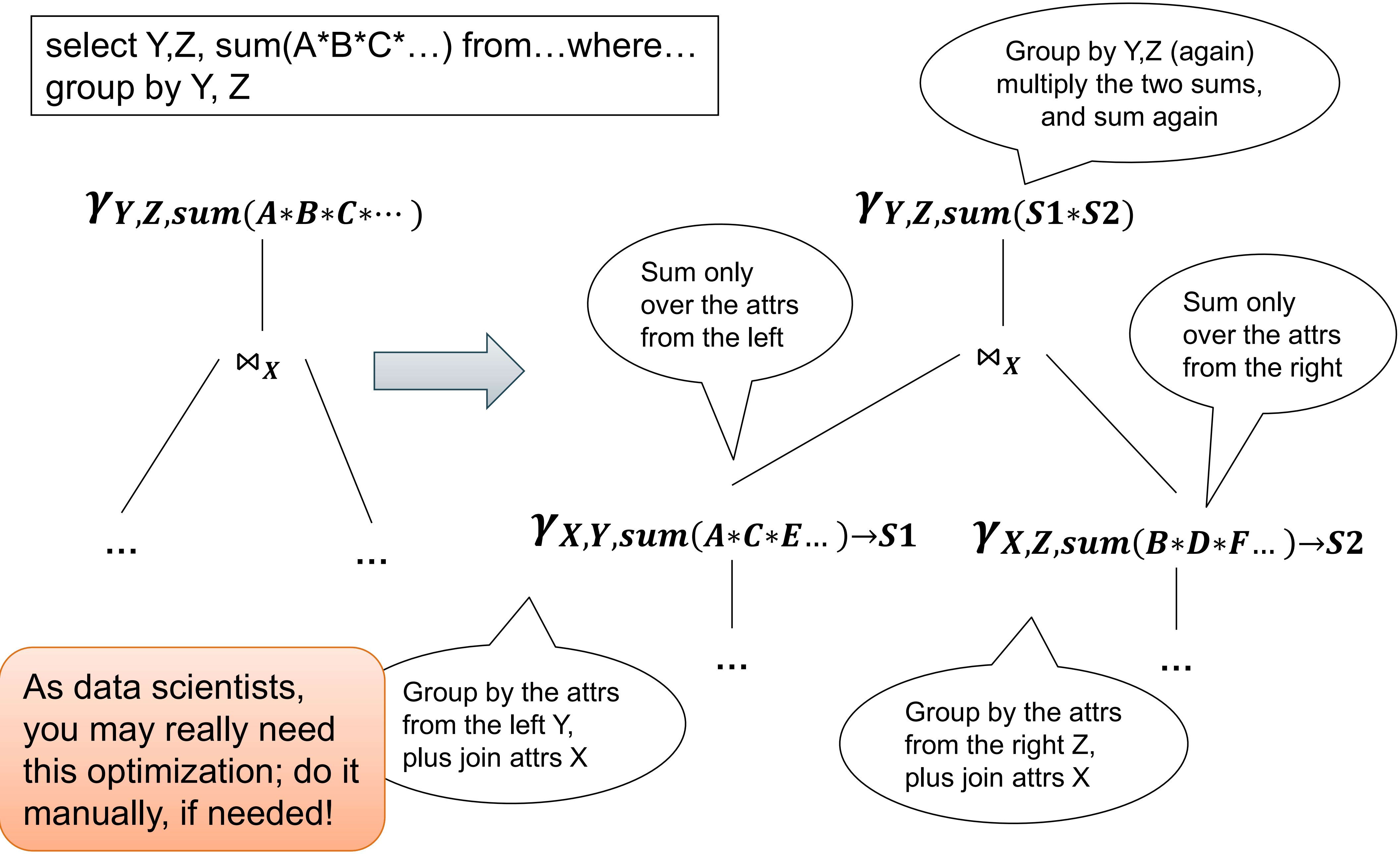
select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z



As data scientists, you may really need this optimization; do it manually, if needed!

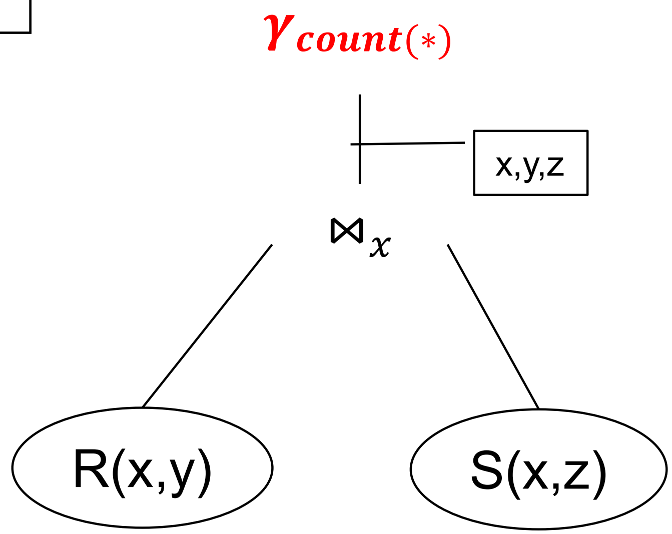
Pushing Aggregates Down

select Y,Z, sum(A*B*C*…) from…where…
group by Y, Z



Example 1

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



Example 1

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

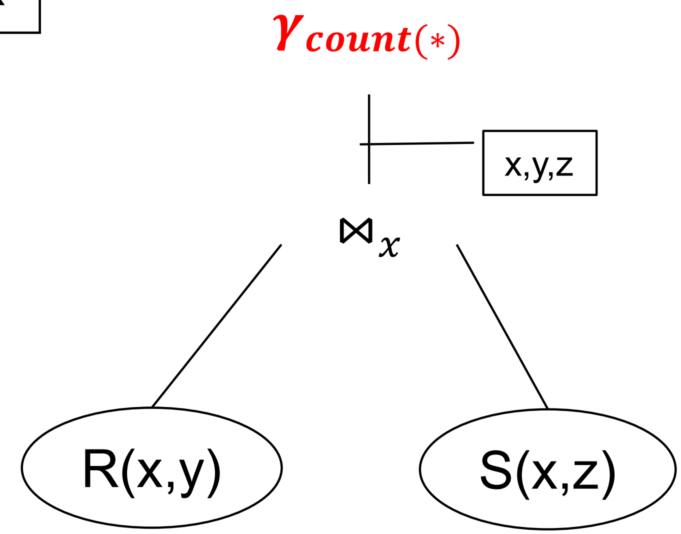
R:

x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = ????



Example 1

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

R:

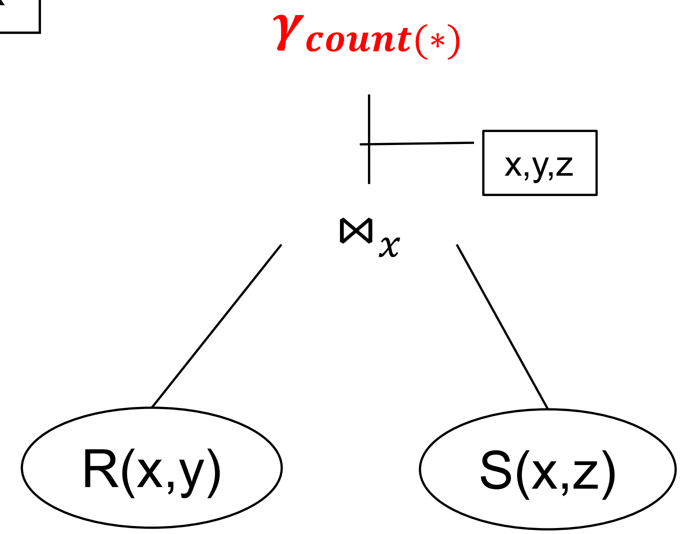
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



Example 1

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

R:

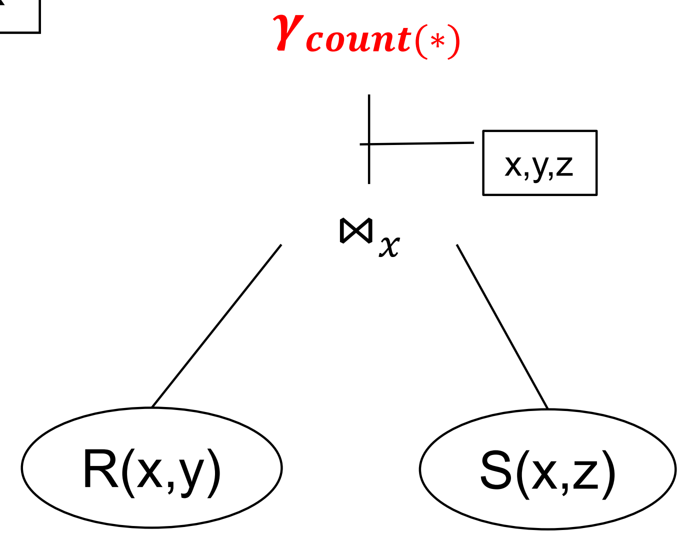
x	y
b	a
b	c
f	d
h	g

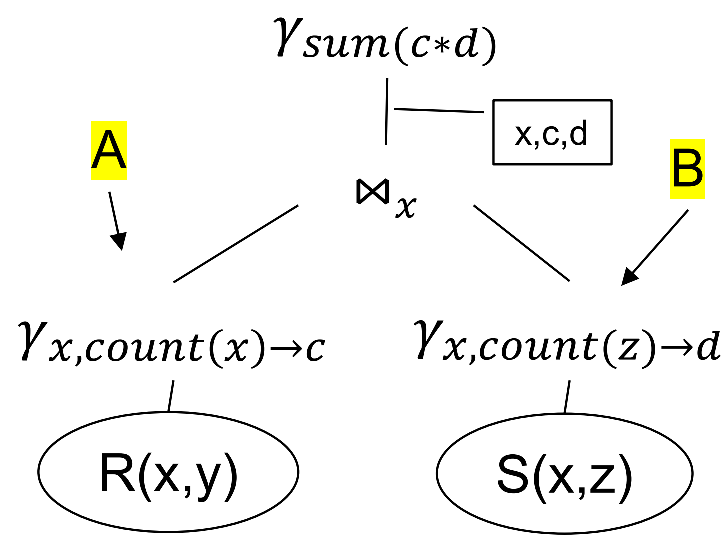
S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = O(N^2)





Example 1

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

R:

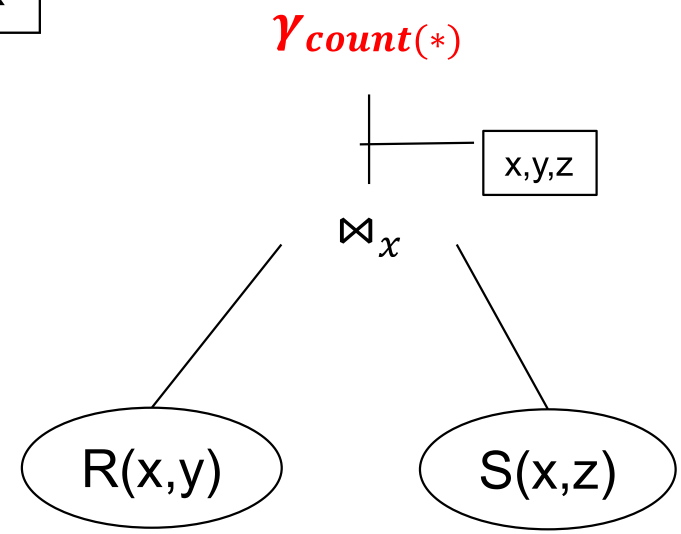
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A:

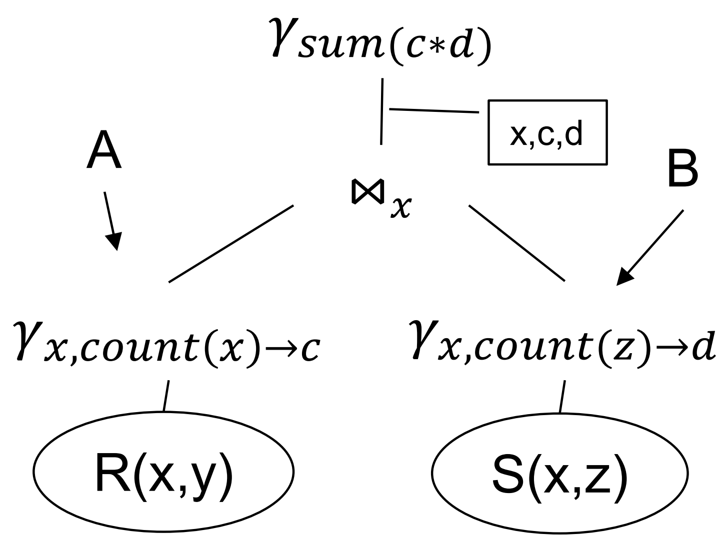
x	c
b	2
f	1
h	1

B:

x	d
b	2
h	1

A⋈B

x	c	d
b	2	2
h	1	1



Example 1

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

R:

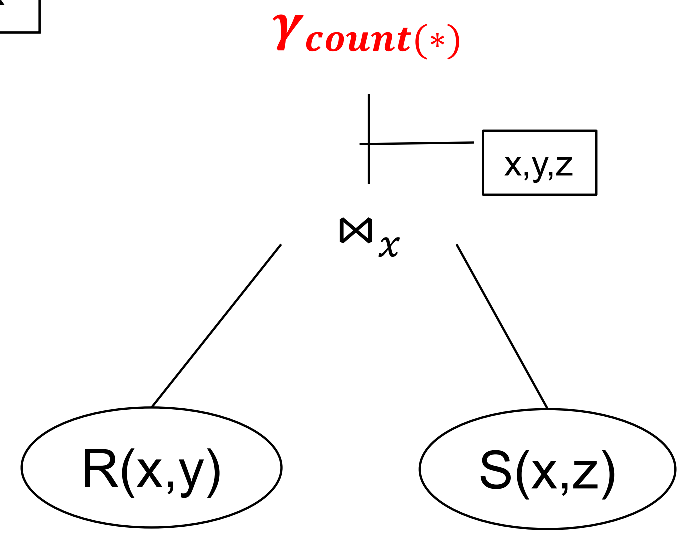
x	y
b	a
b	c
f	d
h	g

S:

x	z
b	g
b	k
h	m

Answer = 5

Runtime = O(N^2)



Answer = 5

Runtime = O(N)

A:

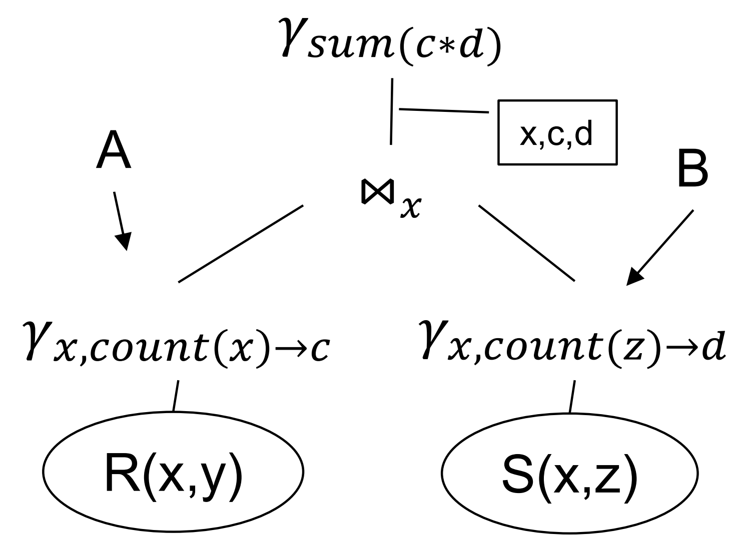
x	c
b	2
f	1
h	1

B:

x	d
b	2
h	1

A⋈B

x	c	d
b	2	2
h	1	1



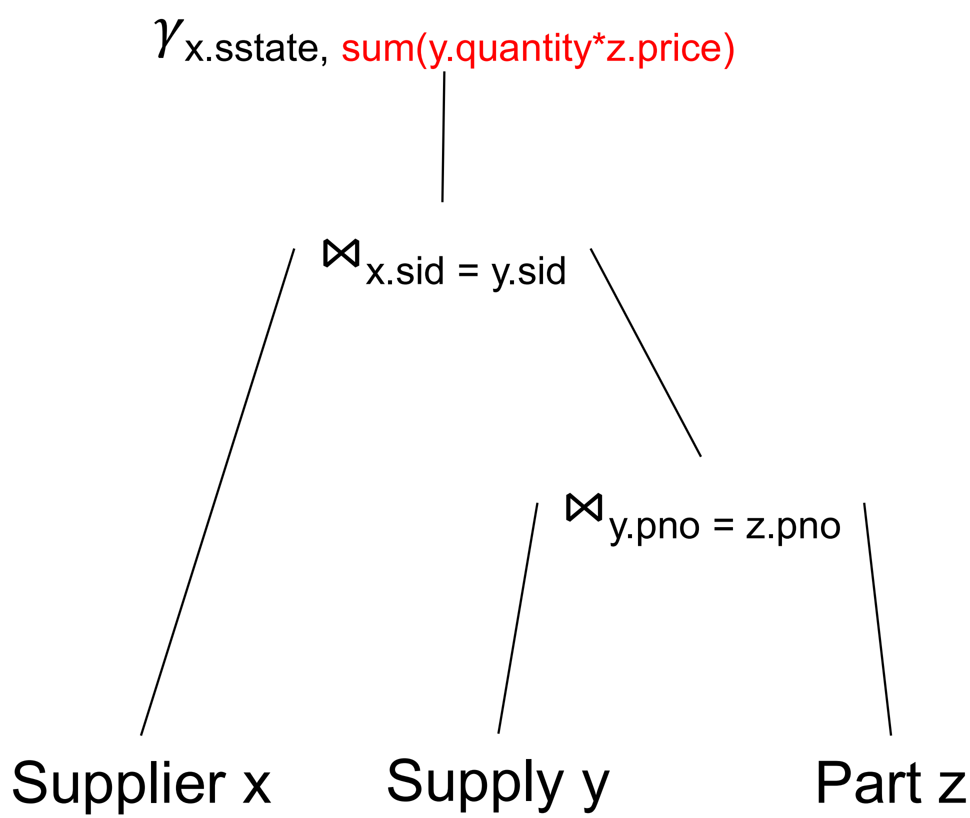
Example 2

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

```
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)

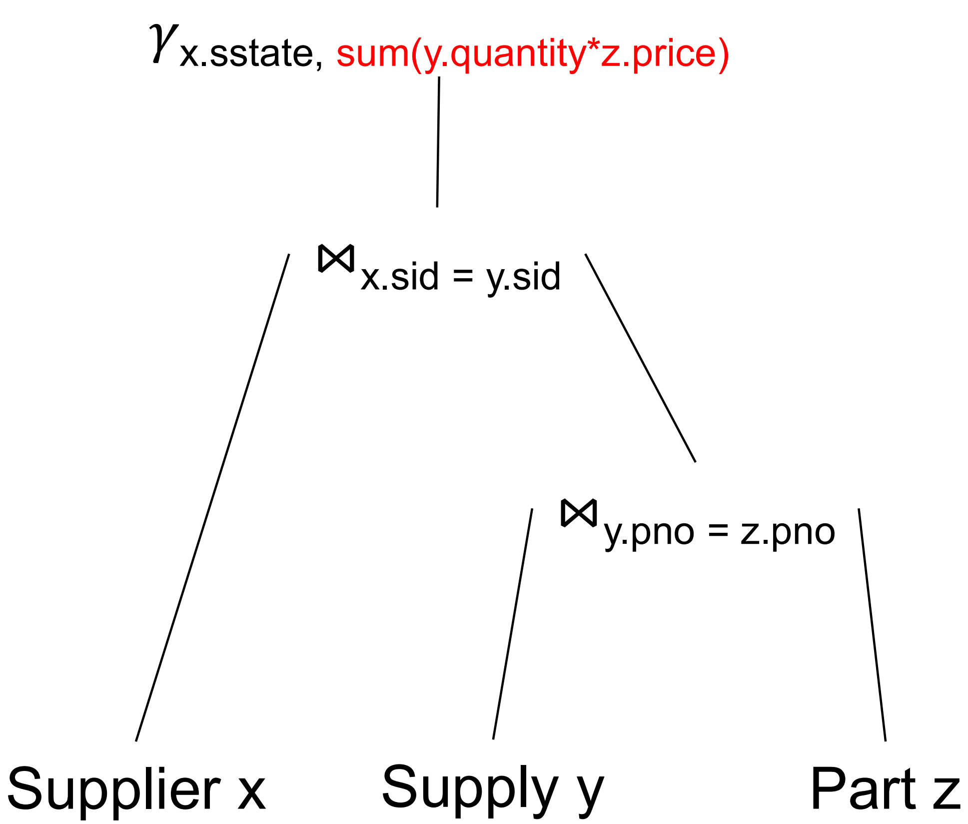
Example 2

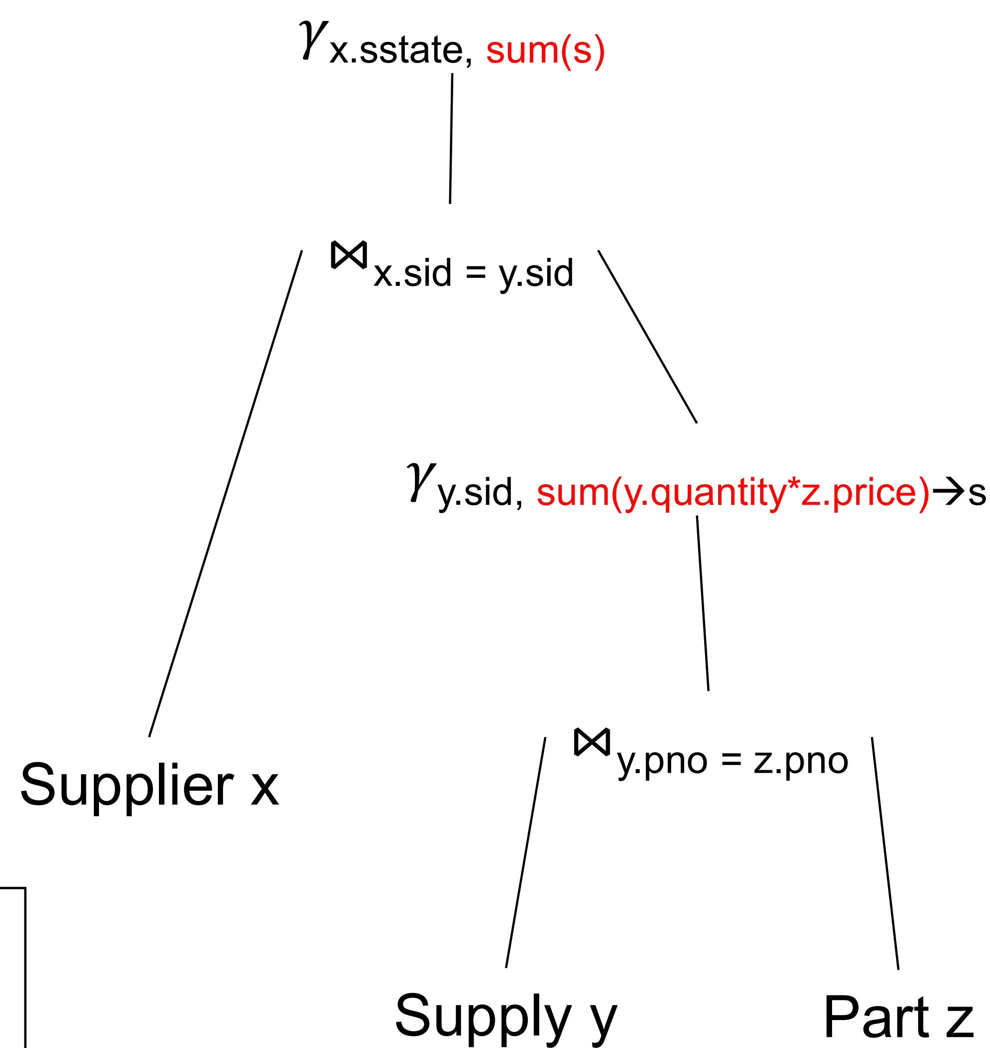



```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

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

Example 2






```
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
```

Discussion

- Join-aggregates: common in data science
- Implementation in RDBMS seems spotty:
  - Postgres: NO (someone started, abandoned)
  - Redshift: NO (I don't know the status)
  - SQL Server: YES (at least a few years back)
  - Snowflake: ??
- You may have to force this manually, by writing nested SQL queries
- Let's make sure we understand it (next)

Redundant Foreign-key / key Joins

- Simple, highly effective
- Almost all engines implement this

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

Foreign-Key / 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)

Foreign-Key / 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)

Foreign-Key / 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

Summary of Rules

- Database optimizers typically have a database of rewrite rules
- E.g. SQL Server: 400+ rules
- Rules become complex as they need to serve specialized types of queries

Query Optimization

1. Search space
2. Cardinality and cost estimation

Discussed already

3. Plan enumeration algorithms

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
- Today's systems combine both

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: $Q = \{R_1, \ldots, R_n\}$

Avoid cartesian products; possibly restrict tree shapes

Details

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

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

Details

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

- Size = $T(R_i)$
- Best plan: $scan(R_i)$
- Cost = $c*T(R_i)$ // c=the cost to read one tuple

Details

Step k = 2…n:
For each $Q = \{R_{i_1}, \ldots, R_{i_k}\}$ // w/o cartesian product

- Size = estimate the size of Q

- For each j=1,…,k:
  - Let: $Q' = Q - \{R_{i_j}\}$
  - Let: $\mathrm{Plan}(Q') \bowtie R_{i_j}$ $\quad \mathrm{Cost}(Q') + \mathrm{CostOf}(\bowtie)$

- $\mathrm{Plan}(Q), \mathrm{Cost}(Q) =$ cheapest of the above

[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

- Optimizer has three components:
  - Search space
  - Cardinality and cost estimation
  - Plan enumeration algorithms
- Optimizer realizes *physical data independence*
- Weakest link: cardinality estimation
  - Poor plans are almost always due to that

Spark

Distributed or Parallel Query Processing

- Clusters:
  - More servers → more in main memory
  - More servers → more computing power
  - Clusters are now cheaply available in the cloud
  - *Distributed* query procesing
- Multicores:
  - The end of Moore's law
  - *Parallel* query processing

Motivation

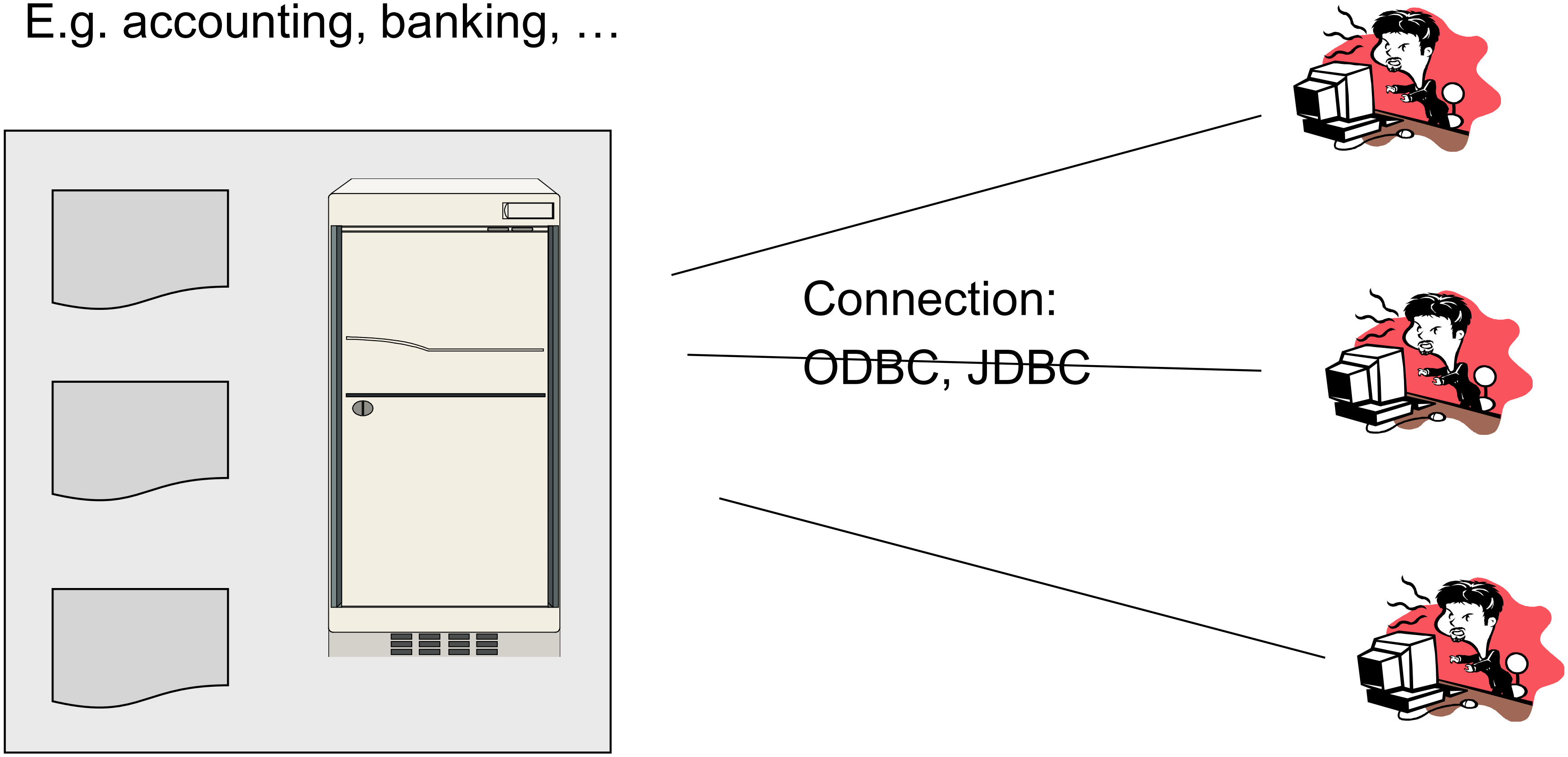
- Limitations of relational database systems:
  - Single server (at least traditionally)
  - SQL is a limited language (eg no iteration)
- Spark:
  - Distributed system
  - Functional language (Java/Scala) good for ML
- Implementation:
  - Extension of MapReduce
  - Distributed physical operators

Review: Single Client

E.g. data analytics

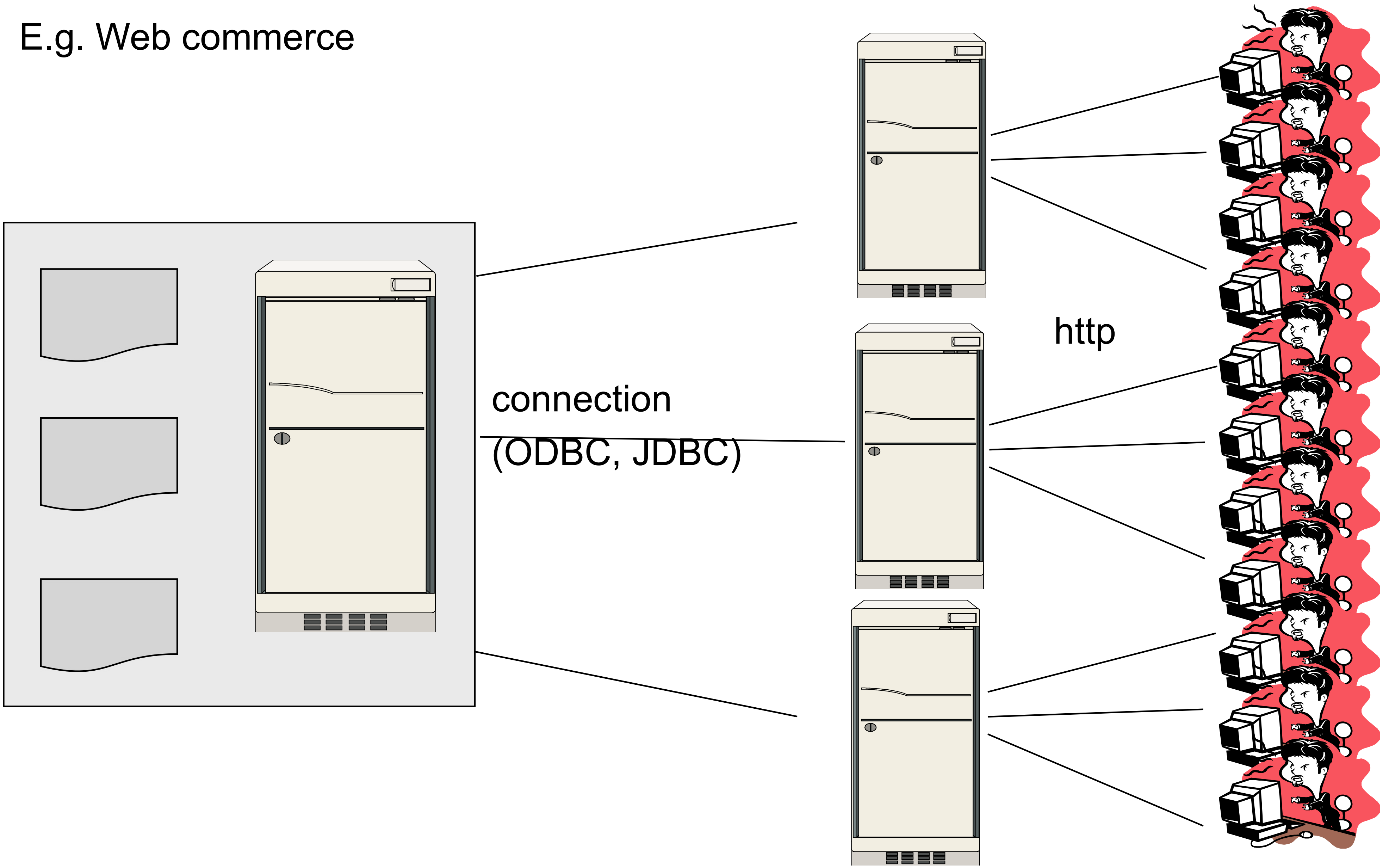
Review: Client-Server

E.g. accounting, banking, …

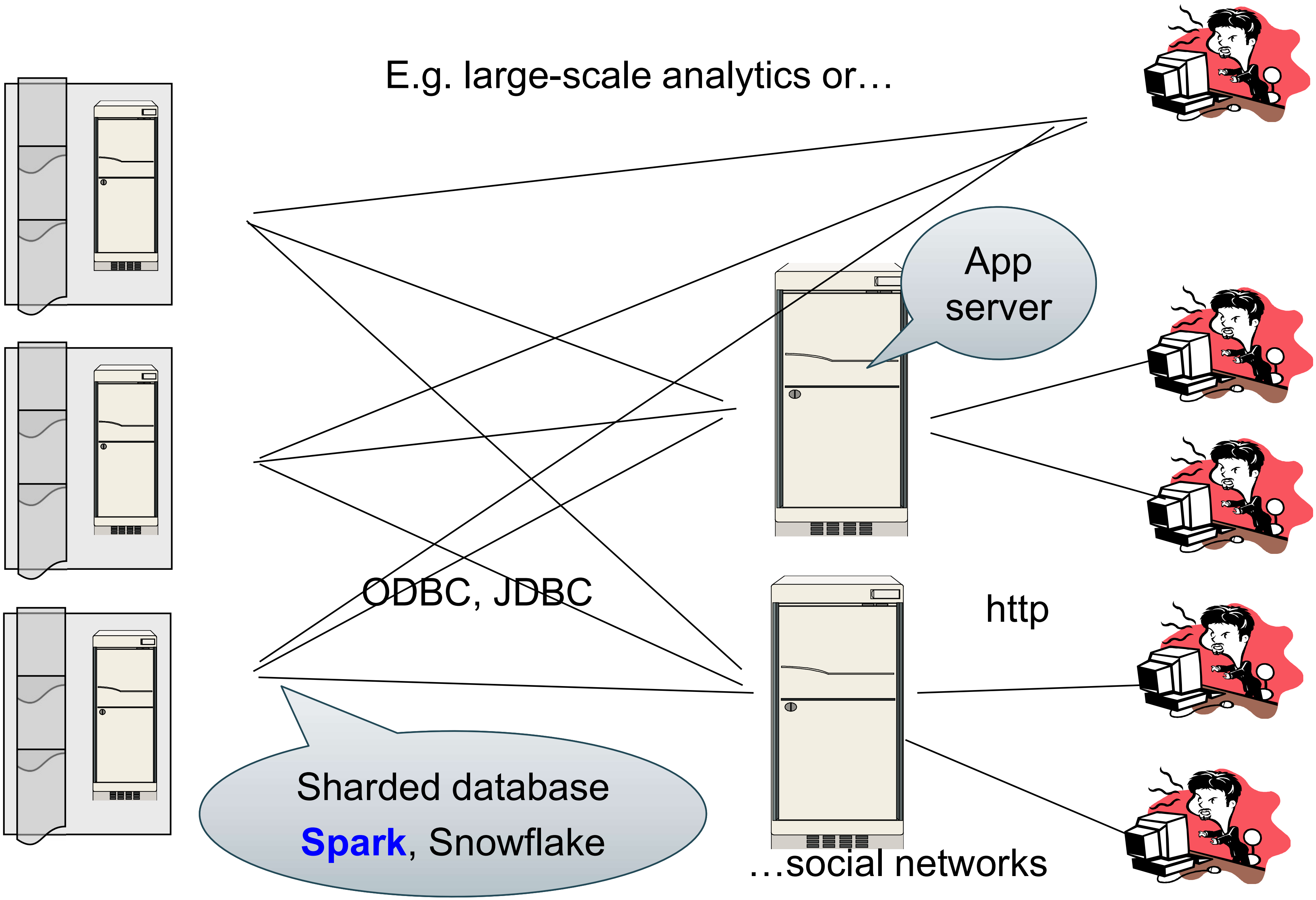


Review: Three-tier

E.g. Web commerce



Review: Distributed Database



Programming in Spark

- A Spark program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- Eager: operators are executed immediately
- Lazy: operators are not executed immediately
  - A *operator tree* is constructed in memory instead
  - Similar to a relational algebra tree

Collections in Spark

RDD<T> = an RDD collection of type T

- Distributed on many servers, not nested
- Operations are done in parallel
- Recoverable via lineage; more later

Seq<T> = a sequence

- Local to one server, may be nested
- Operations are done sequentially

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")).count()
```

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile(“hdfs://…”)

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith(“Error”))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains(“MySQL”)).count()
```

Transformation: Not executed yet…

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")).count()
```

Transformation: Not executed yet…

Action: triggers execution of entire program

Anonymous Functions

A.k.a. lambda expressions, starting in Java 8

```
errors = lines.filter(x -> x.startsWith("Error"))
```

Chaining Style

```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

The RDD s:

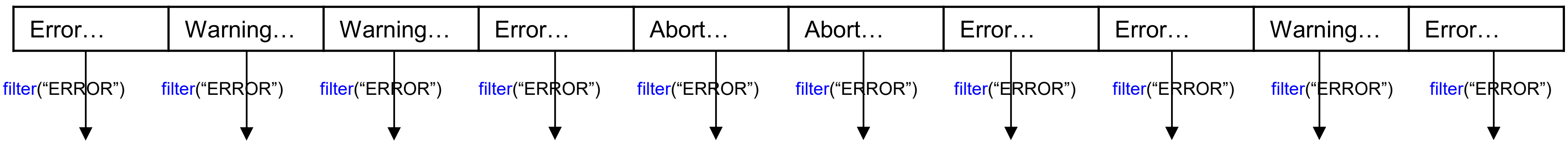
Error…	Warning…	Warning…	Error…	Abort…	Abort…	Error…	Error…	Warning…	Error…

```
sqlerrors = spark.textFile(“hdfs://…”)
            .filter(x -> x.startsWith(“ERROR”))
            .filter(x -> x.contains(“sqlite”))
            .collect();
```

Example

Parallel step 1

The RDD s:

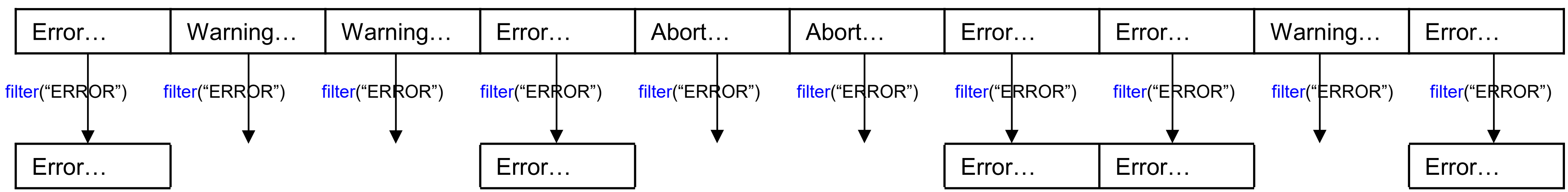



```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

Parallel step 1

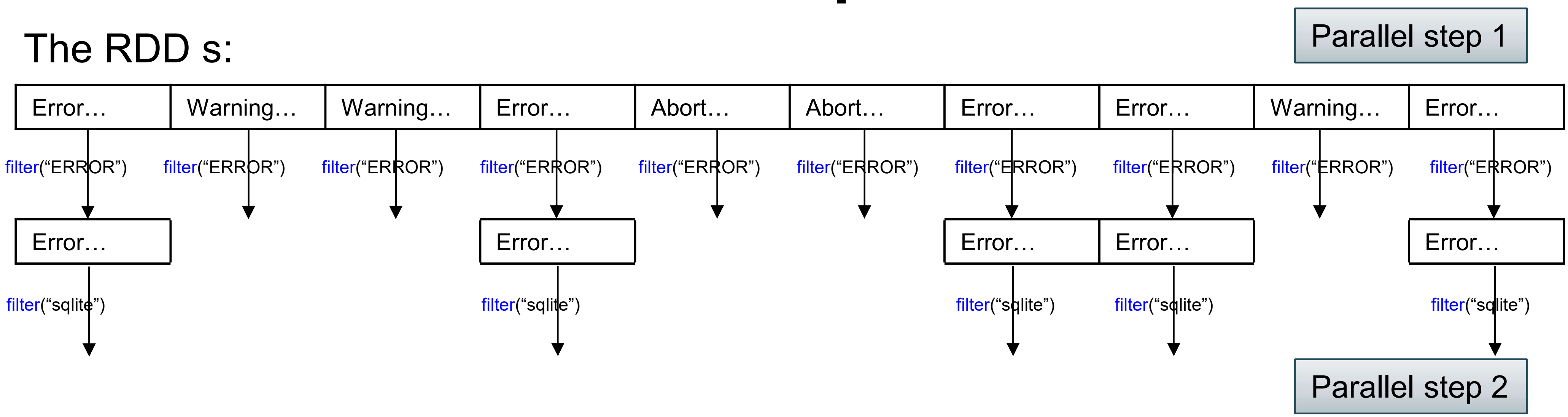
The RDD s:




```
sqlerrors = spark.textFile("hdfs://…")
              .filter(x -> x.startsWith("ERROR"))
              .filter(x -> x.contains("sqlite"))
              .collect();
```

Example

The RDD s:




```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

More on Programming Interface

Large set of pre-defined transformations:

- Map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, crossProduct, …

Small set of pre-defined actions:

- Count, collect, reduce, lookup, and save

Programming interface includes **iterations**

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS