CSE544
Data Management

Lecture 16
Query Optimization Wrapup

Announcements

- Project: MAJOR milestone next Friday!
- 1-1 Meetings on Monday, Dec. 1st
- Presentations on Wednesday, Dec 3rd
- Last HW: datalog + theory questions

Paper Discussion

- How Good Are Query Optimizers, Really? VLDB'2015

Questions in the paper

- How good are cardinality estimators?
- How important are they for the optimizer?
- How large does the plan space need to be?

[How good are they]

Cardinality Estimators

- Standard database benchmark: TPC-H
- They designed a new benchmark. Why?

Cardinality Estimators

- Standard database benchmark: TPC-H
- They designed a new benchmark. Why?
- Because TPC-H is synthetically generated, unrealistically uniform

Cardinality Estimators

What type of queries are in IMDB/JOB?

Cardinality Estimators

What type of queries are in IMDB/JOB?

- For CE: select * multijoin queries
- For runtime: replace * with min Why?

Cardinality Estimators

What type of queries are in IMDB/JOB?

- For CE: select * multijoin queries
- For runtime: replace * with min Why?

- Materializing * is expensive…
- …and postgres does not push min down the plan

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

Single Table Estimation

What technique helped here? (conjectured)

	median	90th	95th	
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

Single Table Estimation

	median	90th	95th	
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 f elections

What technique helped here? (conjectured)

Sampling. E.g. Hyper: 1000 rows

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

Single Table Estir

Why queries still lead to poor estimates?

	median	90th	95th	ax
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

[How good are they]

Single Table Estim

	median	90th	95th	ax
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 b ons

Why queries still lead to poor estimates?

Low selectivity: 10^{-5} – 10^{-6}

[How good are they]

Single Table Estimation

- 1d Histograms:
  - Good for single equality or range predicate
  - Poor for multiple predicates
  - Useless for LIKE
- Samples:
  - Good for multiple predicates, LIKE
  - Poor for low selectivity predicates

[How good are they]

Joins (0 to 6)

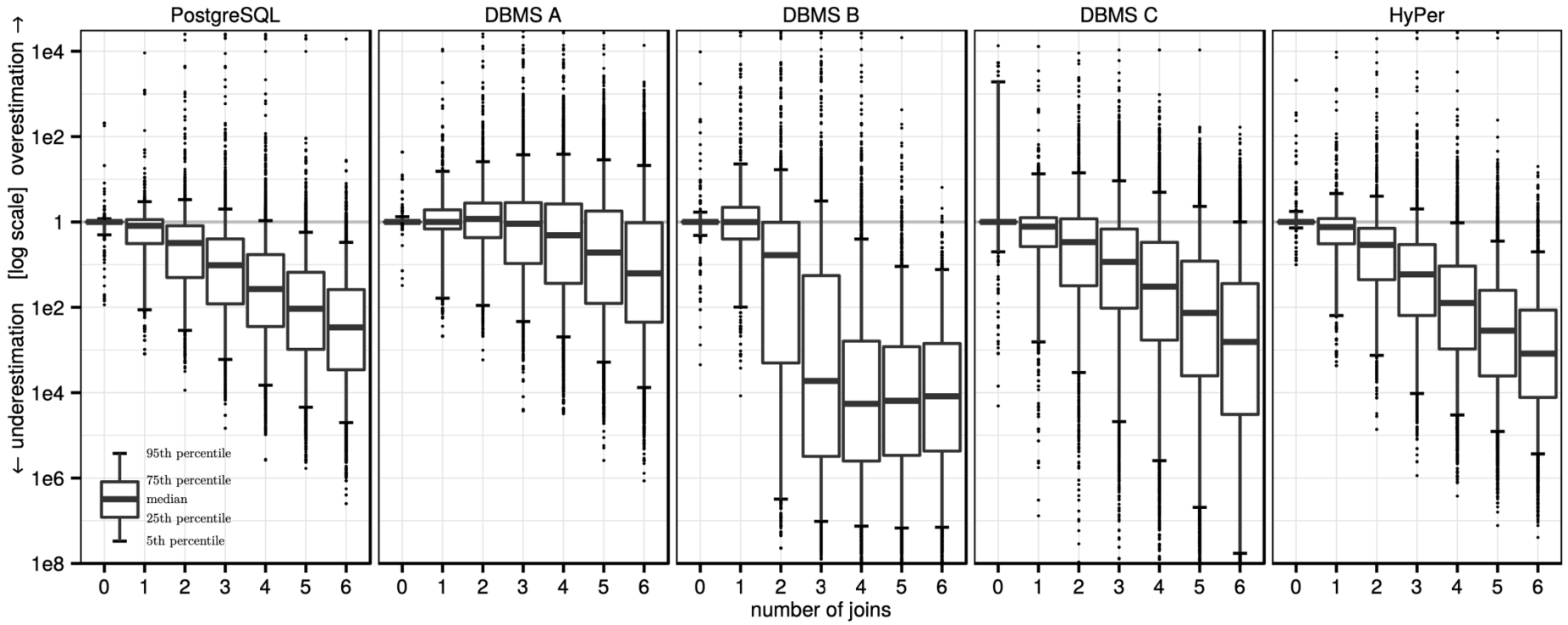


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

[How good are they]

Joins (0 to 6)

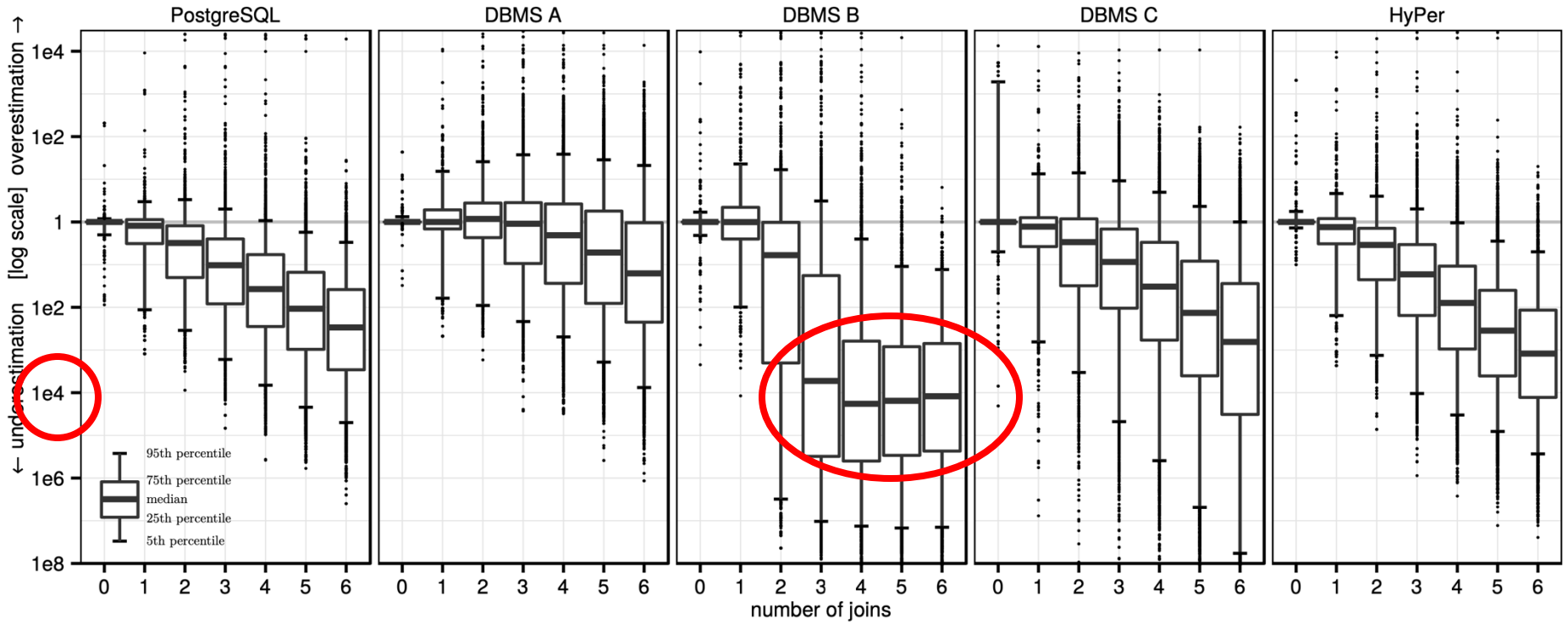


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

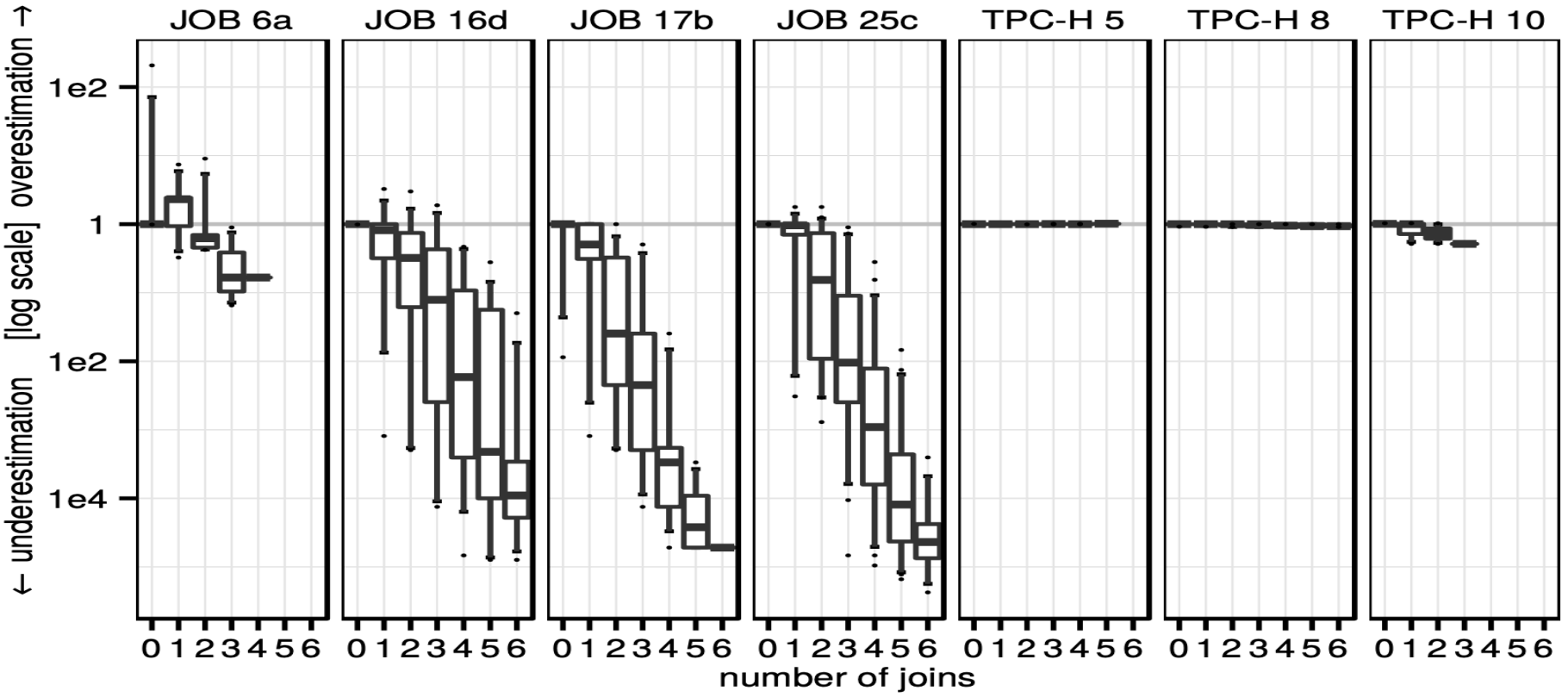
Estimation of Joins

- Error increases exponentially with the number of joins
  - This was known from [Ioannidis'91]

- Underestimate, because of positive correlations

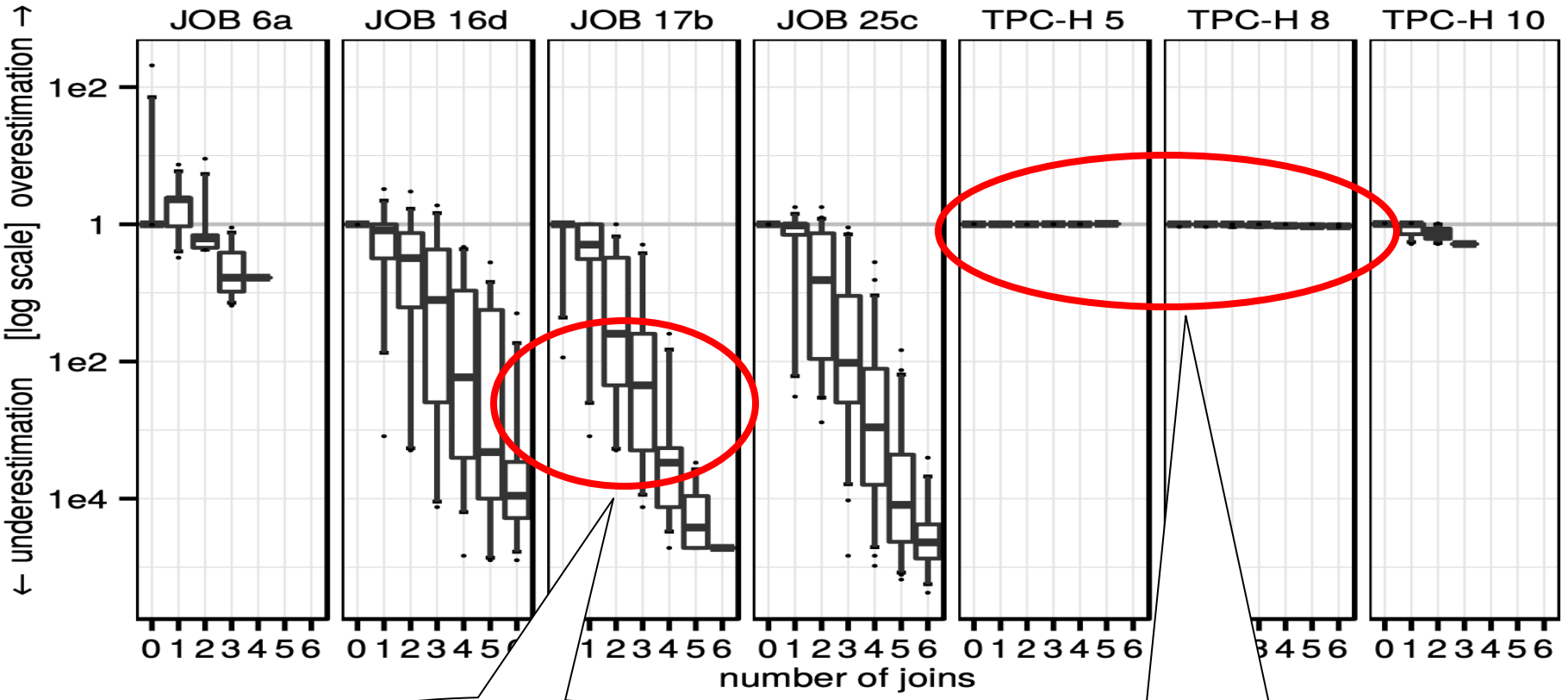
[How good are they]

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



[How good are they]

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



Impact of Mis-estimates

- Question: how much does a good/poor CE matter for the quality of a query plan
- How did they measure that?

Impact of Mis-estimates

- Question: how much does a good/poor CE matter for the quality of a query plan
- How did they measure that?
  - Inject into postgres other systems' estimates – won't discuss this
  - Inject into postgres true cardinalities; call it optimal plan, compare with regular plan
- Two configs of indexes: PK and PK+FK

Impact of Mis-estimates

PK indexes

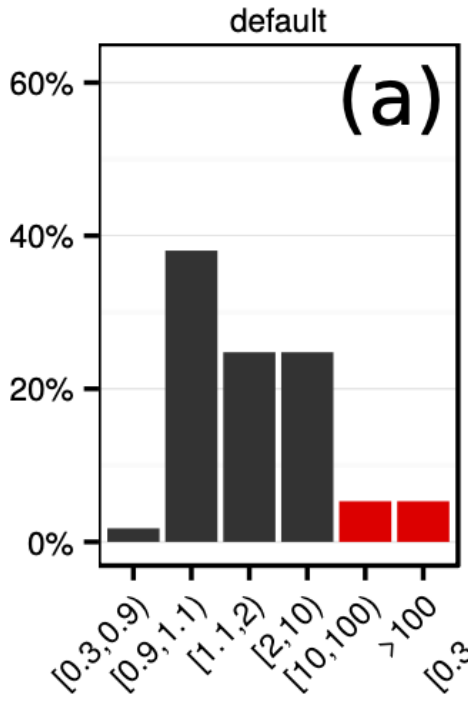


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes

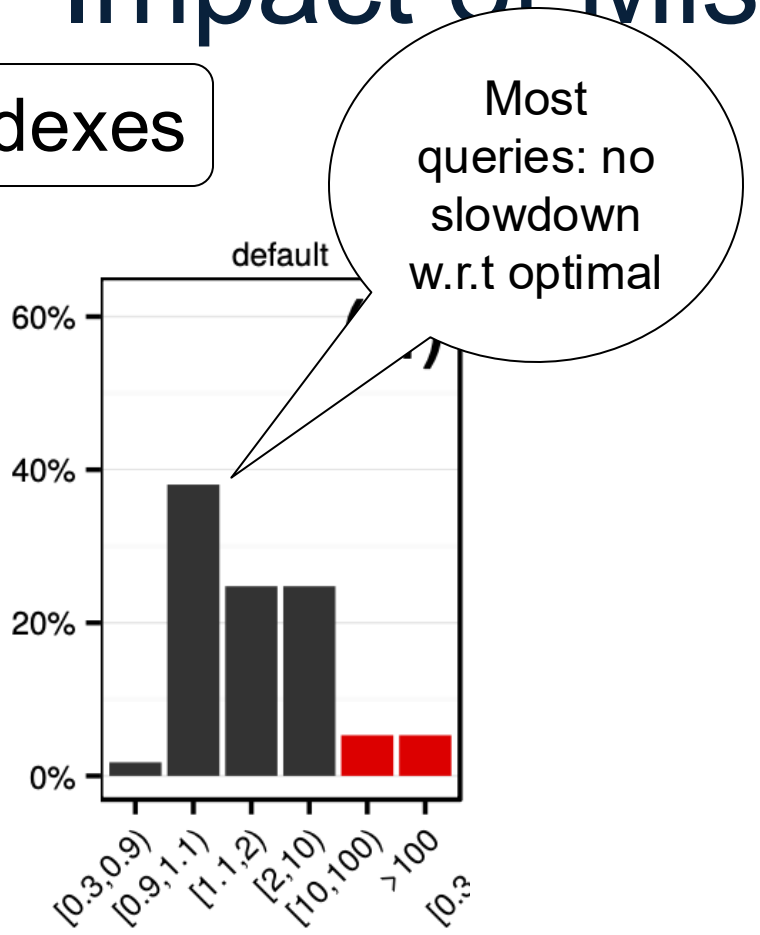


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes

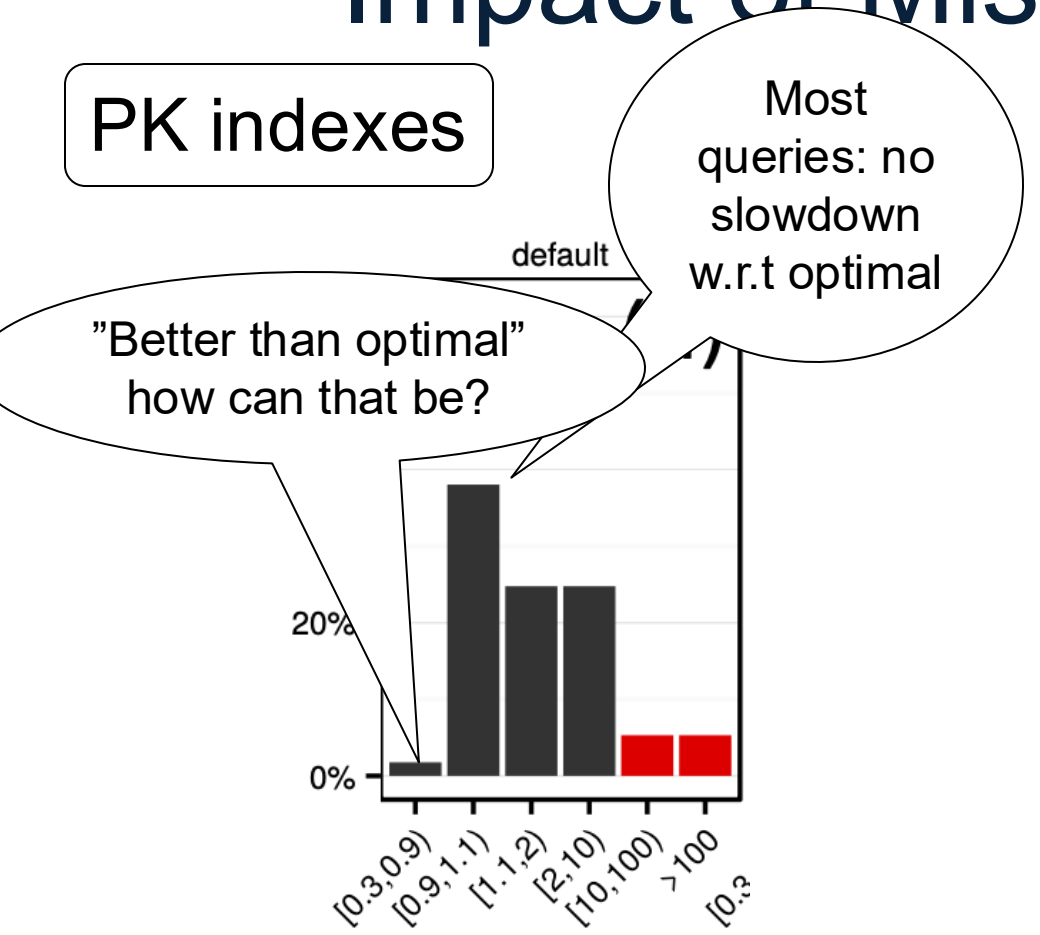


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes

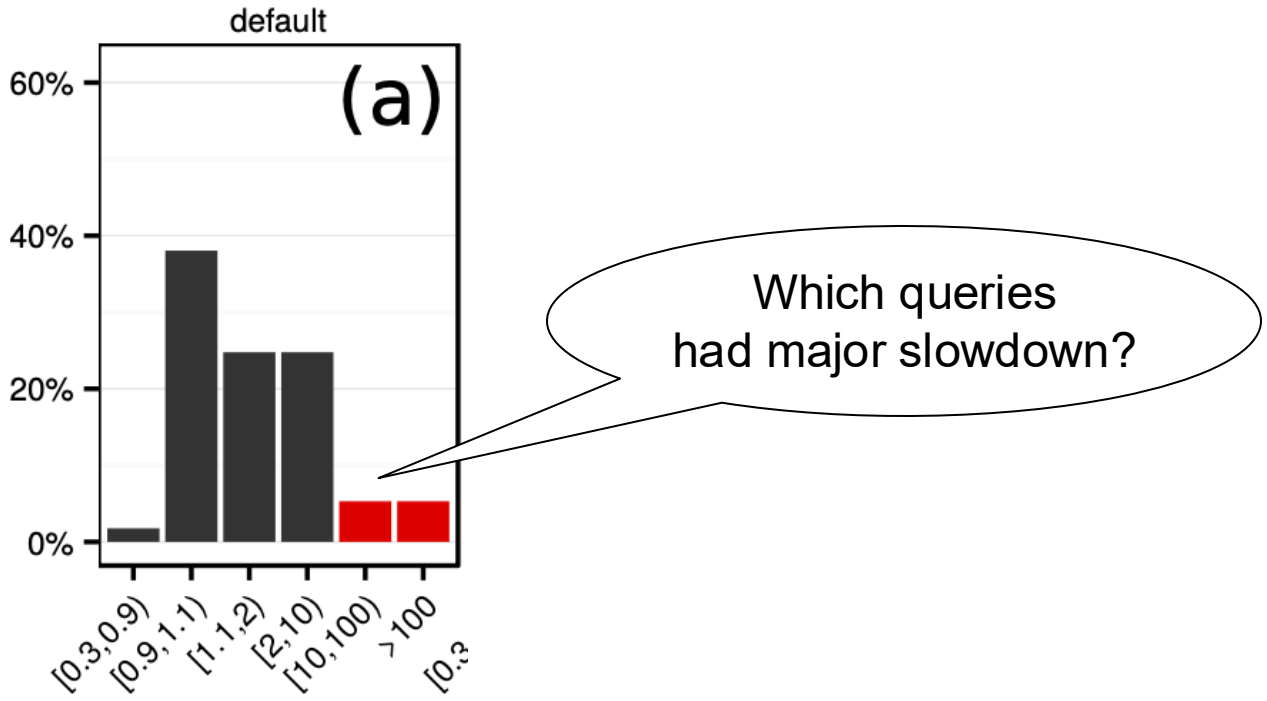


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes

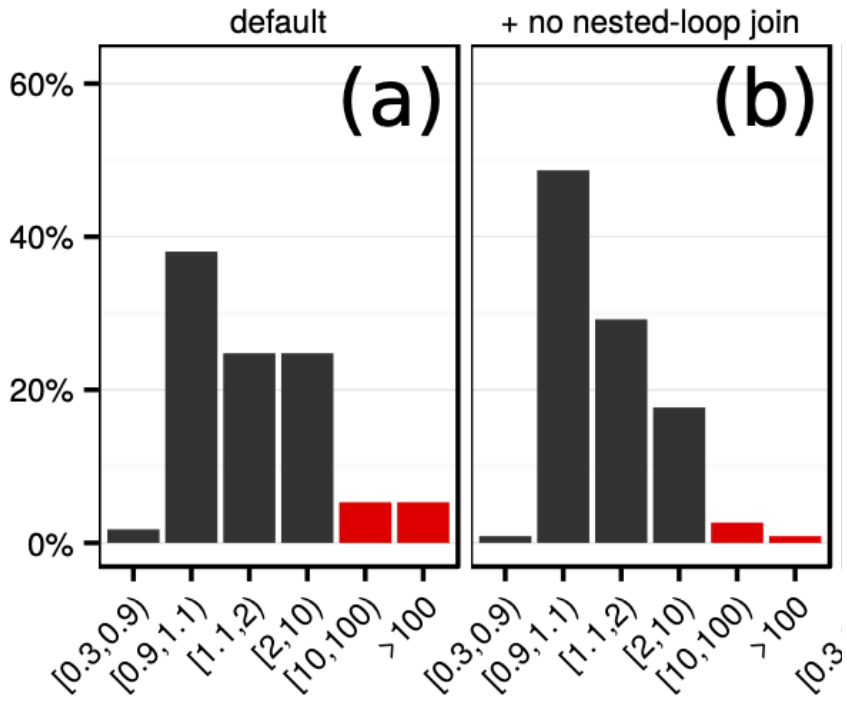
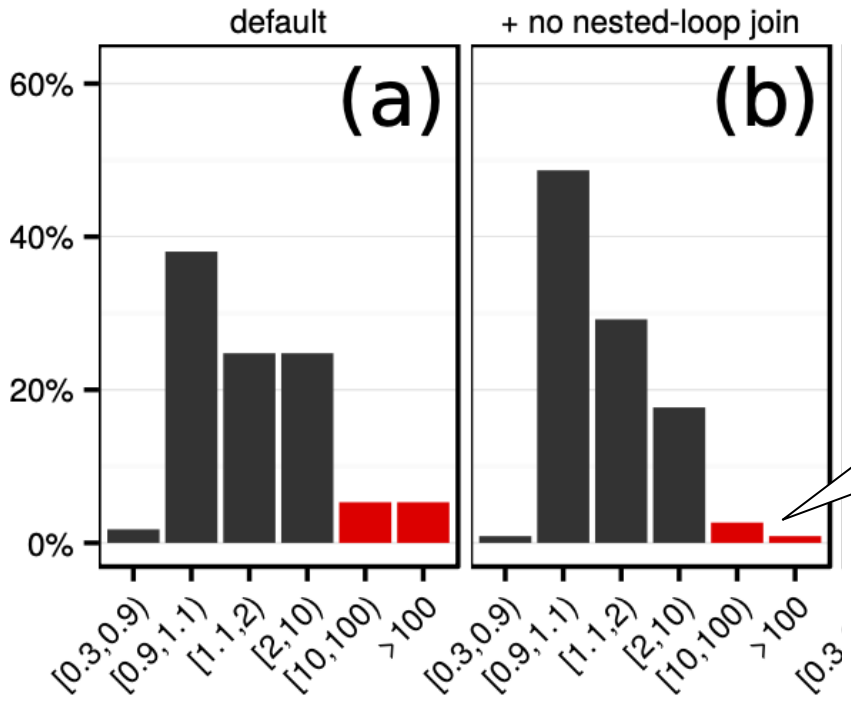


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes



Still some queries significantly slower.

Why?

Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

PK indexes

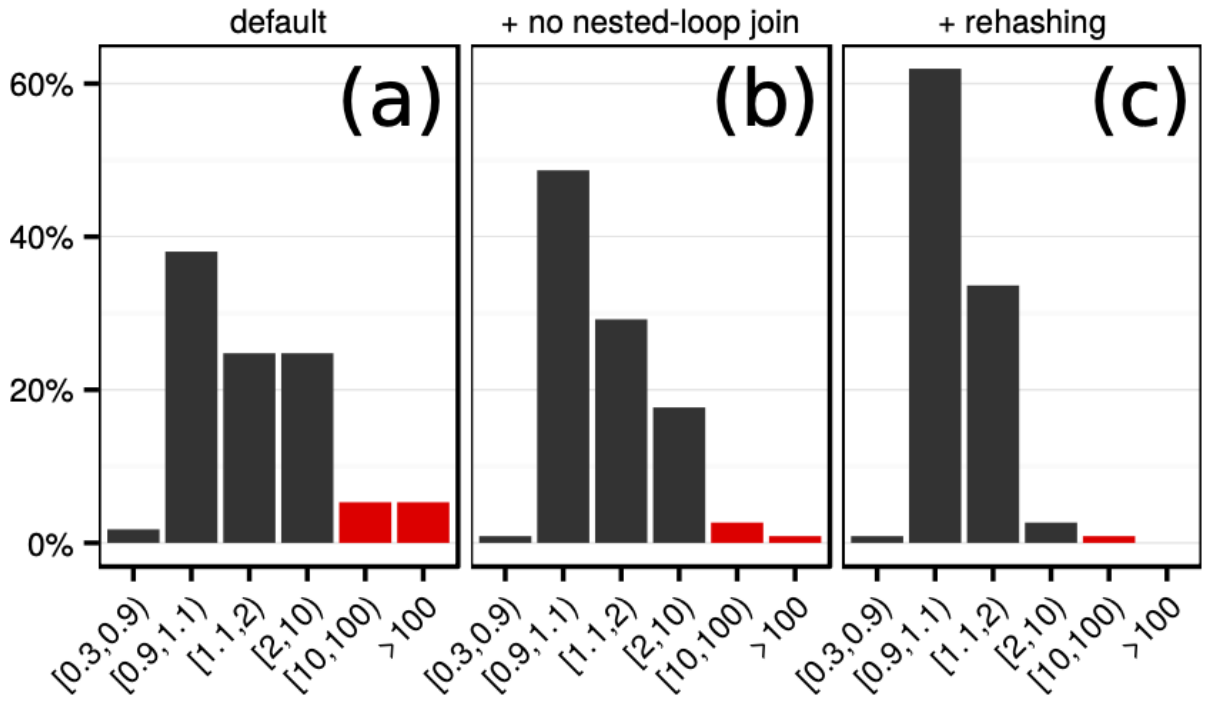


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

Impact of Mis-estimates

Indexes on PK only

- Low sensitivity to CE, because the “fact” table needs to be scanned anyway
- Plans most sensitive to CE errors:
  - Plans with nested-loop joins
  - Hash-table preallocation
- Discuss “robust query optimization”

Impact of Mis-estimates

FK/PK indexes

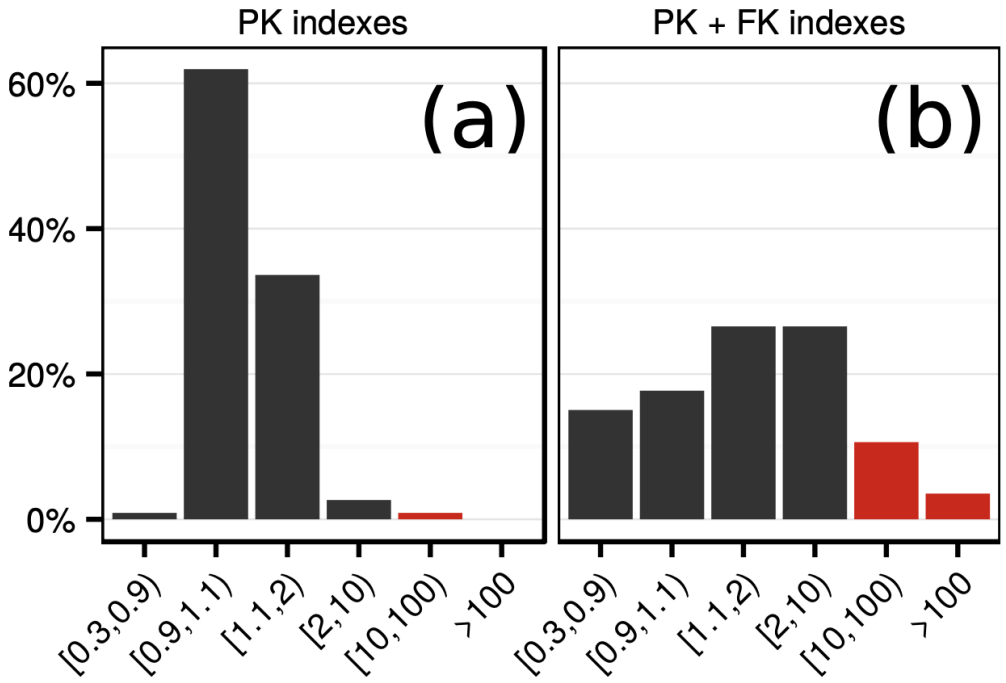


Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)

Impact of Mis-estimates

FK/PK indexes

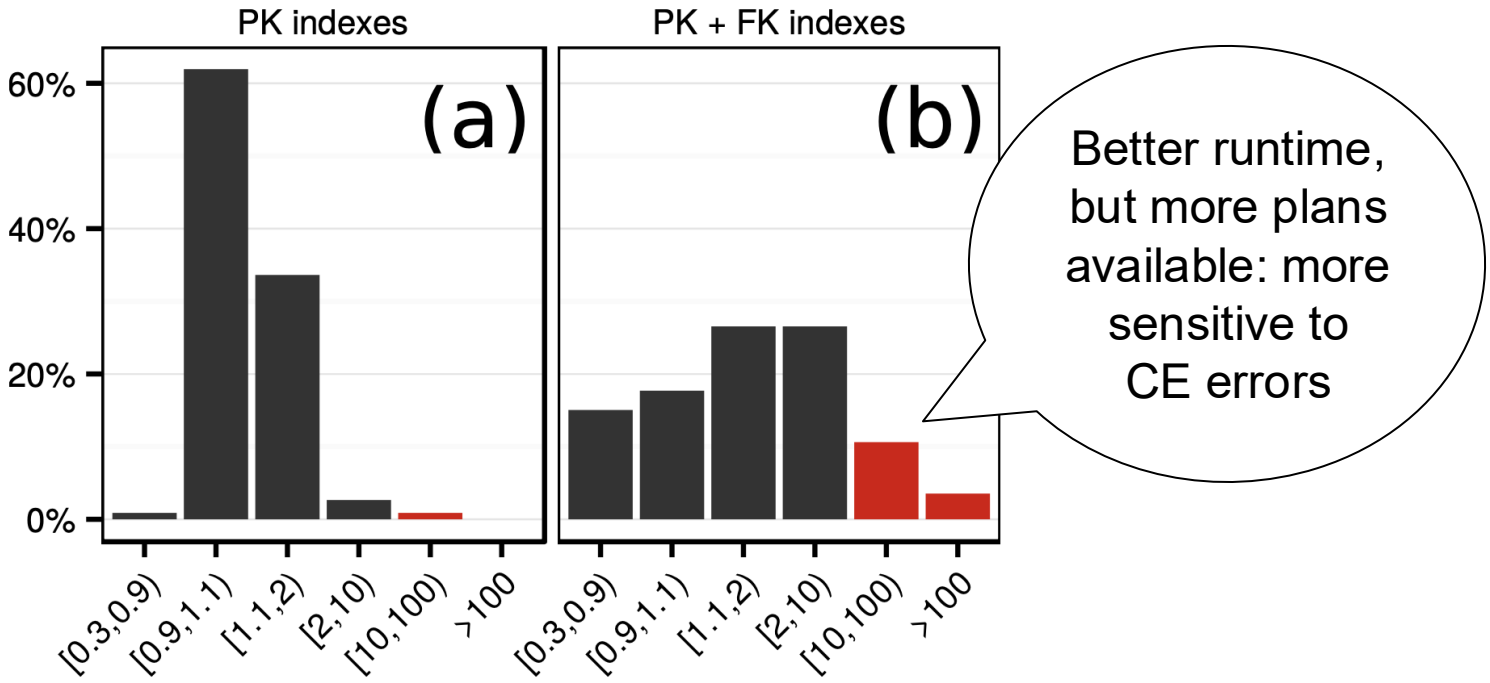


Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)

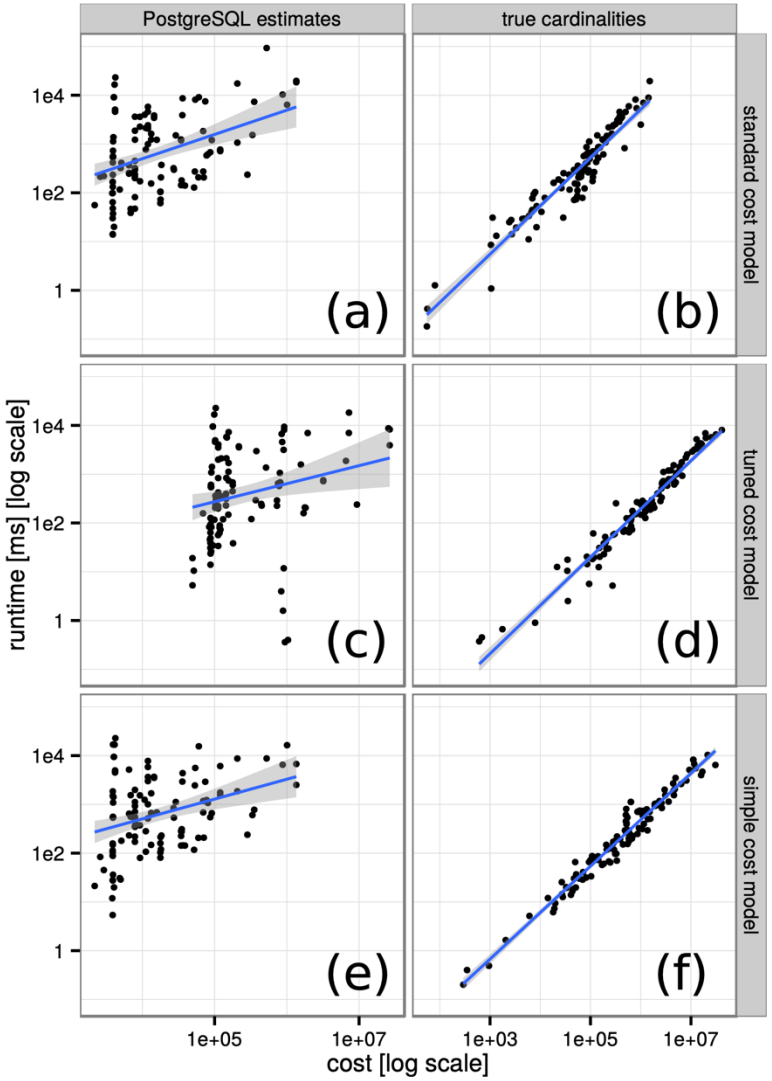
Discussion

- When PK indexes only, optimizer chooses a good plan anyway; impact of CE is limited; confirmed by others too

- When indexes on PK+FK, performance improves, but sensitivity to CE higher

[How good are they]

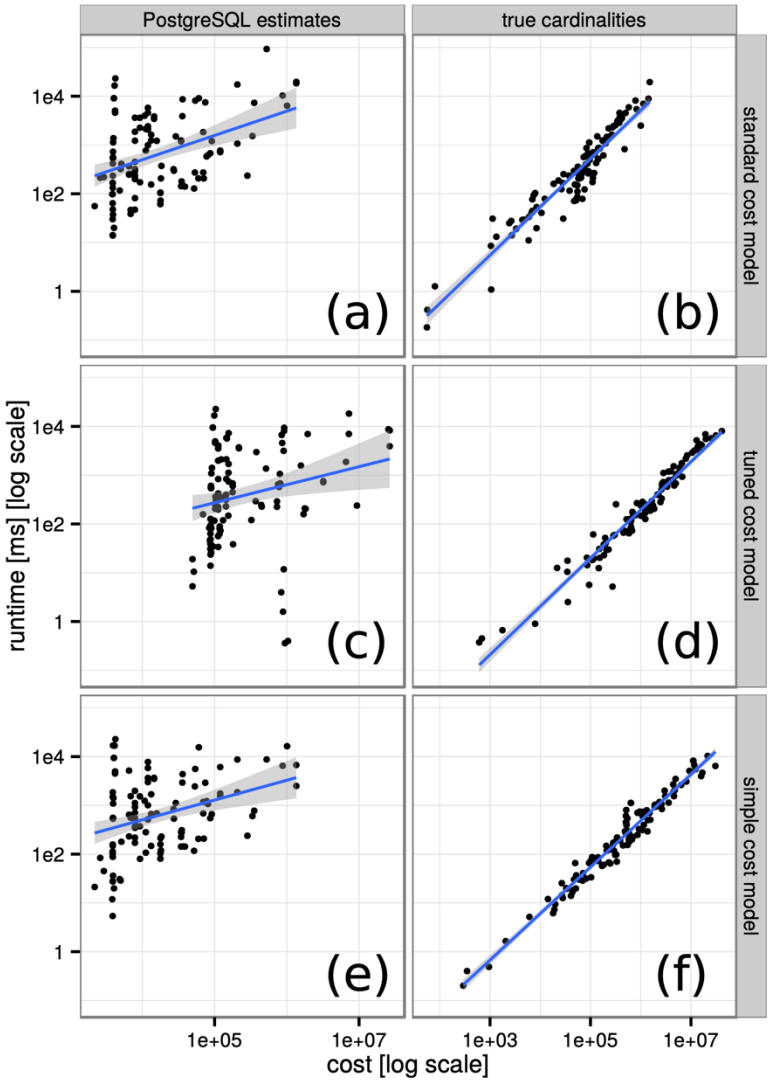
Cardinalities to Cost



[How good are they]

Cardinalities to Cost

Postgres cost

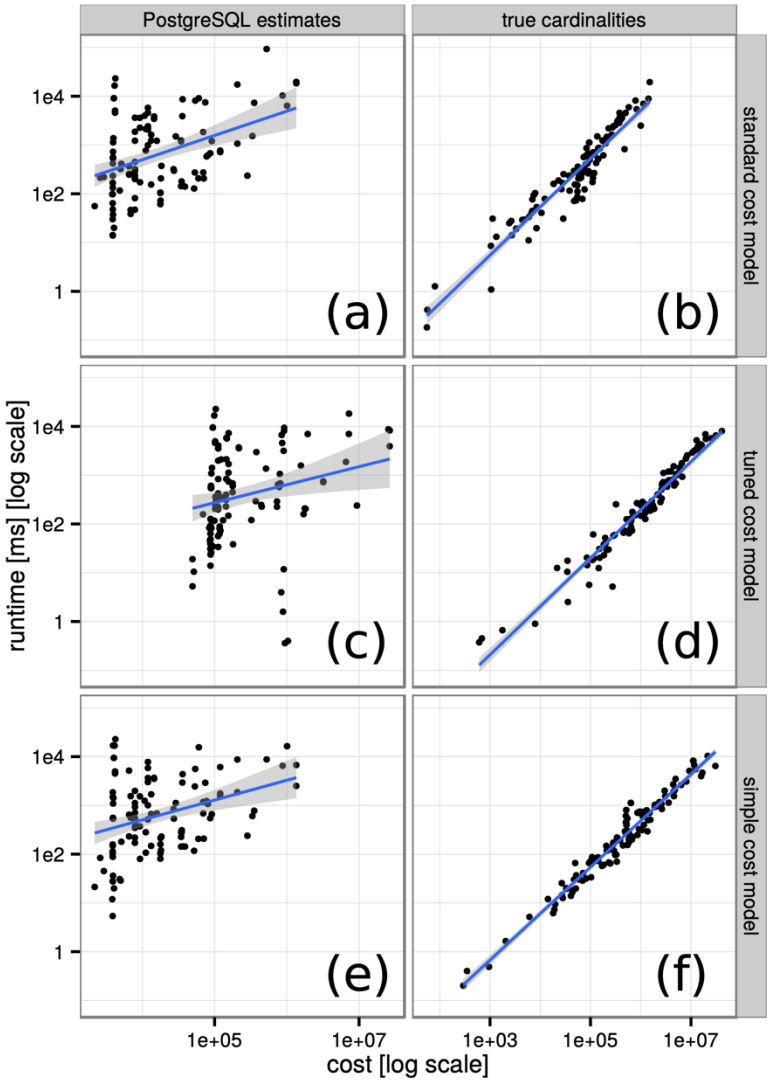


[How good are they]

Cardinalities to Cost

Postgres cost

No I/O, keep only CPU



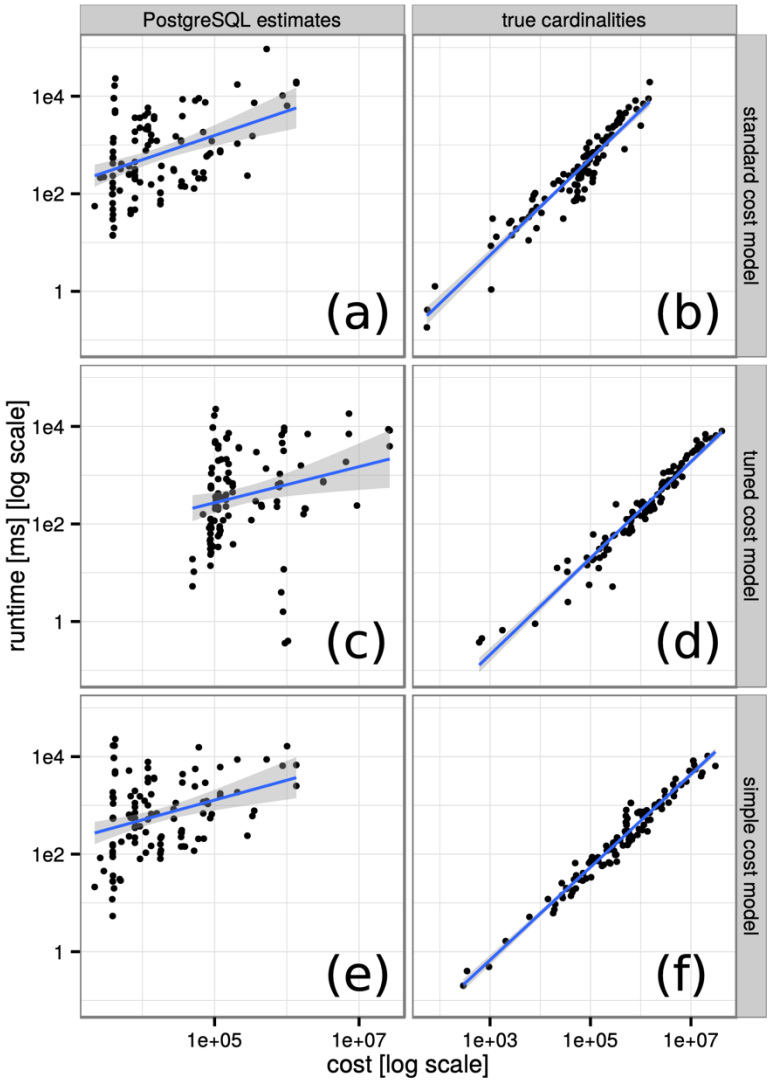
[How good are they]

Cardinalities to Cost

Postgres cost

No I/O, keep only CPU

Their own simple formula



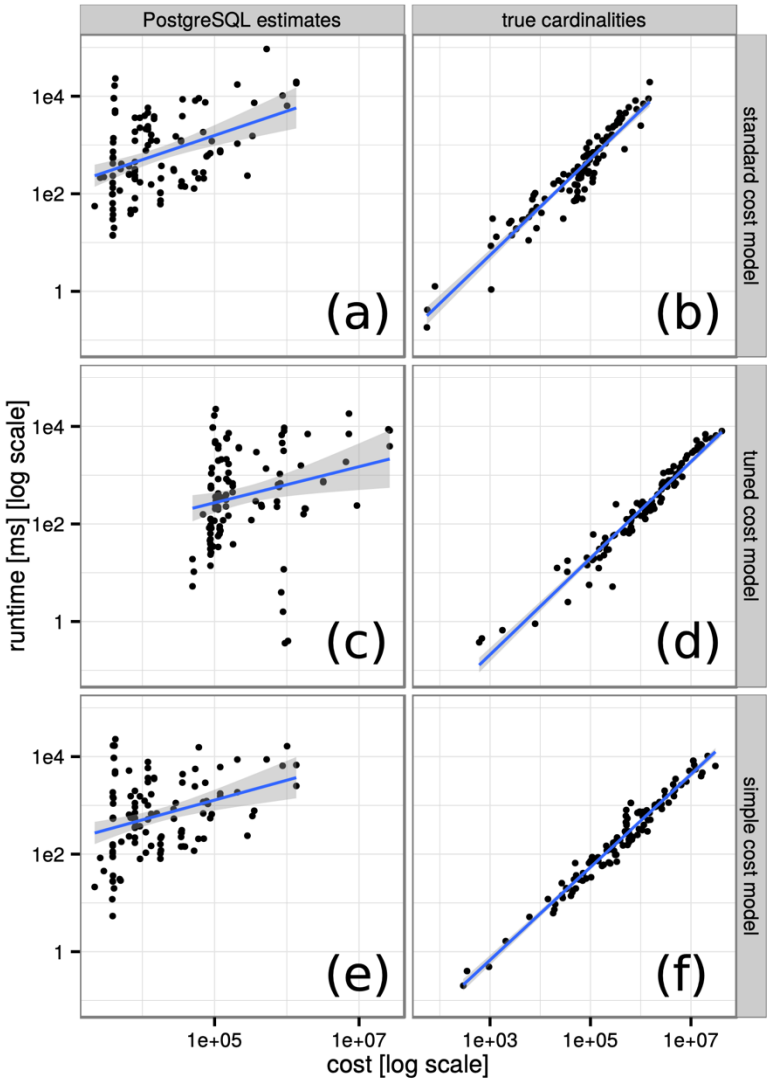
Cardinalities to Cost

- CE accounts for largest errors
- Cost models: both simple or complex are fine

Postgres cost

No I/O, keep only CPU

Their own simple formula



Importance of the Plan Space

- Do we need to explore a large space, or should we pick a plan at random?
- Do we need bushy trees, or are left-, or right-, or zigzag-trees enough?
- Do we need dynamic programming, or is greedy enough?

[How good are they]

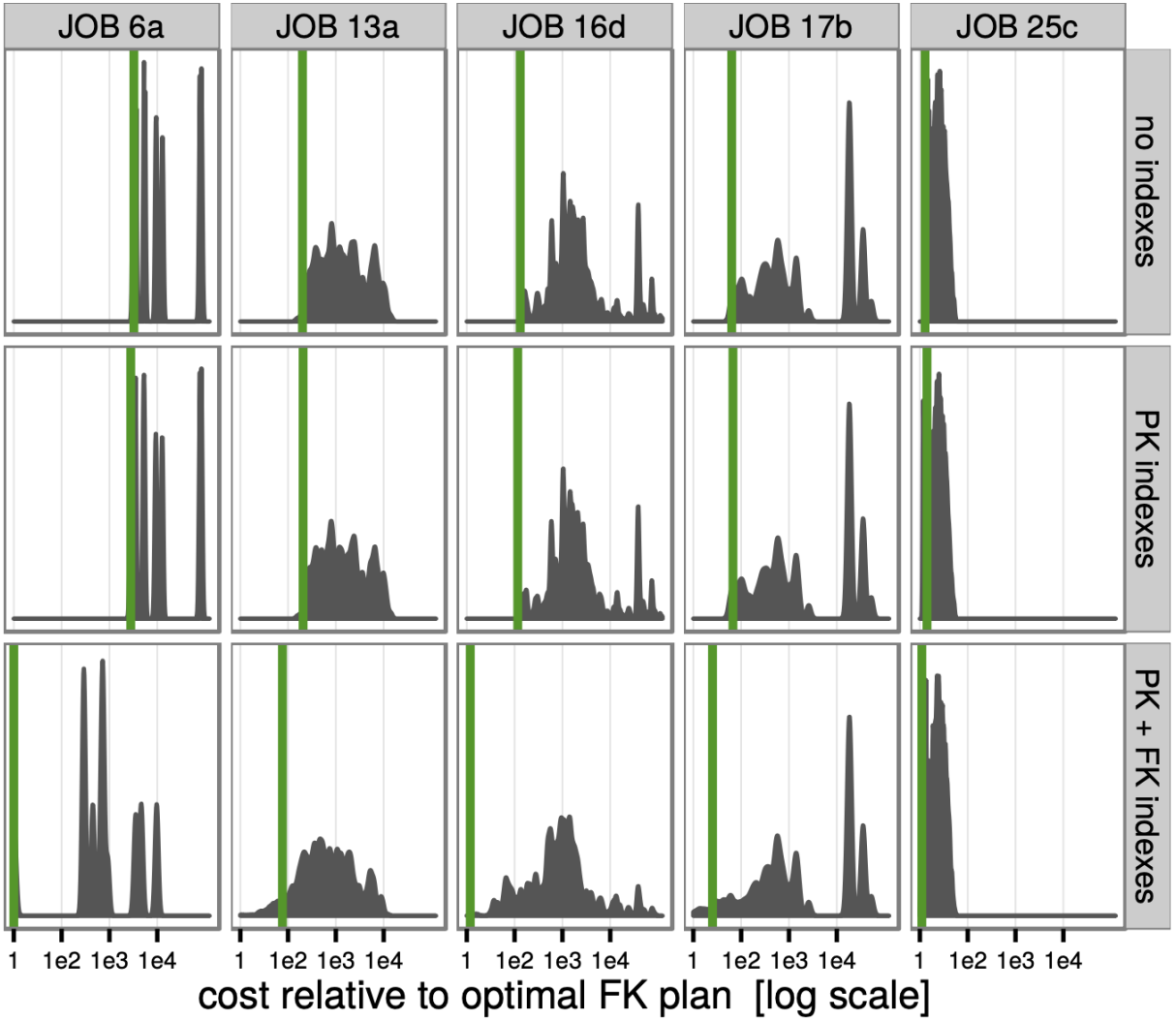


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

[How good are they]

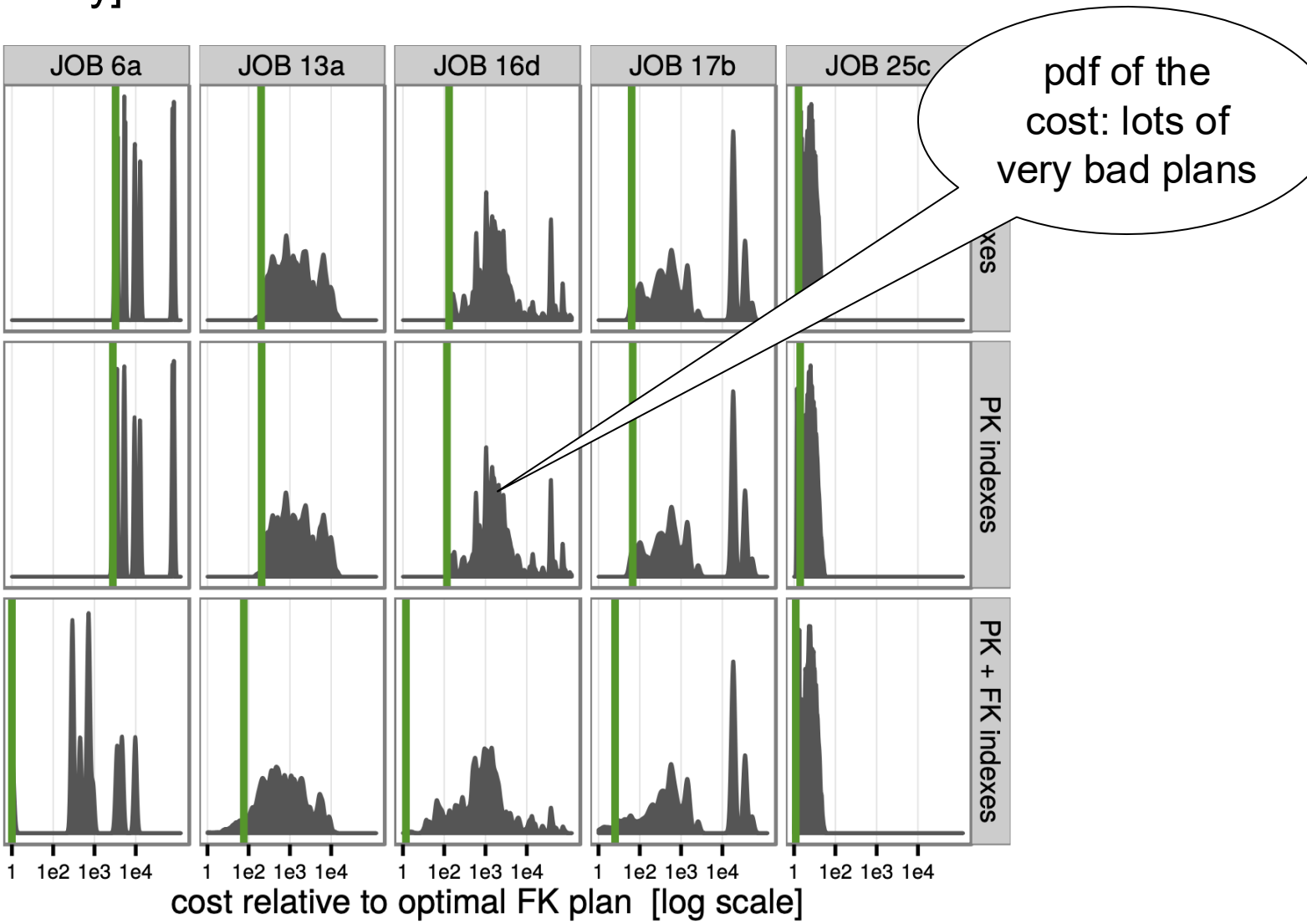


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

[How good are they]

	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)

[How good are they]

Generally, not much worse than optimal…

	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

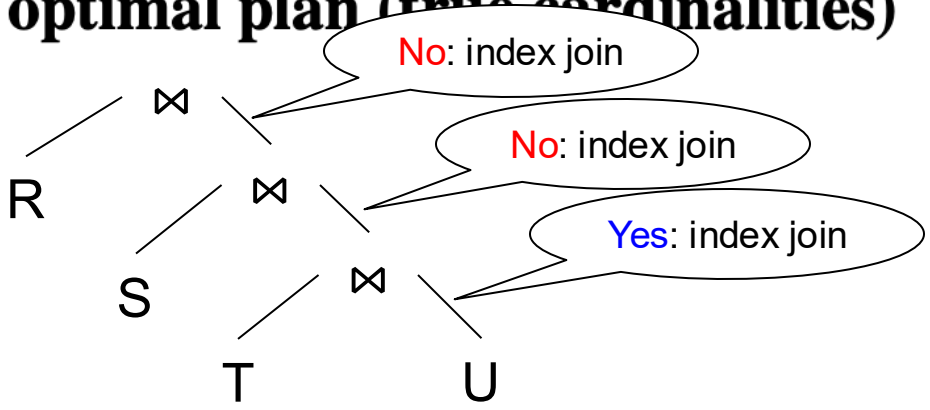
Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

[How good are they]

Generally, not much worse than optimal…

	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)



…except here. Right-deep plans prevent index joins.

[How good are they]

	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

Final Comments

- Paper had wide impact; clear description of current query engines and their limitations. Test of Time Award.
- Results confirmed by others (e.g. Microsoft's SQL Server)
- JOB Benchmark also widely influential, but limited to single block SQL; now: SQLStorm.

Yannakakis' Algorithm

Motivation

- What is the time complexity to compute these queries using a typical plan?
  - Q(X,Y,Z) = R(X,Y), S(Y,Z)
  - Q(X,Y,Z,U,V)=R(X,Y),S(Y,Z),T(Z,U),K(U,V)

- If |R| = |S| = … = N, then the time for the queries above is $O(N^2)$, and $O(N^4)$

Motivation

- Yannakakis' algorithm computes any acyclic query in time O(N+OUT), where OUT is the size of the output

- There are two key ideas:
  - It works only for acyclic queries
  - It performs a semi-join reduction before computing the joins

Semi-Join Reduction

Semi-join definition:

$$R \ltimes S = \Pi_{attr(R)}(R \bowtie S)$$

Semi-Join Reduction

Semi-join definition:

$$R \ltimes S = \Pi_{attr(R)}(R \bowtie S)$$

Basic law:

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

Example 1

- Example:

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

Example 1

- Example:

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

- A semijoin reducer is:

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

Example 1

- Example:

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

- A semijoin reducer is:

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

- The rewritten query is:

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

Example 2

$$Q(y,z,u) = R(\text{‘a’}, y), S(y,z), T(z,u), K(u,\text{’b’})$$

Semi-join reducer:

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)
K’(u) :- K(u,’b’) ⋉ T’(z,u)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)
K’(u) :- K(u,’b’) ⋉ T’(z,u)
T’’(z,u) :- T’(z,u) ⋉ K’(u)

Example 2

Q(y,z,u) = R(‘a’, y), S(y,z), T(z,u), K(u,’b’)

Semi-join reducer:

S’(y,z) :- S(y,z) ⋉ R(‘a’, y)
T’(z,u) :- T(z,u) ⋉ S’(y,z)
K’(u) :- K(u,’b’) ⋉ T’(z,u)
T’’(z,u) :- T’(z,u) ⋉ K’(u)
S’’(y,z) :- S’(y,z) ⋉ T’’(z,u)
R’’(y) :- R(‘a’,y) ⋉ S’’(y,z)

Example 2

$Q(y,z,u) = R(\text{‘a’}, y), S(y,z), T(z,u), K(u,\text{’b’})$

Semi-join reducer:

$S'(y,z) :- S(y,z) \ltimes R(\text{‘a’}, y)$
$T'(z,u) :- T(z,u) \ltimes S'(y,z)$
$K'(u) :- K(u,\text{’b’}) \ltimes T'(z,u)$
$T''(z,u) :- T'(z,u) \ltimes K'(u)$
$S''(y,z) :- S'(y,z) \ltimes T''(z,u)$
$R''(y) :- R(\text{‘a’},y) \ltimes S''(y,z)$

Reduced query:

$Q(y,z,u) = R''(y), S''(y,z), T''(z,u), K''(u)$

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

T is called *join tree*

Acyclic Queries

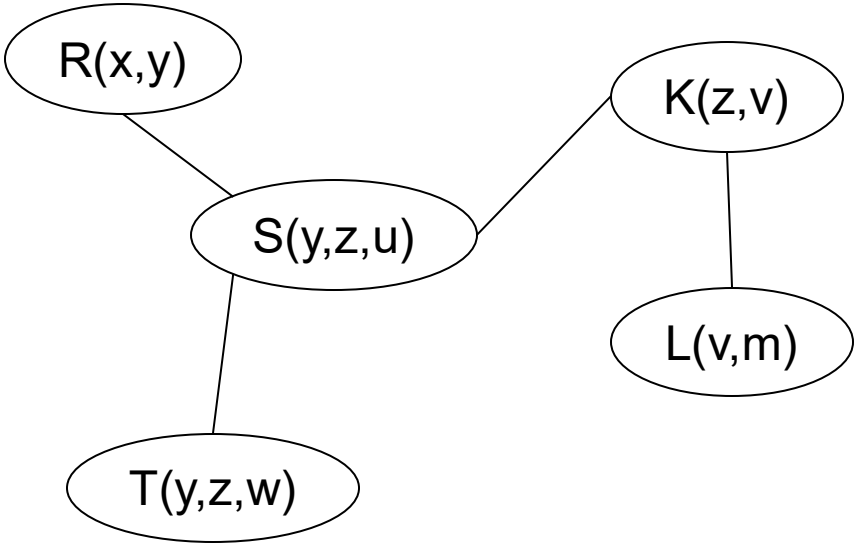
Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

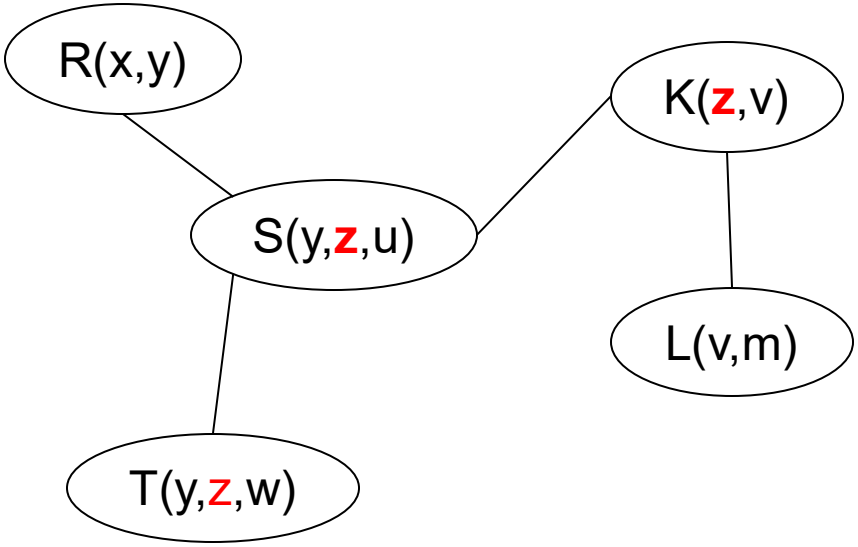


R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



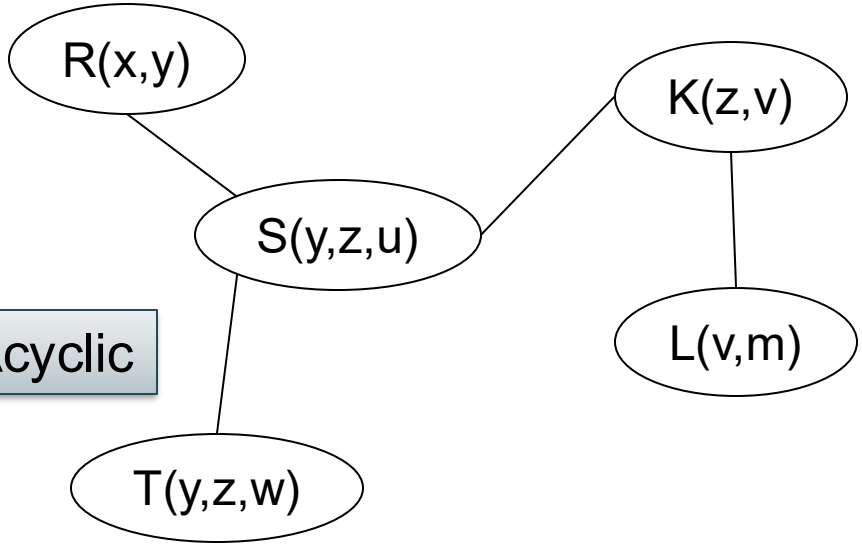
E.g. **z** forms a connected component

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



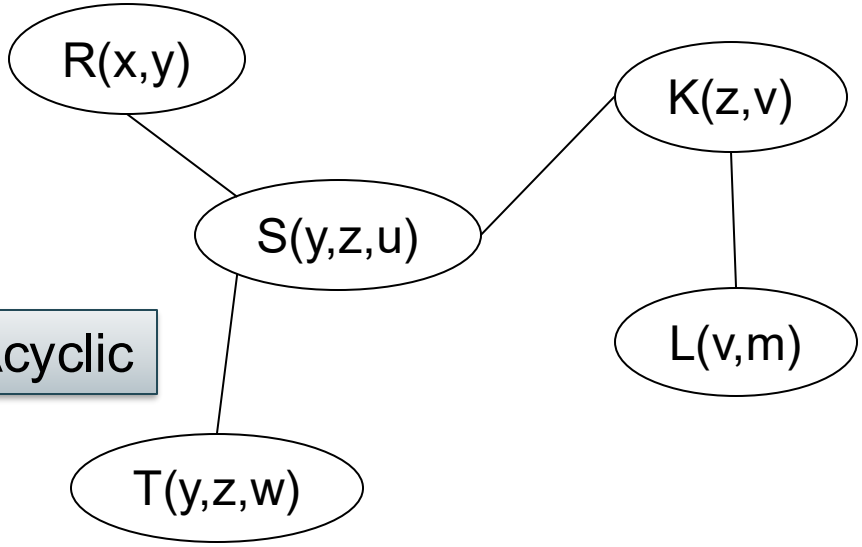
Acyclic

R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component



Acyclic

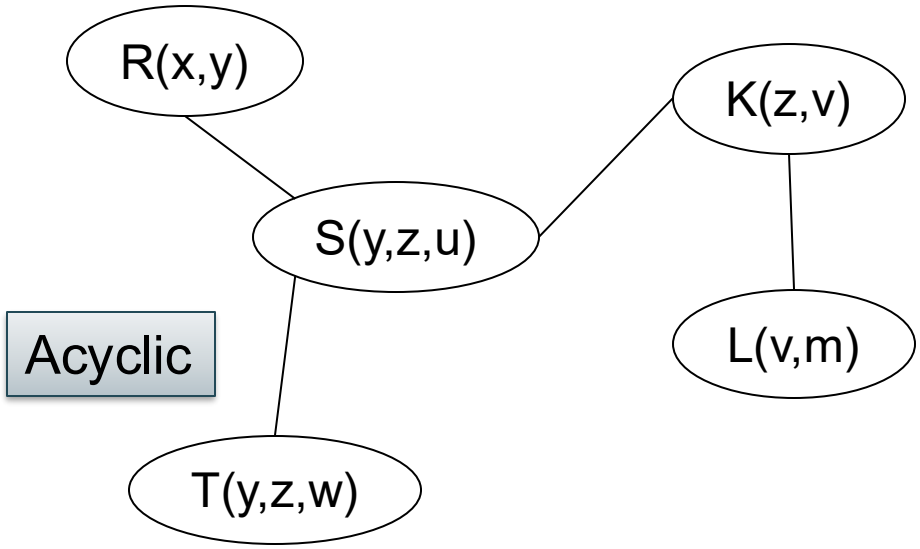
R(x,y),S(y,z),T(z,x)

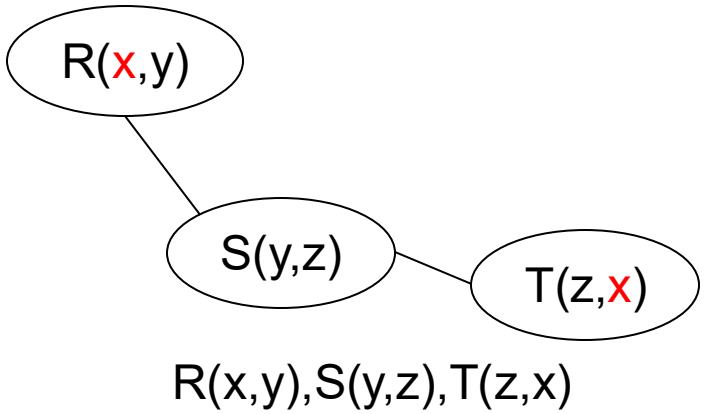
R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

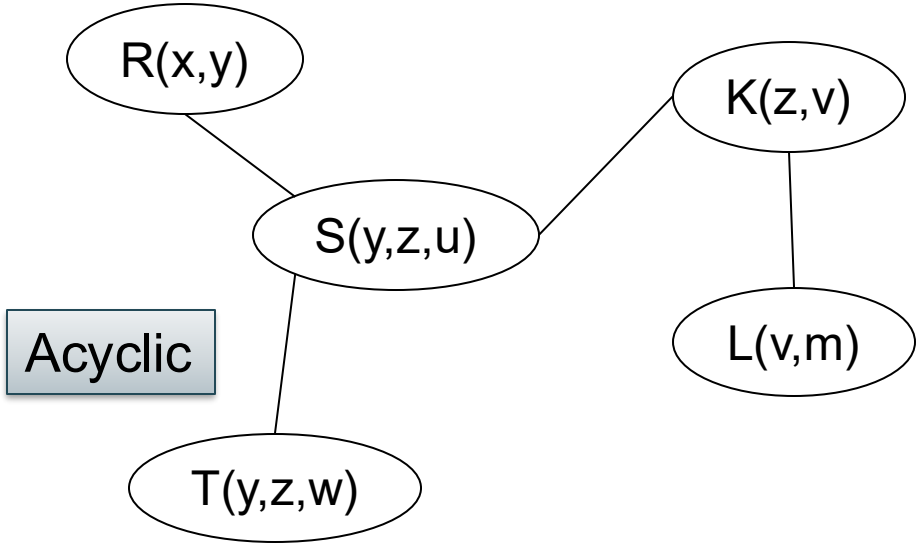


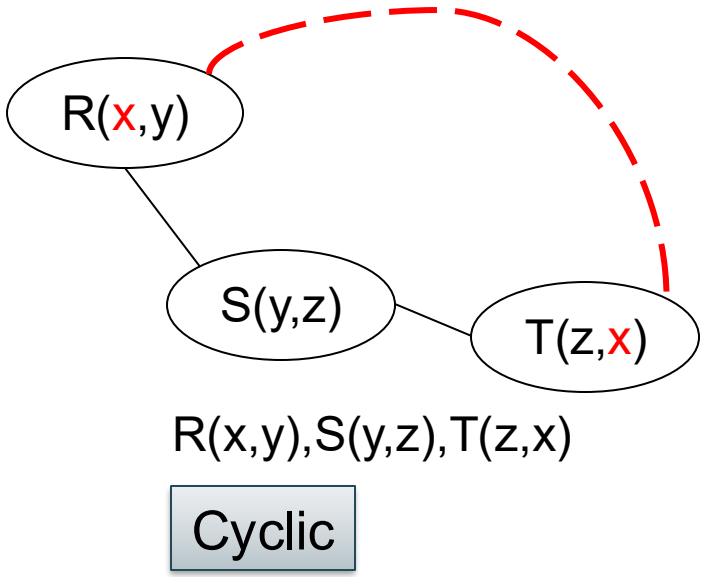


T is called *join tree*

Acyclic Queries

Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component





A Theorem

Q = an acyclic query Q that is:

- Boolean, or
- Full, or
- Aggregate with $\leq$1 group-by variable

Theorem Q can be computed in time*:

$$\tilde{O}(|Input| + |Output|)$$

* $\tilde{O}$ means *plus a logarithmic factor (for sorting)*

Yannakakis Algorithm

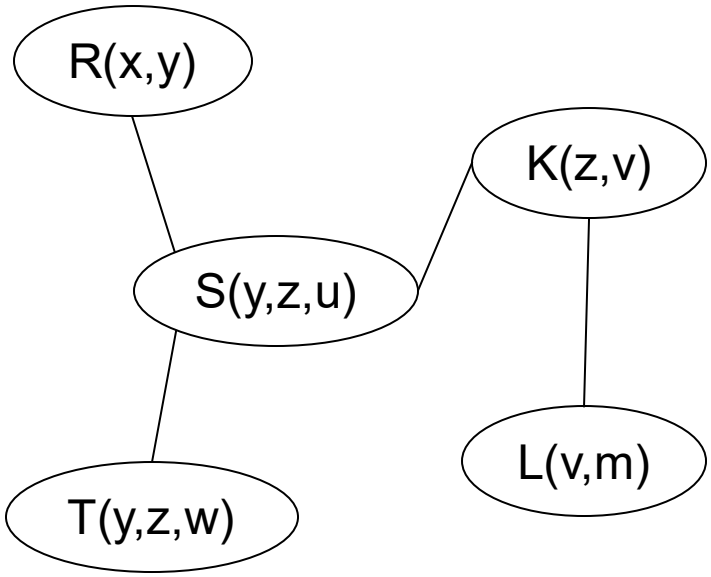
- Step 1: semi-join reduction
  - Pick any root node in the join tree of Q
  - Semi-join reduction from leaves to root
  - Semi-join reduction from root to leaves
- Step 2:
  - Compute the joins bottom up,
  - Push group-by down

Example: Full CQ

$$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$$

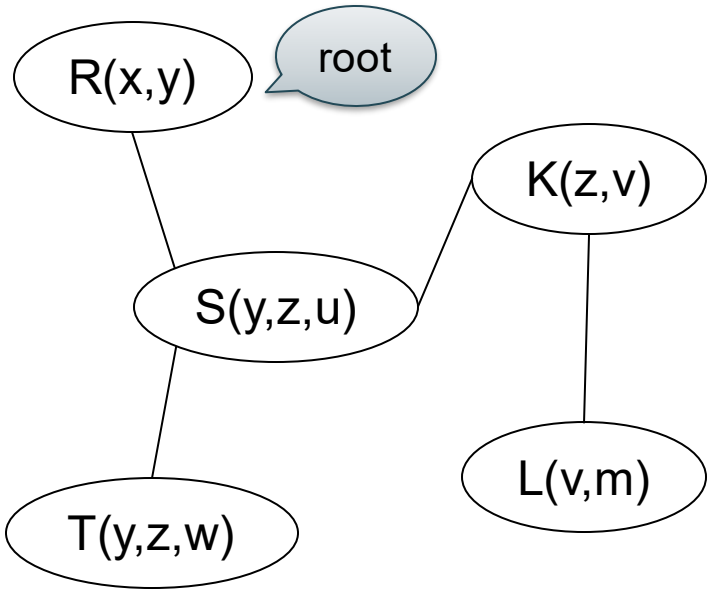
Example: Full CQ

$$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$$



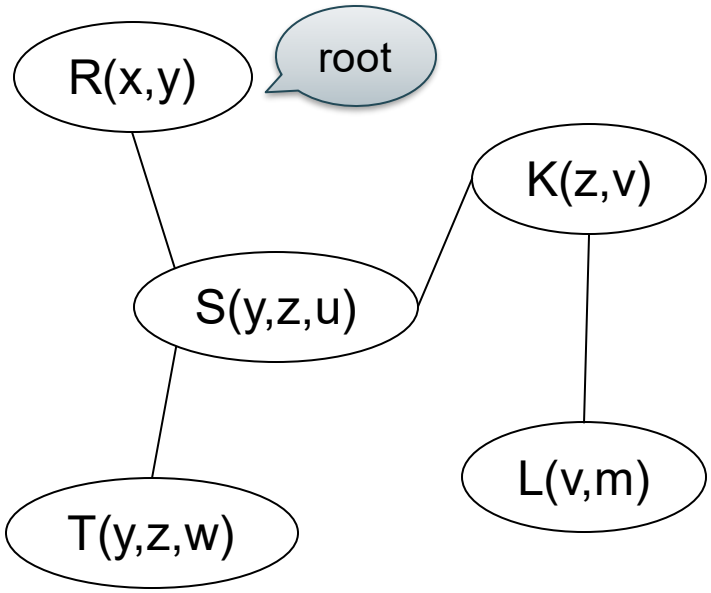
Example: Full CQ

$$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$$



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

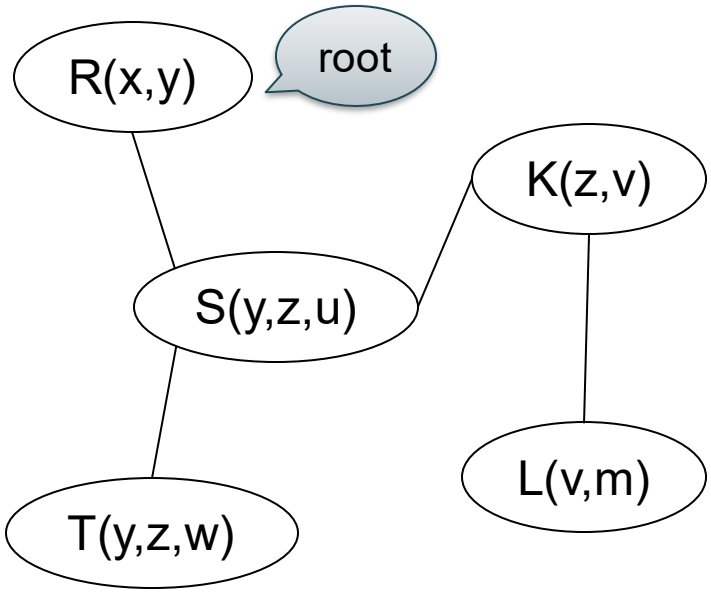


-- Leaves to root:

$K :- K \ltimes L$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



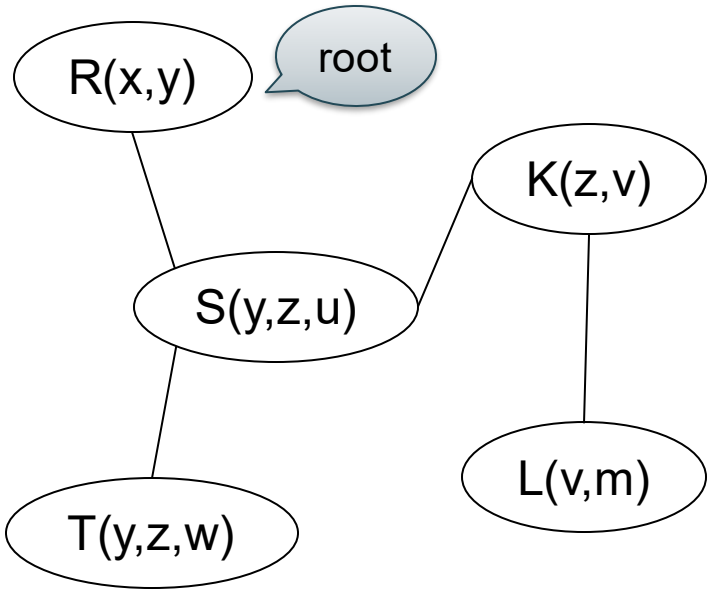
-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

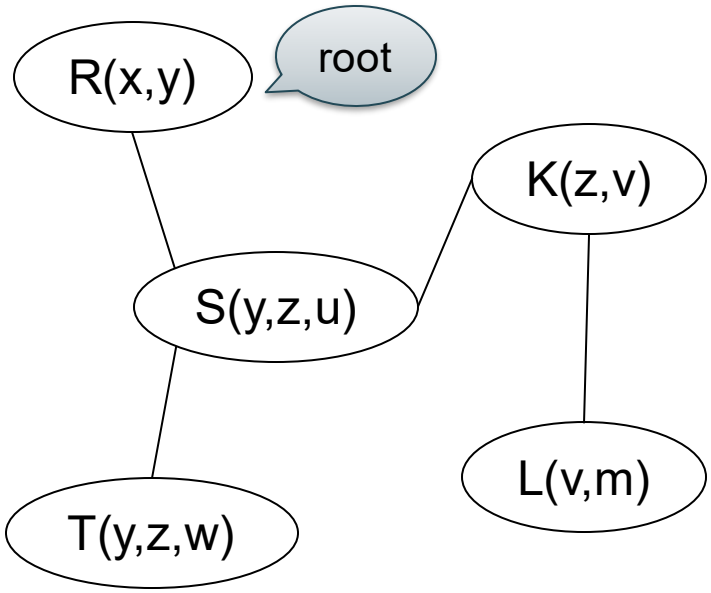
$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

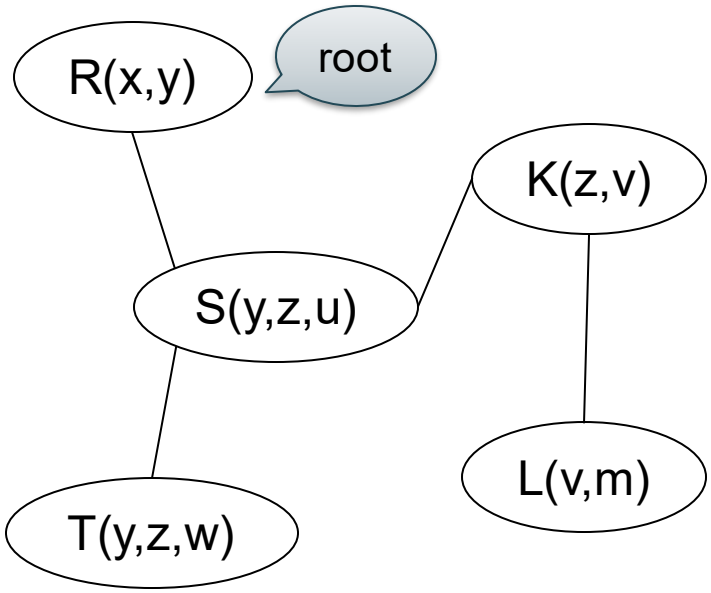
$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

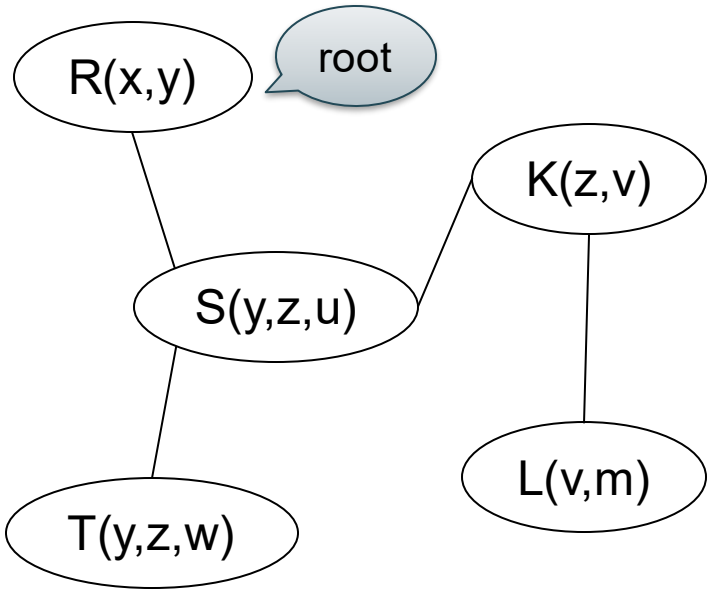
$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

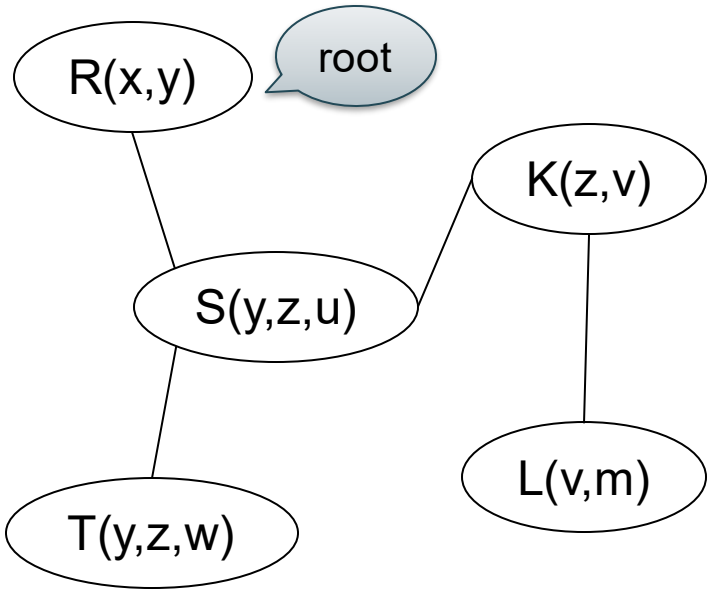
$R :- R \ltimes S$

-- Root to leaves:

$S :- S \ltimes R$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

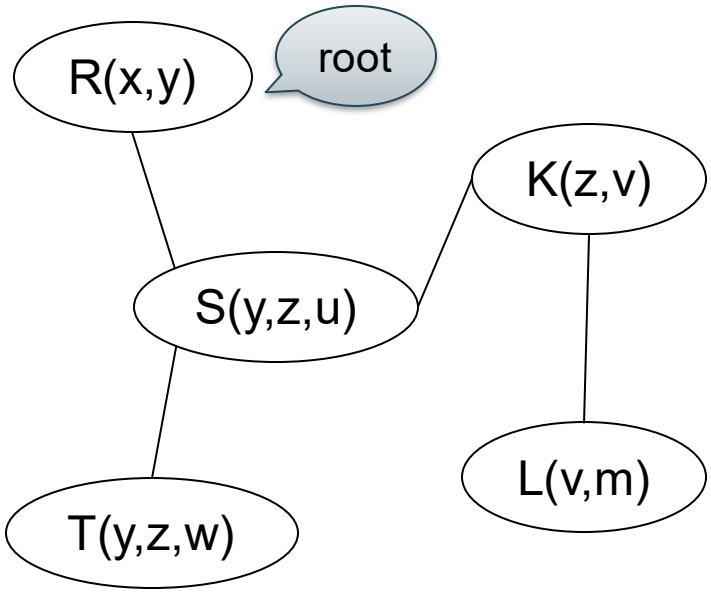
-- Root to leaves:

$S :- S \ltimes R$

$T :- T \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

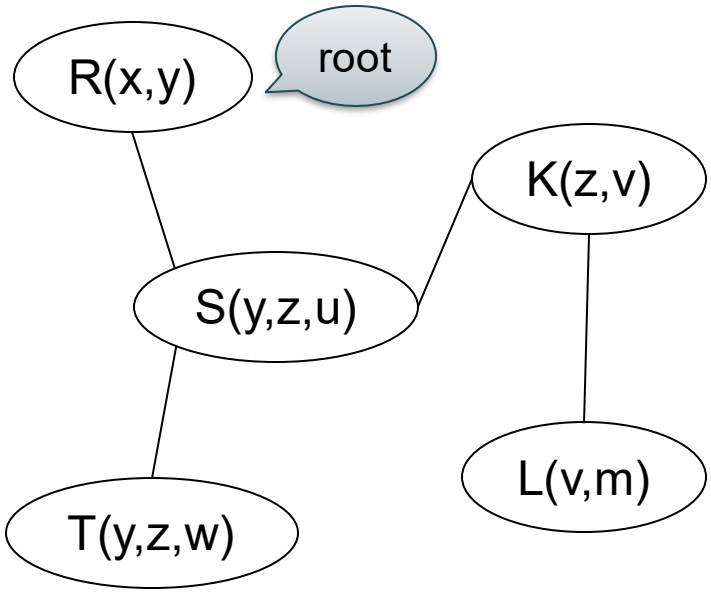
$S :- S \ltimes R$

$T :- T \ltimes S$

$K :- K \ltimes S$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

$S :- S \ltimes R$

$T :- T \ltimes S$

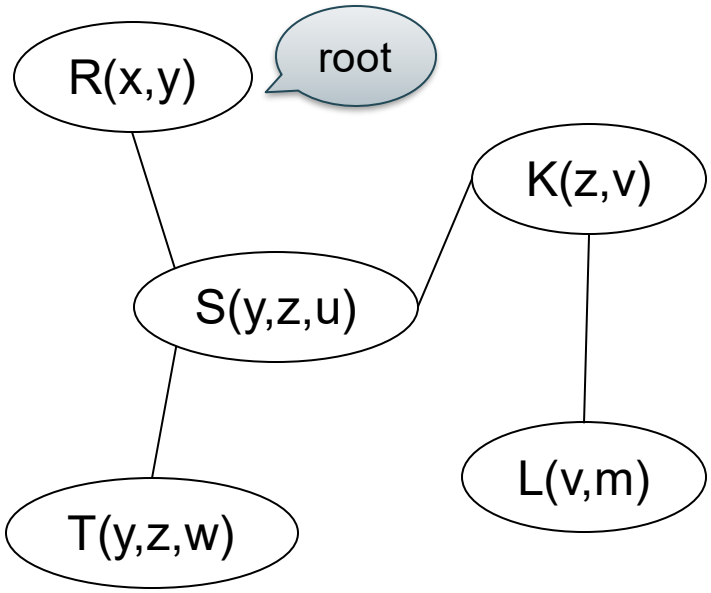
$K :- K \ltimes S$

$L :- L \ltimes K$

Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

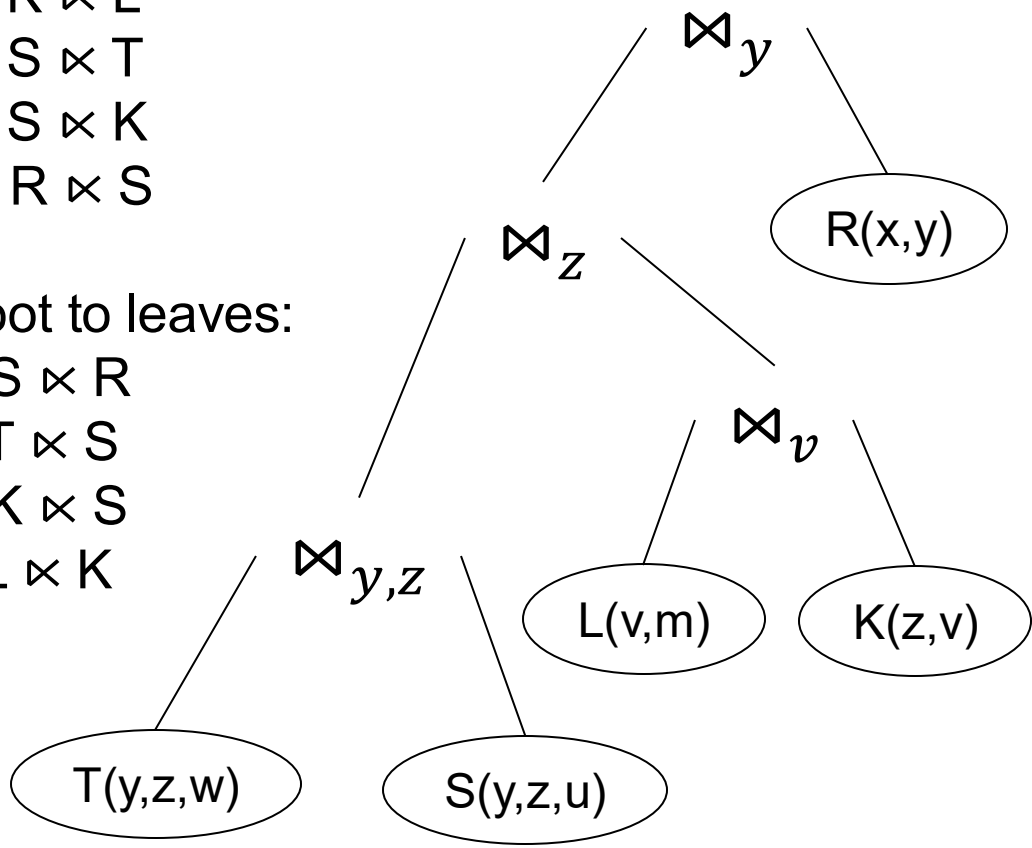
-- Root to leaves:

S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

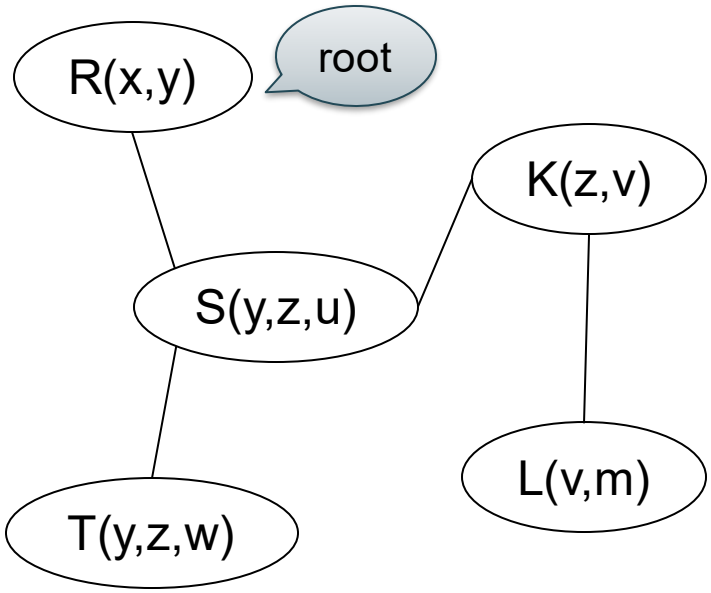
L :- L ⋉ K



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

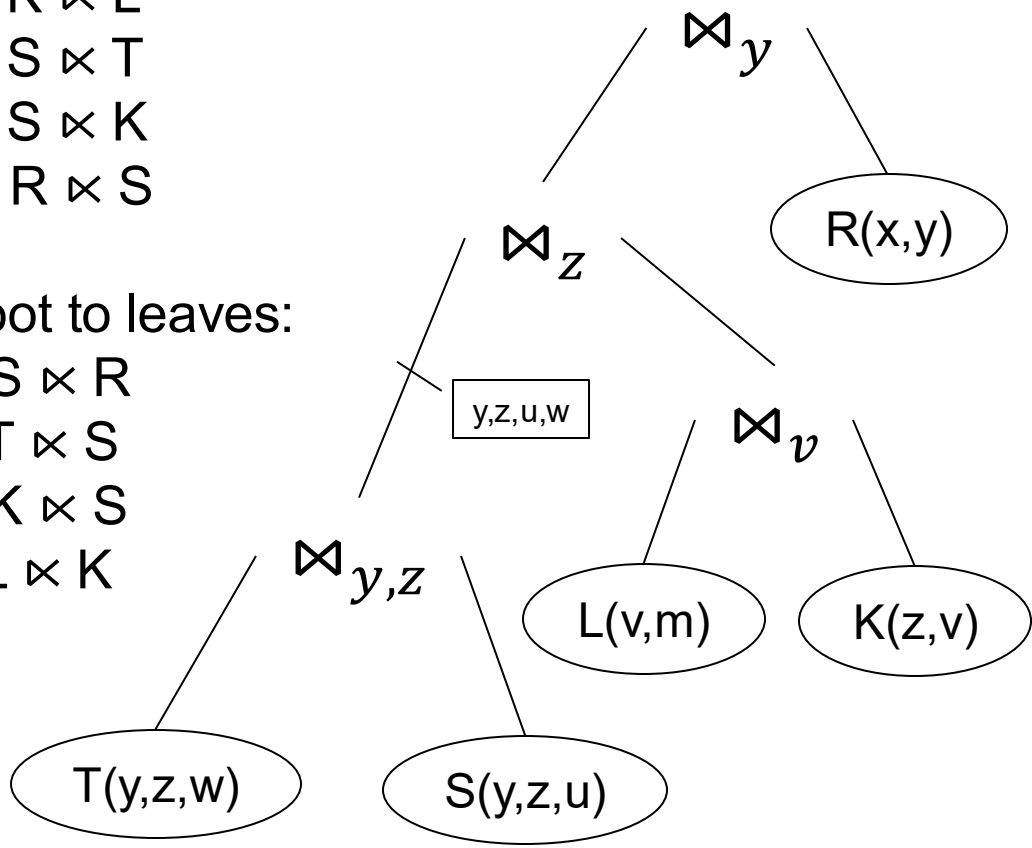
-- Root to leaves:

S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

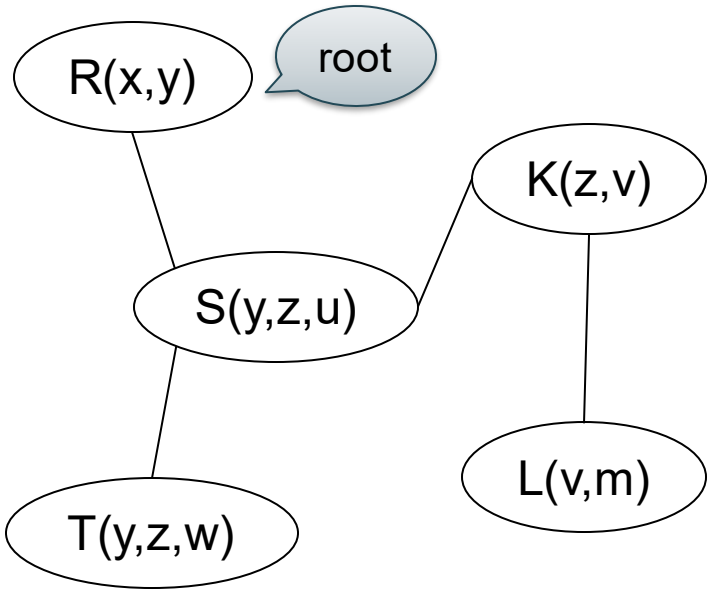
L :- L ⋉ K



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

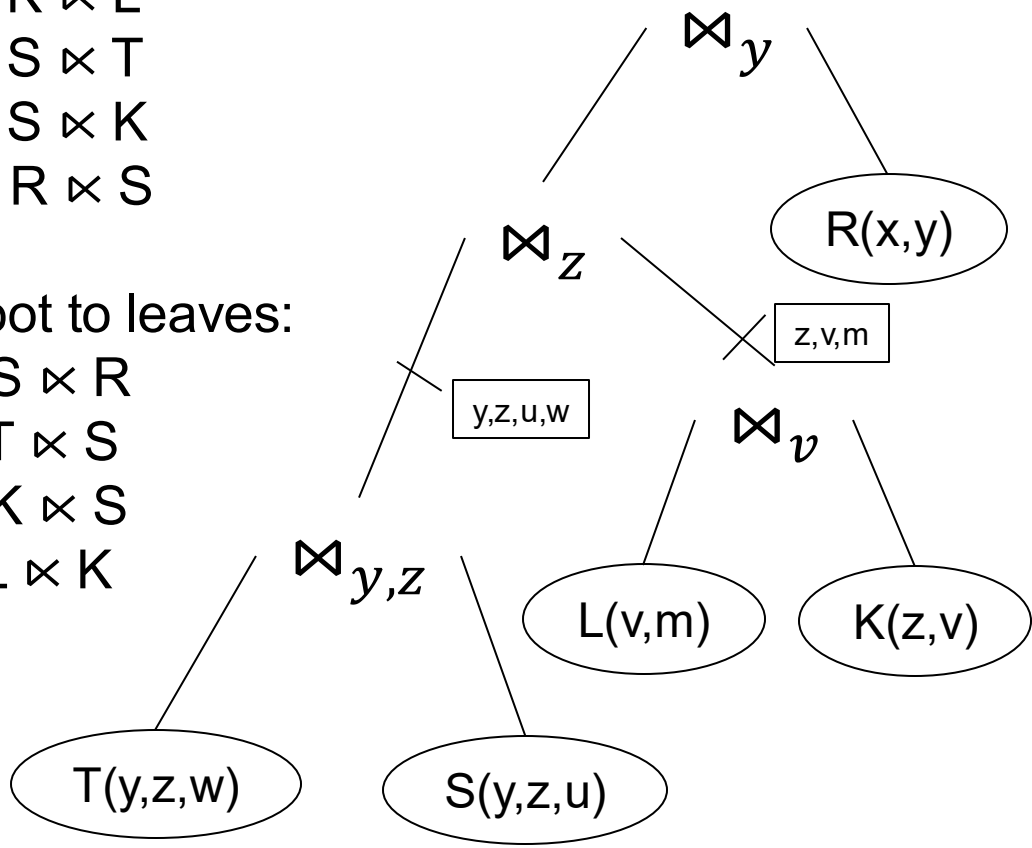
-- Root to leaves:

S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ S

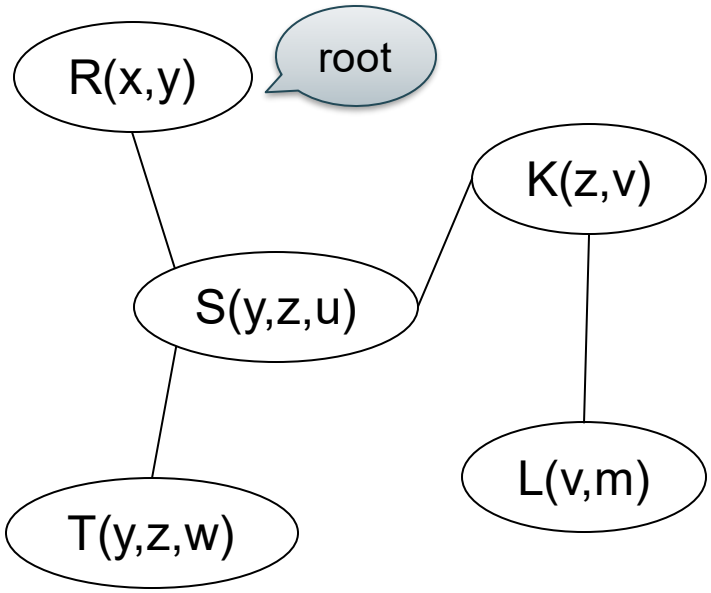
L :- L ⋉ K



Example: Full CQ

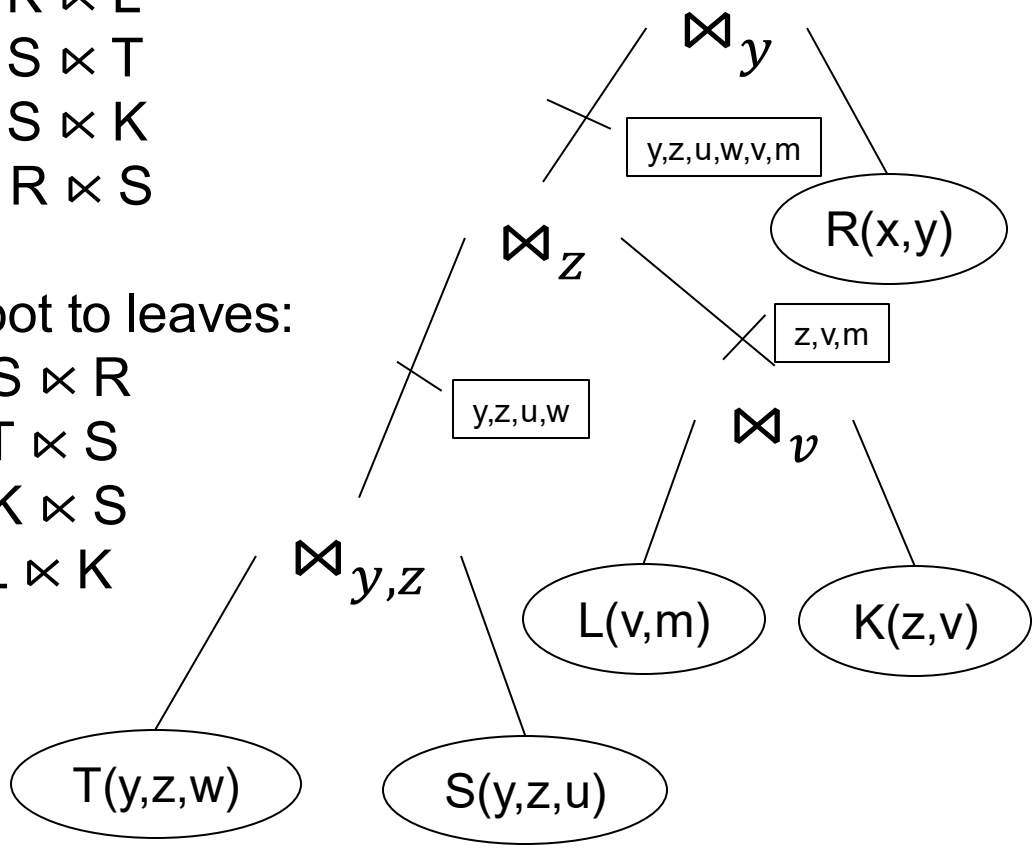
$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$

Join (any order in the tree)



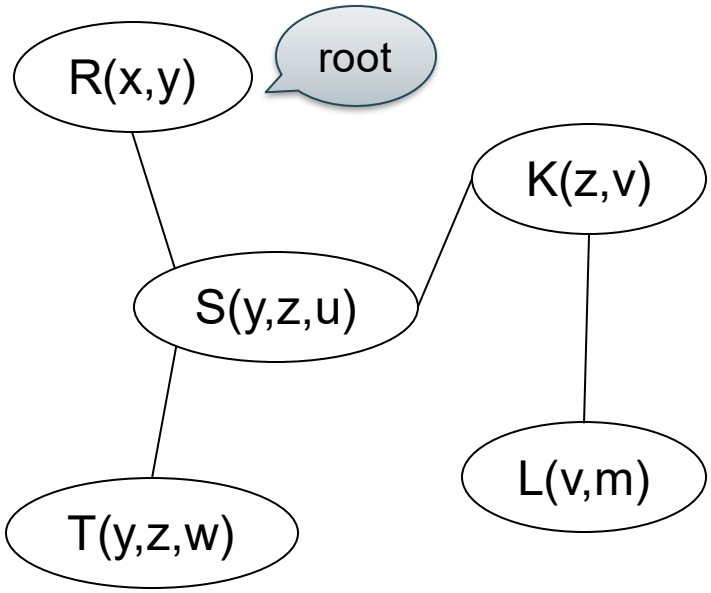
-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K



Example: Full CQ

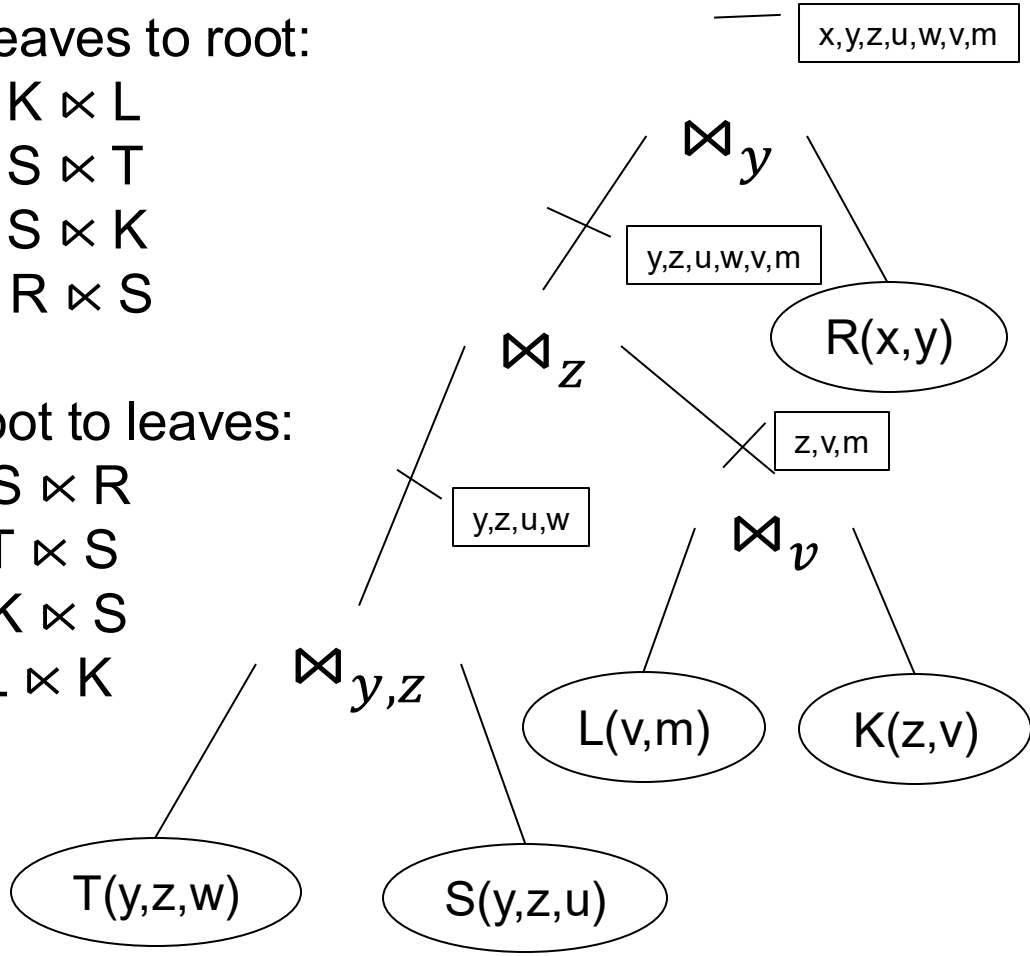
$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

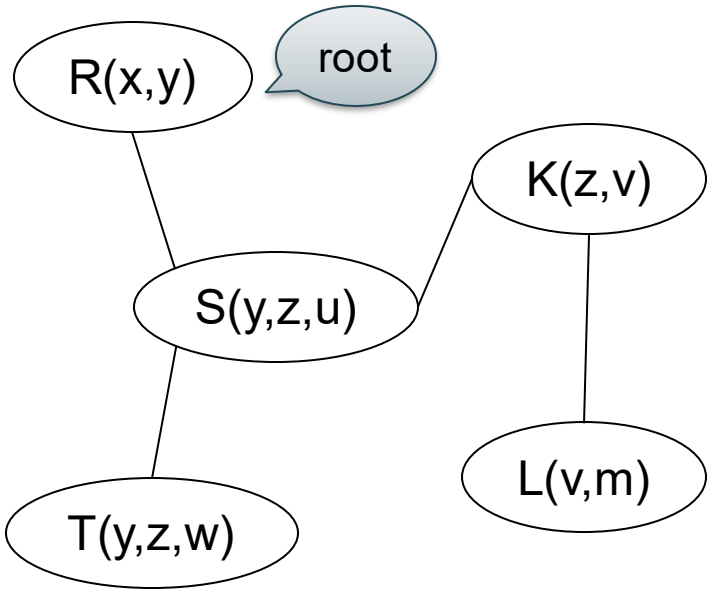
-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K

Join (any order in the tree)



Example: Full CQ

Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

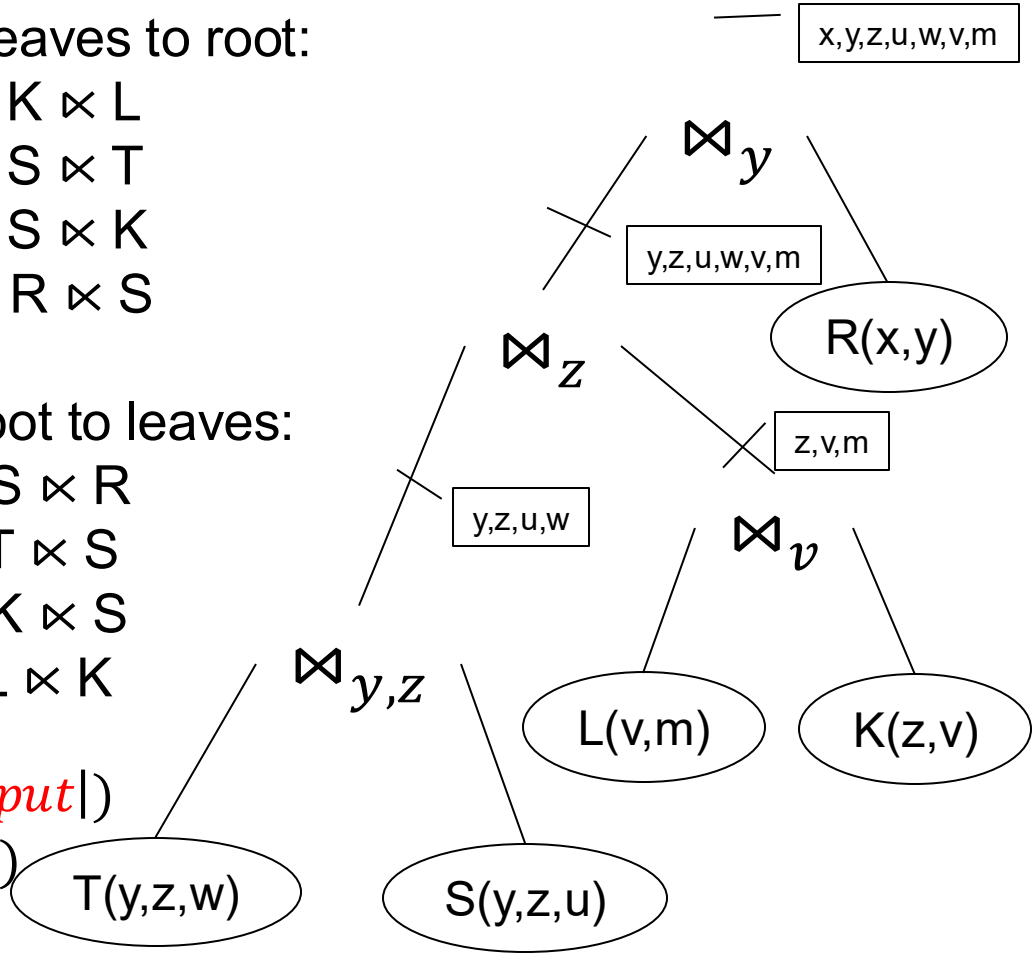


-- Leaves to root:
K :- K ⋉ L
S :- S ⋉ T
S :- S ⋉ K
R :- R ⋉ S

-- Root to leaves:
S :- S ⋉ R
T :- T ⋉ S
K :- K ⋉ S
L :- L ⋉ K

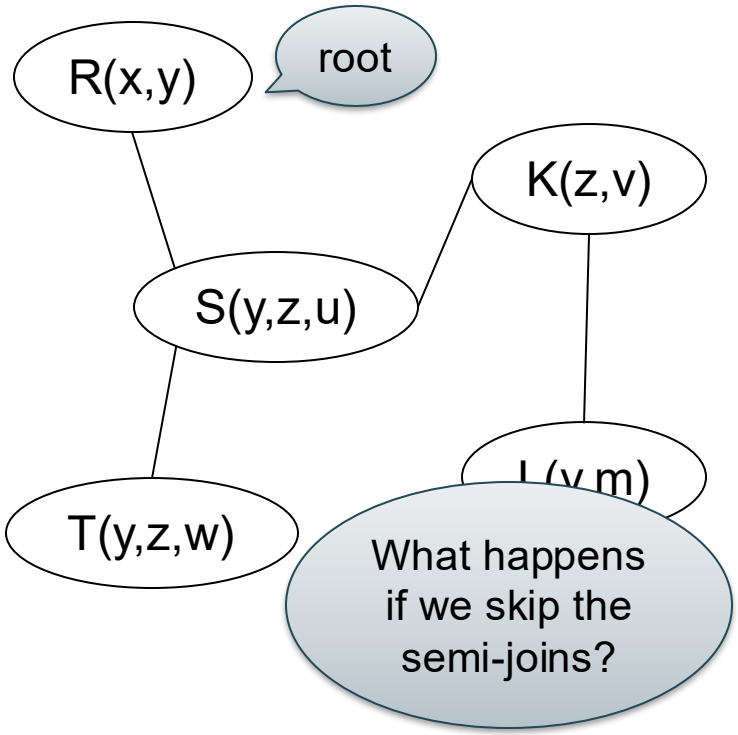
Runtime:
- Every semi-join takes time $\tilde{O}(|Input|)$
- Every join takes time $\tilde{O}(|Output|)$

Join (any order in the tree)



Example: Full CQ

$Q(*) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)$



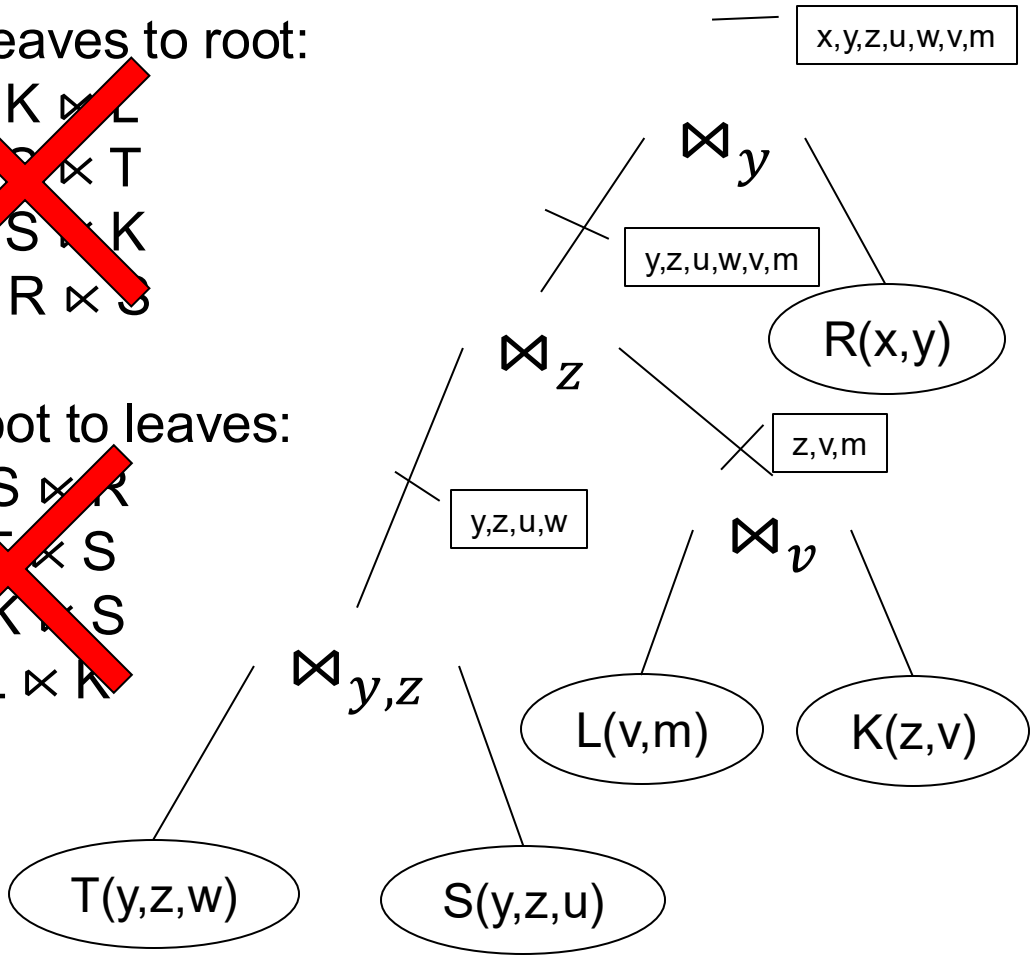
-- Leaves to root:

~~K :- K ⋉ L~~
~~S :- S ⋉ T~~
~~S :- S ⋉ K~~
~~R :- R ⋉ S~~

-- Root to leaves:

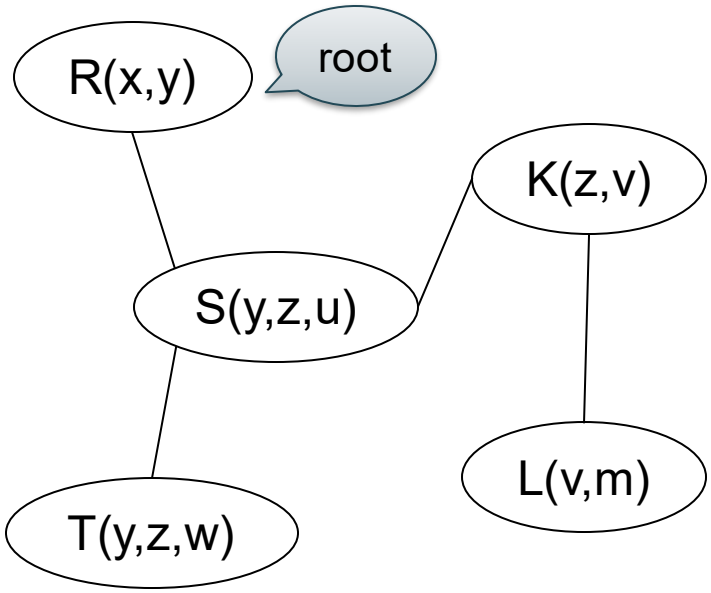
~~S :- S ⋉ R~~
~~T :- T ⋉ S~~
~~K :- K ⋉ S~~
~~L :- L ⋉ K~~

Join (any order in the tree)

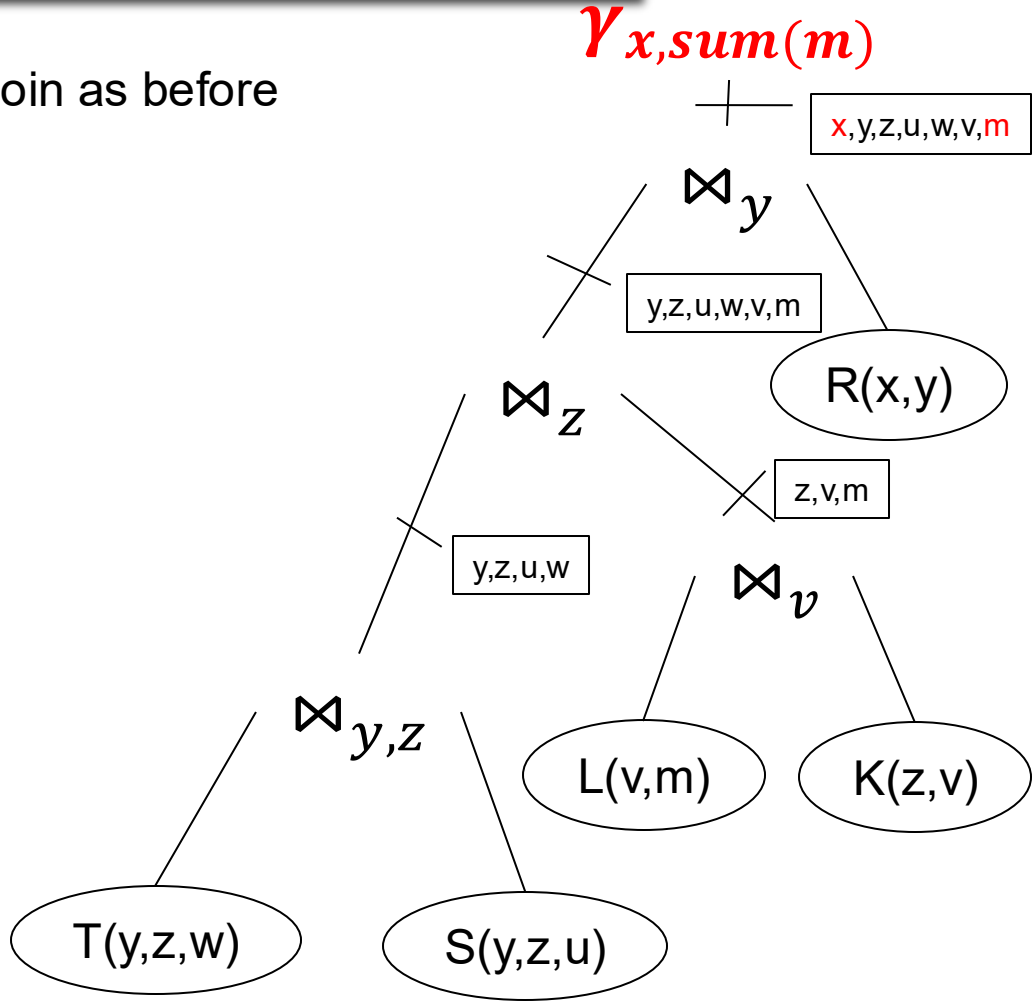


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

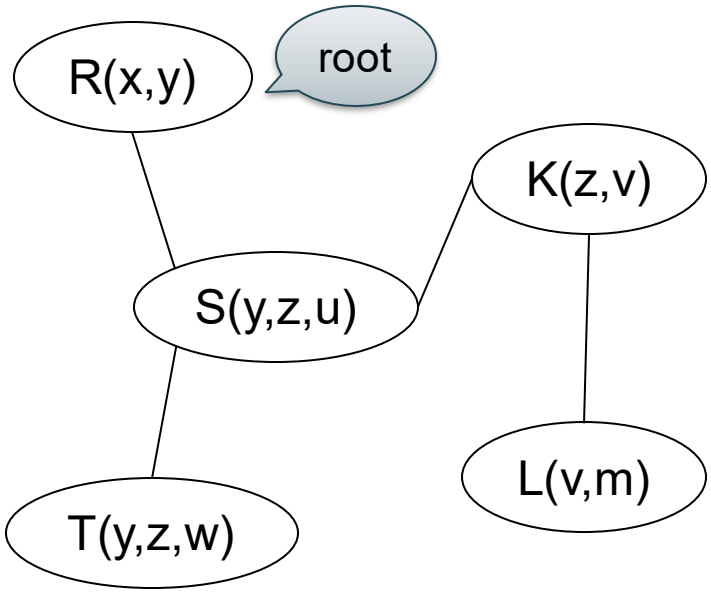


Semi-join as before

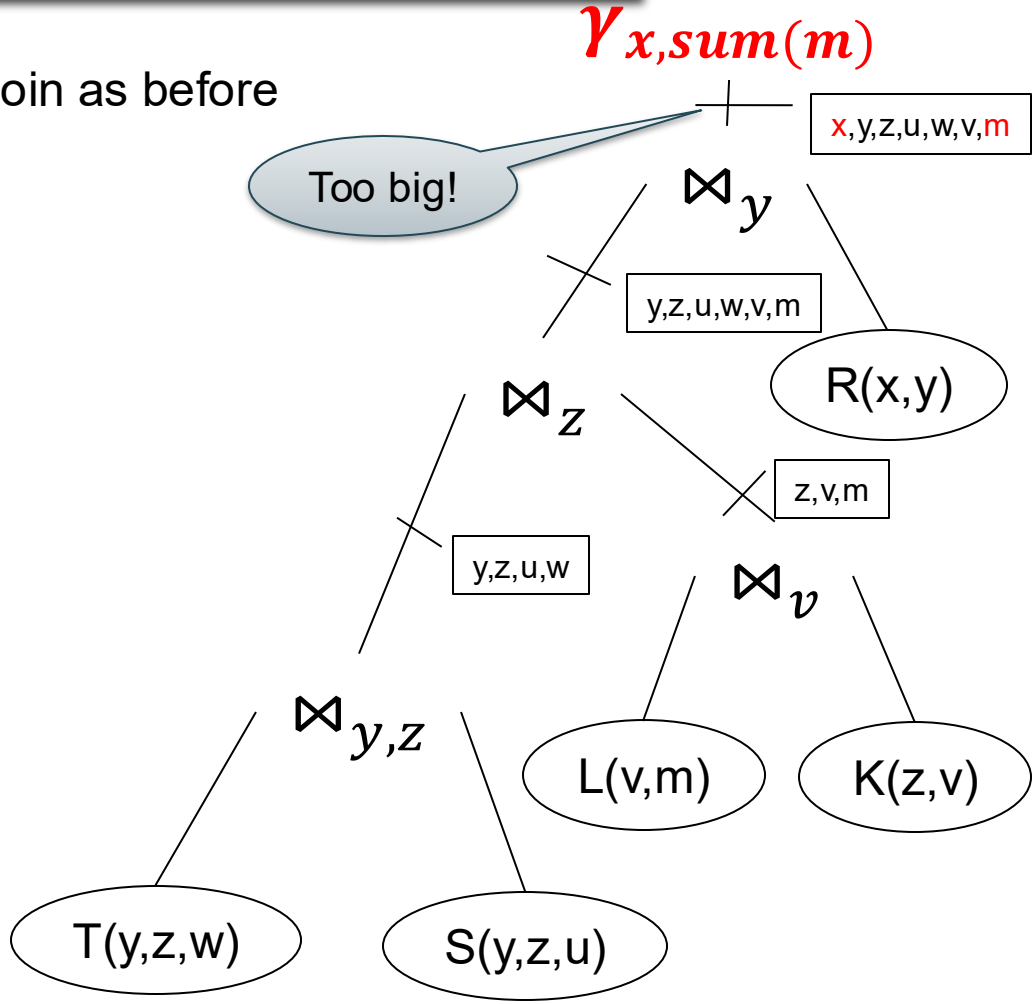


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

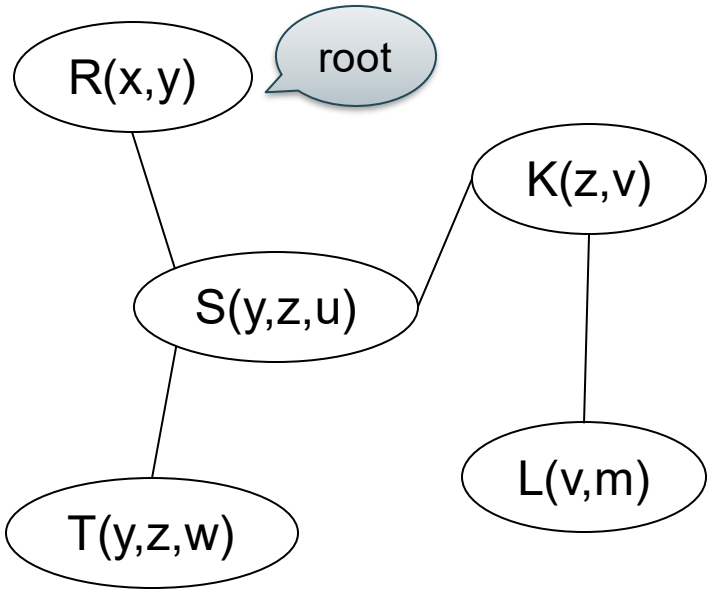


Semi-join as before

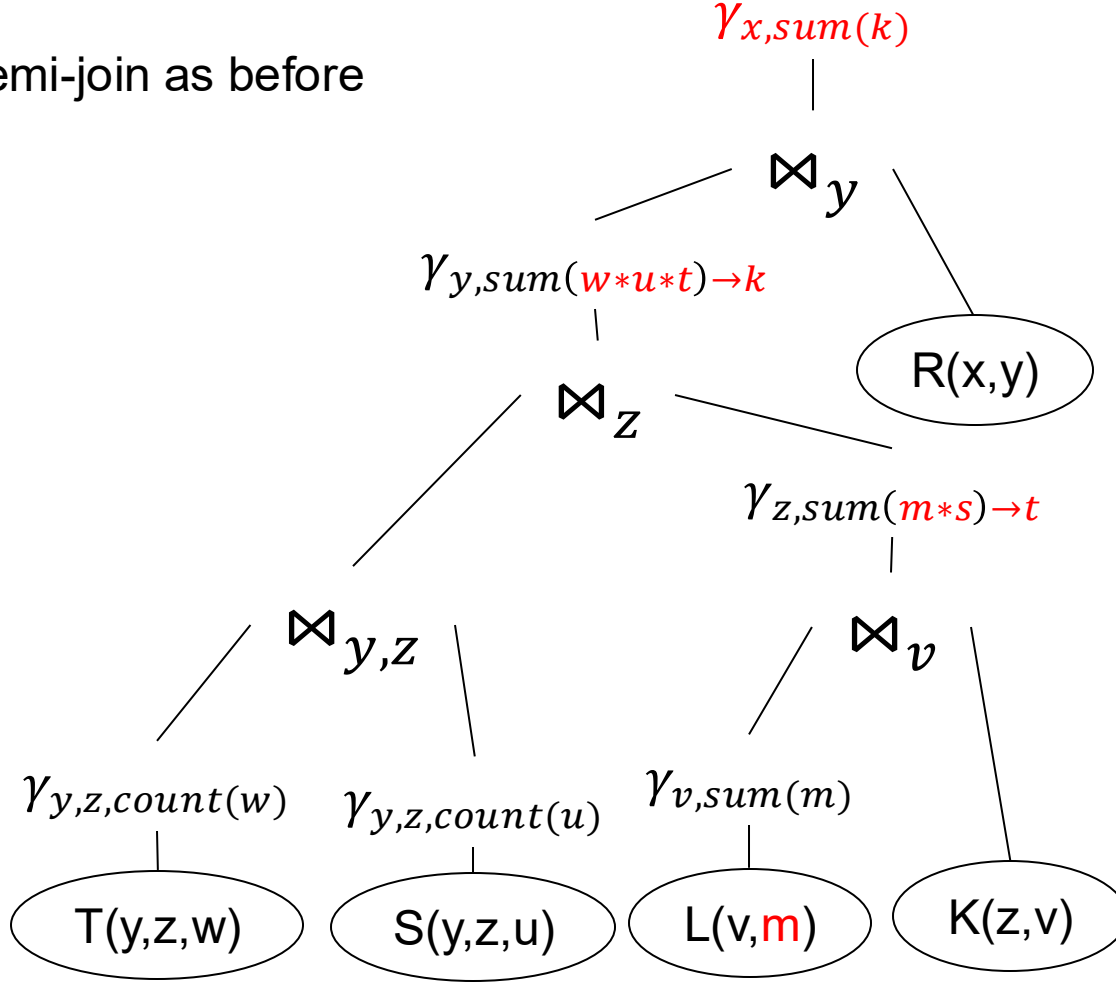


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

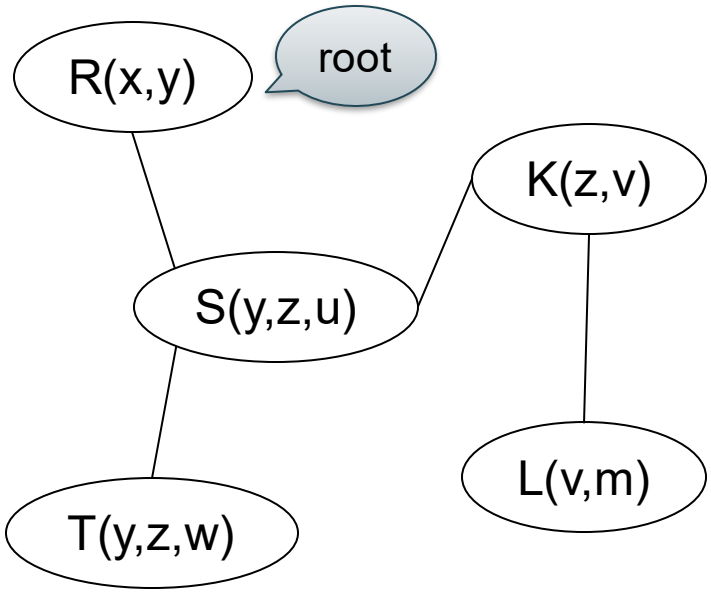


Semi-join as before

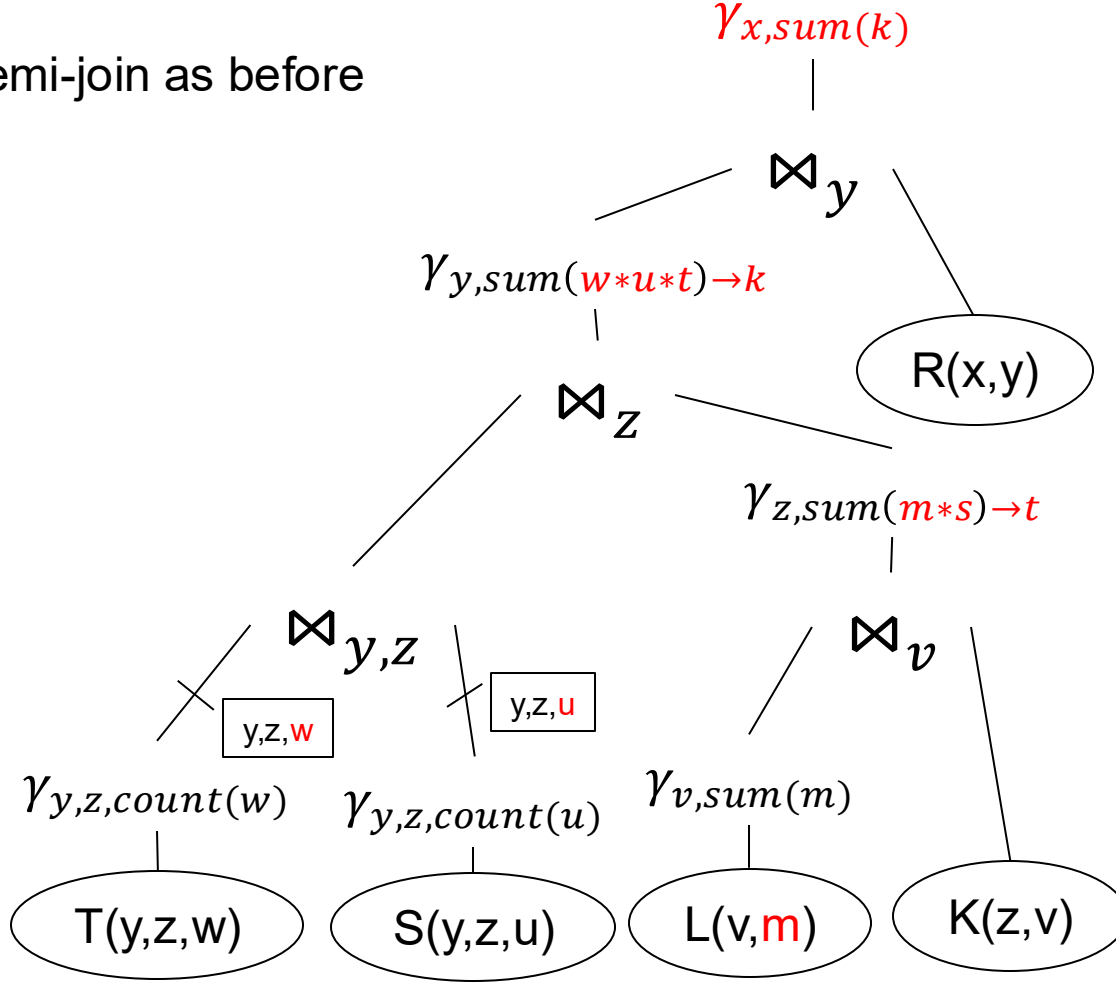


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

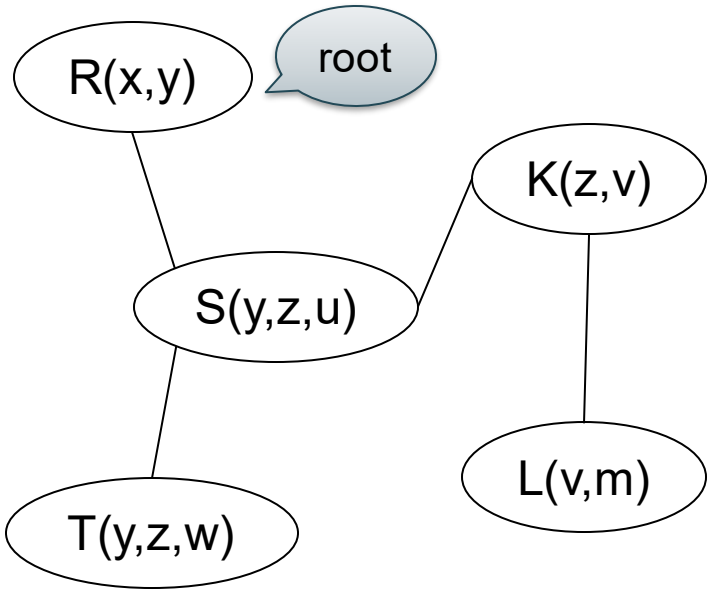


Semi-join as before

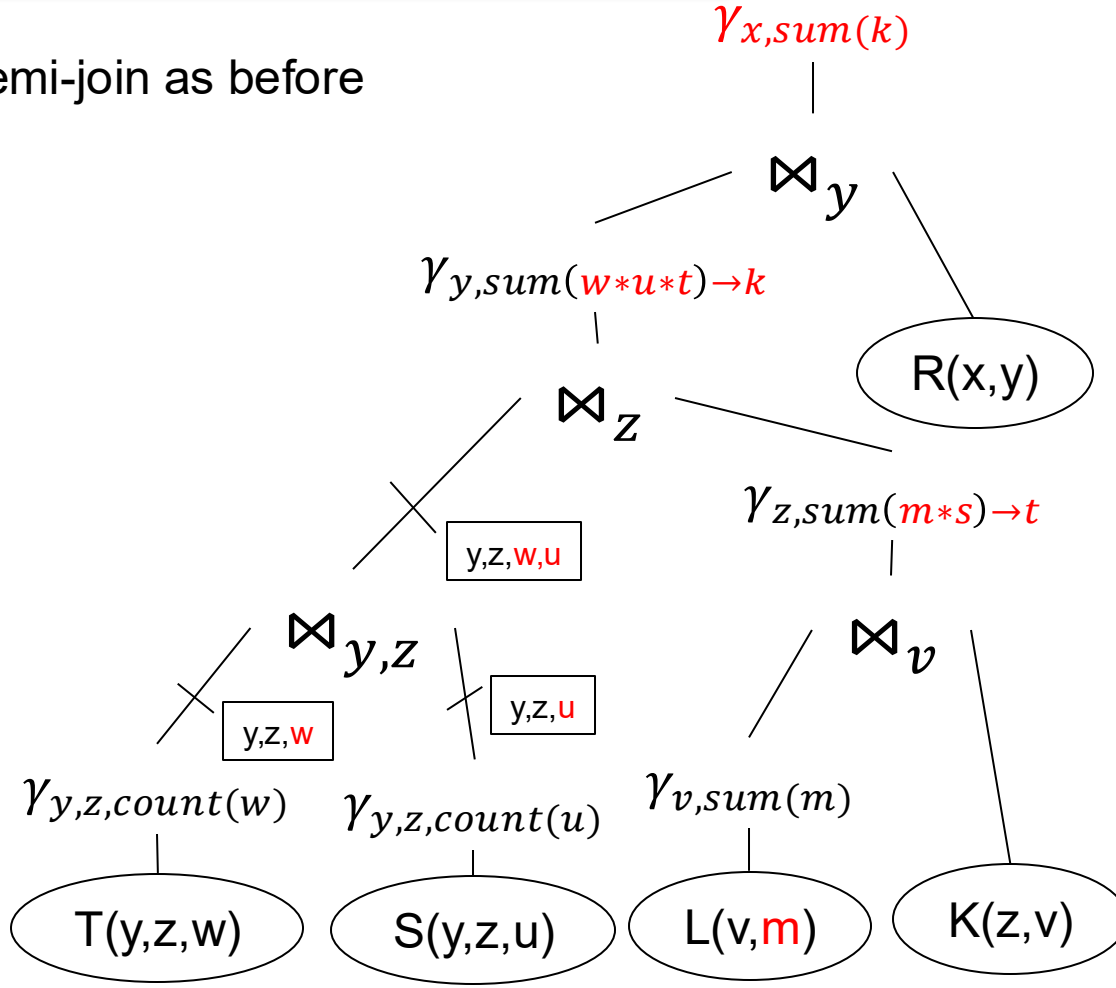


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

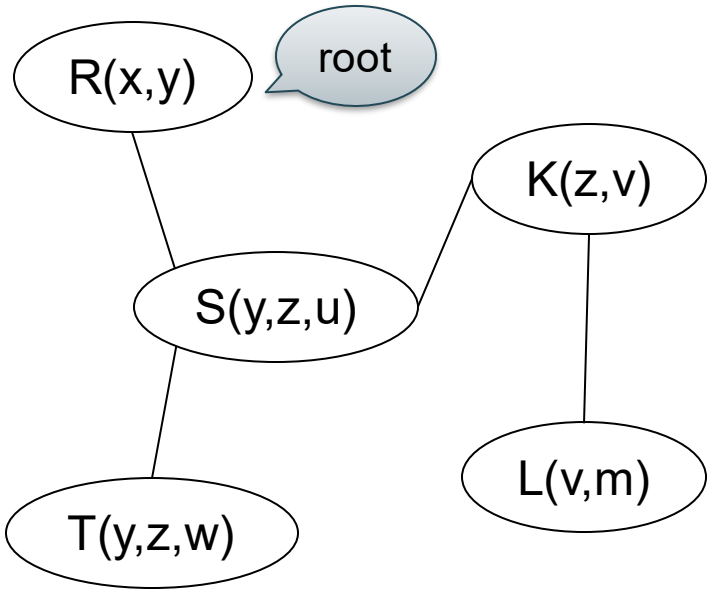


Semi-join as before

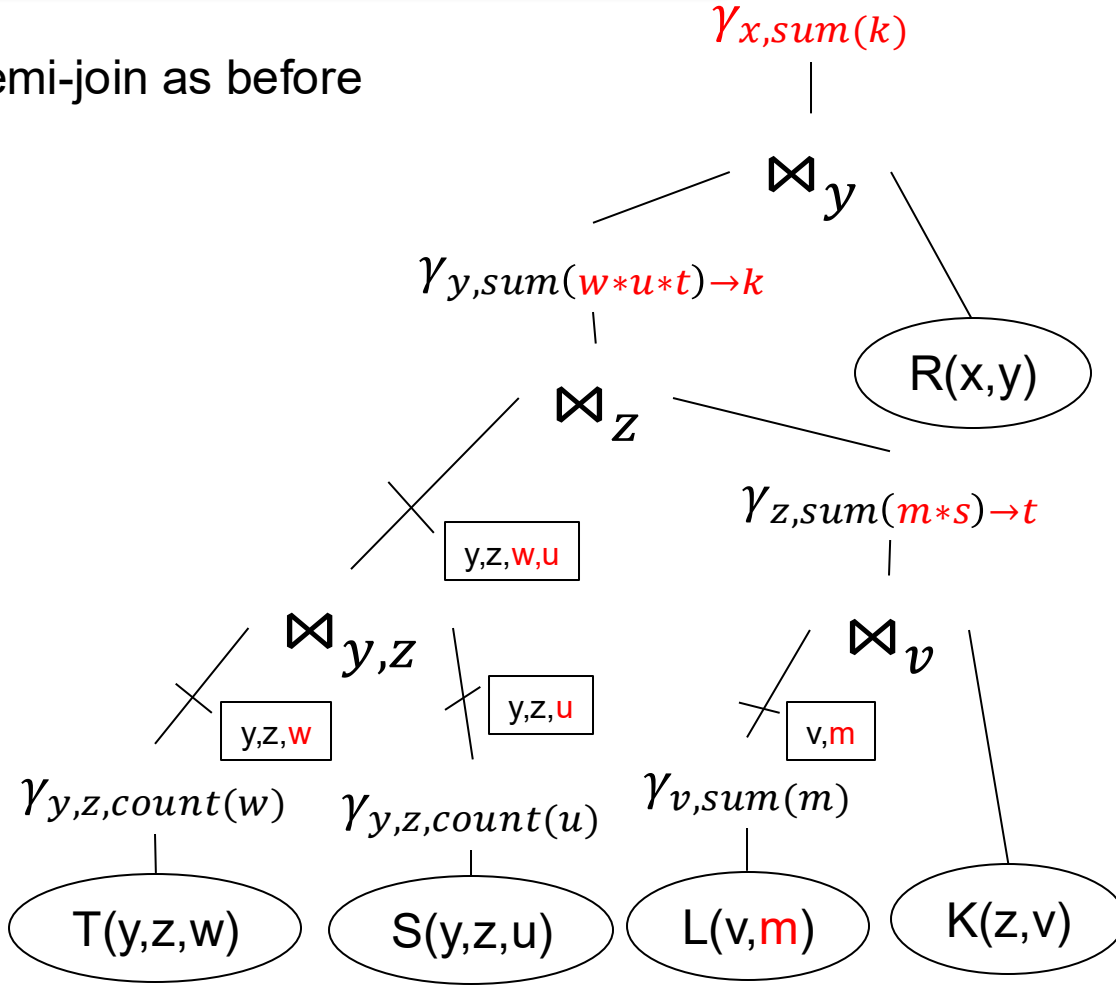


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

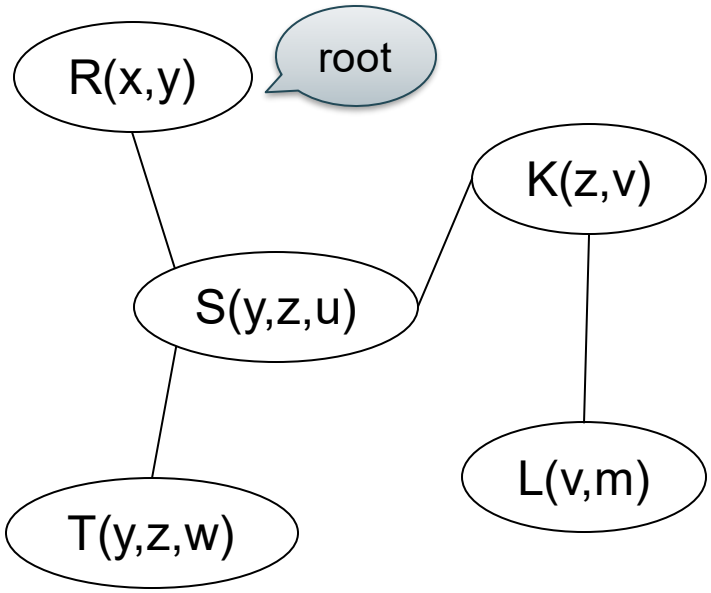


Semi-join as before

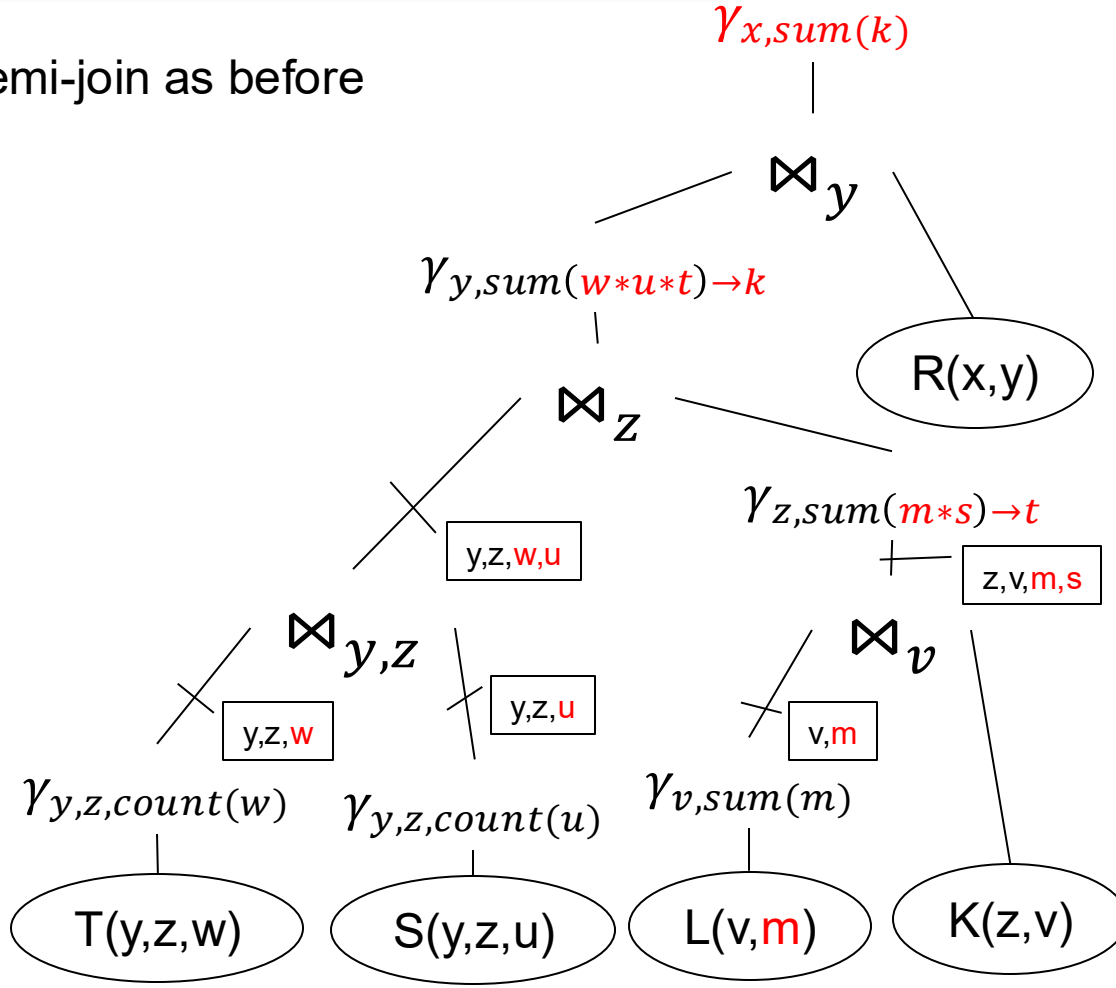


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

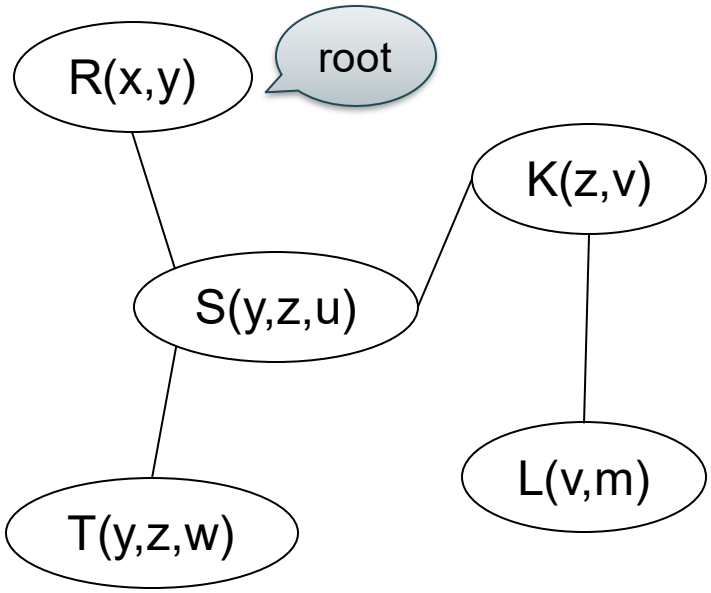


Semi-join as before

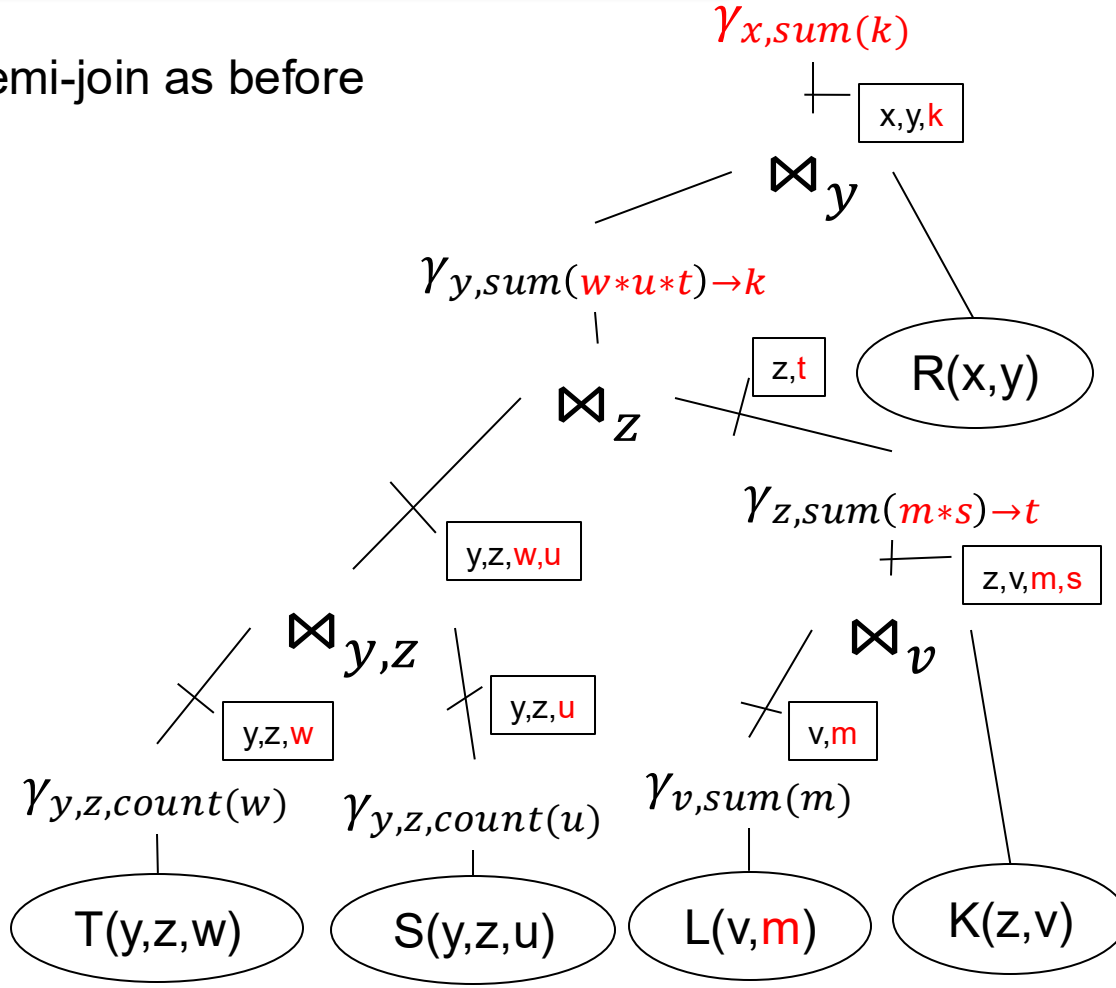


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

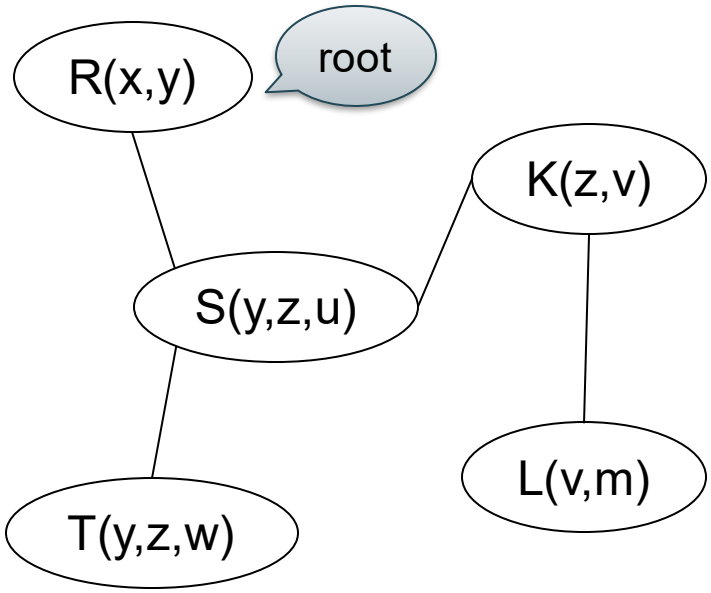


Semi-join as before

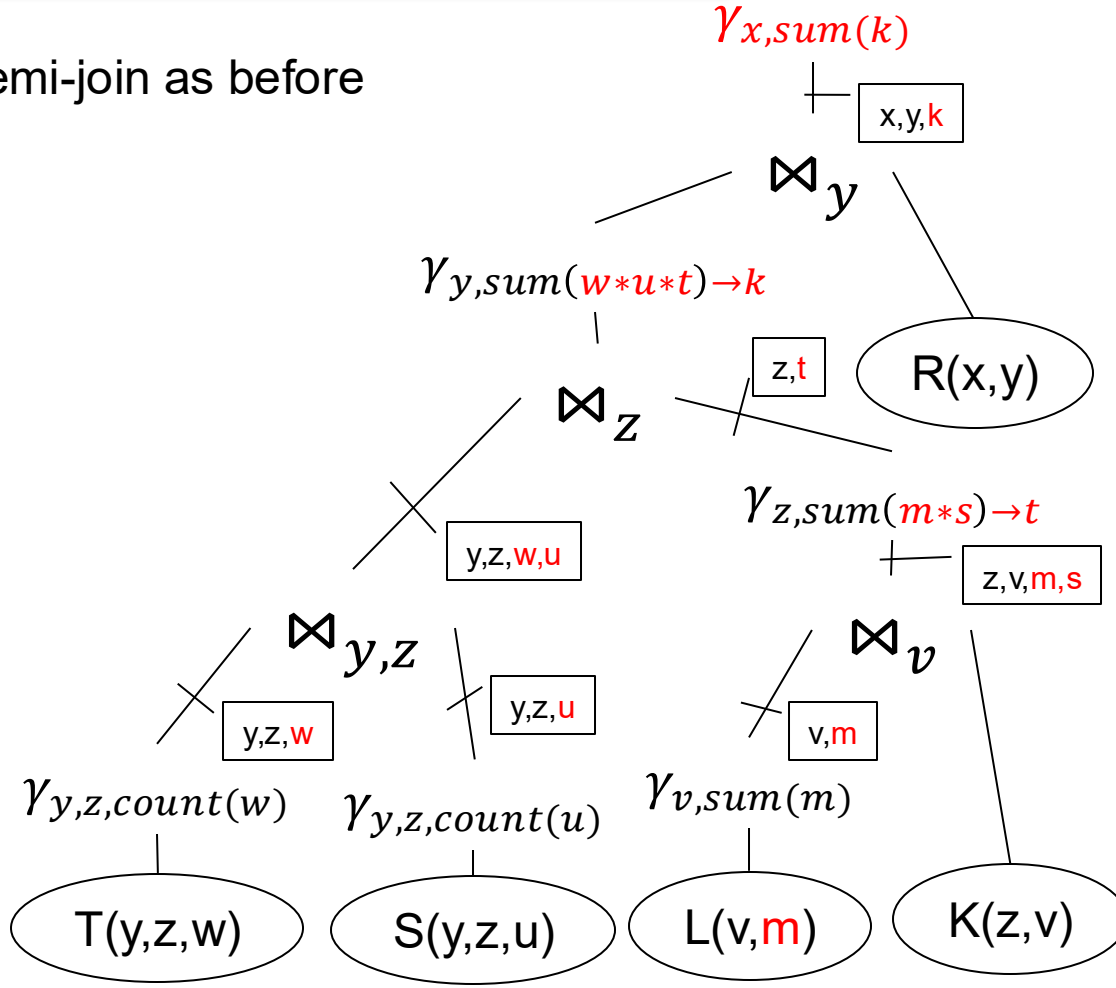


Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



Semi-join as before

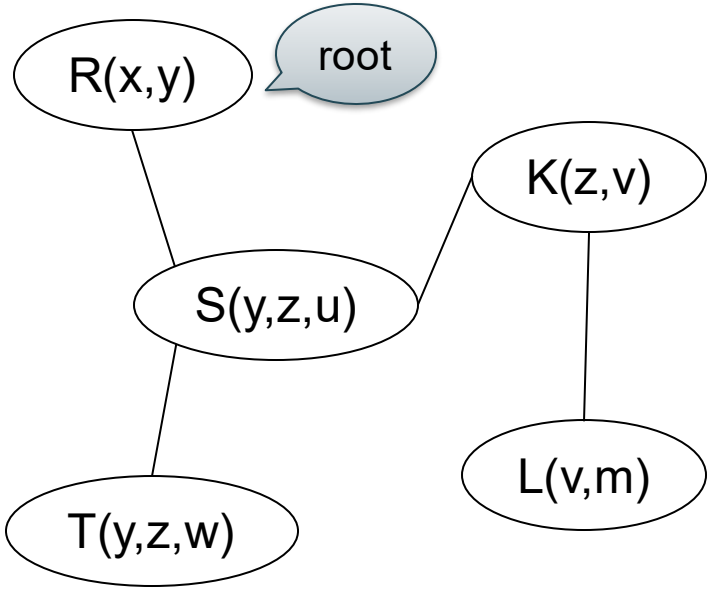


Runtime:

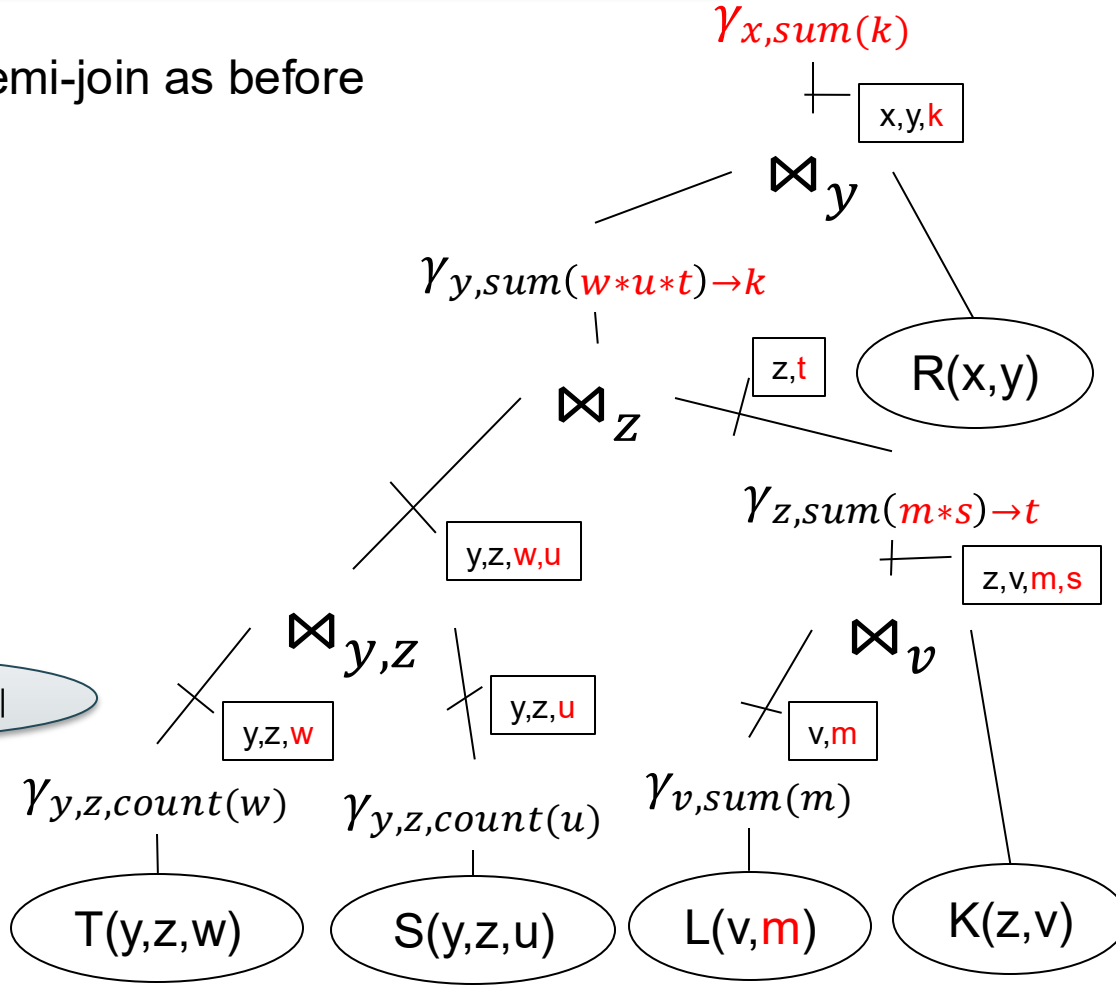
- Semi-joins: $\tilde{O}(|Input|)$
- Join/group-by: $\tilde{O}(|Input|)$

Example: CQ with Aggregates

Q(x,sum(m)) = R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)



Semi-join as before



$|Output| \leq |Input|$

Runtime:

- Semi-joins: $\tilde{O}(|Input|)$
- Join/group-by: $\tilde{O}(|Input|)$

Discussion

- Database optimizers rarely do semi-join reduction (sometimes called *magic sets*)
- Reason: when semi-join is ineffective, then it increases cost by a factor of 3*
- Recent research aims at adapting Yannakakis' algorithm to avoid the factor 3 amplification

*can be reduced to 2, by skipping the 2nd semijoin reduction, and joining from root to leaves

Discussion

- Recent research aims at adapting Yannakakis' algorithm to avoid the factor 3 amplification