CSE544
Data Management

Lectures 11-12

Advanced Query Processing

Announcements

- HW3 due on Friday
- No lecture on Monday (President’s day)
- Project milestone due next Friday

Advanced Query Processing

Current optimization techniques:
optimal plan given current statistics

- Ignores asymptotic runtime
- Sometimes asymptotic is provably bad

Advanced techniques: find optimal *asymptotic* runtime

Examples

```
SELECT count(*)
FROM Author;
```

Answer: 2419705
Time: < 1s
Asymptotic: O(N)

```
SELECT count(*)
FROM Publication;
```

Answer: 4659997
Time: < 1s
Asymptotic: O(N)

```
SELECT count(*)
FROM Author, Publication;
```

Answer: 2419705 * 4659997
Timeout
Asymptotic: $O(N^2)$
Should be: O(N)

Examples

```
SELECT count(*)
FROM Author x,
        Authored y,
        Publication z
WHERE x.authorid=y.authorid
      and y.pubid=z.pubid
      and z.year < 2015
```

Optimize this! (At home…)

Outline

- Acyclic queries, Yannakakis algorithm
- Tree decomposition of cyclic queries
- Worst-case optimal algorithm; next week

Conjunctive Queries

- A CQ is:

$$Q(X) \text{ :- } R_1(X_1), R_2(X_2), \ldots, R_m(X_m)$$

- Same as a single datalog rule

Types of CQ

- **Full** CQ: all variables are head variables
  Q(x,y,z,u) :- R(x,y),S(y,z),T(z,u)
  Q(*) :- …

- **Boolean** CQ: no variables are head variables
  Q() :- R(x,y),S(y,z),T(z,u)

- CQ with aggregates:
  Q(x,sum(u)) :- R(x,y),S(y,z),T(z,u)

Generalized Distributivity Law

- Basic idea: group-by commutes with join if we write it the right way

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

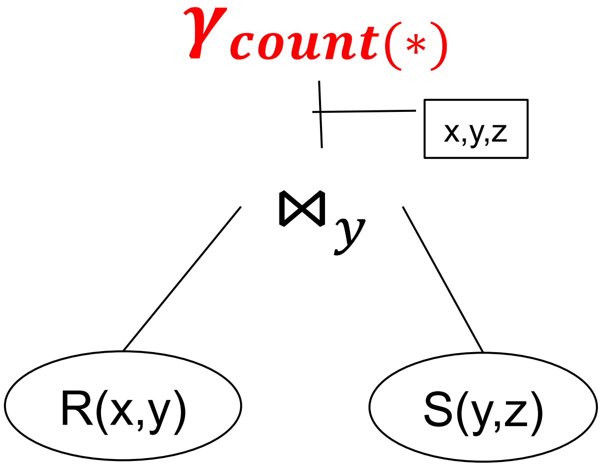
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

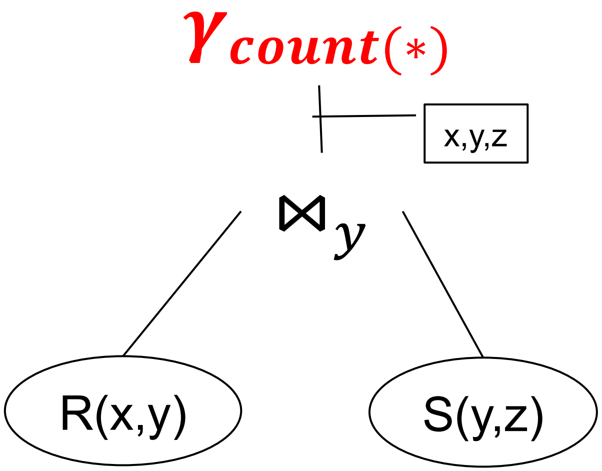
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A(y,count(x) as c) = R(x,y)
B(y,count(z) as d) = S(y,z)
Q(sum(c*d))= A(y,c),B(y,d)

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

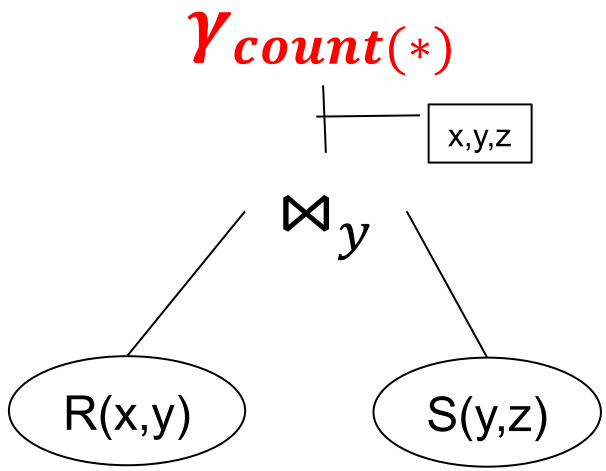
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A(y,count(x) as c) = R(x,y)
B(y,count(z) as d) = S(y,z)
Q(sum(c*d))= A(y,c),B(y,d)

A:

y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

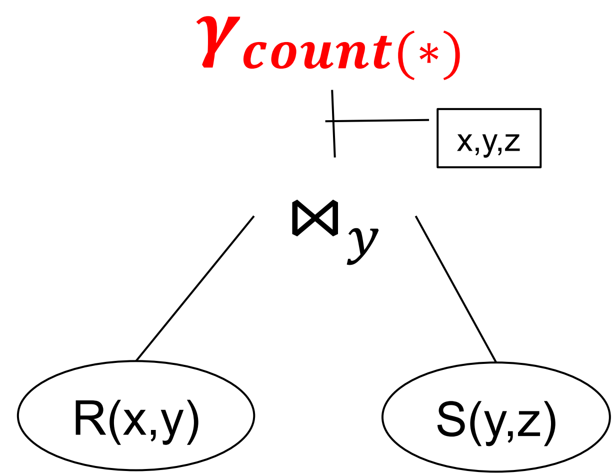
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A(y,count(x) as c) = R(x,y)
B(y,count(z) as d) = S(y,z)
Q(sum(c*d))= A(y,c),B(y,d)

A:

y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

A⋈B

y	c	d
b	2	2
h	1	1

Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

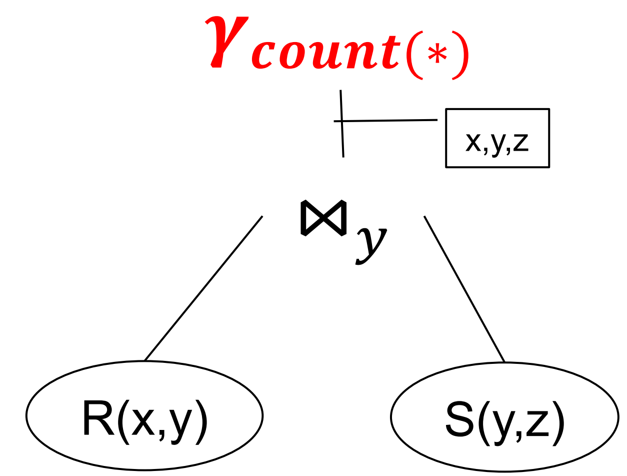
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A(y,count(x) as c) = R(x,y)
B(y,count(z) as d) = S(y,z)
Q(sum(c*d))= A(y,c),B(y,d)

A:

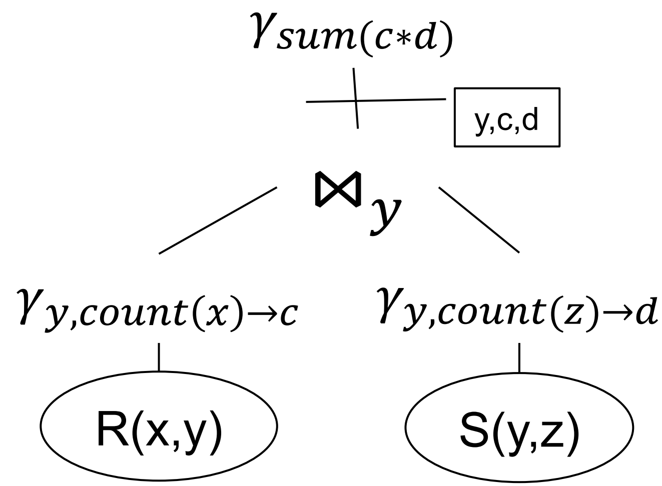
y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

A⋈B

y	c	d
b	2	2
h	1	1



Generalized Distributivity Law

Q(count(*)) = R(x,y),S(y,z)

R:

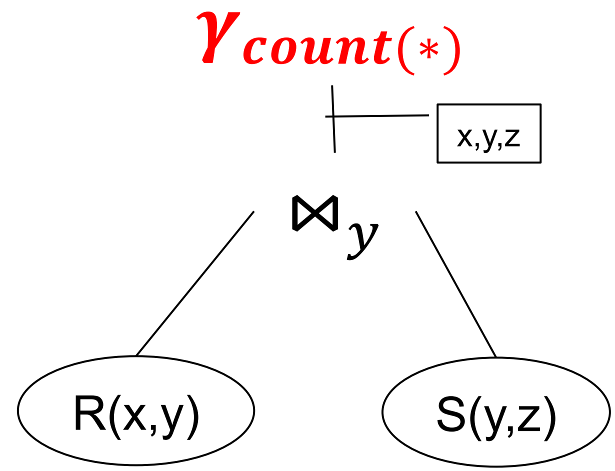
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

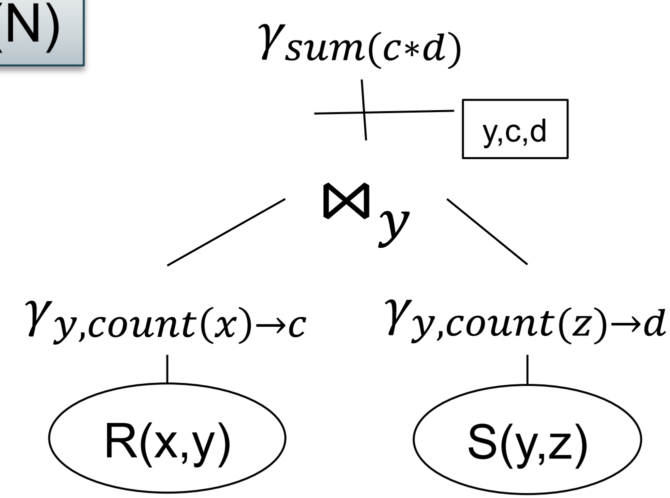
Answer = 5

Runtime = $O(N^2)$



A(y,count(x) as c) = R(x,y)
B(y,count(z) as d) = S(y,z)
Q(sum(c*d))= A(y,c),B(y,d)

Runtime = O(N)



A:

y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

A⋈B

y	c	d
b	2	2
h	1	1

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

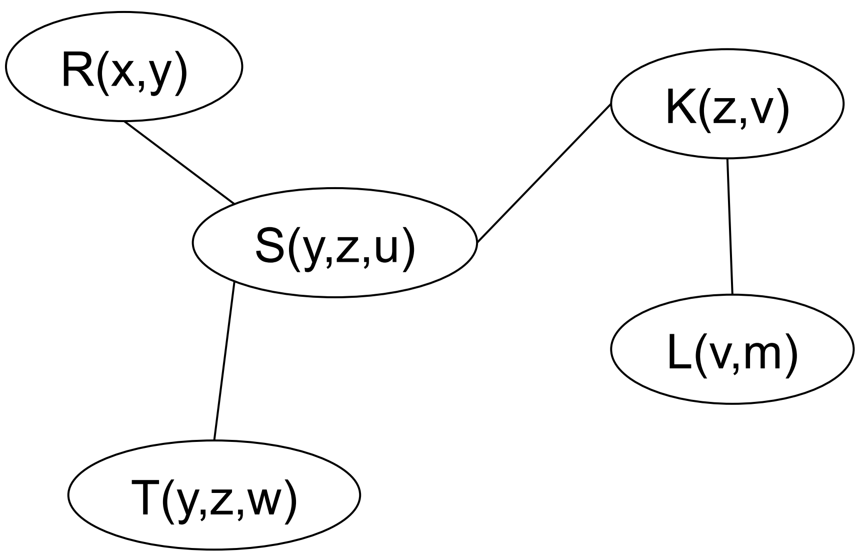
Q is *acyclic* if its atoms can be placed in a tree T such that for every variable the set of nodes that contain that variable form a connected component

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

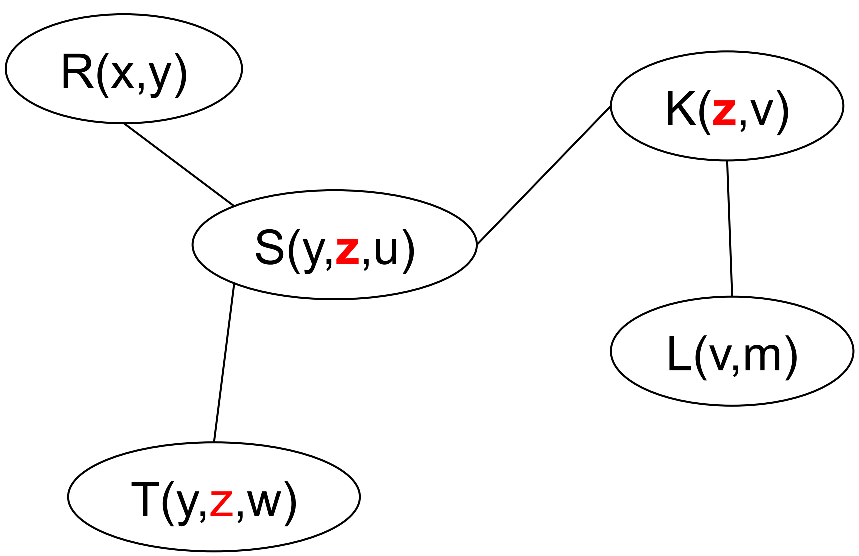


R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

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



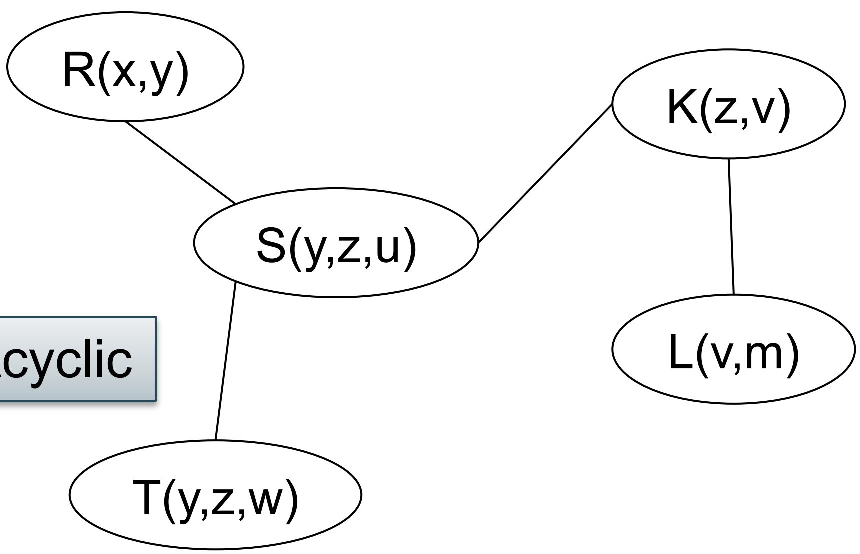
E.g. **z** forms a connected component

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



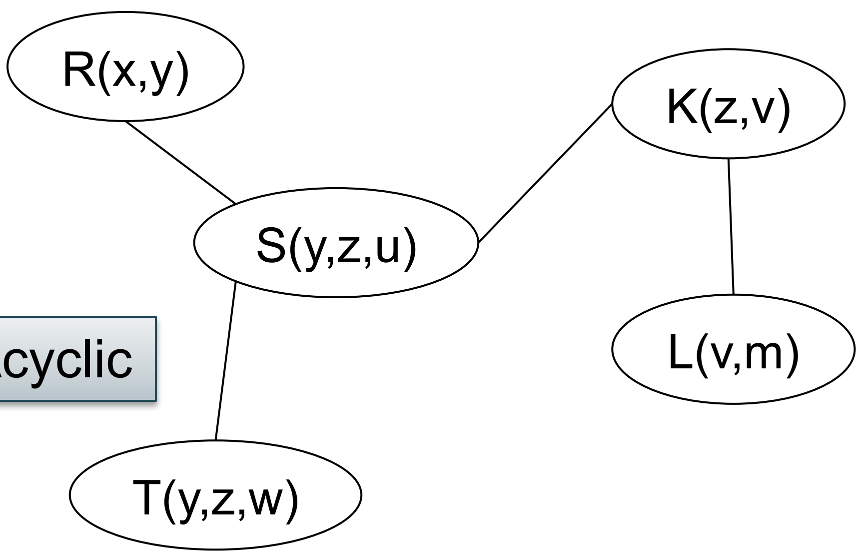
Acyclic

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

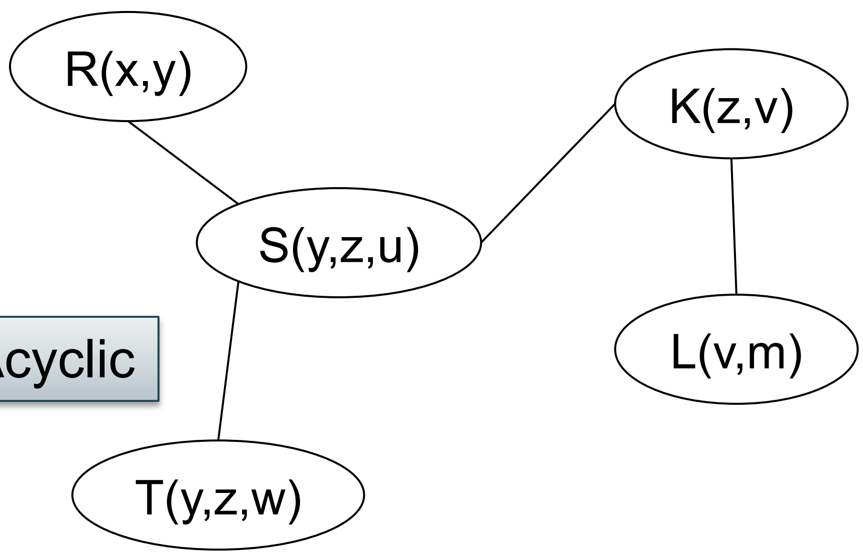
R(x,y),S(y,z,u),T(y,z,w),K(z,v),L(v,m)

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

T is called *join tree*

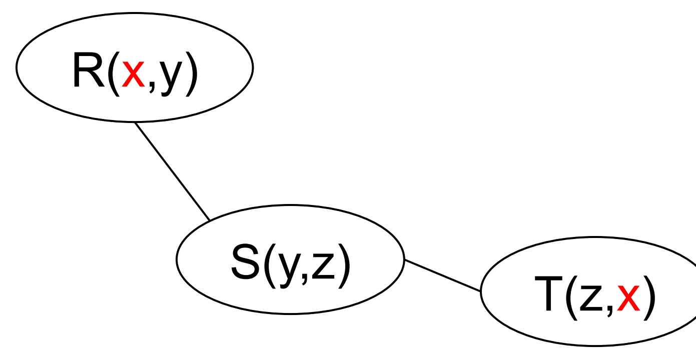
Acyclic Queries

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)

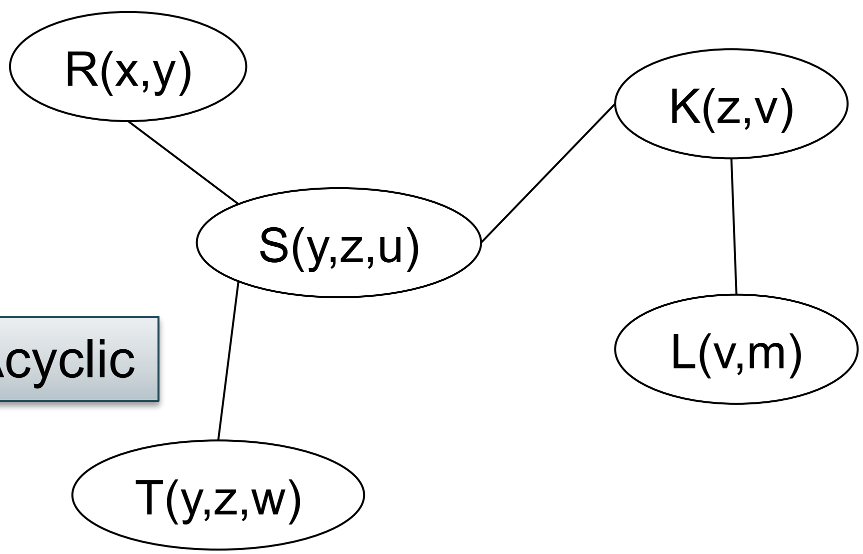


R(x,y),S(y,z),T(z,x)

T is called *join tree*

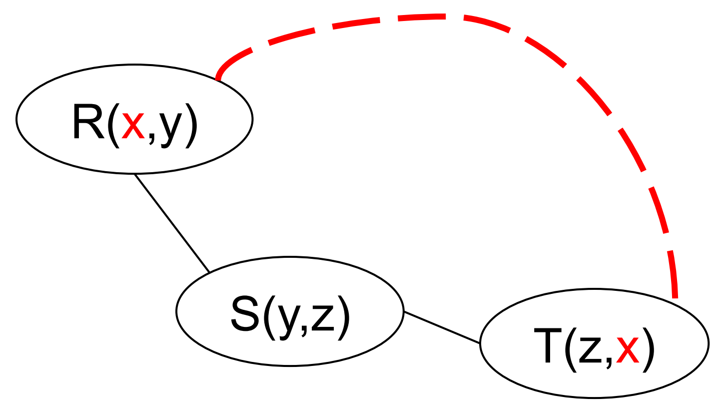
Acyclic Queries

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)



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

Cyclic

A Theorem

Q = an acyclic query Q that is:

- Boolean, or
- Full, or
- Aggregate with ≤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

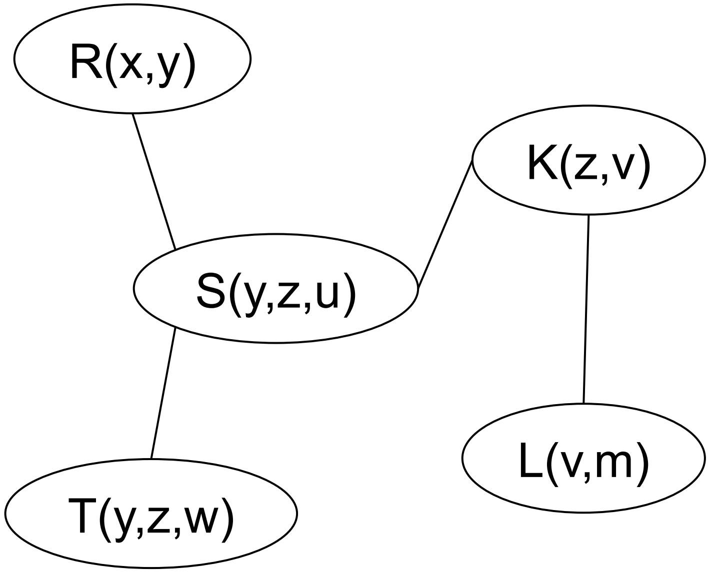
- Step 1: semi-join reduction
  - Pick any root node in the join tree of Q
  - Semi-join reduction from leaves to root
  - Semi-join reduction from root to leaves
- Step 2:
  - Compute the joins bottom up,
  - Push group-by down

Example: Full CQ

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

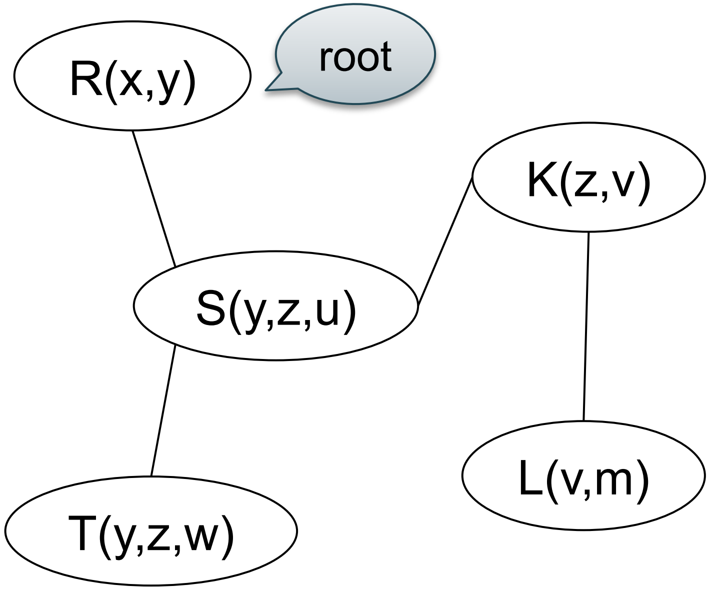
Example: Full CQ

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



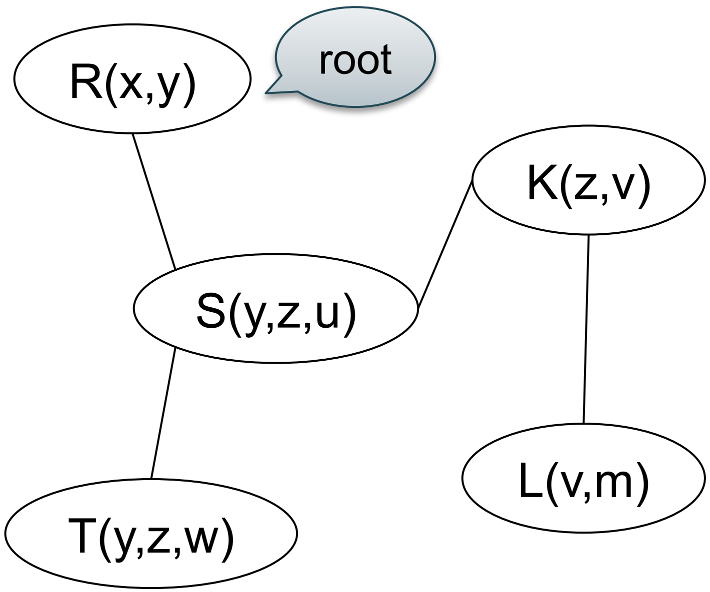
Example: Full CQ

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)

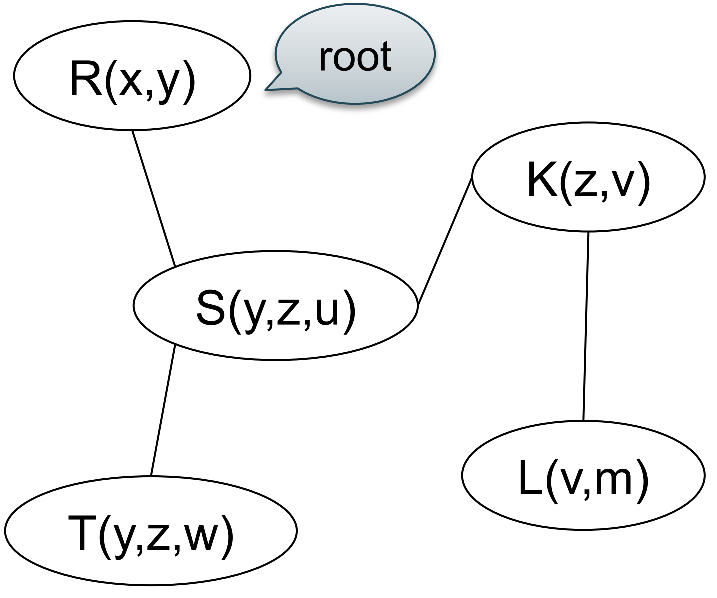


-- Leaves to root:

K :- K ⋉ L

Example: Full CQ

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



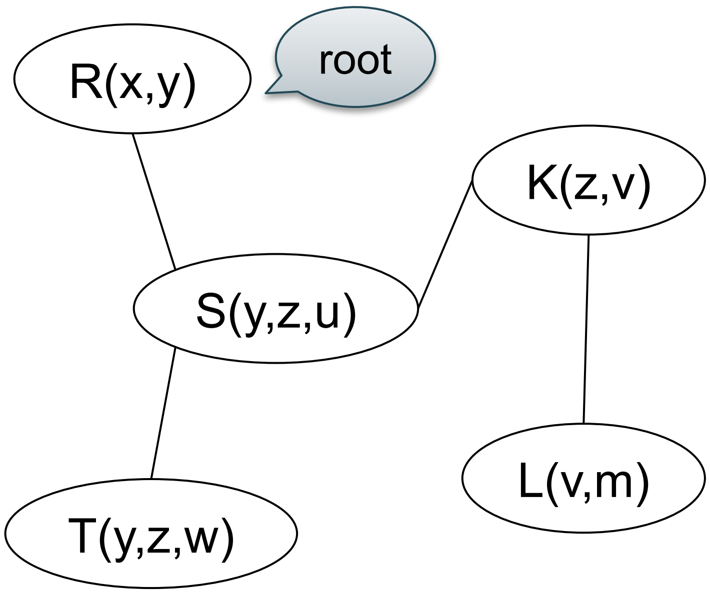
-- Leaves to root:

K :- K ⋉ L

S :- S ⋉ 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:

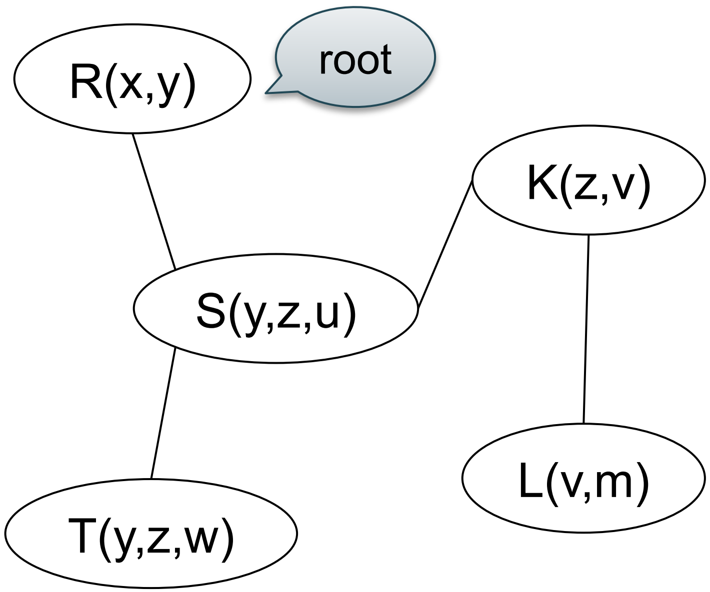
K :- K $\ltimes$ L

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 ⋉ L

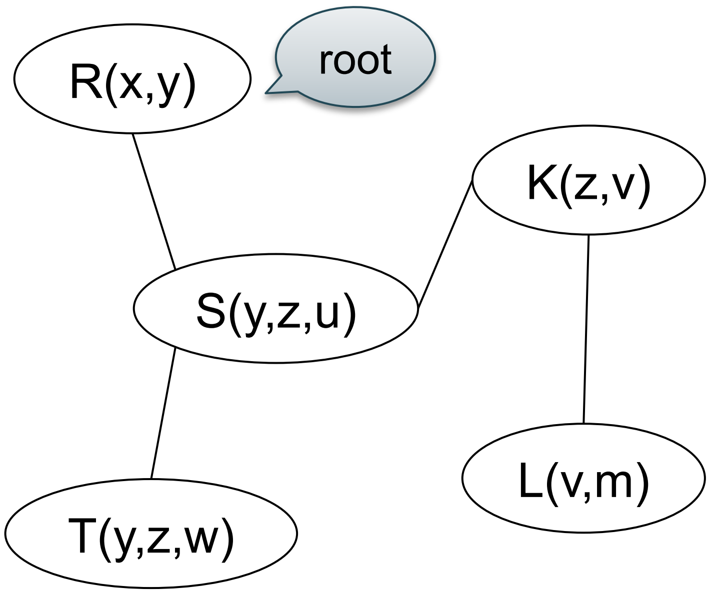
S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ 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 ⋉ L

S :- S ⋉ T

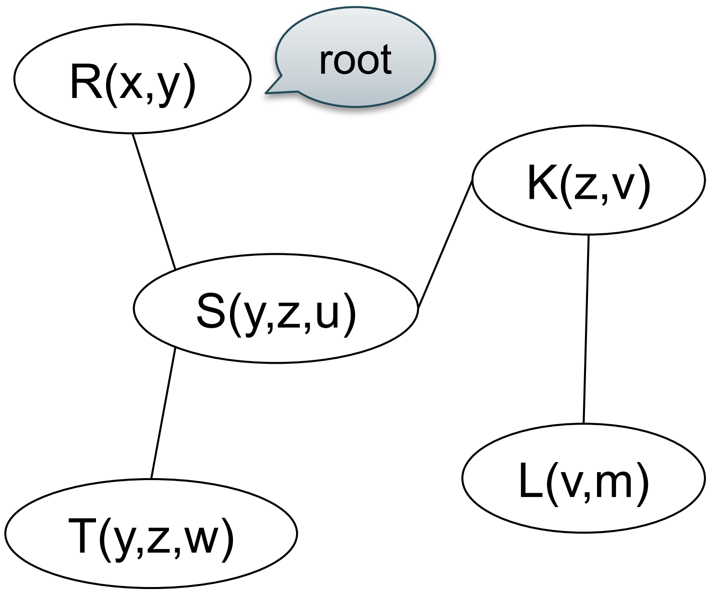
S :- S ⋉ K

R :- R ⋉ 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 ⋉ L

S :- S ⋉ T

S :- S ⋉ K

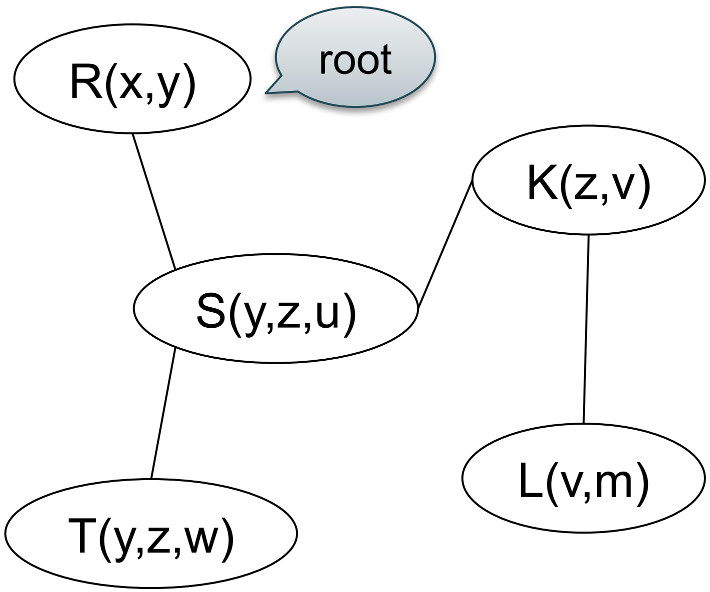
R :- R ⋉ S

-- Root to leaves:

S :- S ⋉ 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

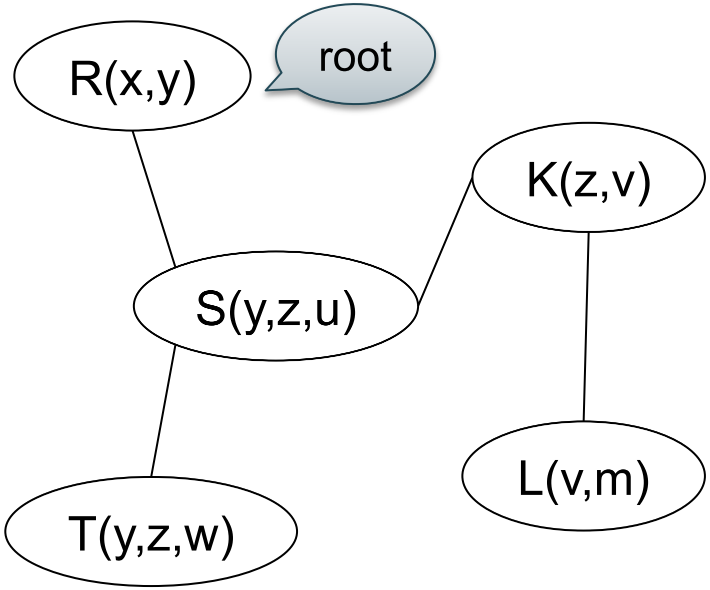
-- Root to leaves:

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 ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

-- Root to leaves:

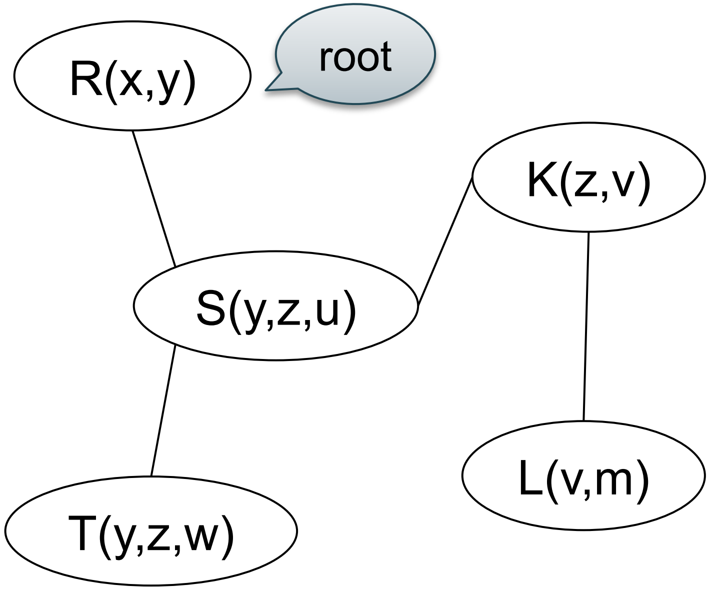
S :- S ⋉ R

T :- T ⋉ S

K :- K ⋉ 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 ⋉ L

S :- S ⋉ T

S :- S ⋉ K

R :- R ⋉ S

-- Root to leaves:

S :- S ⋉ R

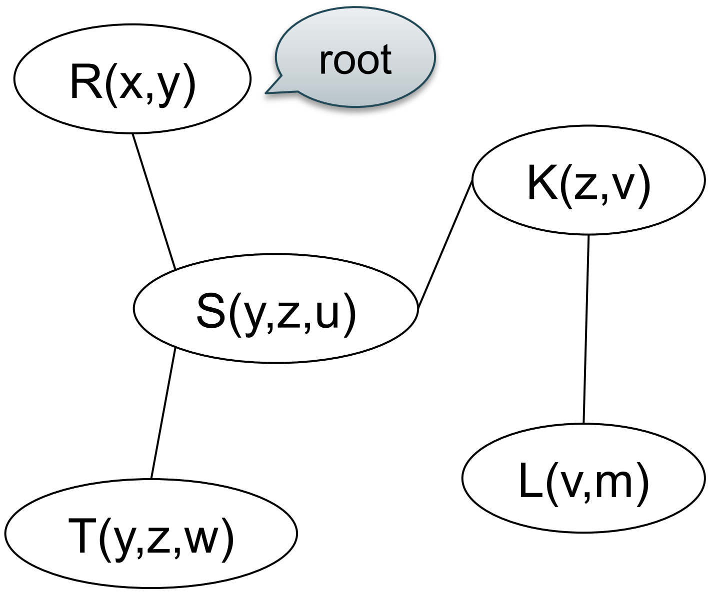
T :- T ⋉ S

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 $\ltimes$ L

S :- S $\ltimes$ T

S :- S $\ltimes$ K

R :- R $\ltimes$ S

-- Root to leaves:

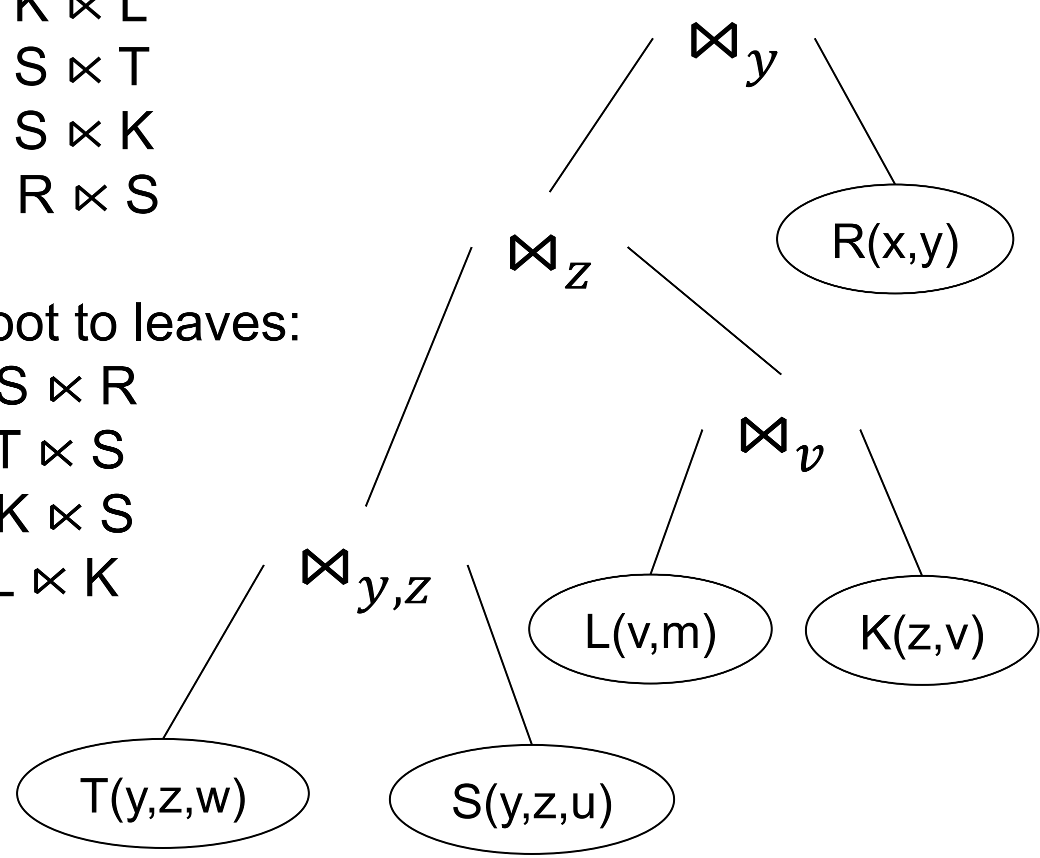
S :- S $\ltimes$ R

T :- T $\ltimes$ S

K :- K $\ltimes$ S

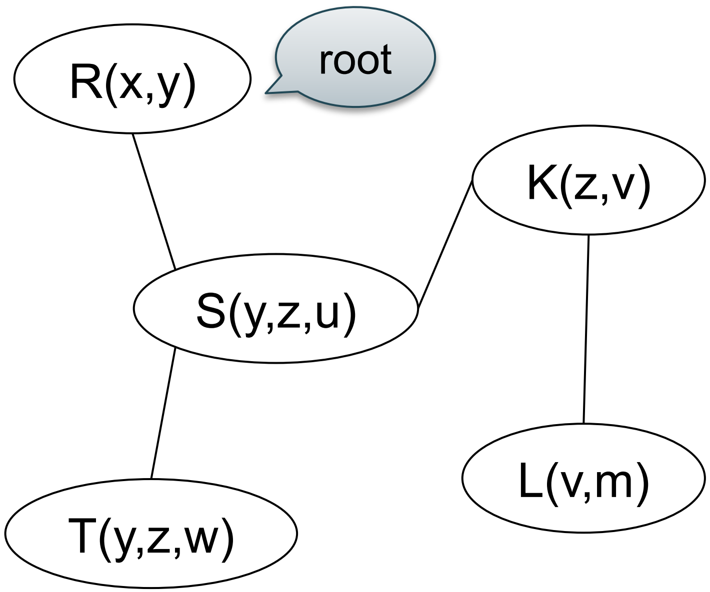
L :- L $\ltimes$ K

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:

$K :- K \ltimes L$

$S :- S \ltimes T$

$S :- S \ltimes K$

$R :- R \ltimes S$

-- Root to leaves:

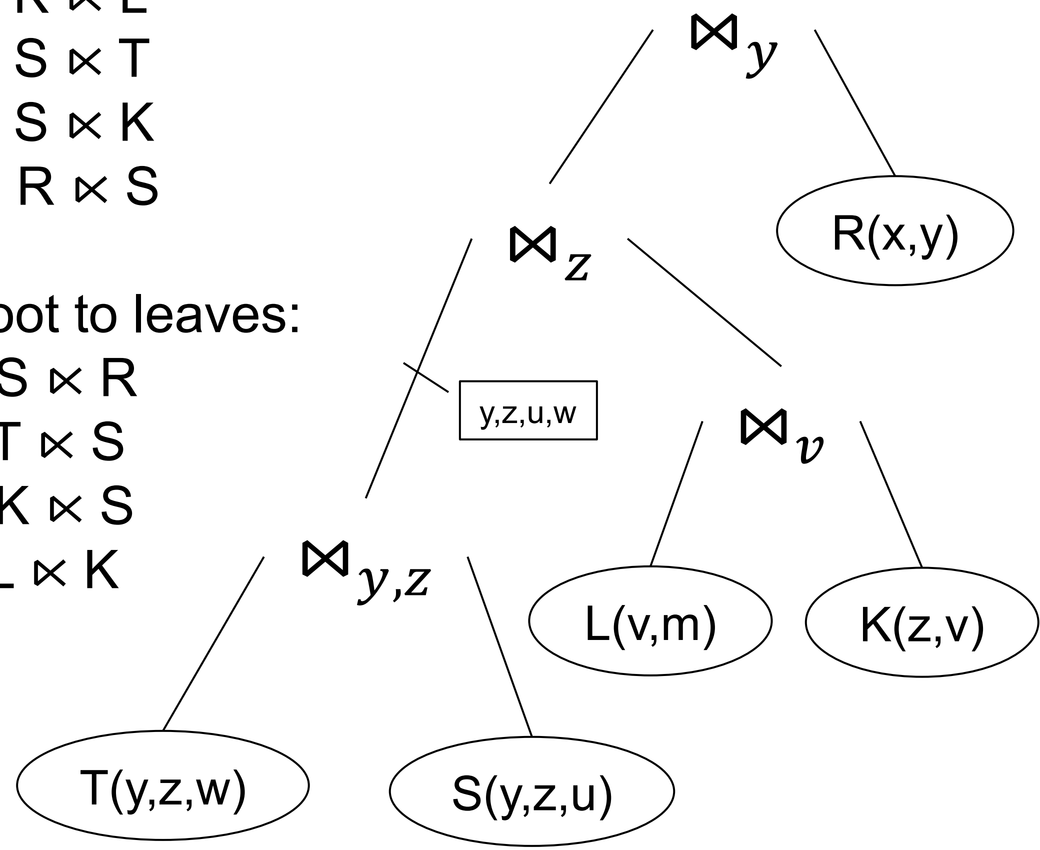
$S :- S \ltimes R$

$T :- T \ltimes S$

$K :- K \ltimes S$

$L :- L \ltimes K$

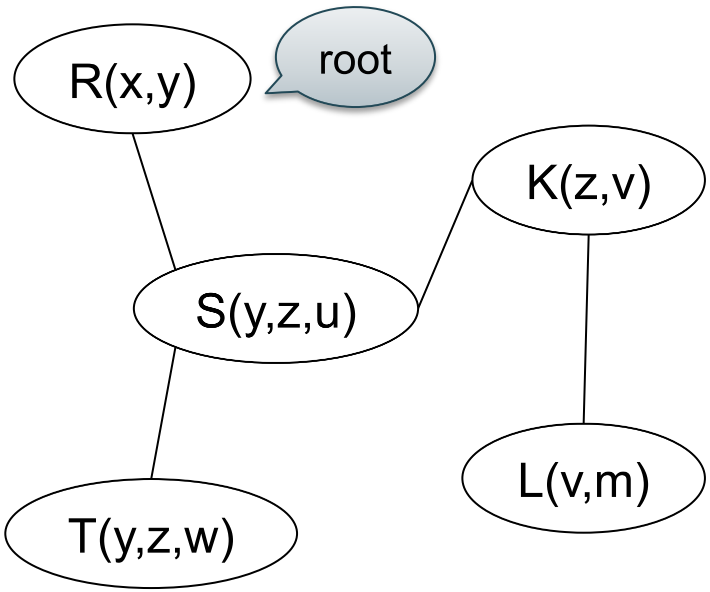
Join (any order in the tree)



Example: Full CQ

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

Join (any order in the tree)

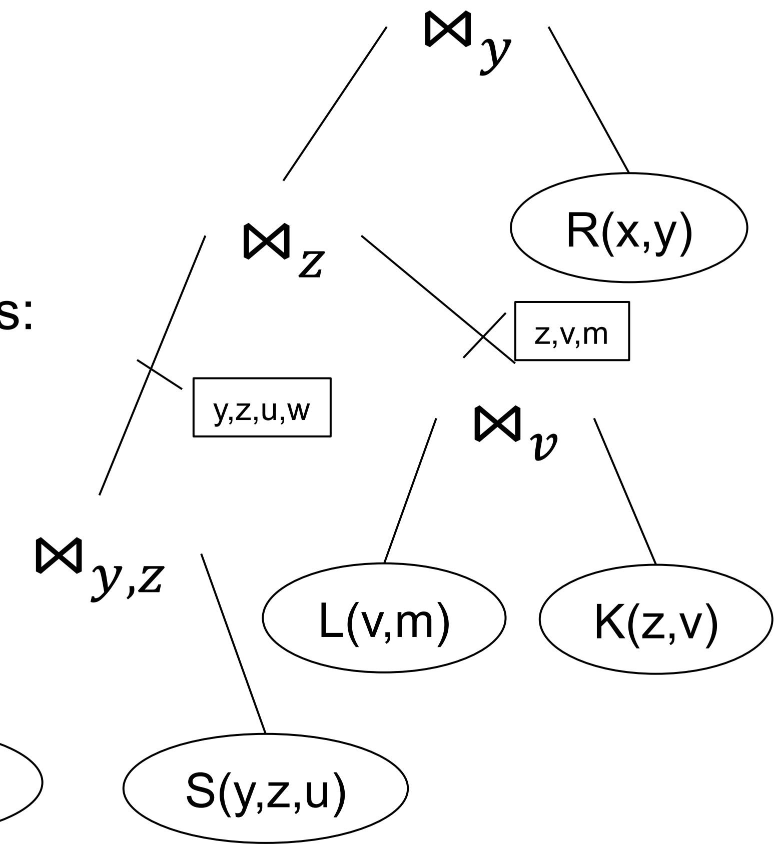


-- Leaves to root:

K :- K $\ltimes$ L
S :- S $\ltimes$ T
S :- S $\ltimes$ K
R :- R $\ltimes$ S

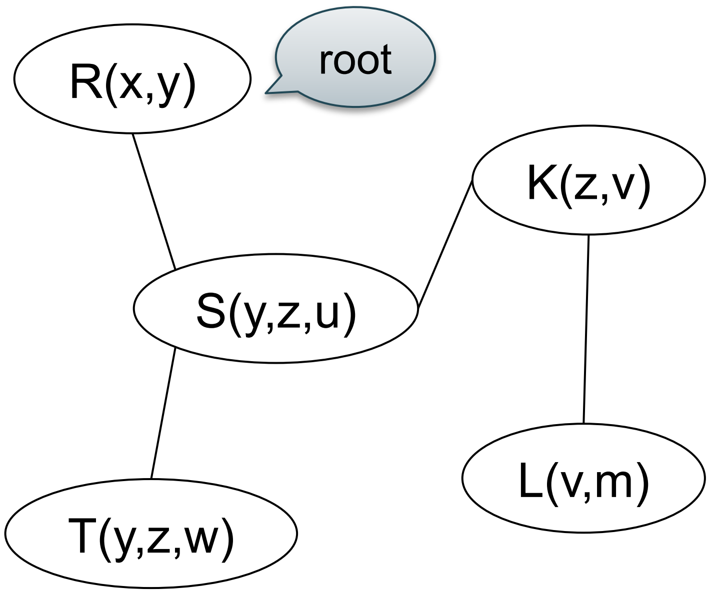
-- Root to leaves:

S :- S $\ltimes$ R
T :- T $\ltimes$ S
K :- K $\ltimes$ S
L :- L $\ltimes$ K



Example: Full CQ

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



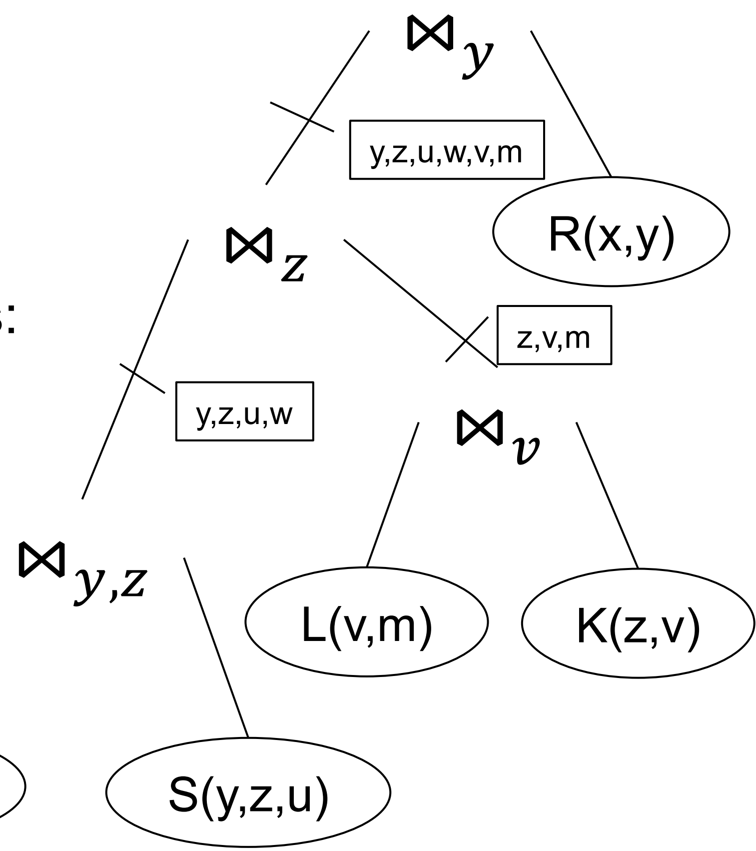
-- Leaves to root:

K :- K $\ltimes$ L
S :- S $\ltimes$ T
S :- S $\ltimes$ K
R :- R $\ltimes$ S

-- Root to leaves:

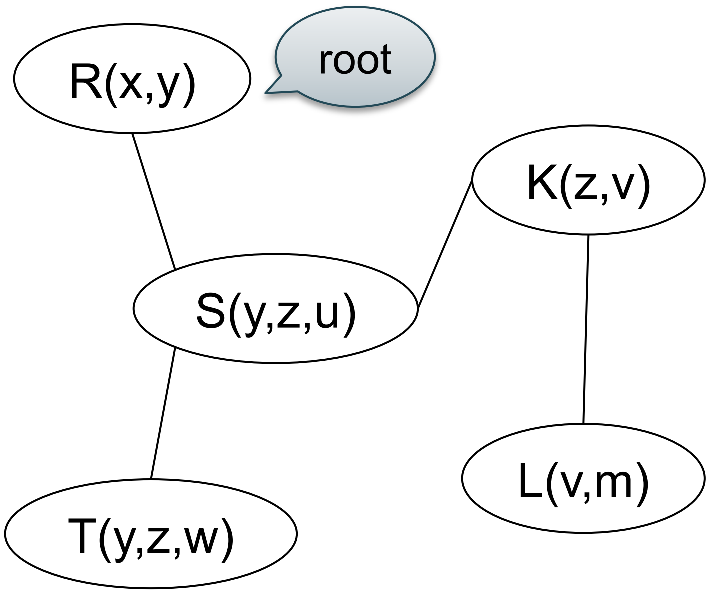
S :- S $\ltimes$ R
T :- T $\ltimes$ S
K :- K $\ltimes$ S
L :- L $\ltimes$ K

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 $\ltimes$ L

S :- S $\ltimes$ T

S :- S $\ltimes$ K

R :- R $\ltimes$ S

-- Root to leaves:

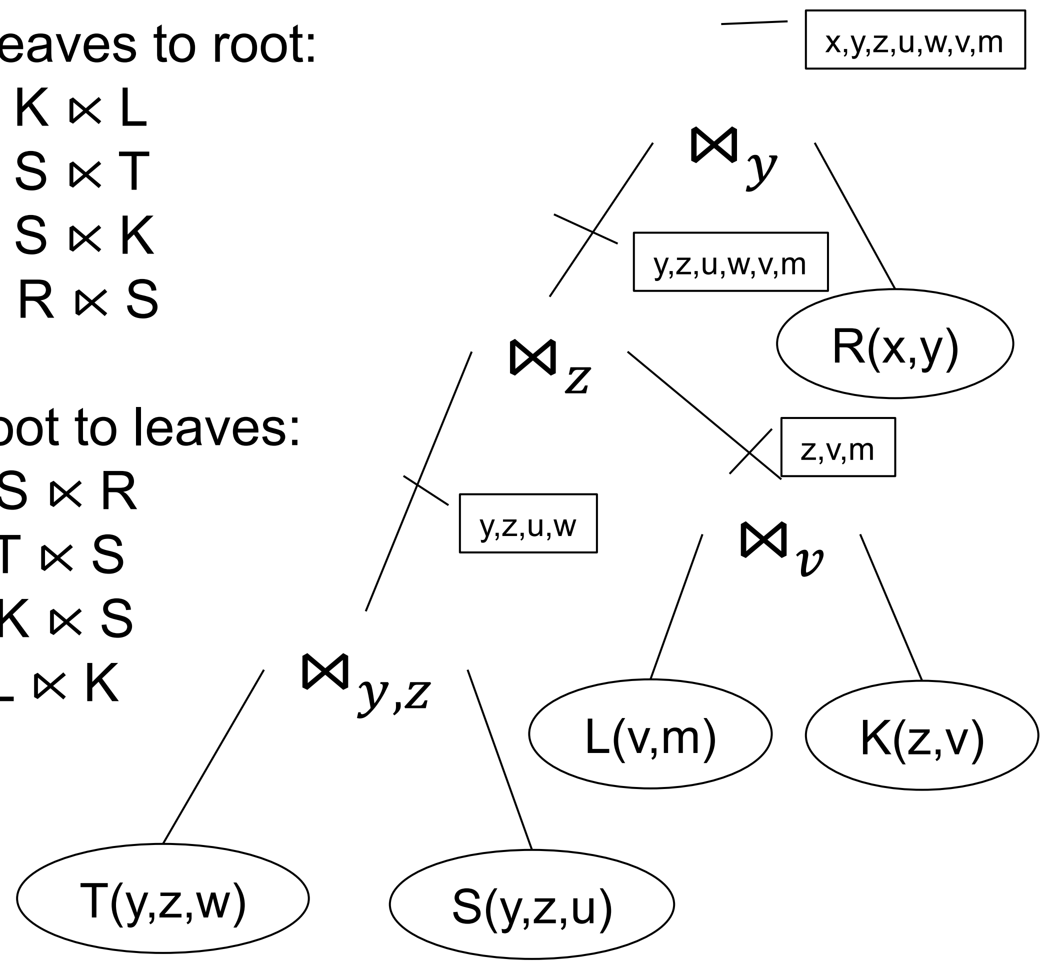
S :- S $\ltimes$ R

T :- T $\ltimes$ S

K :- K $\ltimes$ S

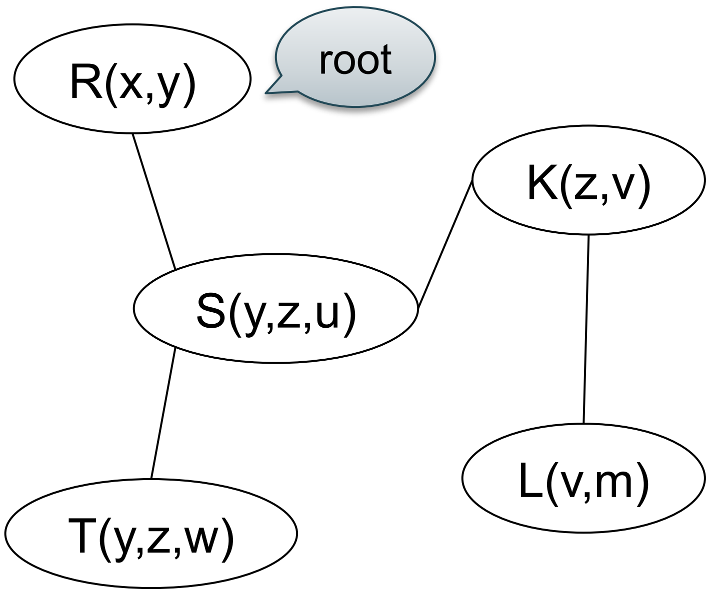
L :- L $\ltimes$ K

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 $\ltimes$ L

S :- S $\ltimes$ T

S :- S $\ltimes$ K

R :- R $\ltimes$ S

-- Root to leaves:

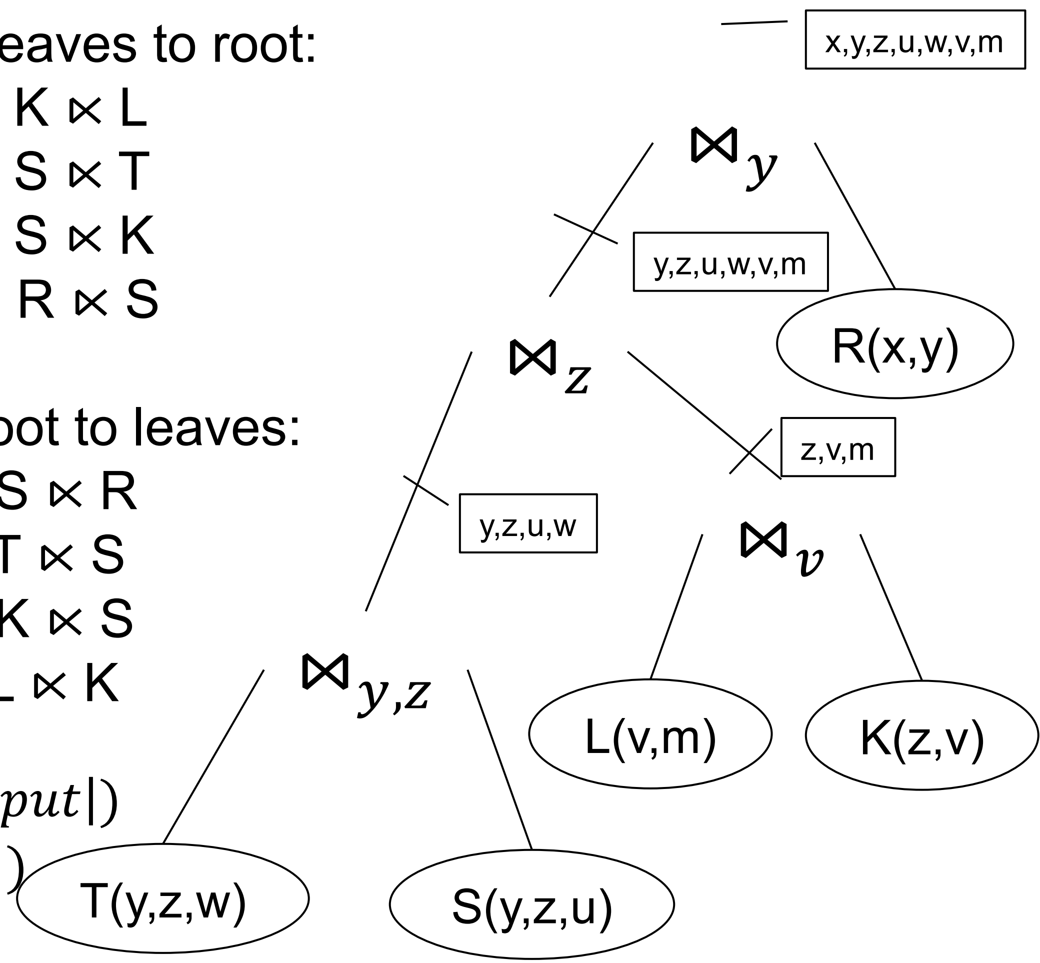
S :- S $\ltimes$ R

T :- T $\ltimes$ S

K :- K $\ltimes$ S

L :- L $\ltimes$ K

Join (any order in the tree)

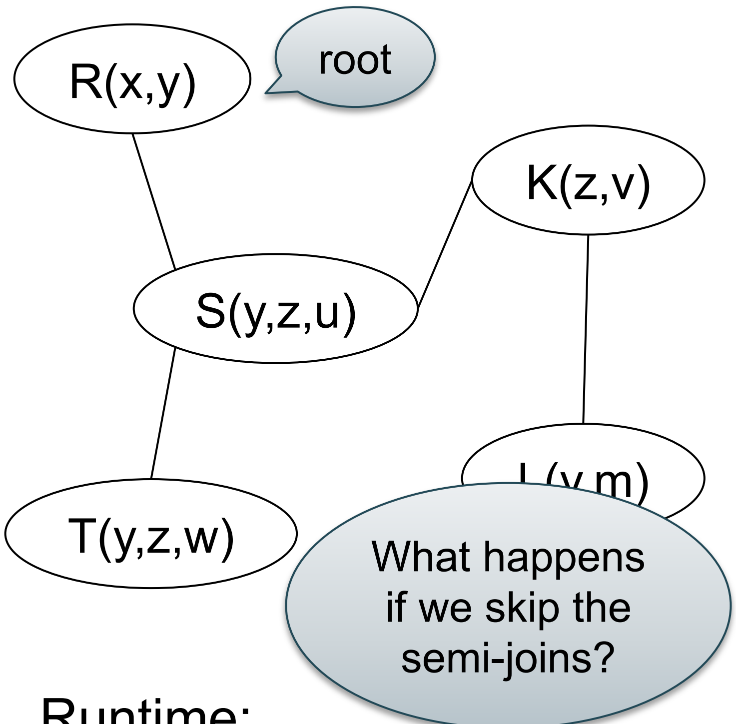


Runtime:

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

Example: Full CQ

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



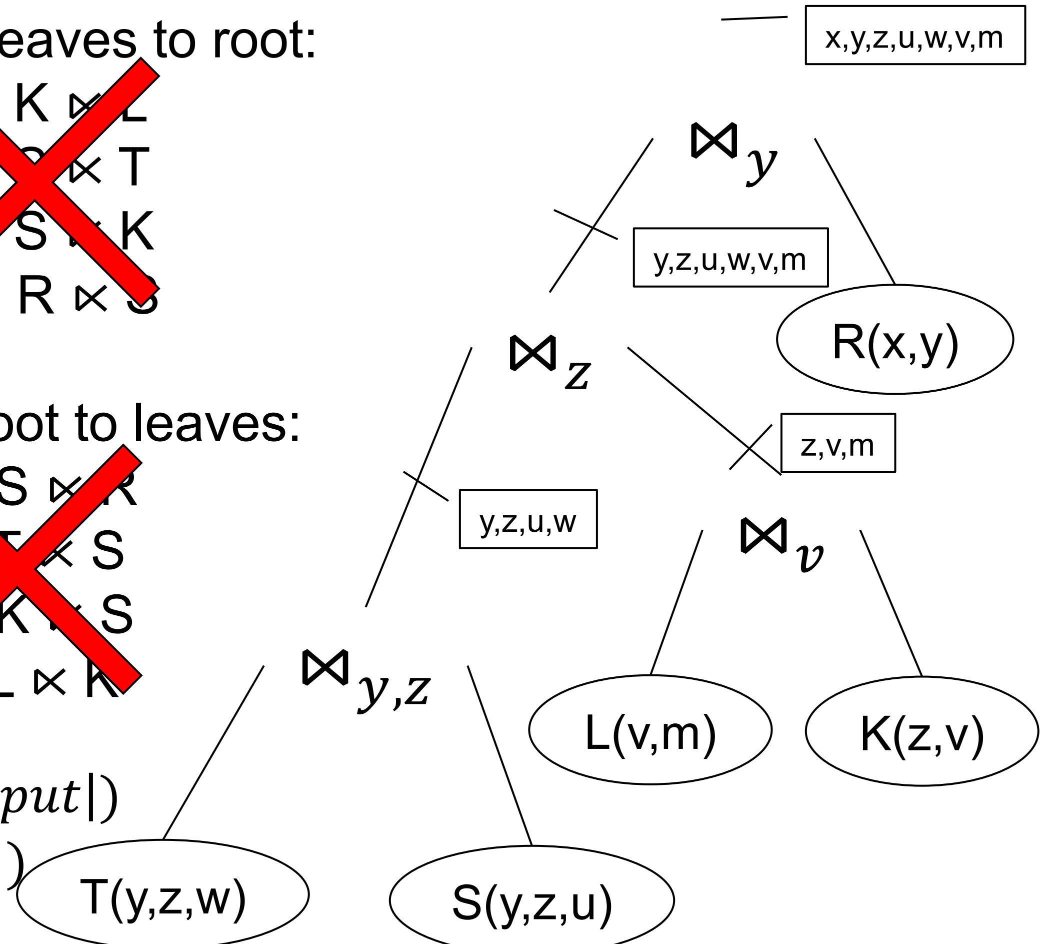
-- Leaves to root:

~~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)



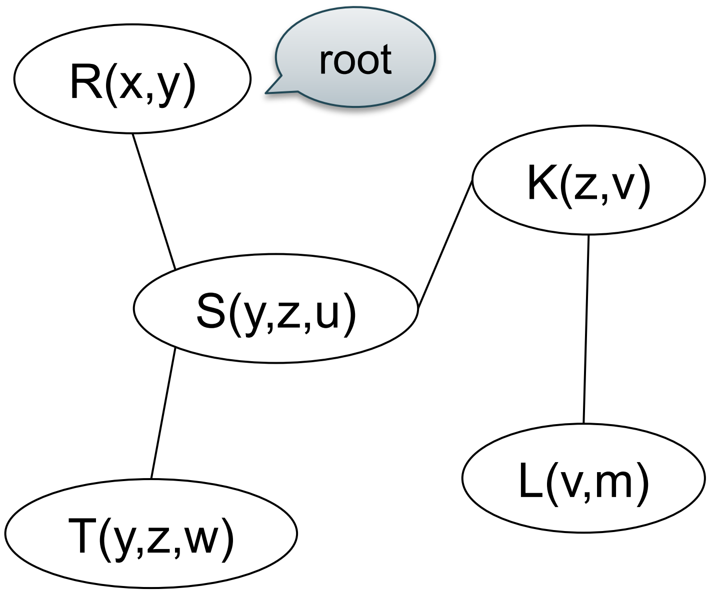
Runtime:

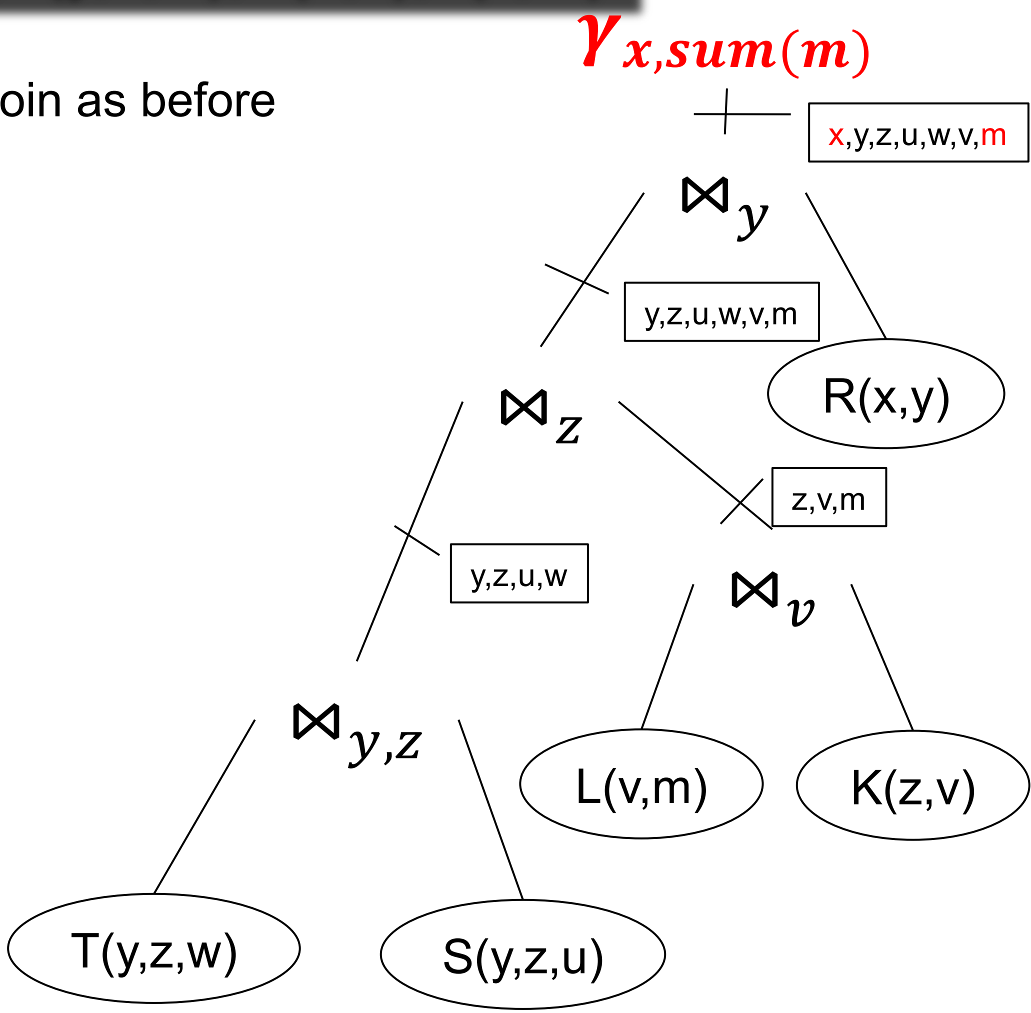
- Every semi-join takes time $\tilde{O}(|Input|)$
- Every join takes time $\tilde{O}(|Output|)$

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)

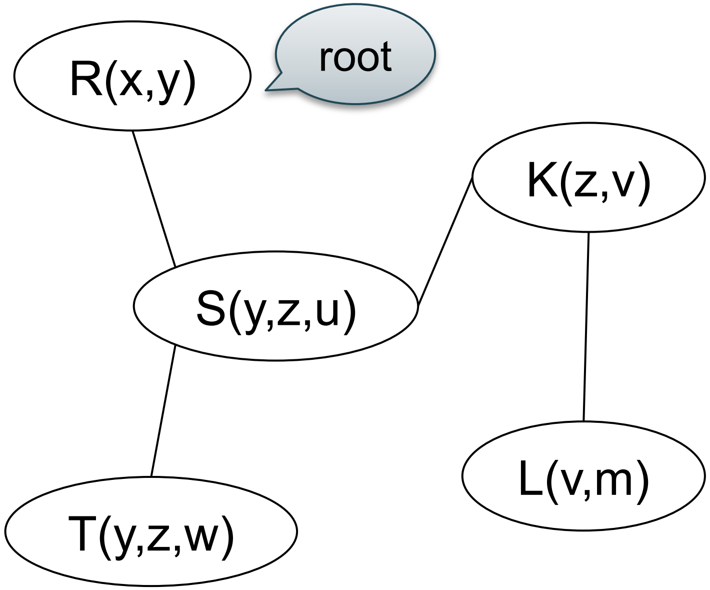
Semi-join as before



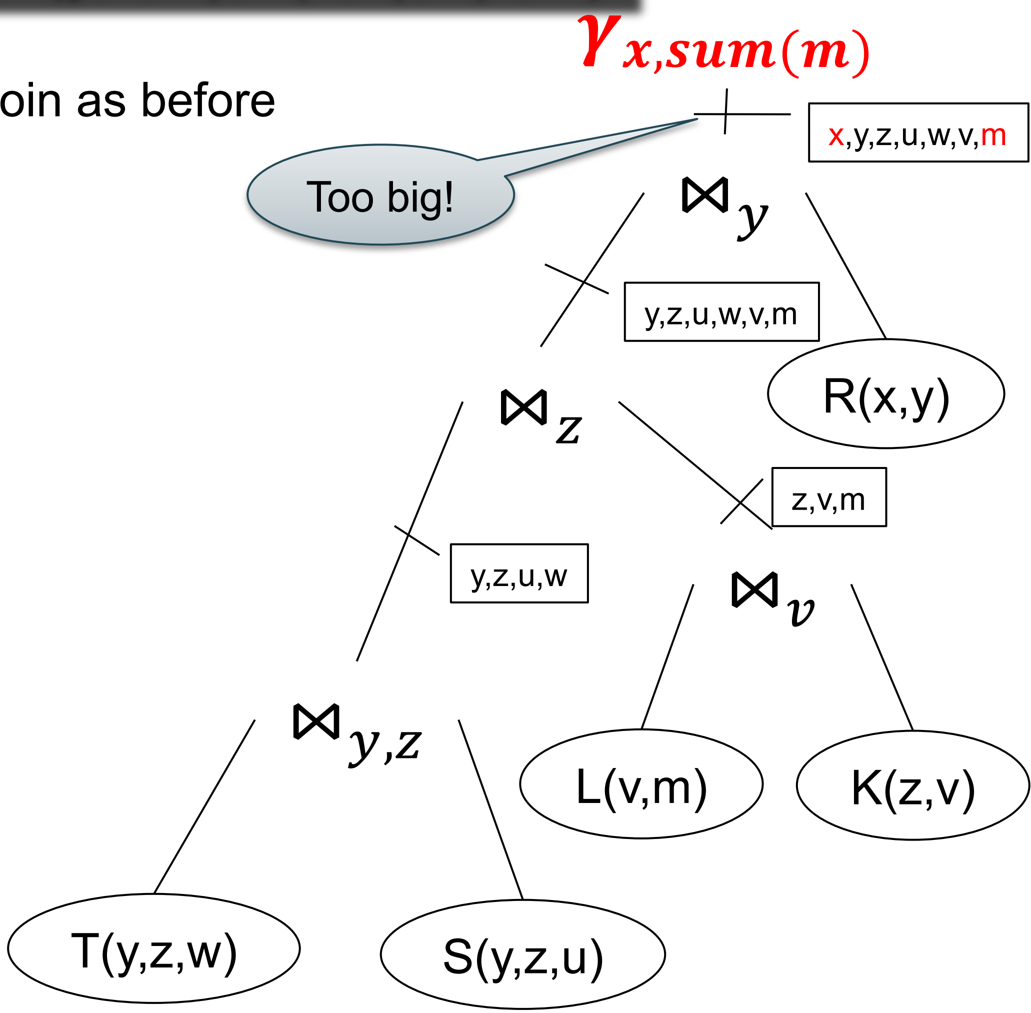


Example: CQ with Aggregates

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

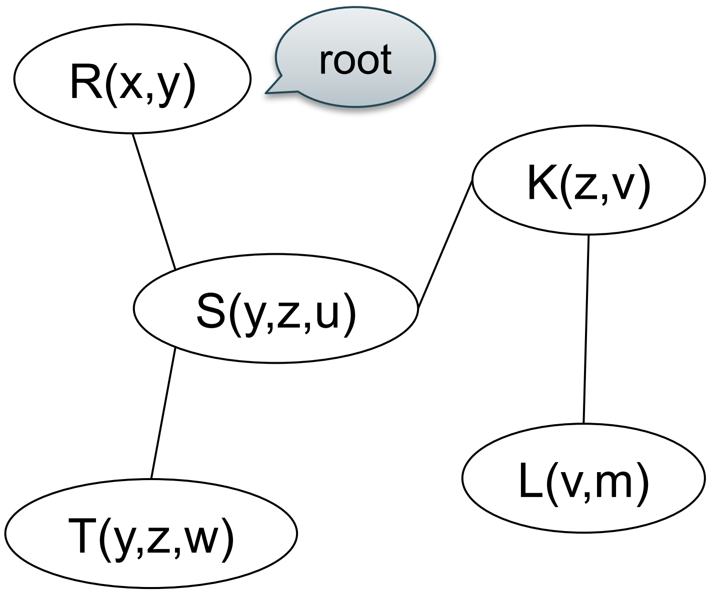


Semi-join as before

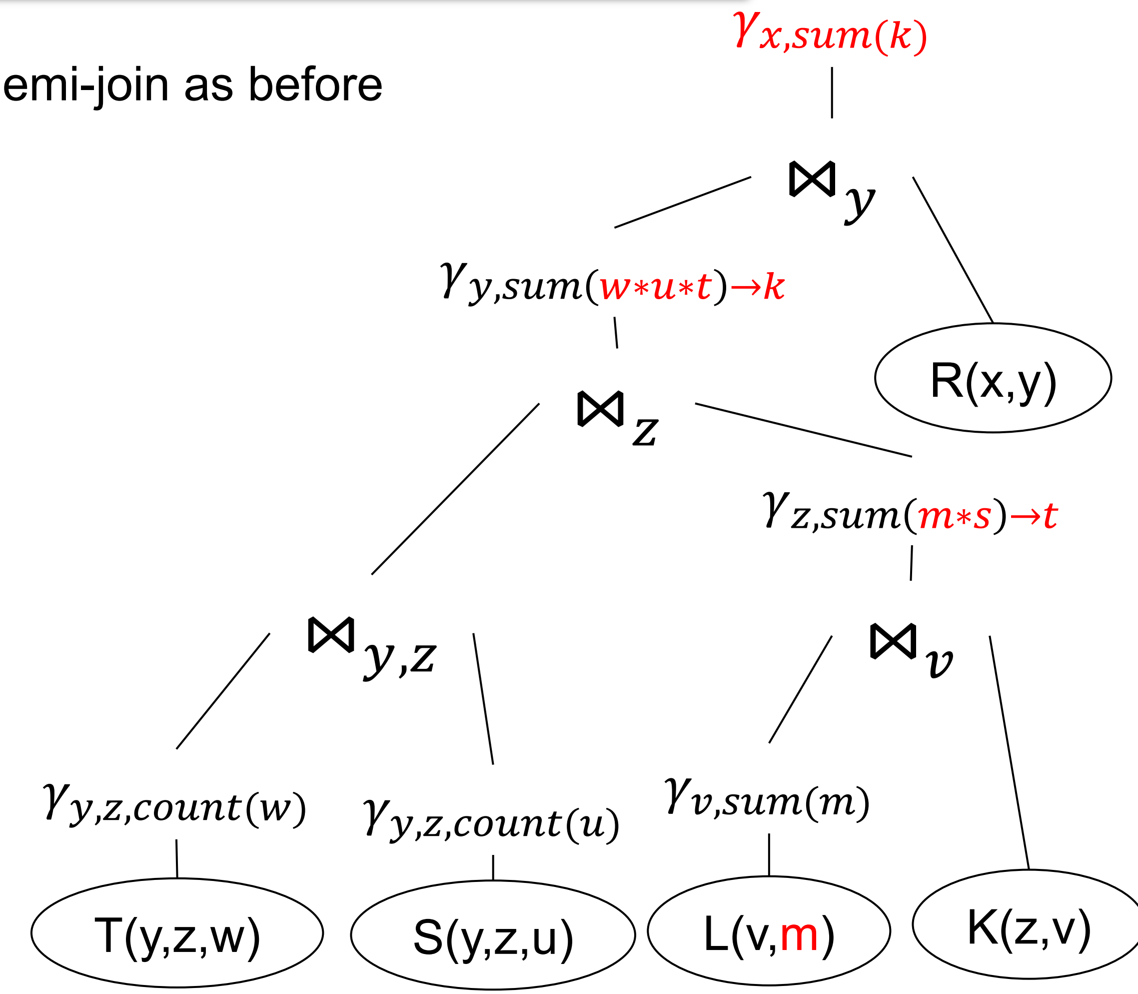


Example: CQ with Aggregates

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

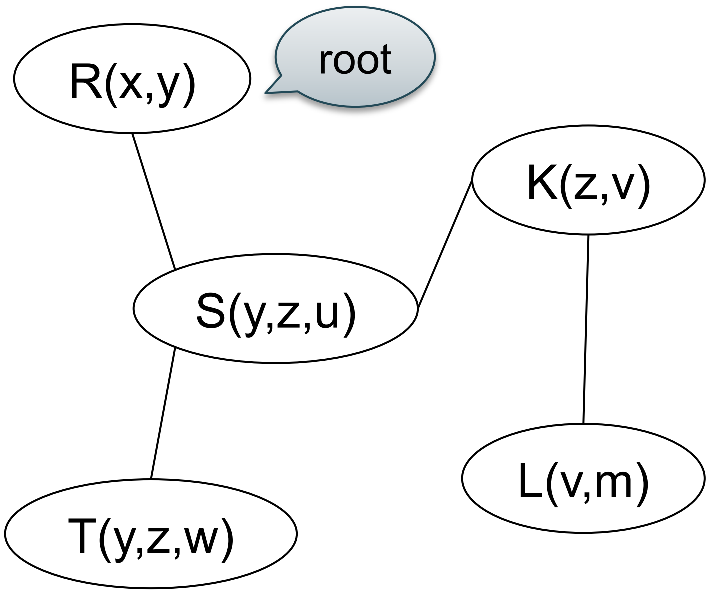


Semi-join as before

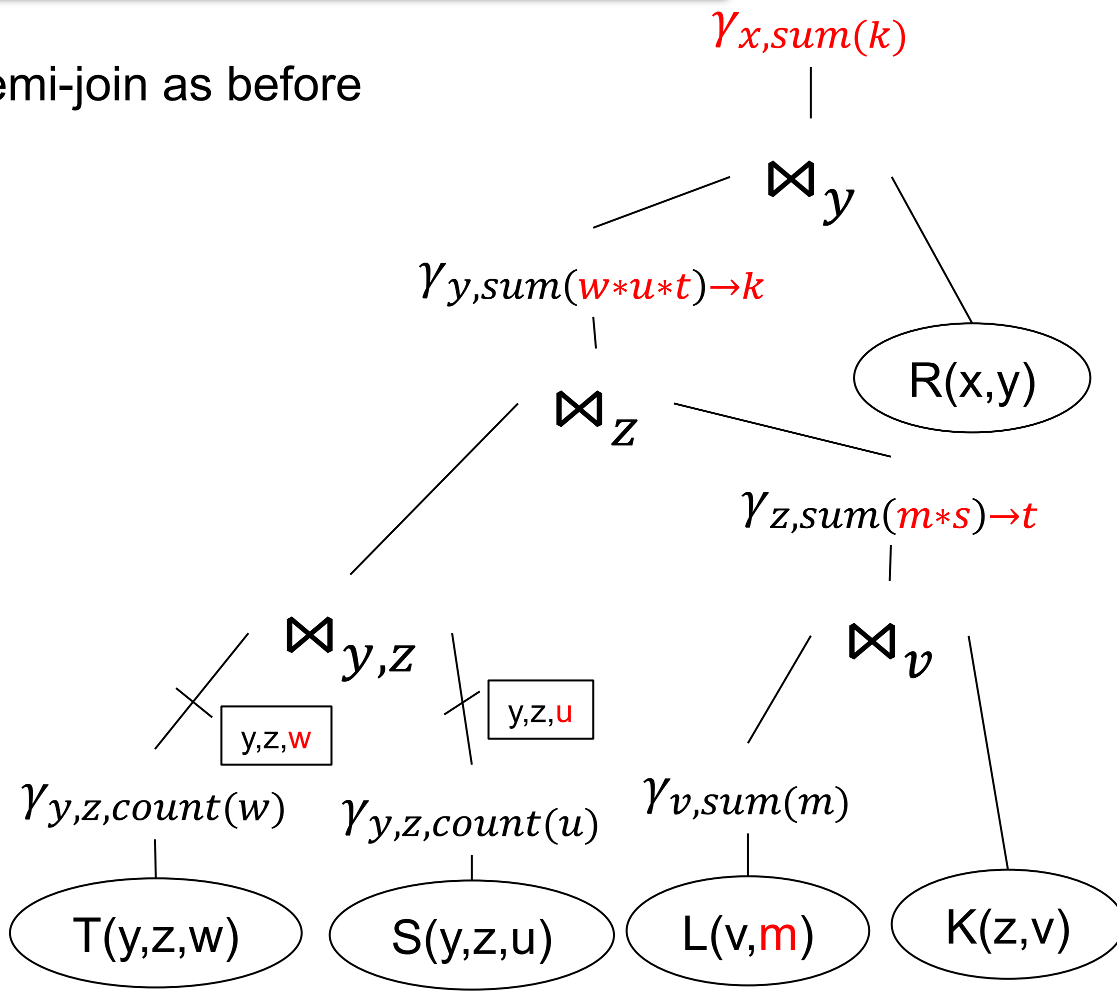


Example: CQ with Aggregates

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

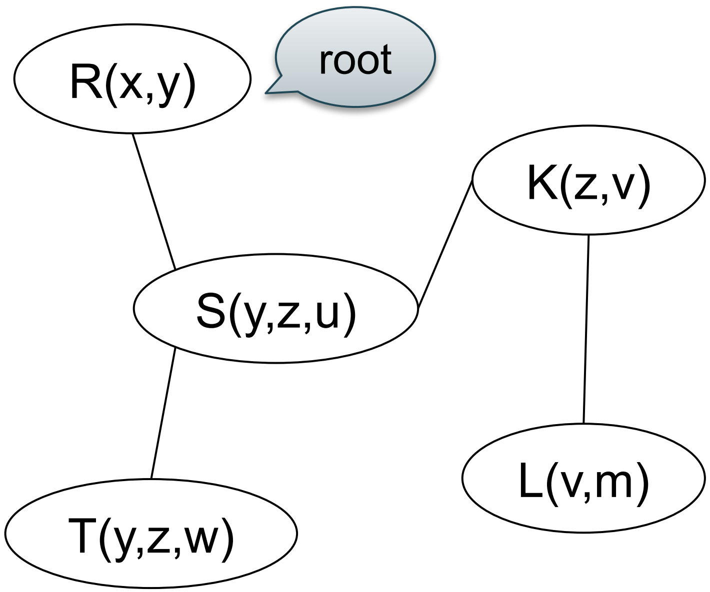


Semi-join as before

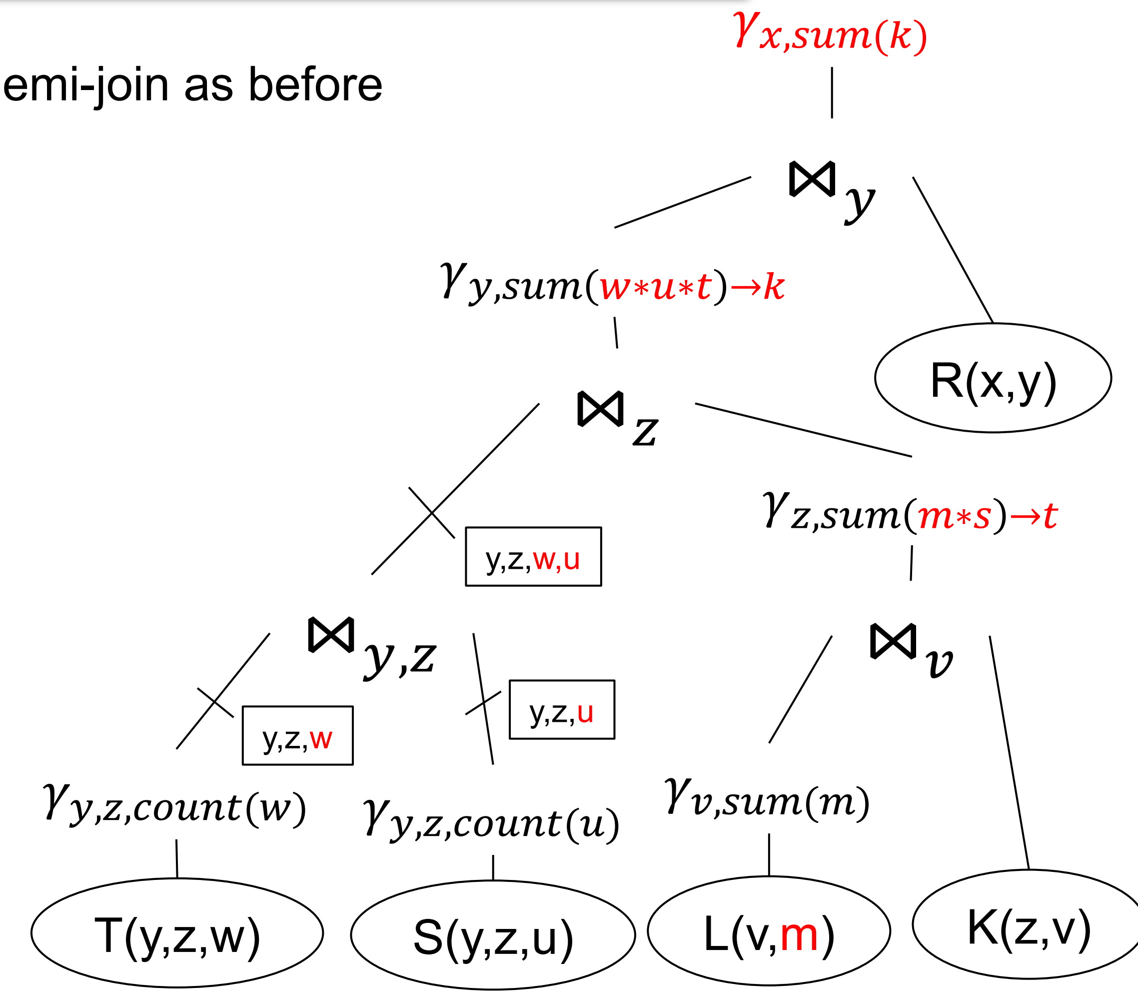


Example: CQ with Aggregates

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

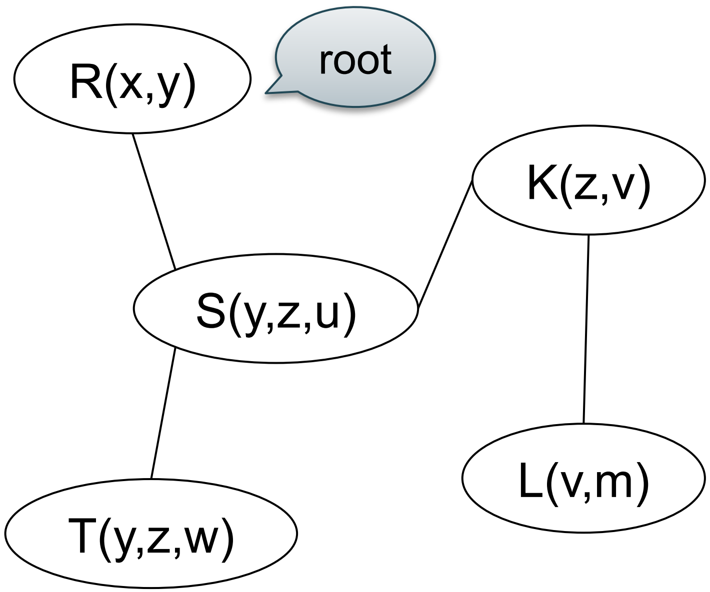


Semi-join as before

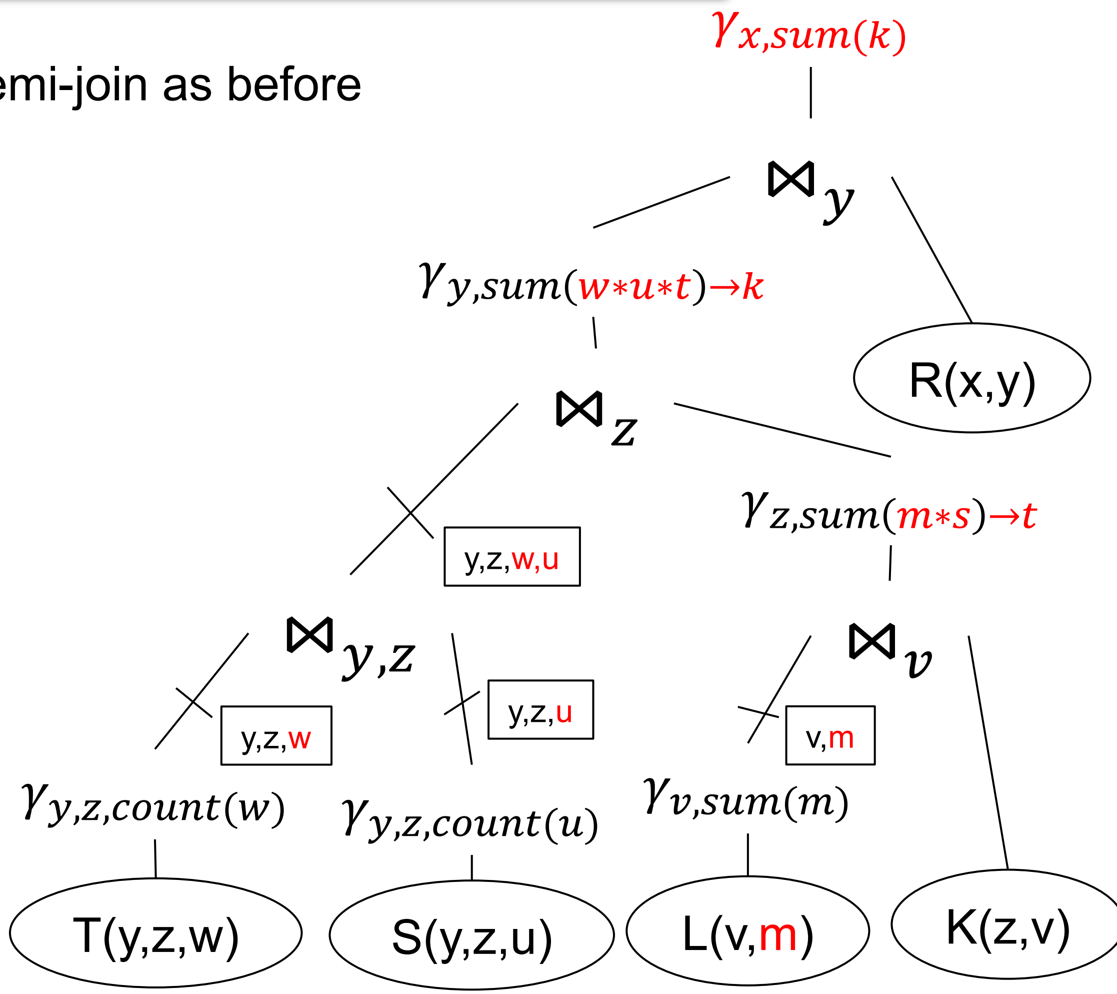


Example: CQ with Aggregates

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

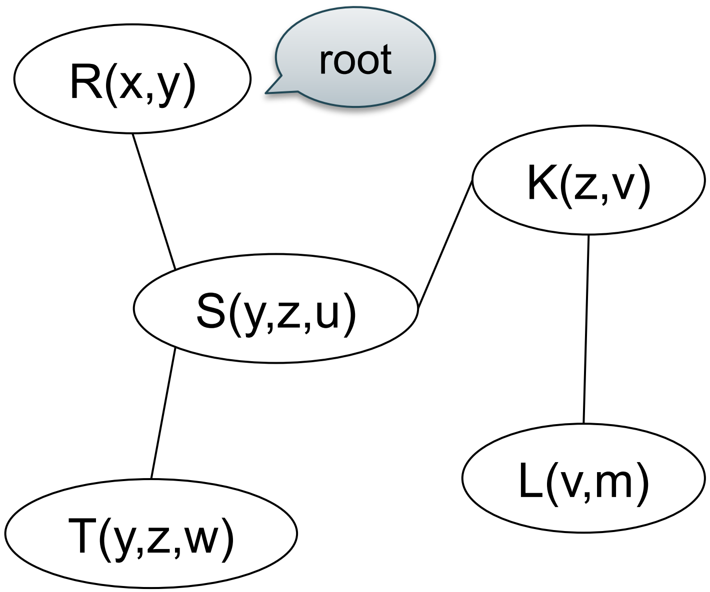


Semi-join as before

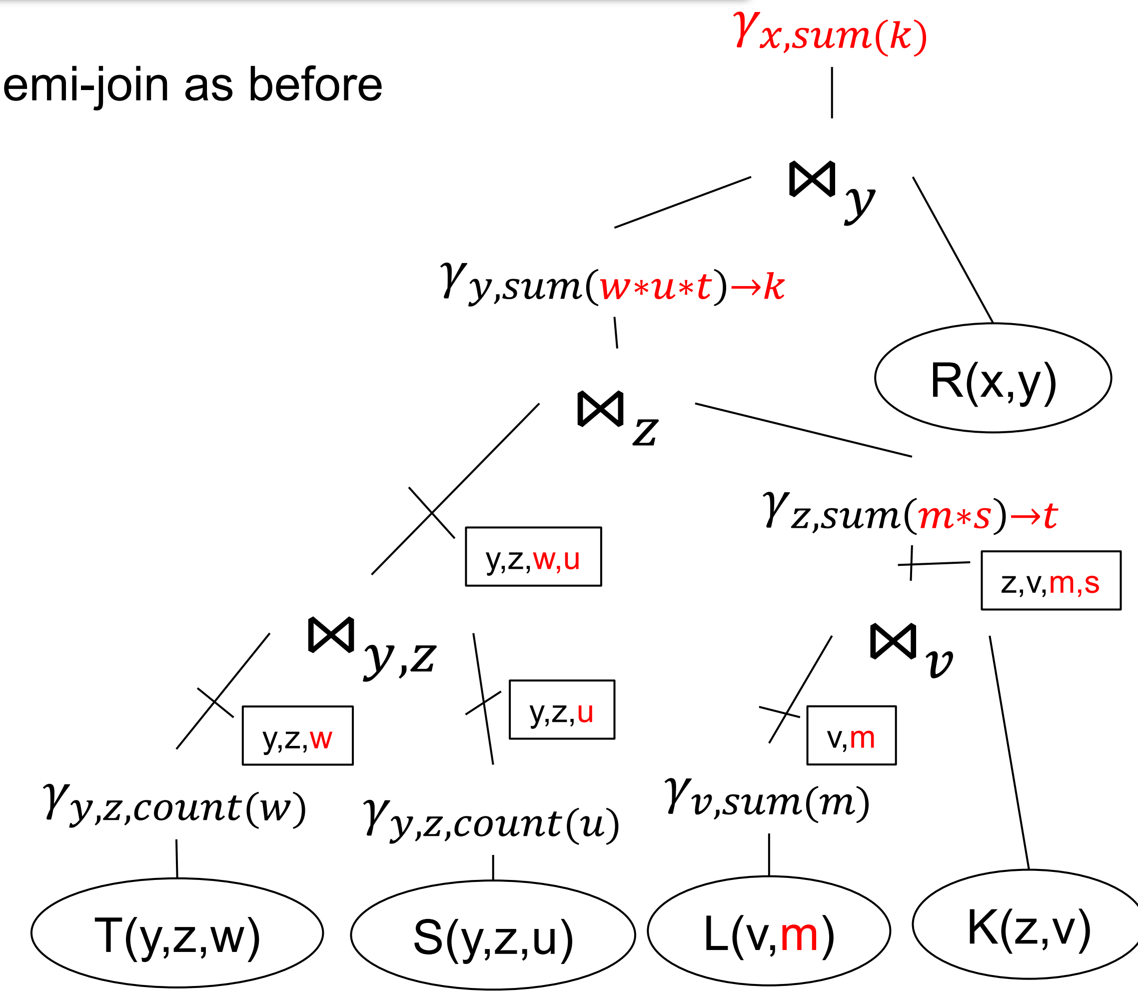


Example: CQ with Aggregates

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

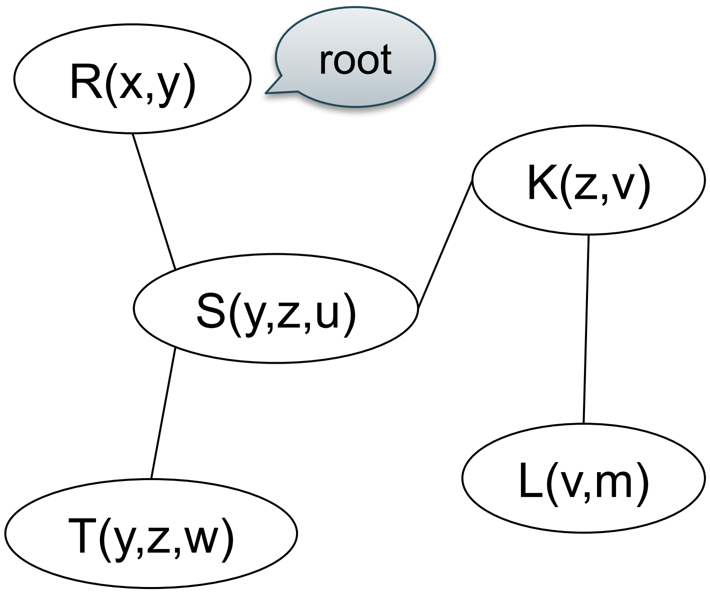


Semi-join as before

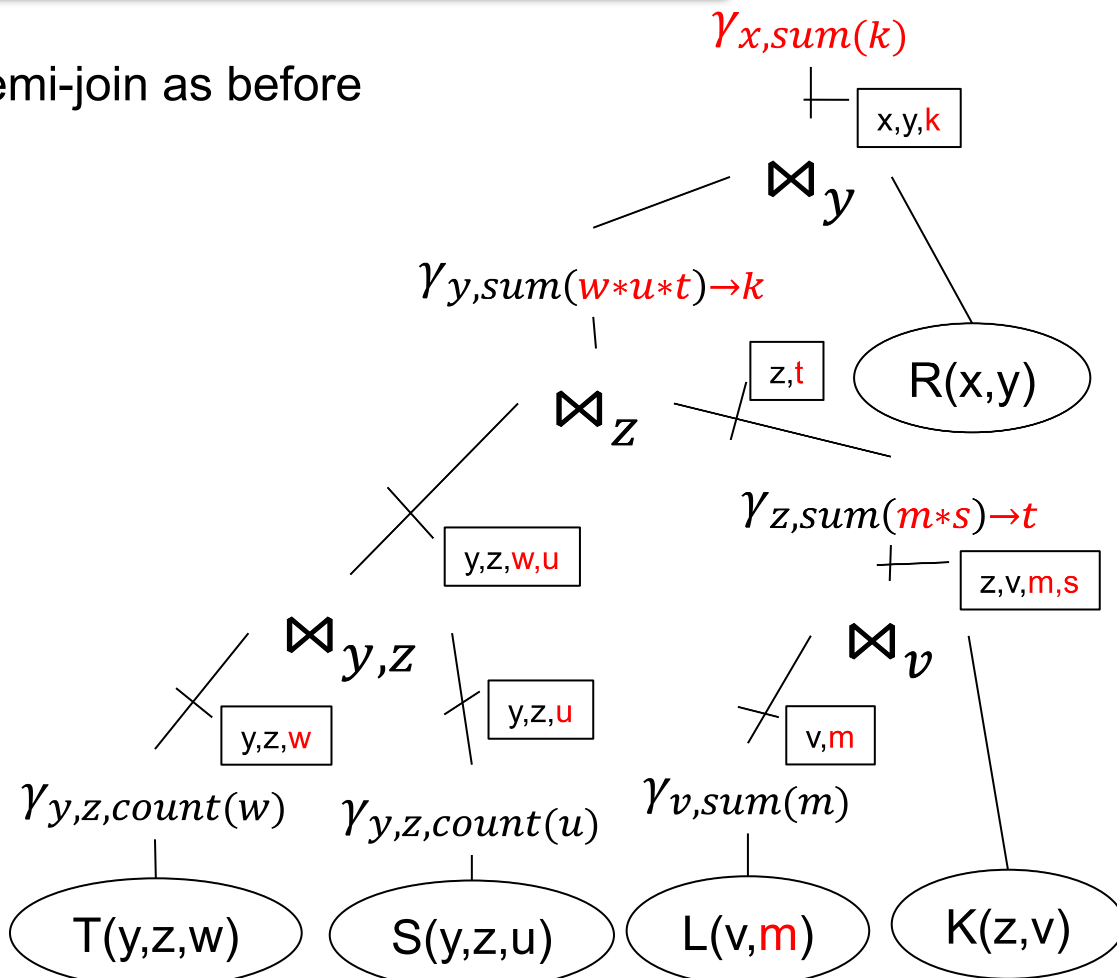


Example: CQ with Aggregates

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

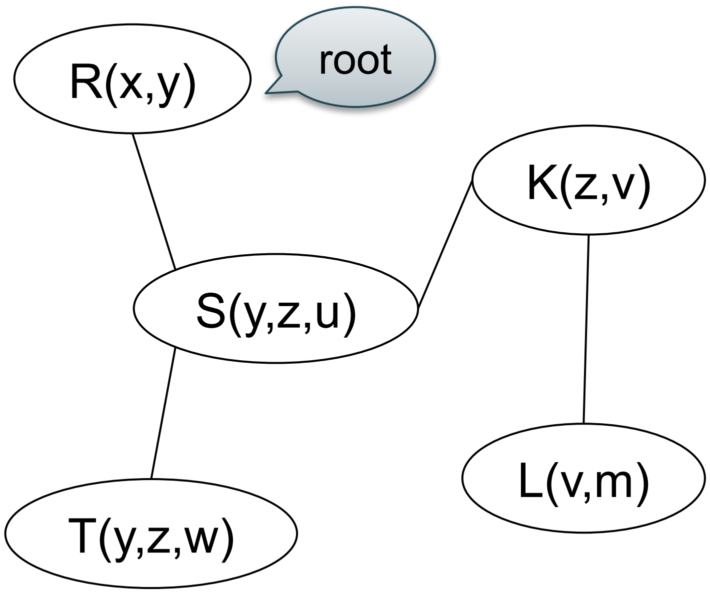


Semi-join as before

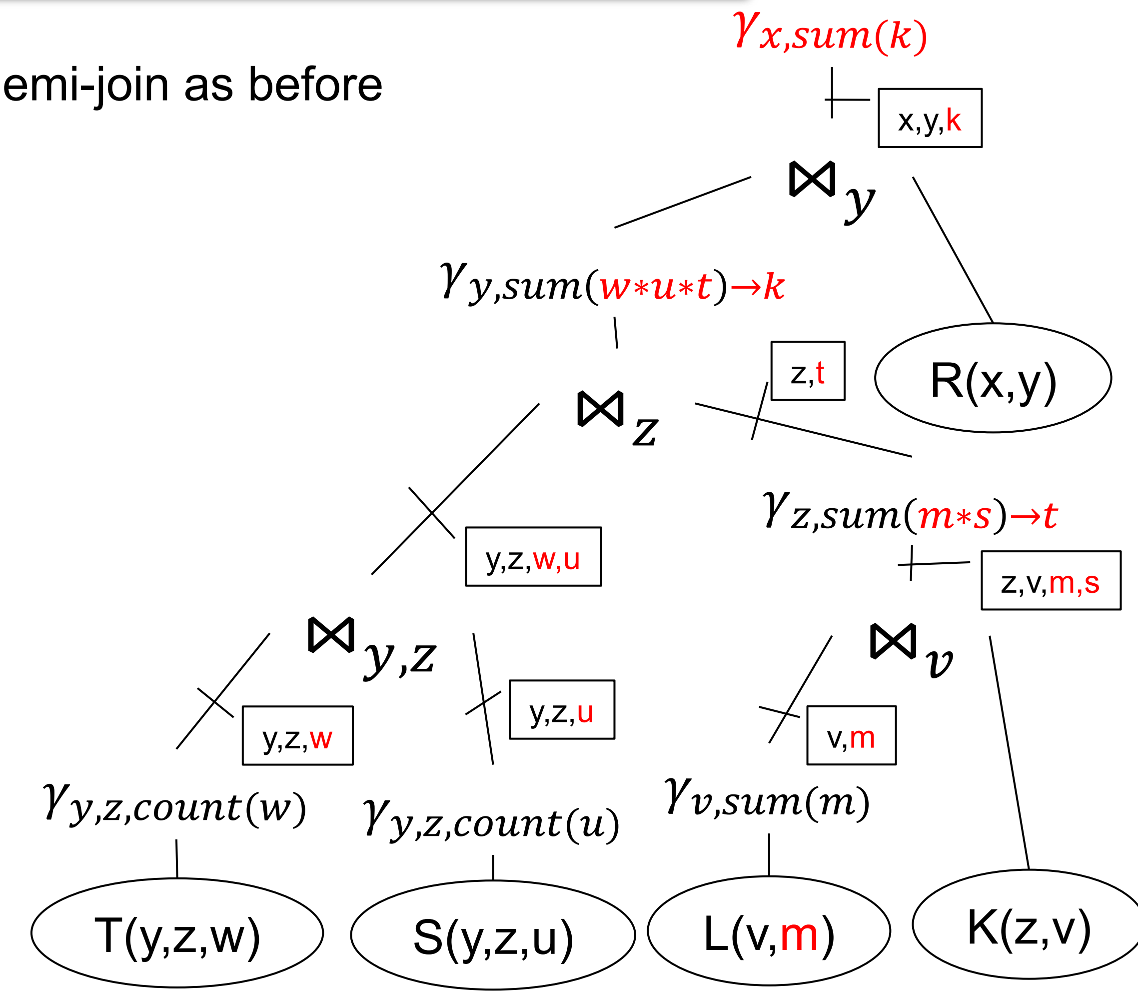


Example: CQ with Aggregates

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

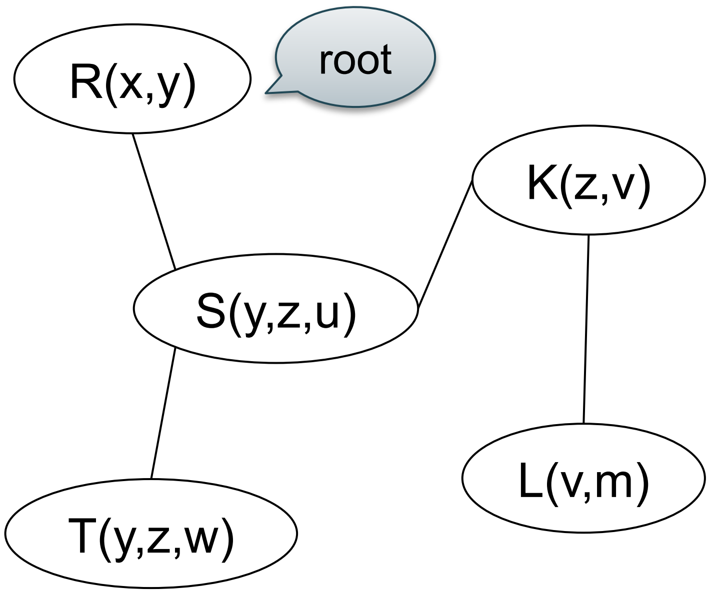


Runtime:

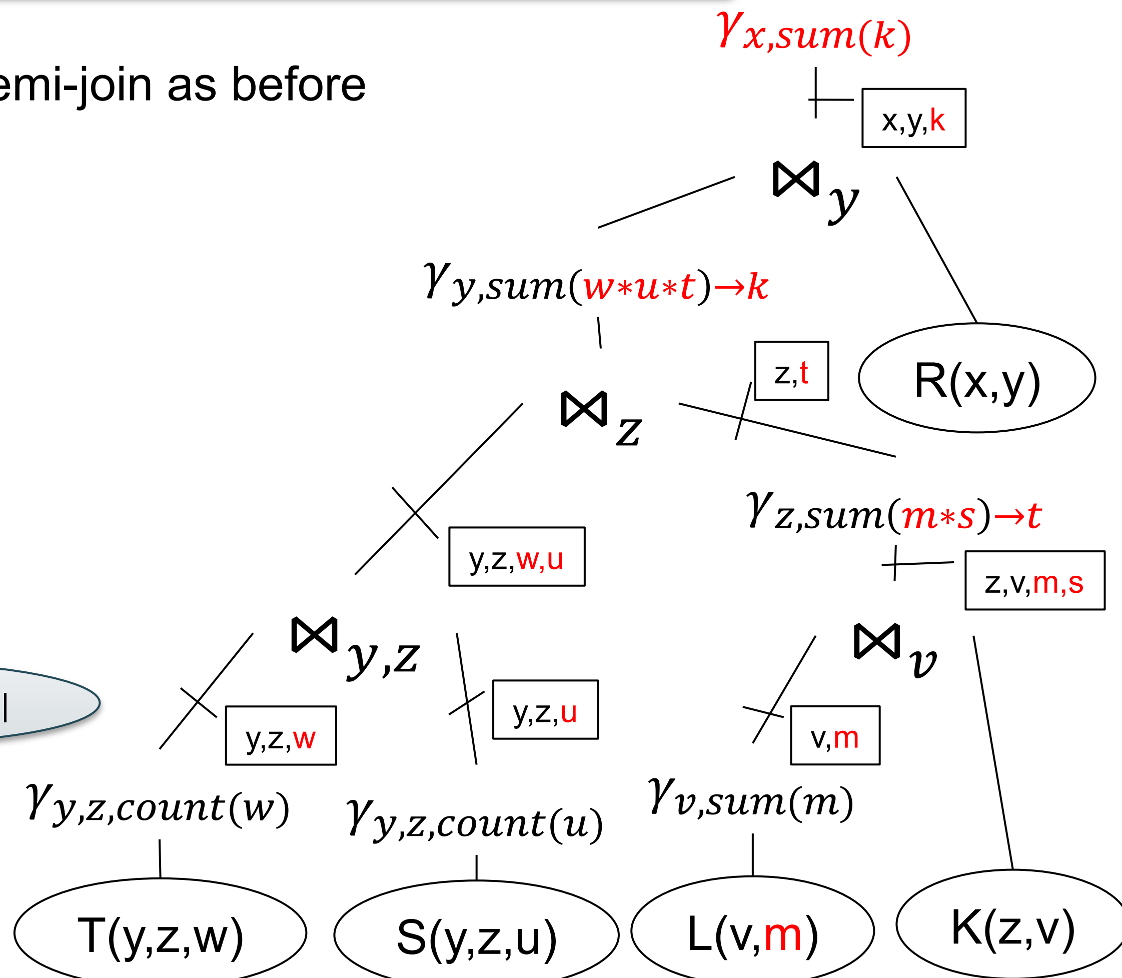
- Semi-joins: $\tilde{O}(|Input|)$
- Join/group-by: $\tilde{O}(|Input|)$

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

What about group-by multiple attributes?

Q(x,z,m,sum(w)) :- ….

- Can apply the same principle, but runtime may be polynomial in Input or Output

Discussion

What is the query is disconnected?

```
SELECT count(*)
FROM Author, Publication;
```

```
SELECT x.firstName, y.year, count(*)
FROM Author x, Publication y
GROUP By x.firstName, y.year;
```

- Simply compute each connected component separately, then take their cartesian product, or regular product, as needed.

Discussion

Which join order do we choose?

- Yannakakis algorithm doesn't specify: a*ny* join order ensures runtime is:
$$\tilde{O}(|Input| + |Output|)$$
- BUT: join order may impact the constant significantly, and in practice that matters

Discussion

- Some acyclic queries have more than one join tree, and each tree has several join orders
- Example: Q(x) = R(x), S(x), T(x)

Discussion

- Database optimizers rarely do semi-join reduction
  - When they do, they sometimes call it a *magic set optimization* (we'll explain next)
- Reason: when semi-join is ineffective, then it increases cost by a factor of 3

Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated

Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated

```
T(x,y) :- Parent(x,y)
T(x,y) :- T(x,z),Parent(z,y)
Q(y)   :- T('Alice',y)
```

Discussion

- Magic set optimizations
- Semi-join reductions can also be applied to recursive datalog program
- Called *magic set optimizations*; quite complicated

```
T(x,y) :- Parent(x,y)
T(x,y) :- T(x,z),Parent(z,y)
Q(y)   :- T('Alice',y)
```

```
Q(y) :- Parent('Alice',y)
Q(y) :- Q(x),Parent(x,y)
```

Discussion

- A *full reducer* for Q is a sequence of semi-joins after which every tuple contributes to at least one answer

- **Theorem**. Q has a full reducer iff it is acyclic
- **Proof**: if Q is acyclic, then Yannakakis' algorithm.
  If Q is cyclic, assume w.l.o.g.:

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

(show in class that it has no full reducer)

Testing if Q is Acyclic

An *ear* of Q is an atom R(X) with the following property:

- Let $X' \subseteq X$ be the set of join variables (occurring some other atom)
- There exists some other atom S(Y) such that $X' \subseteq Y$

Testing if Q is Acyclic

An *ear* of Q is an atom R(X) with the following property:

- Let X' ⊆ X be the set of join variables (occurring some other atom)
- There exists some other atom S(Y) such that X' ⊆ Y

GYO algorithm (Graham,Yu,Özsoyoğlu) for acyclicity:

- While Q has an ear R(X), remove R(X) from Q
- If all atoms were removed, then Q is acyclic
- If atoms remain but there is no ear, then Q is cyclic

Outline

- Acyclic queries, Yannakakis algorithm
- Tree decomposition of cyclic queries
- Worst-case optimal algorithm; next week

Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

Tree Decomposition

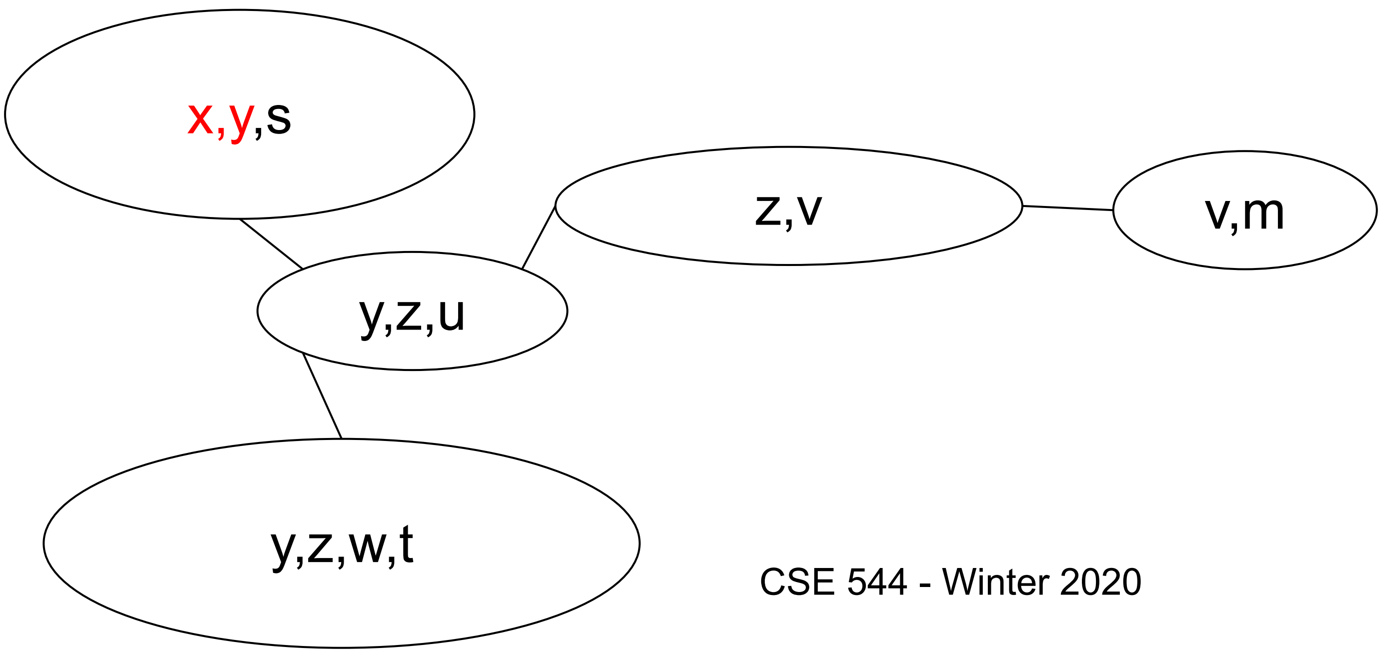
Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) ∀A∈Atoms(Q) ∃t∈ Nodes(T), Vars(A) ⊆ χ(t)
(2) ∀x∈Vars(Q), {t | x ∈χ(t)} is connected

Q(x,...,m) = R(x,y)∧A(y,s)∧B(x,s)∧S(y,z,u)∧T(y,z,w)∧C(z,w,t)∧D(w,t,y)
∧E(t,y,z)∧K(z,v)∧F(z,v)∧L(v,m)

Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{Vars(Q)}$ s.t.:
(1) ∀A∈Atoms(Q) ∃t∈ Nodes(T), Vars(A) ⊆ χ(t)
(2) ∀x∈Vars(Q), {t | x ∈χ(t)} is connected

Q(x,...,m) = R(x,y)∧A(y,s)∧B(x,s)∧S(y,z,u)∧T(y,z,w)∧C(z,w,t)∧D(w,t,y)
∧E(t,y,z)∧K(z,v)∧F(z,v)∧L(v,m)

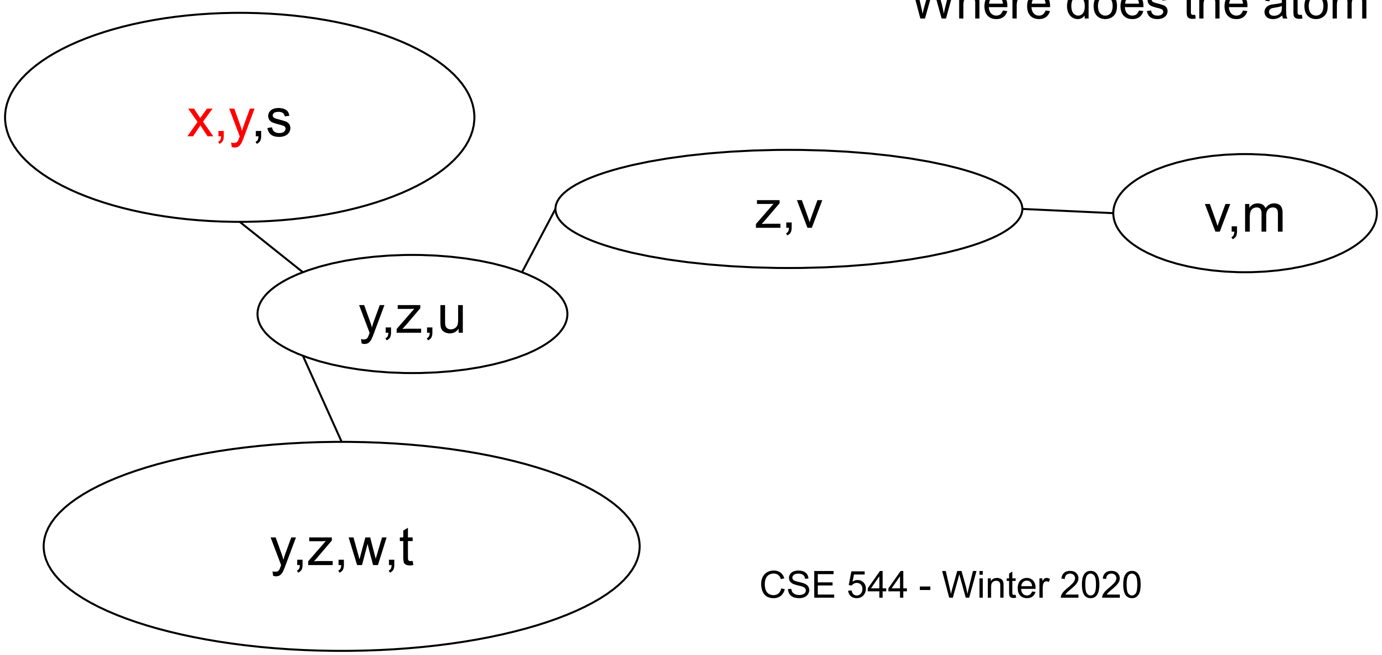


Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)$\rightarrow 2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

$Q(x,\ldots,m) = R(x,y) \wedge A(y,s) \wedge B(x,s) \wedge S(y,z,u) \wedge T(y,z,w) \wedge C(z,w,t) \wedge D(w,t,y) \wedge E(t,y,z) \wedge K(z,v) \wedge F(z,v) \wedge L(v,m)$

Where does the atom R(x,y) occur?

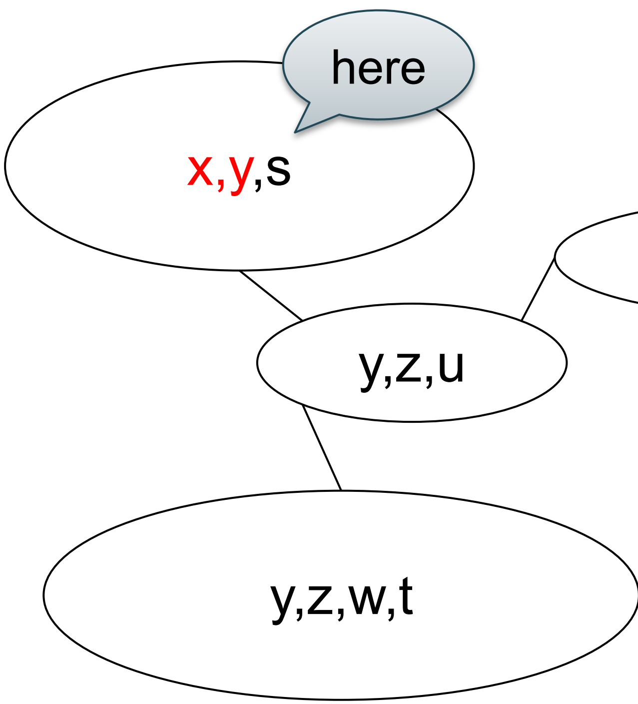


Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{Vars(Q)}$ s.t.:
(1) ∀A∈Atoms(Q) ∃t∈ Nodes(T), Vars(A) ⊆ χ(t)
(2) ∀x∈Vars(Q), {t | x ∈χ(t)} is connected

Q(x,...,m) = R(x,y)∧A(y,s)∧B(x,s)∧S(y,z,u)∧T(y,z,w)∧C(z,w,t)∧D(w,t,y)
∧E(t,y,z)∧K(z,v)∧F(z,v)∧L(v,m)

Where does the atom R(x,y) occur?

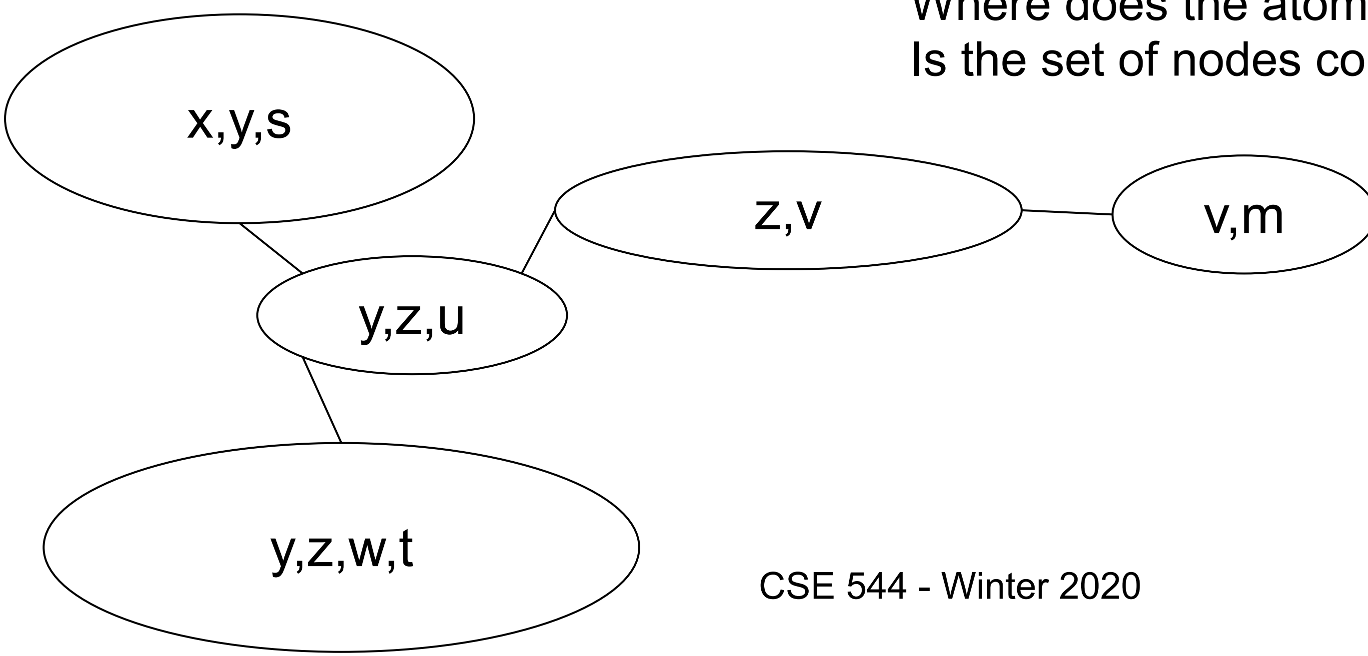


Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) ∀A∈Atoms(Q) ∃t∈ Nodes(T), Vars(A) ⊆ χ(t)
(2) ∀x∈Vars(Q), {t | x ∈χ(t)} is connected

Q(x,...,m) = R(x,y)∧A(y,s)∧B(x,s)∧S(y,z,u)∧T(y,z,w)∧C(z,w,t)∧D(w,t,y)
∧E(t,y,z)∧K(z,v)∧F(z,v)∧L(v,m)

Where does the atom R(x,y) occur?
Is the set of nodes containing z connected?

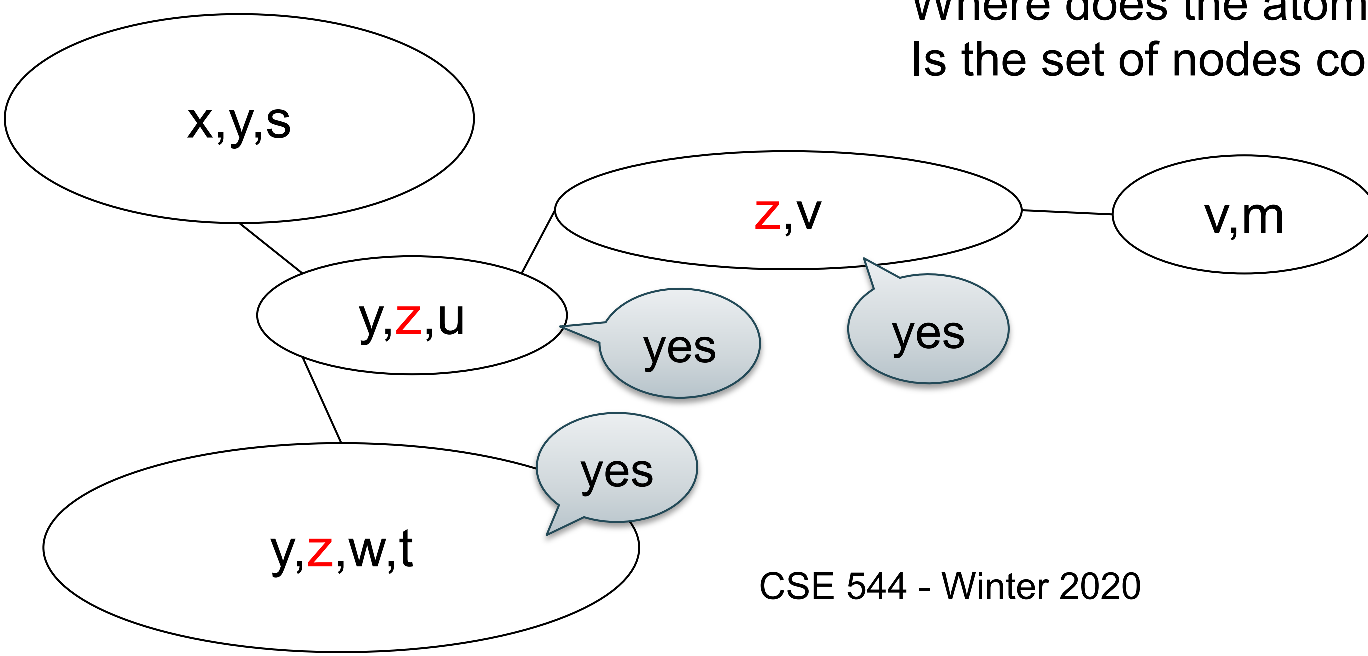


Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

Q(x,...,m) = R(x,y)∧A(y,s)∧B(x,s)∧S(y,z,u)∧T(y,z,w)∧C(z,w,t)∧D(w,t,y)
∧E(t,y,z)∧K(z,v)∧F(z,v)∧L(v,m)

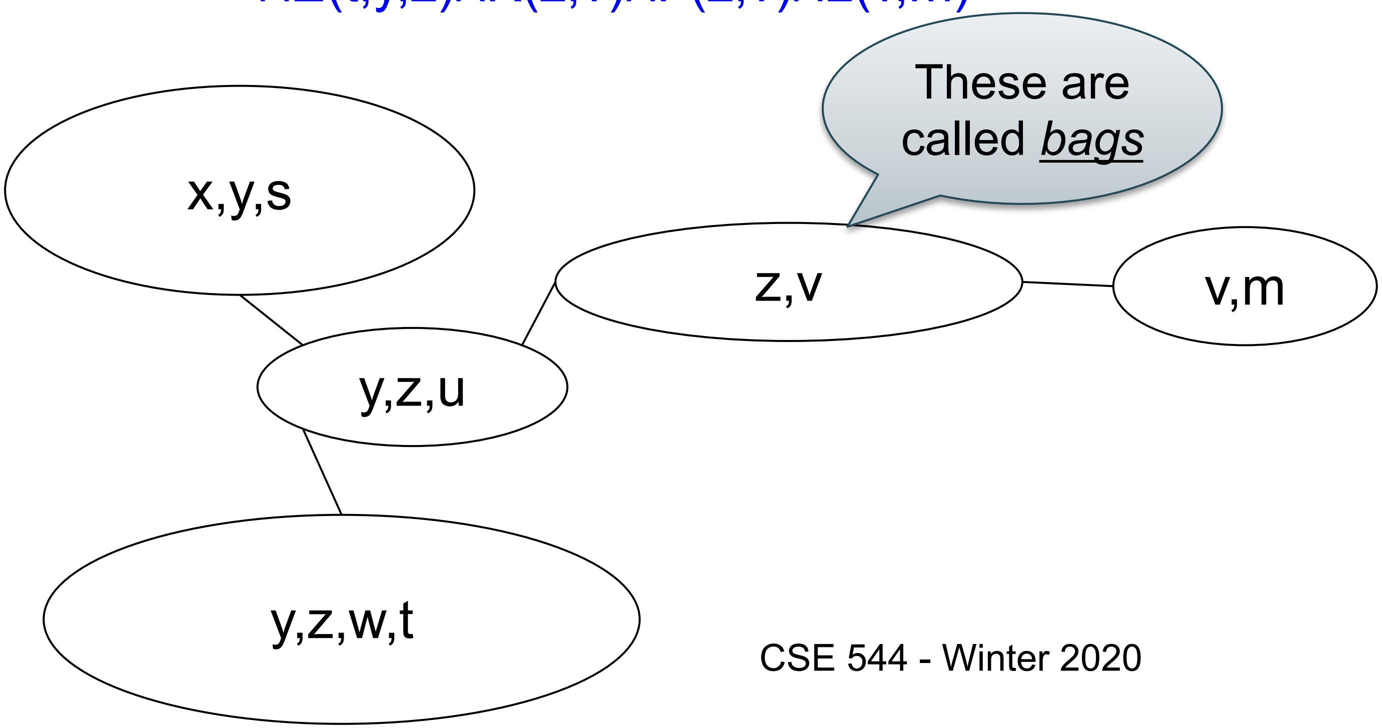
Where does the atom R(x,y) occur?
Is the set of nodes containing z connected?



Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

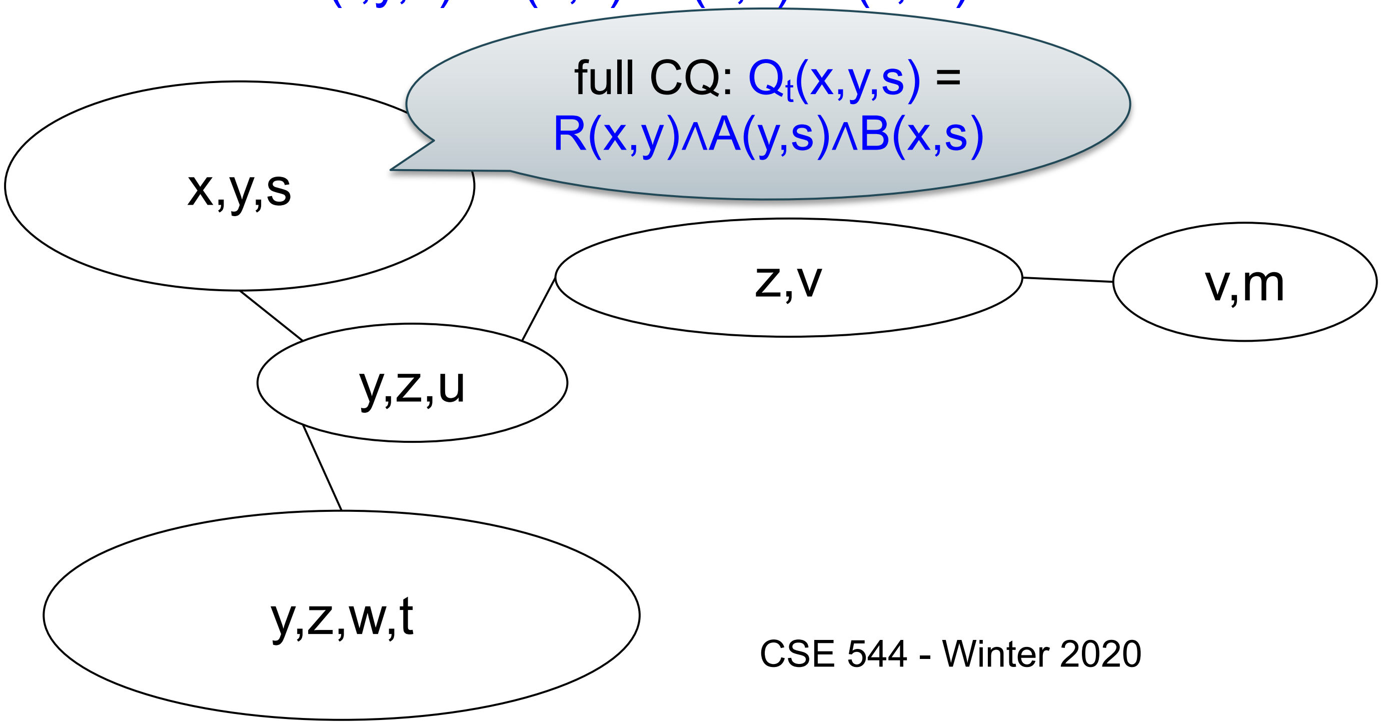
$Q(x,\ldots,m) = R(x,y) \wedge A(y,s) \wedge B(x,s) \wedge S(y,z,u) \wedge T(y,z,w) \wedge C(z,w,t) \wedge D(w,t,y)$
$\wedge E(t,y,z) \wedge K(z,v) \wedge F(z,v) \wedge L(v,m)$



Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)$\rightarrow 2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

$Q(x,\ldots,m) = R(x,y) \wedge A(y,s) \wedge B(x,s) \wedge S(y,z,u) \wedge T(y,z,w) \wedge C(z,w,t) \wedge D(w,t,y)$
$\wedge E(t,y,z) \wedge K(z,v) \wedge F(z,v) \wedge L(v,m)$



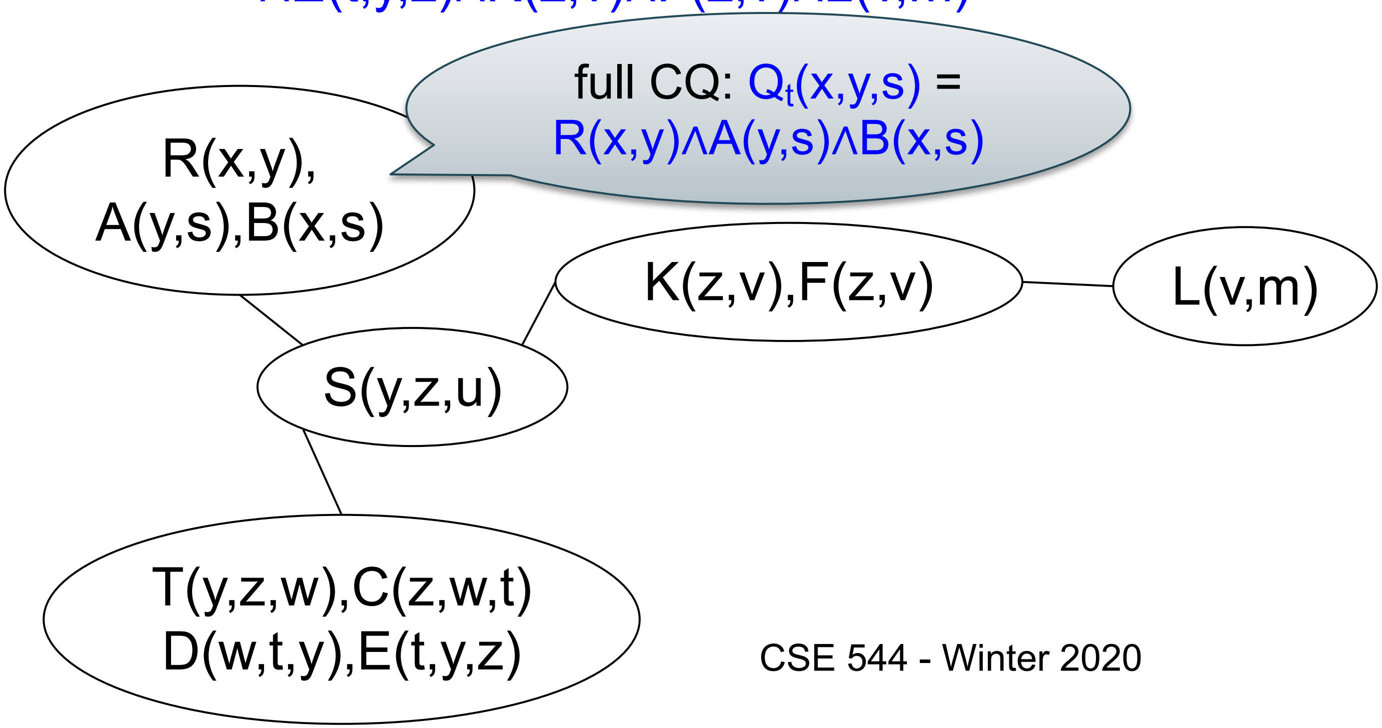
Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{Vars(Q)}$ s.t.:

(1) $\forall A \in Atoms(Q)\ \ \exists t \in Nodes(T),\ \ Vars(A) \subseteq \chi(t)$

(2) $\forall x \in Vars(Q)$, $\{t \mid x \in \chi(t)\}$ is connected

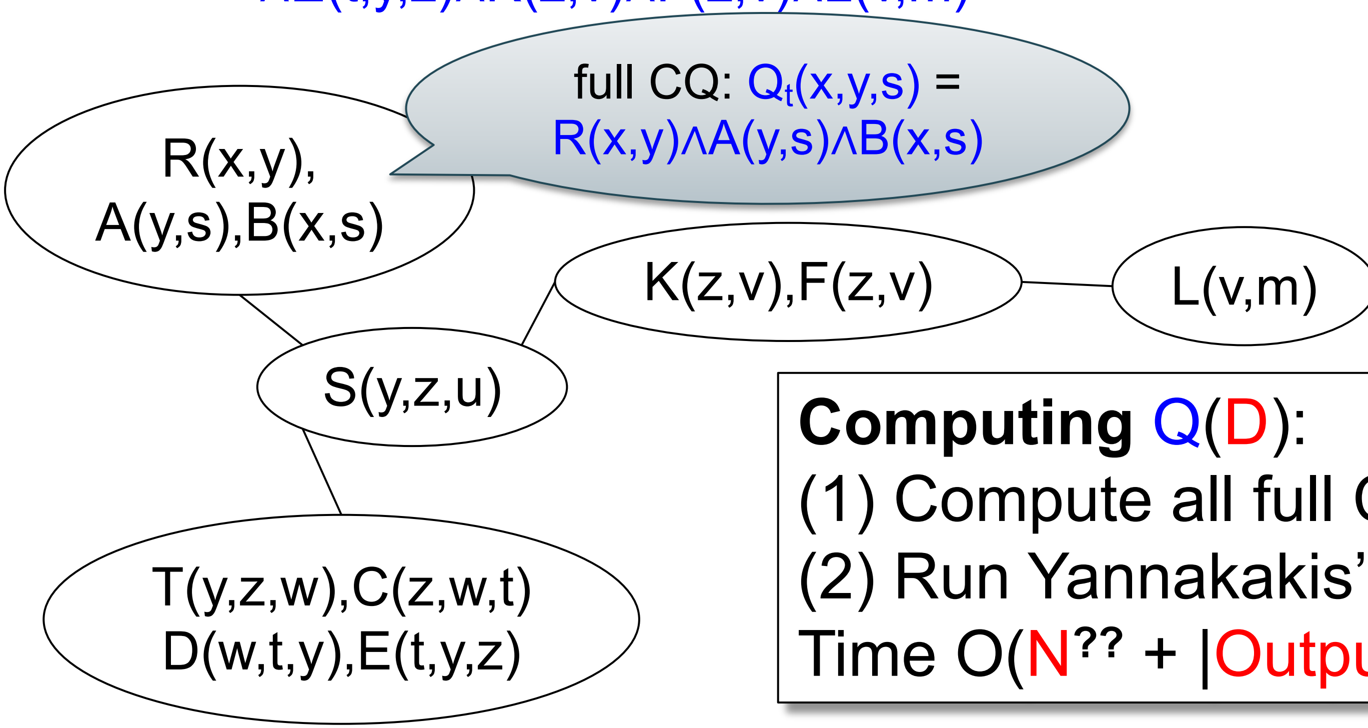
$Q(x,\ldots,m) = R(x,y) \wedge A(y,s) \wedge B(x,s) \wedge S(y,z,u) \wedge T(y,z,w) \wedge C(z,w,t) \wedge D(w,t,y) \wedge E(t,y,z) \wedge K(z,v) \wedge F(z,v) \wedge L(v,m)$



Tree Decomposition

Def *Tree decomposition* is (T, χ), χ:Nodes(T)→$2^{\text{Vars}(Q)}$ s.t.:
(1) $\forall A \in$ Atoms(Q) $\exists t \in$ Nodes(T), Vars(A) $\subseteq \chi(t)$
(2) $\forall x \in$ Vars(Q), $\{t \mid x \in \chi(t)\}$ is connected

$Q(x,\ldots,m) = R(x,y) \wedge A(y,s) \wedge B(x,s) \wedge S(y,z,u) \wedge T(y,z,w) \wedge C(z,w,t) \wedge D(w,t,y)$
$\wedge E(t,y,z) \wedge K(z,v) \wedge F(z,v) \wedge L(v,m)$



Computing Q(D):
(1) Compute all full CQ's Q_t
(2) Run Yannakakis' on the join tree
Time $O(N^{??} + |\text{Output}|)$

Recap

To compute a query Q proceed as follows

1. Find a tree decomposition of Q
2. For each tree node ("bag") compute its local query Q_t
3. Run Yannakakis on the resulting acyclic query

Runtime is dominated step 2

Will discuss step 2 next

Tree-width

Def $tw(Q) = \min_T \max_{t \in Nodes(T)} |\chi(t)| - 1$

Tree-width

This is the standard definition in graph theory

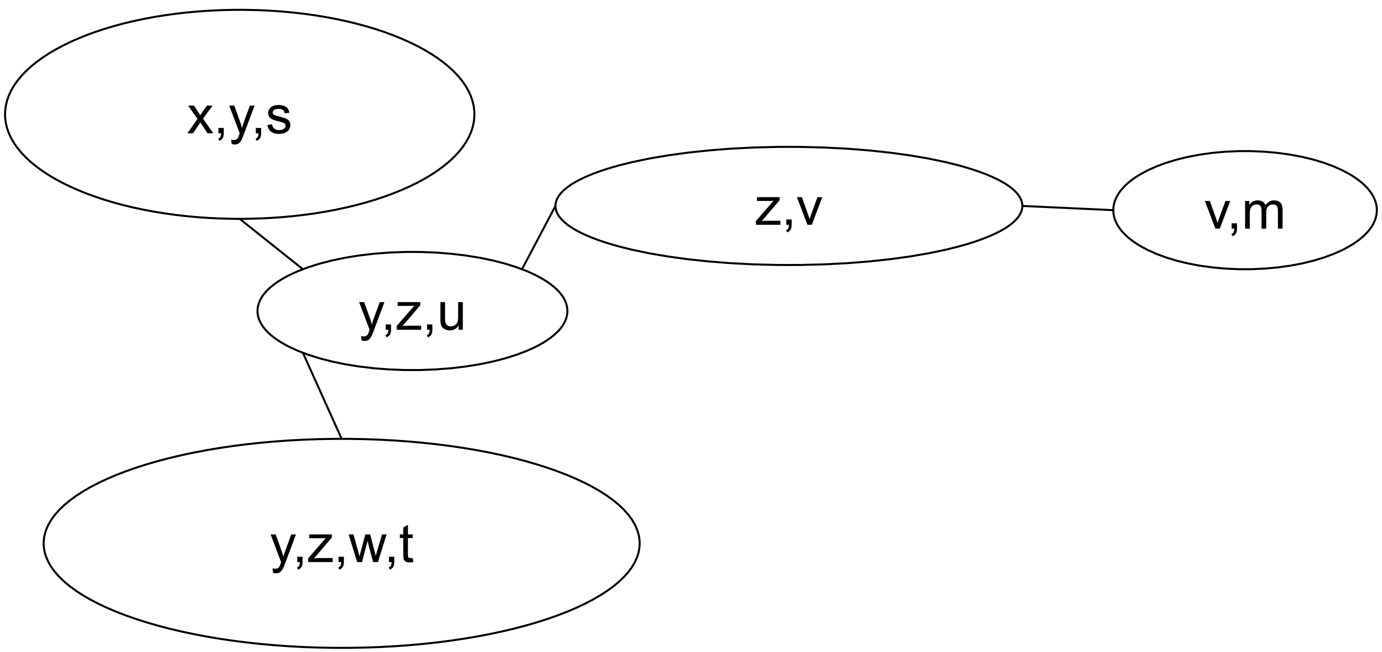
Def $tw(Q) = \min_T \max_{t \in Nodes(T)} |\chi(t)| - 1$

Tree-width

This is the standard definition in graph theory

Def $tw(Q) = \min_T \max_{t \in Nodes(T)} |\chi(t)| - 1$

$tw(Q) = 3$

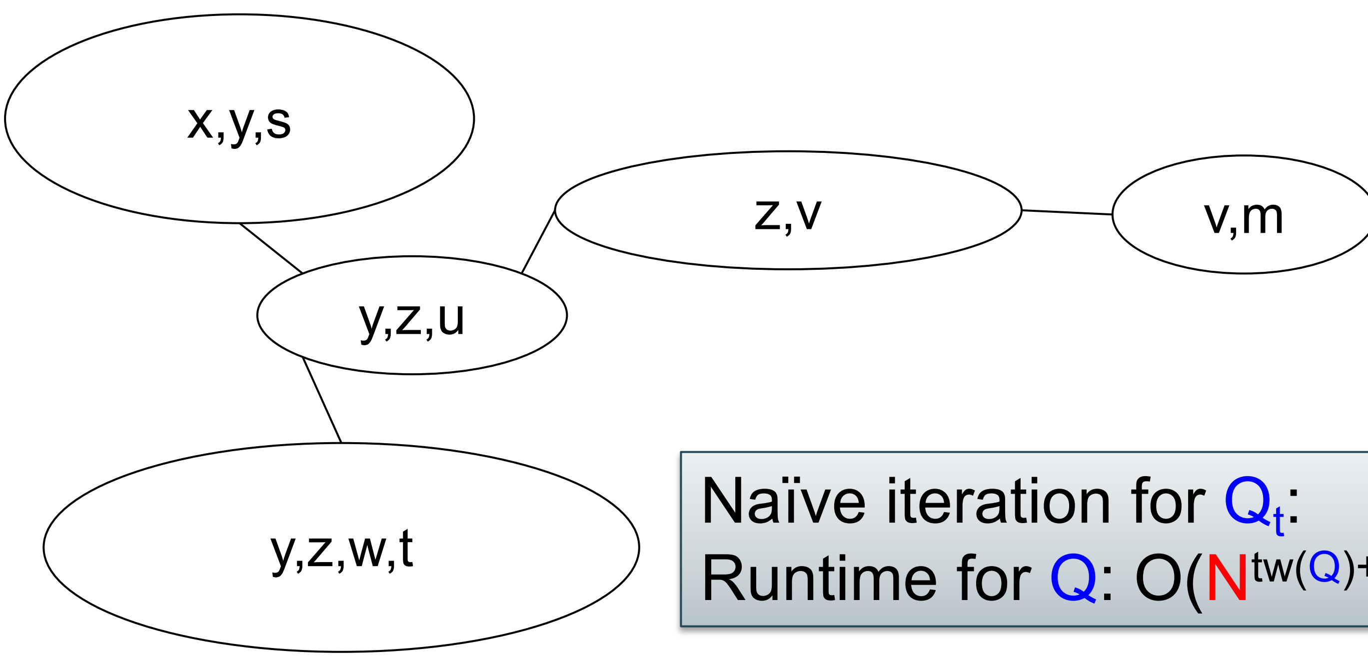


Tree-width

This is the standard definition in graph theory

Def tw(Q) = $\min_T \max_{t \in Nodes(T)} |\chi(t)| - 1$

tw(Q) = 3



Naïve iteration for Q_t:
Runtime for Q: $O(N^{tw(Q)+1} + |Output|)$

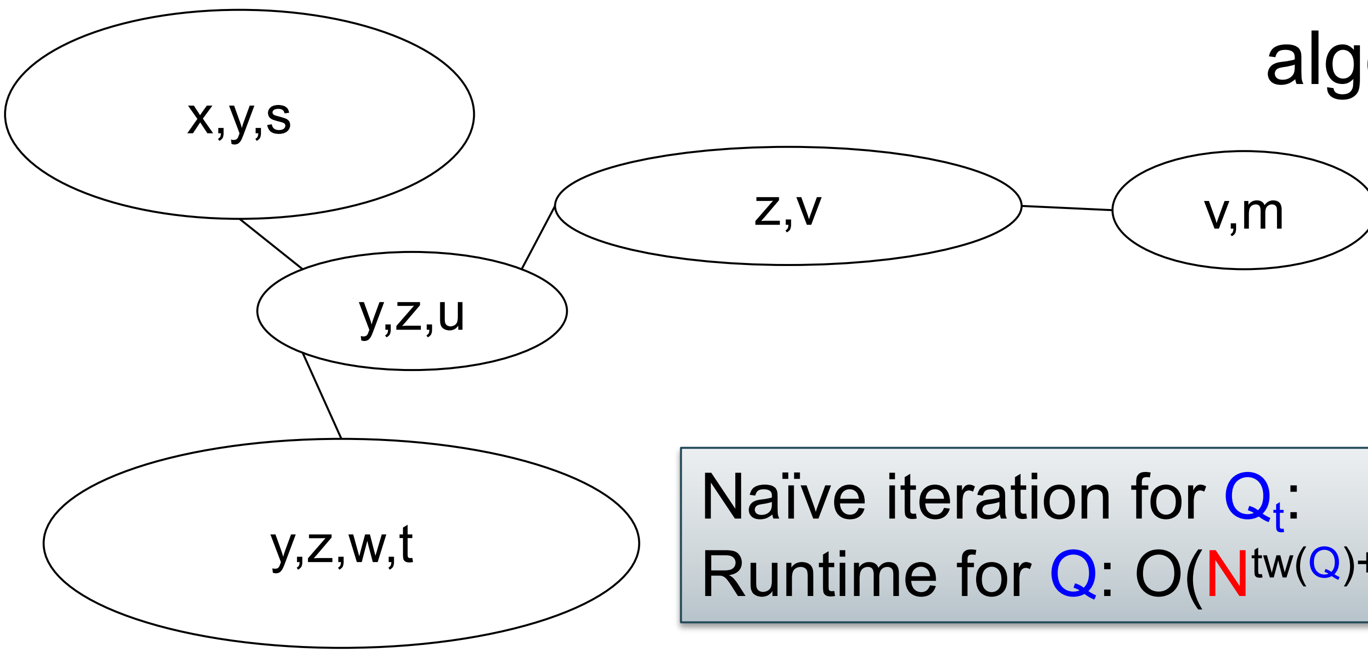
Tree-width

This is the standard definition in graph theory

Def tw(Q) = $\min_T \max_{t \in Nodes(T)} |\chi(t)| - 1$

tw(Q) = 3

Tree-width gives the complexity of the most naïve algorithm



Naïve iteration for Q_t:
Runtime for Q: $O(N^{tw(Q)+1} + |Output|)$

Generalized Hypertree Width

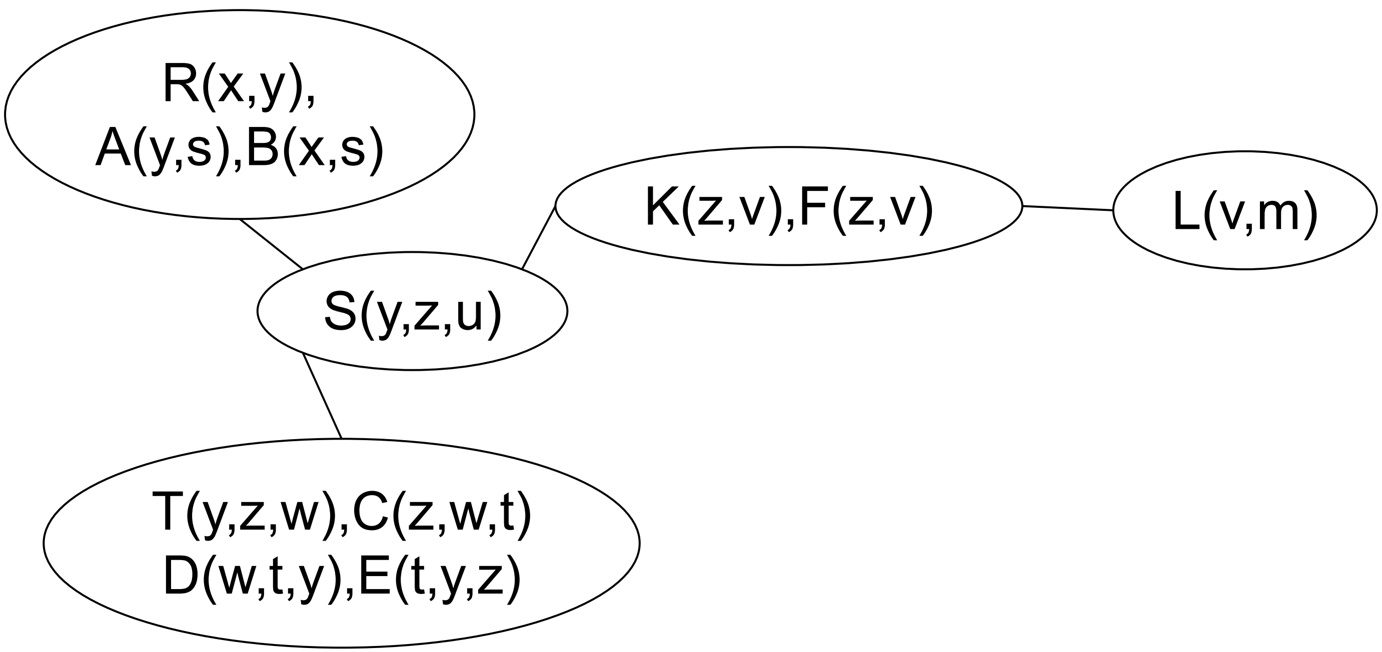
Def $\text{ghtw}(Q) = \min_T \max_{t \in \text{Nodes}(T)} \rho(Q_t)$

ρ = edge covering number

Generalized Hypertree Width

Def ghtw(Q) = $\min_T \max_{t \in Nodes(T)} \rho(Q_t)$

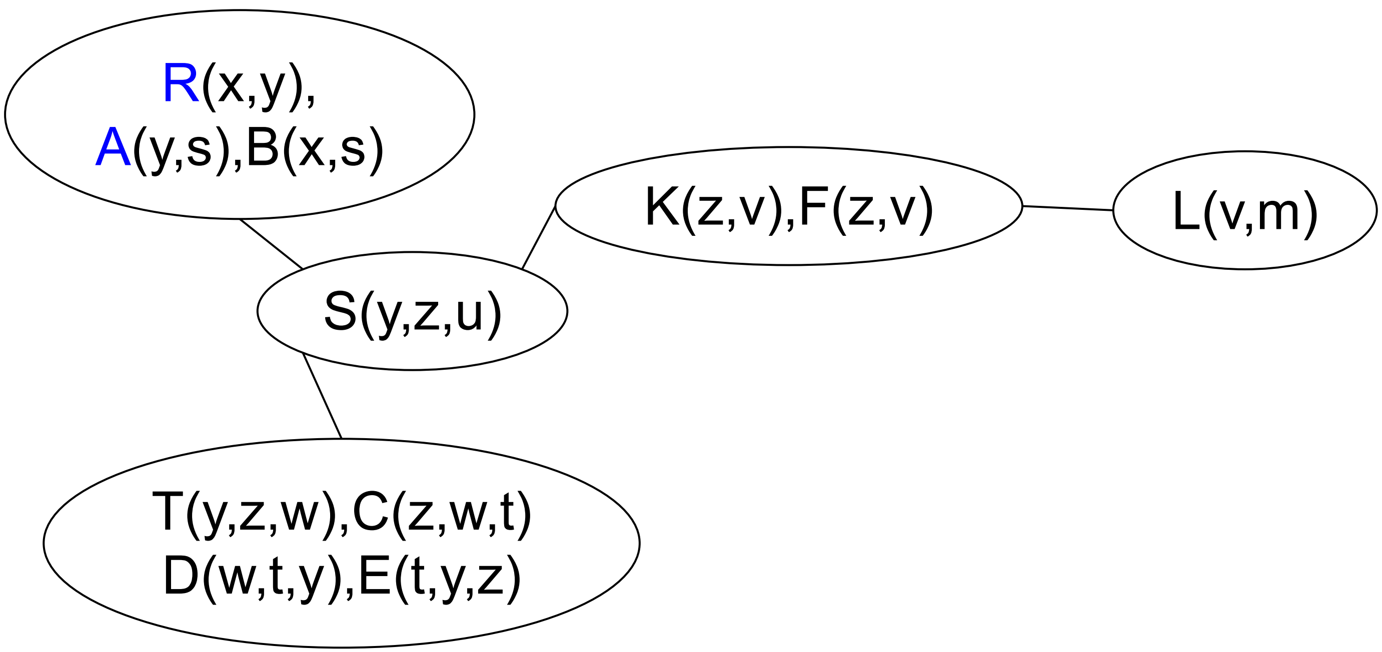
ρ = edge covering number



Generalized Hypertree Width

Def $\mathrm{ghtw}(Q) = \min_T \max_{t \in \mathrm{Nodes}(T)} \rho(Q_t)$

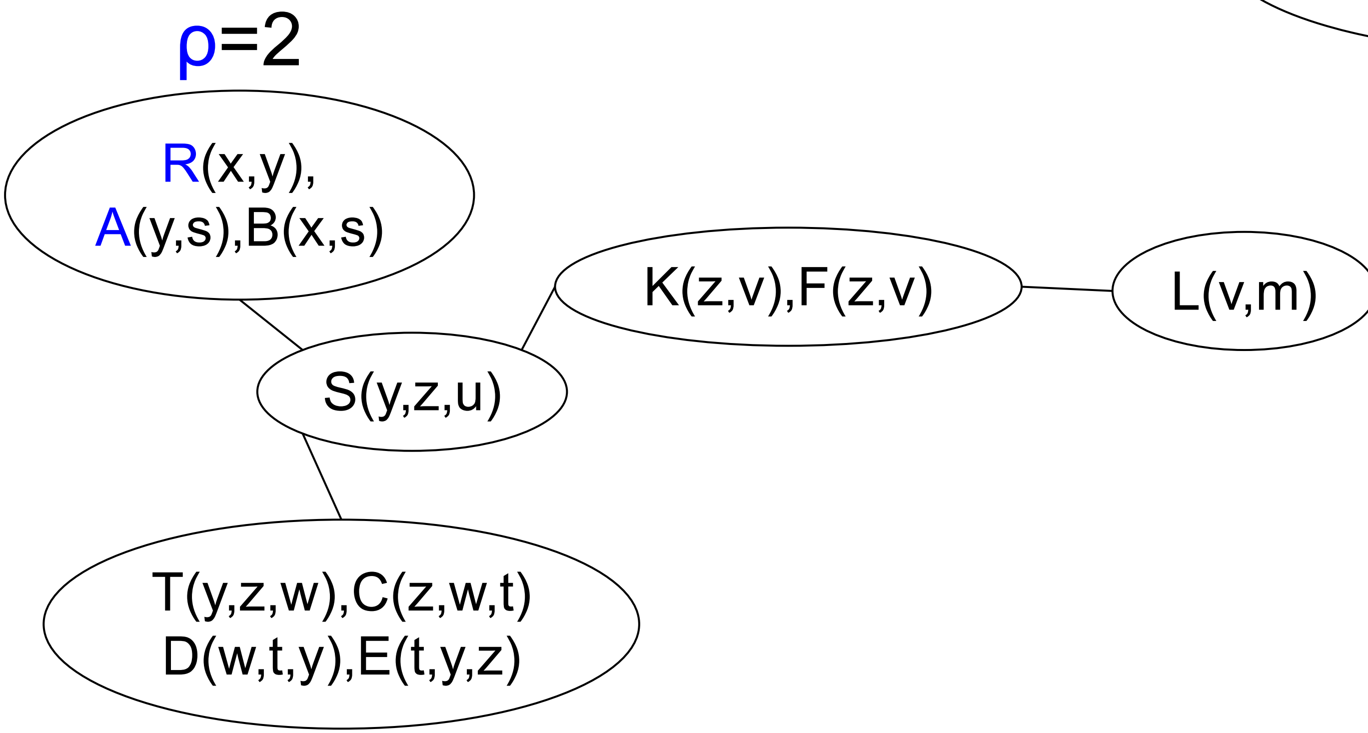
ρ = edge covering number



Generalized Hypertree Width

Def $ghtw(Q) = \min_T \max_{t \in Nodes(T)} \rho(Q_t)$

ρ = edge covering number

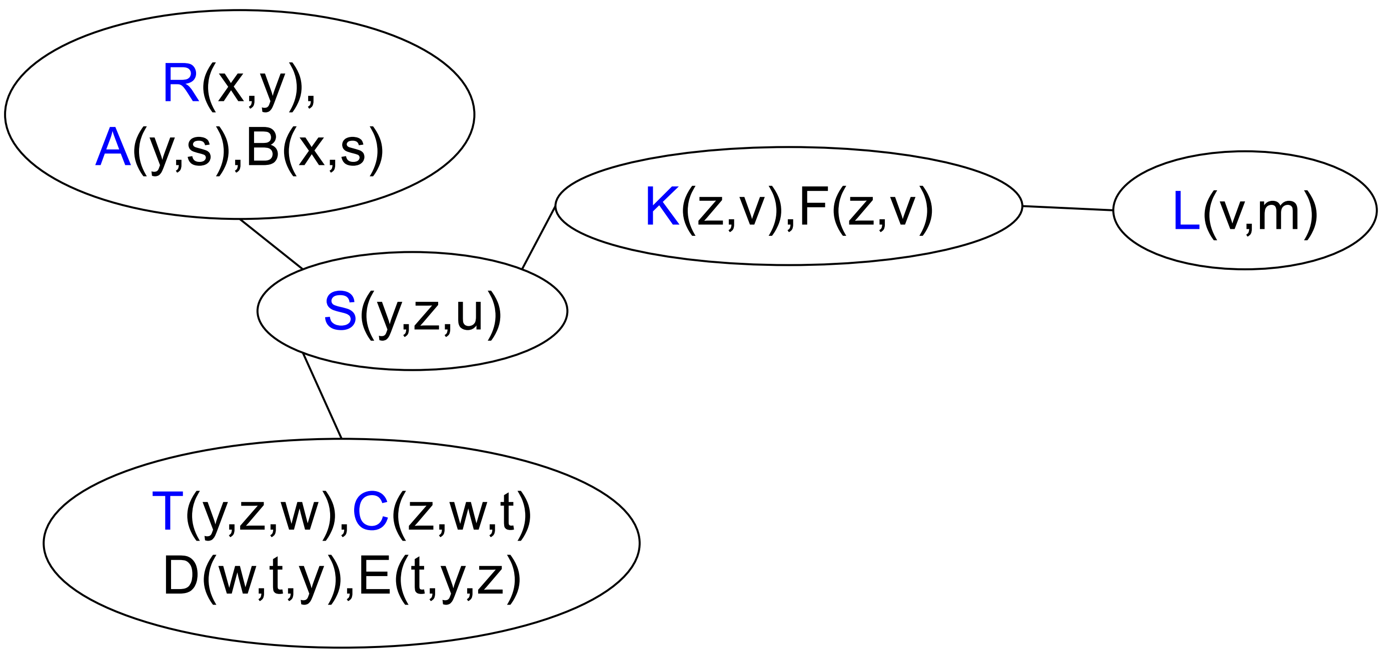


Generalized Hypertree Width

Def $ghtw(Q) = \min_T \max_{t \in Nodes(T)} \rho(Q_t)$

ρ = edge covering number

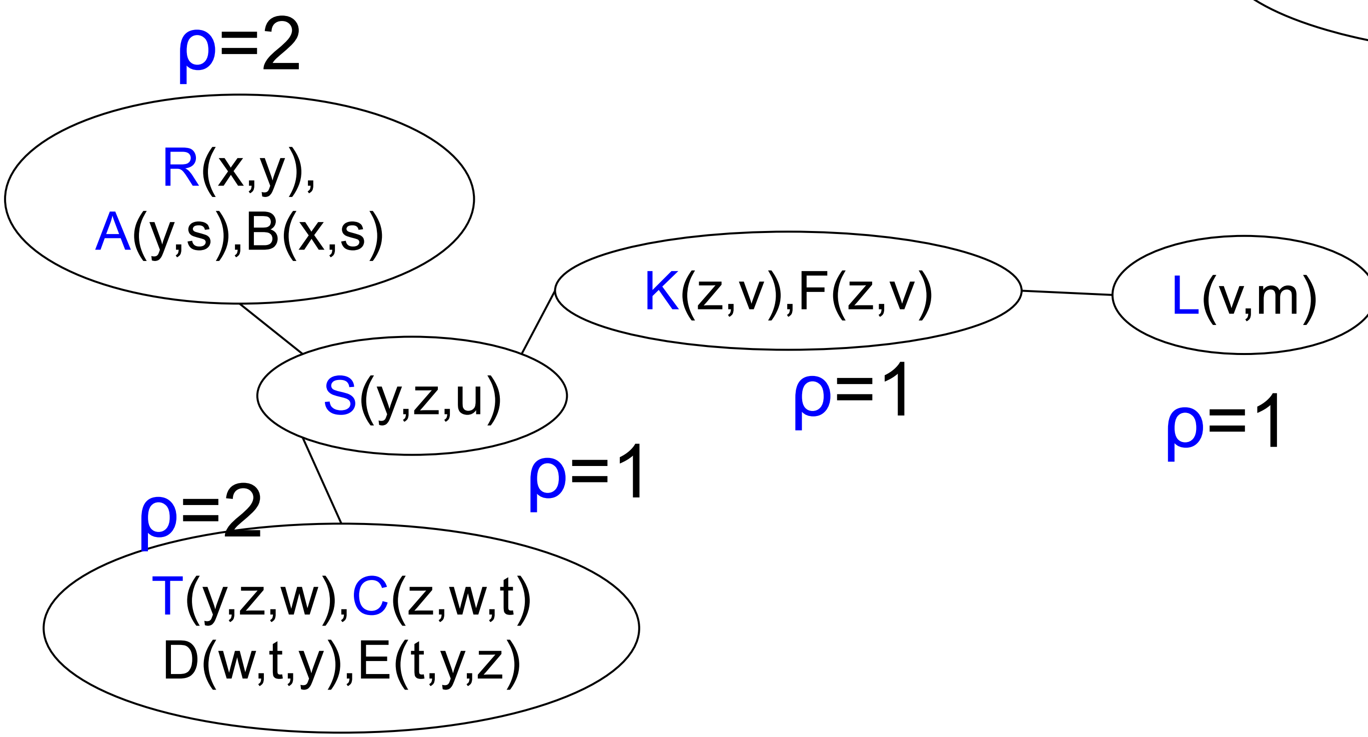
$\rho=2$



Generalized Hypertree Width

Def $\text{ghtw}(Q) = \min_T \max_{t \in \text{Nodes}(T)} \rho(Q_t)$

ρ = edge covering number

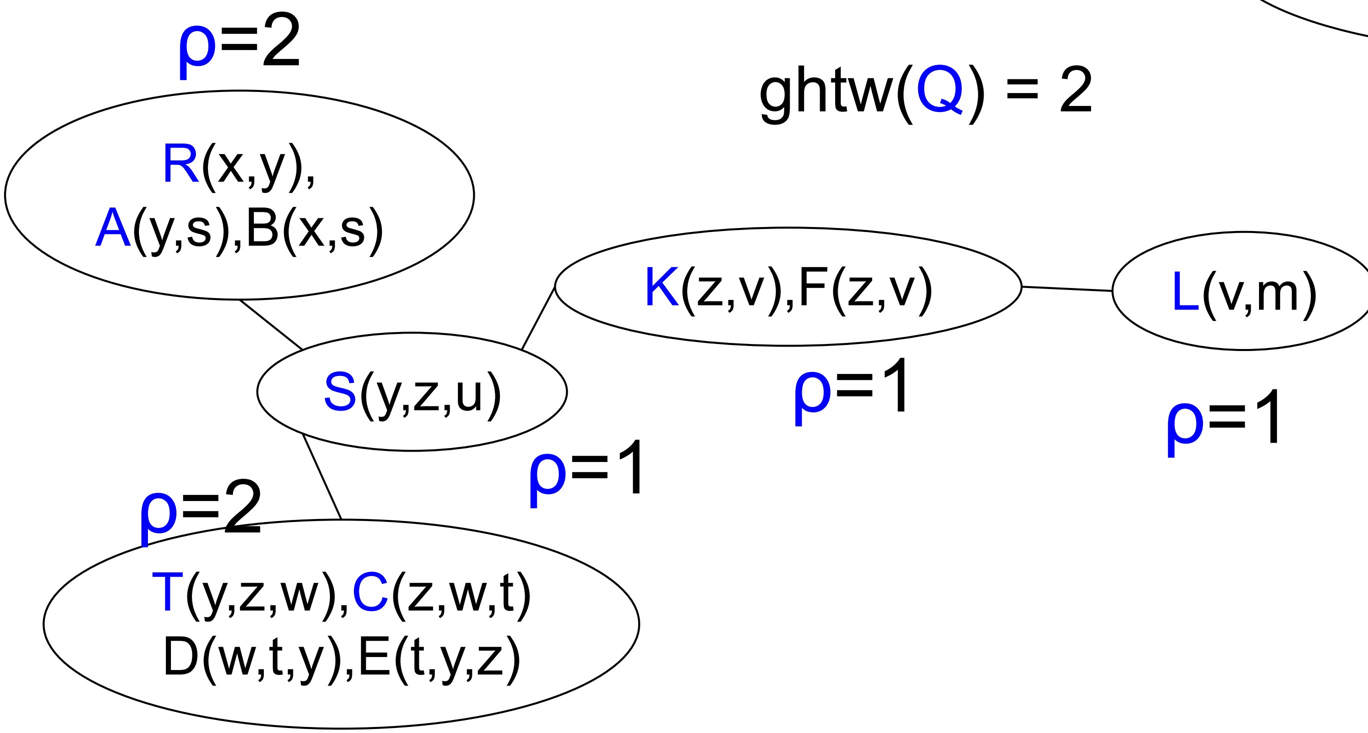


Generalized Hypertree Width

Def $\mathrm{ghtw}(Q) = \min_T \max_{t \in \mathrm{Nodes}(T)} \rho(Q_t)$

ρ = edge covering number

$\mathrm{ghtw}(Q) = 2$

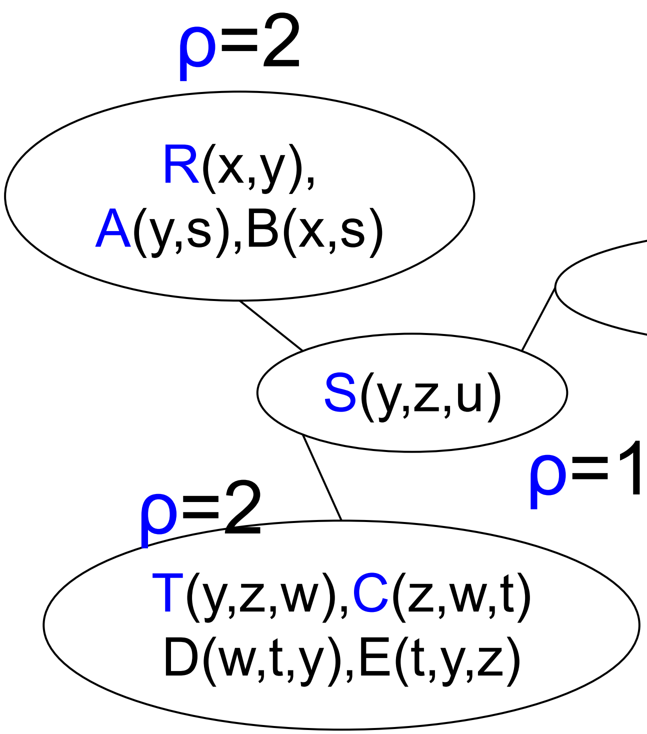


Generalized Hypertree Width

Def ghtw(Q) = $\min_T \max_{t \in Nodes(T)} \rho(Q_t)$

ρ = edge covering number

ghtw(Q) = 2

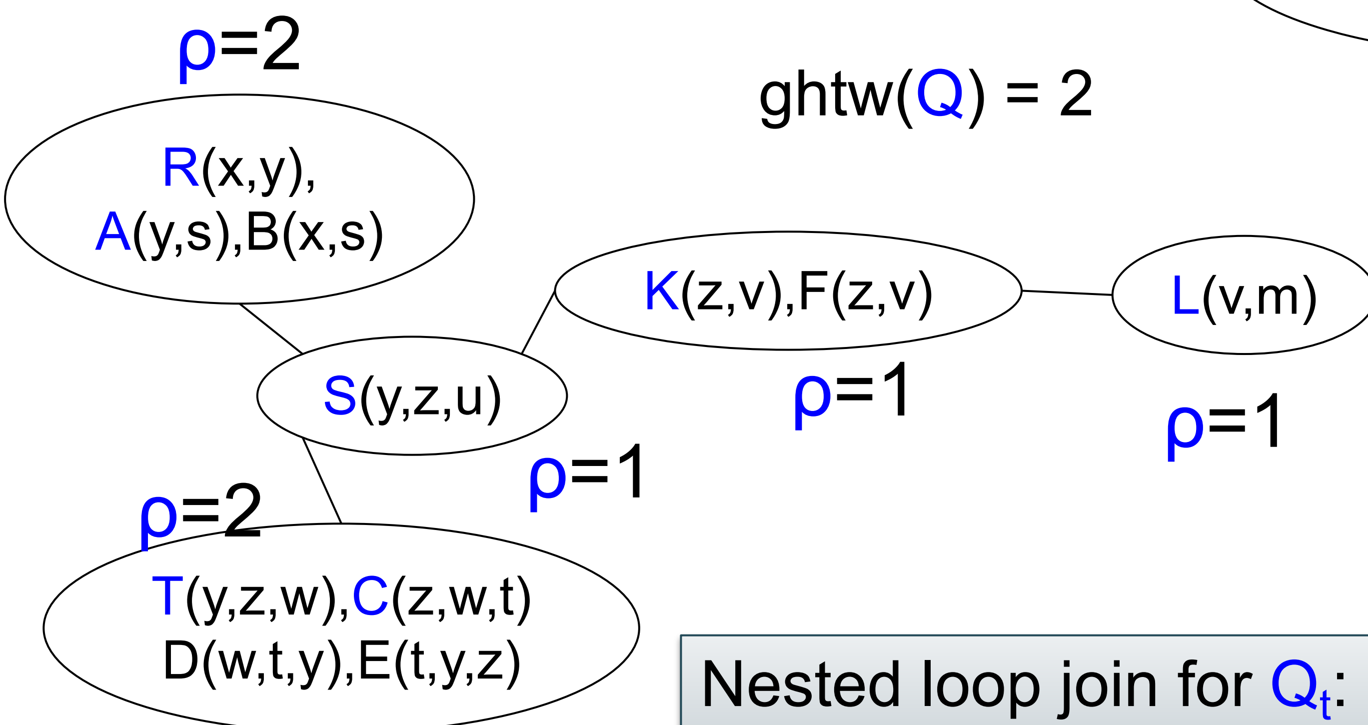


Nested loop join for Q_t:
Runtime for Q: $O(N^{ghtw(Q)} + |Output|)$

Generalized Hypertree Width

Def $ghtw(Q) = \min_T \max_{t \in Nodes(T)} \rho(Q_t)$

ρ = edge covering number



GHTW is the complexity of a still naïve algorithm

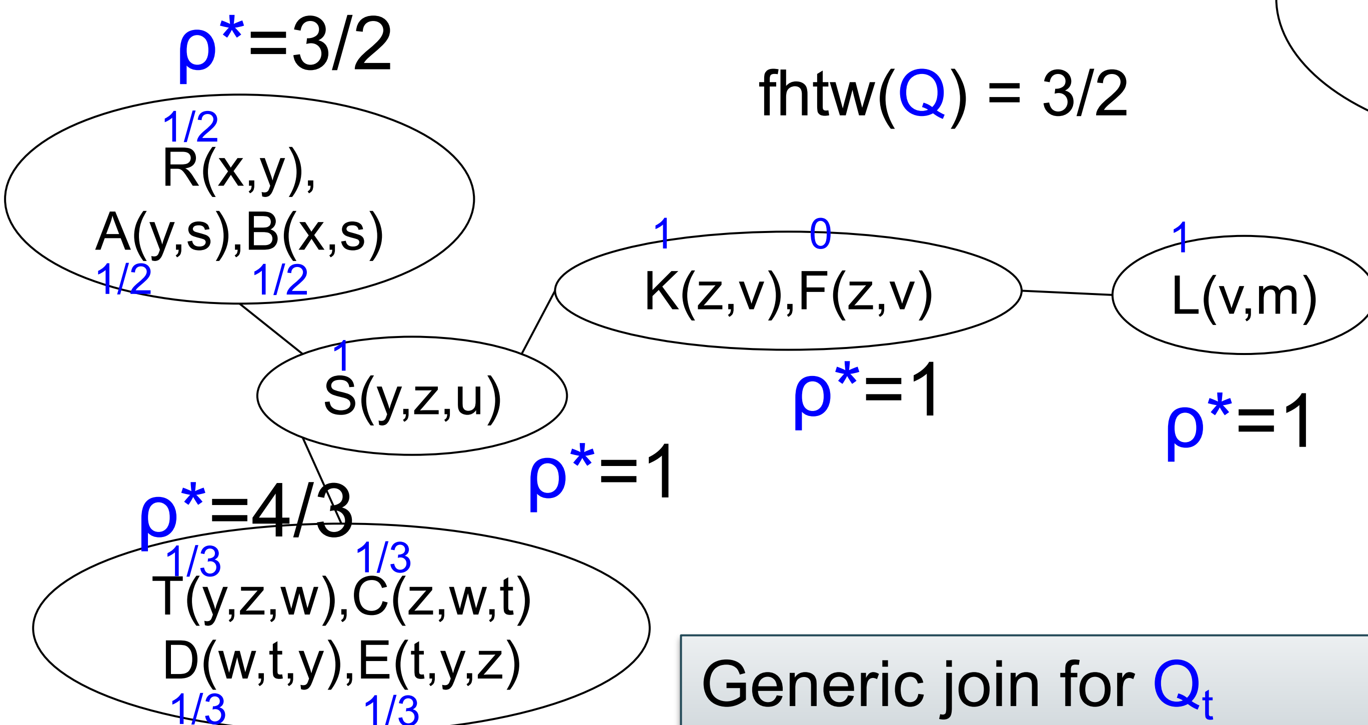
Nested loop join for Q_t:
Runtime for Q: $O(N^{ghtw(Q)} + |Output|)$

Fractional Hypertree Width

Def fhtw(Q) = $\min_T \max_{t \in Nodes(T)} \rho^*(Q_t)$

ρ^* = Fractional edge covering number

fhtw(Q) = 3/2



Next lecture

Generic join for Q_t
Runtime for Q: $O(N^{fhtw(Q)} + |Output|)$

Outline

- Acyclic queries, Yannakakis algorithm
- Tree decomposition of cyclic queries
- Worst-case optimal algorithm; next week

Wednesday!