Database Theory

Conjunctive Queries

Matthias Lanzinger, 2025

TU WIEN Informatics

Motivation

We've seen that we are limited in many things we want to do when it comes to powerful languages like FO/RA.

Let us instead study a restricted subclass of queries that lies at the core of important data retrieval tasks.

Conjunctive Queries

(Can also contain constants.)

$$\{\, \bar{y} \mid \exists \bar{z}\, R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \,\}$$

We call queries of this form Conjunctive Queries (CQs).

That is, conjunctive queries are FO queries using only the connectives $\exists$ and $\wedge$.

Conjunctive Queries

$$\{\, \bar{y} \mid \exists \bar{z}\, R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \,\}$$

$$\pi_{\bar{y}} \left(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k \right)$$

(Assuming attributes for R_i are $\bar{x}_i$, otherwise simply rename)

Conjunctive Queries

$$\{\, \bar{y} \mid \exists \bar{z}\, R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \,\}$$

$$\pi_{\bar{y}} \left(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k \right)$$

Conjunctive queries correspond so-called join queries in RA.
That is, RA queries that only use projection, renaming, selection, and joins.

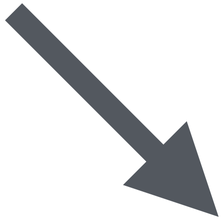
Conjunctive Queries

Recall our SQL example in the lecture on the relational model.

```
SELECT course.name, student.name
FROM course, student, enrolled
WHERE course.name = enrolled.course AND
      course.sem = enrolled.sem AND
      student.id = enrolled.student AND
      enrolled.sem = 'WS24';
```

Conjunctive Queries

```sql
SELECT course.name, student.name
FROM course, student, enrolled
WHERE course.name = enrolled.course AND
      course.sem = enrolled.sem AND
      student.id = enrolled.student AND
      enrolled.sem = 'WS24';
```

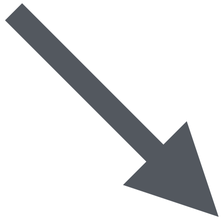


$$\{(c, s) \mid \exists sid, l, dob, active \,.\, Enrolled(c, \text{WS24}, sid) \wedge Course(c, \text{WS24}, l) \wedge Student(sid, s, dob, active)\}$$

Conjunctive Queries

```
SELECT course.name, student.name
FROM course, student, enrolled
WHERE course.name = enrolled.course AND
      course.sem = enrolled.sem AND
      student.id = enrolled.student AND
      enrolled.sem = 'WS24';
```

CQs cover the core part of most SQL queries!



$$\{(c, s) \mid \exists sid, l, dob, active \,.\, Enrolled(c, \text{WS24}, sid) \wedge Course(c, \text{WS24}, l) \wedge Student(sid, s, dob, active)\}$$

Conjunctive Queries

- Conjunctive queries form the key part of most data retrieval tasks.

- Join queries
- Datalog rule bodies are CQs
- Basis for many other query languages, e.g., Conjunctive Regular Path Queries.

Conjunctive Queries

- Conjunctive queries form the key part of most data retrieval tasks.
- Optimising CQs can help to optimise the most expensive part of practical join evaluations

```
select min(ps_supplycost)
from
      partsupp,
      supplier,
      nation,
      region
where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = 'Asia'
```

Real systems will evaluate a CQ first and then evaluate the min aggregate on the result of the CQ.

Conjunctive Queries

- Conjunctive queries form the key part of most data retrieval tasks.
- Optimising CQs can help to optimise the most expensive part of practical join evaluations
- Complexity for results for CQs also give us lower bounds for more complex queries that often have CQs at their core.

```
select min(ps_supplycost)
from
      partsupp,
      supplier,
      nation,
      region
where
      p_partkey = ps_partkey
      and s_suppkey = ps_suppkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = 'Asia'
```

The query intuitively will be at least as hard to solve as the underlying CQ (without the min aggregate).

Equivalence & Containment

Query Containment

- For queries q_1, q_2 we say that q_1 is contained in q_2 (in symbols, $q_1 \subseteq q_2$) if $q_1(D) \subseteq q_2(D)$ for every database D.

- Equivalence of two queries q_1, q_2 (in symbols, $q_1 \equiv q_2$) is defined as

$$q_1 \equiv q_2 \iff q_1 \subseteq q_2 \text{ and } q_2 \subseteq q_1$$

See the Trakthenbrot exercise sheet!

Given q_1, q_2 in RA, there is no algorithm to decide $q_1 \subseteq q_2$.

But if q_1, q_2 are CQs then the problem is decidable!

Query Containment Example

Consider the following two queries

$$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$$

$$q_2 := \{(x, y) \mid R(y, x) \wedge R(x, y)\}$$

R

A	B
1	2
3	3
2	3

Intuitively it seems clear that q_1 describes a weaker requirement: x only needs to reach some z, whereas it needs to reach specifically y in q_2.

$$q_1(D) = \{\ (2,1), (3,2), (3,3)\}$$
$$q_2(D) = \{\ (3,3)\}$$

We would therefore suspect that $q_2(D) \subseteq q_1(D)$ for all D. But how to prove it?

The Tableau of a CQ

Basic Idea: we can represent a CQ as a database.

The tableau $\mathsf{Tbl}(q)$ of a CQ q is the database where the tuples of relation R are all of the term lists that occur for R in the query (+ a relation for the output variables).

Consider the following query:

$$\{\ (x, y) \mid \exists wz\ B(x, y) \wedge R(y, z) \wedge R(y, w) \wedge R(w, y)\ \}$$

We write $\mathsf{Tbl}^*(q)$ for the tables without the special **Out** relation for output variables.

Out

1	*2*
x	y

B

1	*2*
x	y

R

1	*2*
y	z
y	w
w	y

Homomorphisms

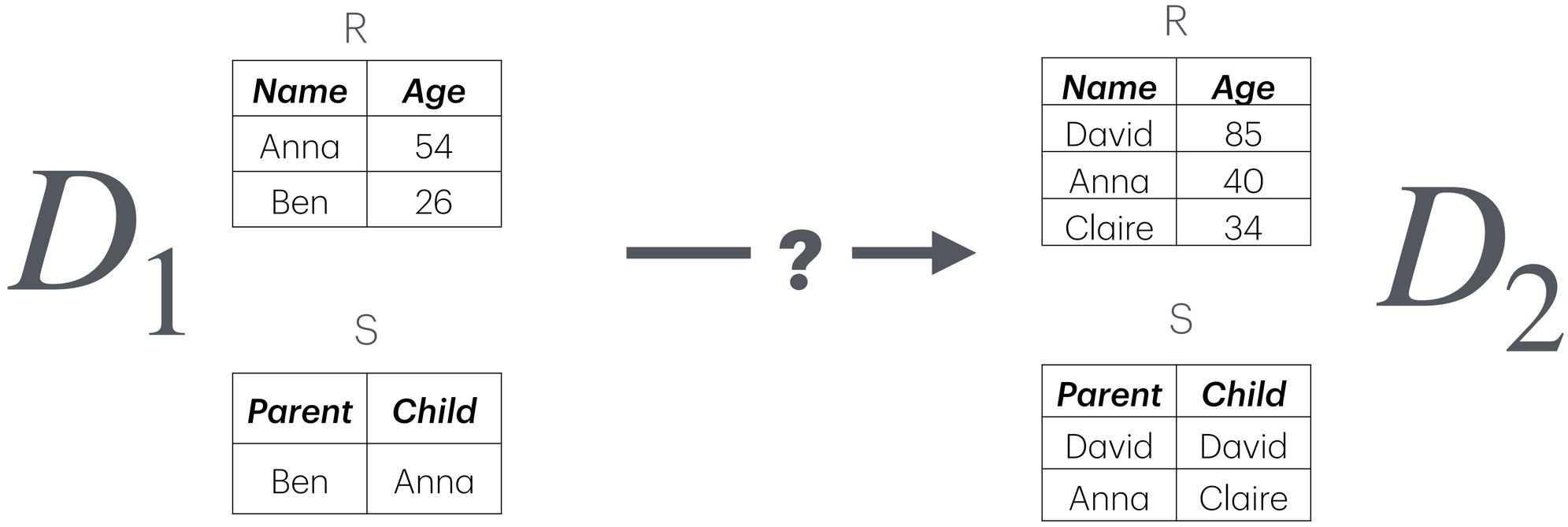
A homomorphism of two databases D_1, D_2 is a function $h : Dom(D_1) \to Dom(D_2)$ such that:

$$(c_1, \ldots, c_n) \in R^{D_1} \implies (h(c_1), \ldots, h(c_n)) \in R^{D_2} \qquad \forall R \in \mathsf{Rel}$$

Example

A homomorphism of two databases D_1, D_2 is a function $h : Dom(D_1) \rightarrow Dom(D_2)$ such that:

$$(c_1, \ldots, c_n) \in R^{D_1} \implies (h(c_1), \ldots, h(c_n)) \in R^{D_2} \qquad \forall R \in \textbf{Rel}$$



Example

A homomorphism of two databases D_1, D_2 is a function $h : Dom(D_1) \to Dom(D_2)$ such that:

$$(c_1, \ldots, c_n) \in R^{D_1} \implies (h(c_1), \ldots, h(c_n)) \in R^{D_2} \qquad \forall R \in \mathbf{Rel}$$

D_1

R

Name	***Age***
Anna	54
Ben	26

S

Parent	***Child***
Ben	Anna

$$Anna \mapsto Claire$$
$$Ben \mapsto Anna$$
$$54 \mapsto 34$$
$$26 \mapsto 40$$

May seem weird in meaning but the "structure" is preserved!

R

Name	***Age***
David	85
Anna	40
Claire	34

S

Parent	***Child***
David	David
Anna	Claire

D_2

Example

A homomorphism of two databases D_1, D_2 is a function $h : Dom(D_1) \to Dom(D_2)$ such that:

$$(c_1, \ldots, c_n) \in R^{D_1} \implies (h(c_1), \ldots, h(c_n)) \in R^{D_2} \qquad \forall R \in \mathbf{Rel}$$

D_1

R

Name	***Age***
Anna	54
Ben	26

S

Parent	***Child***
Ben	Anna

$$Anna \mapsto David$$
$$Ben \mapsto David$$
$$54 \mapsto 85$$
$$26 \mapsto 85$$

Nothing says the function must be injective

R

Name	***Age***
David	85
Anna	40
Claire	34

S

Parent	***Child***
David	David
Anna	Claire

D_2

Example

A homomorphism of two databases D_1, D_2 is a function $h : Dom(D_1) \to Dom(D_2)$ such that:

$$(c_1, \ldots, c_n) \in R^{D_1} \implies (h(c_1), \ldots, h(c_n)) \in R^{D_2} \qquad \forall R \in \mathsf{Rel}$$

D_1

R

Name	***Age***
Anna	54
Ben	26

S

Parent	***Child***
Ben	Anna

Not a hom!

$Anna \mapsto Claire$

$Ben \mapsto Claire$

$54 \mapsto 34$

$26 \mapsto 34$

But that doesn't always work

R

Name	***Age***
David	85
Anna	40
Claire	34

S

Parent	***Child***
David	David
Anna	Claire

D_2

Homomorphism Theorem

Theorem

Let q_1, q_2 be CQs. Then

$$q_1 \subseteq q_2 \quad \text{if and only if} \quad \mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$$

As a result we have an "easy" algorithm for deciding containment for CQs:

1. Compute $\mathsf{Tbl}(\cdot)$ for both queries (trivial).
2. Check if there is a homomorphism (in NP).

Careful!
Only holds for
set semantics!

Homomorphism Theorem

Consider the following two queries

$$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$$

$$q_2 := \{(x, y) \mid \exists wu\, R(y, x) \wedge R(w, x) \wedge R(x, u)\}$$

Tbl(q_1)

Out

1	2
x	y

R

1	2
y	x
x	z

Tbl(q_2)

Out

1	2
x	y

R

1	2
y	x
w	x
x	u

Homomorphism Theorem

Tbl(q_1)

Out

1	2
x	y

R

1	2
y	x
x	z

Tbl(q_2)

Out

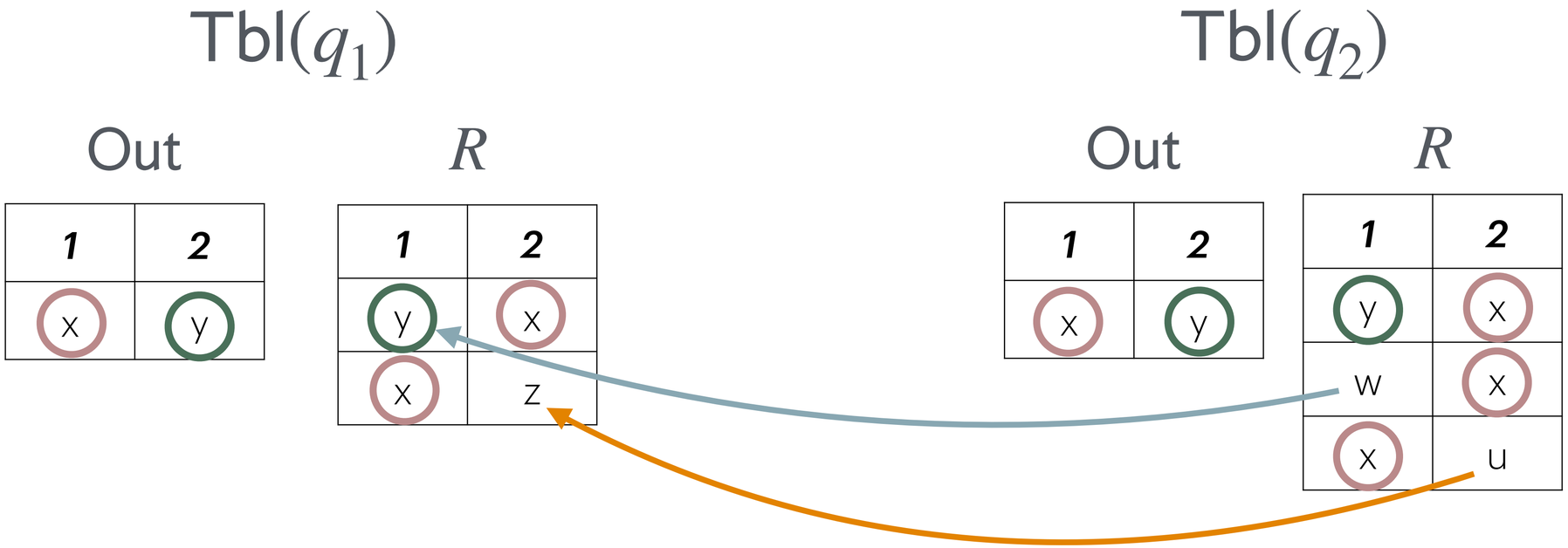
1	2
x	y

R

1	2
y	x
w	x
x	u

Since **Out** has only one tuple, a homomorphism h must necessarily have $h(x) = x$ and $h(y) = y$.

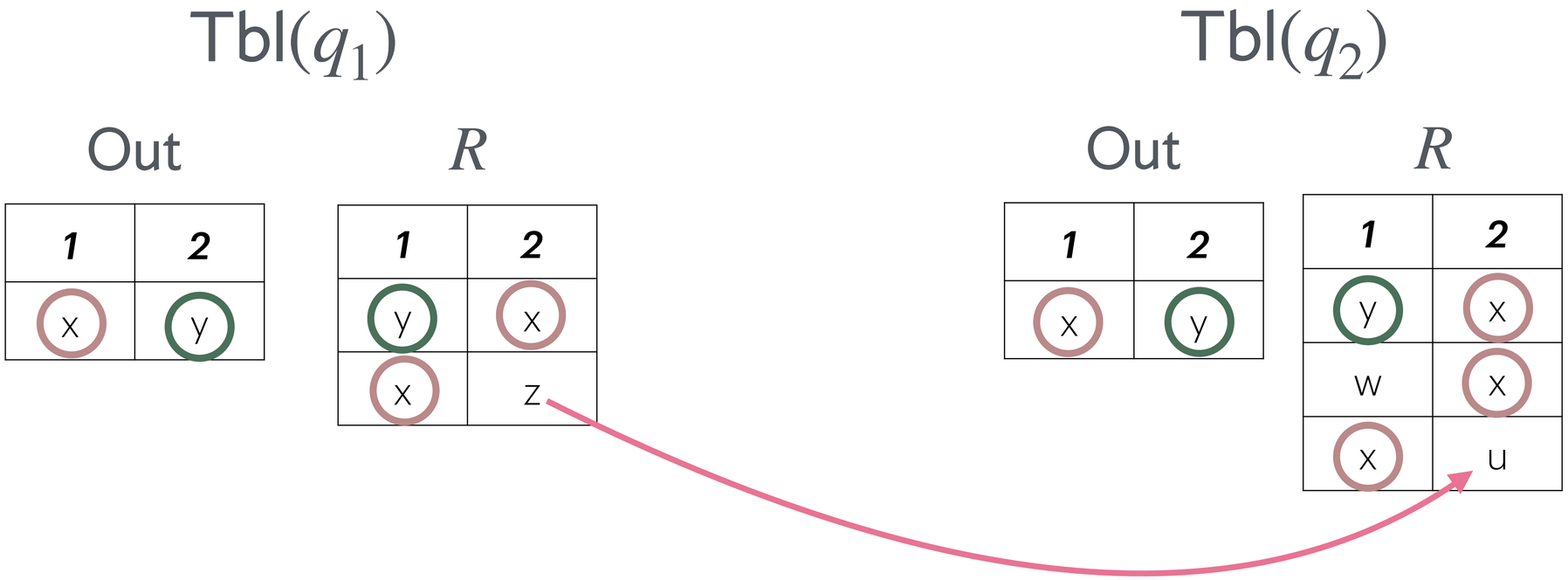
Homomorphism Theorem



Since **Out** has only one tuple, a homomorphism h must necessarily have $h(x) = x$ and $h(y) = y$.

With $h(w) = y$ and $h(u) = z$ we get a homomorphism $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$.

Homomorphism Theorem



Since **Out** has only one tuple, a homomorphism h must necessarily have $h(x) = x$ and $h(y) = y$.

With $h(z) = y$ and $h(u) = z$ we get a homomorphism $\mathsf{Tbl}(q_1) \xrightarrow{hom} \mathsf{Tbl}(q_2)$.

Homomorphism Theorem

Consider the following two queries

$$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$$

$$q_2 := \{(x, y) \mid \exists wu\; R(y, x) \wedge R(w, x) \wedge R(x, u)\}$$

We've seen that $\mathsf{Tbl}(q_1) \xrightarrow{hom} \mathsf{Tbl}(q_2)$ and $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$.

By the Homomorphism Theorem that means $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$, i.e., $q_1 \equiv q_2$!

Why?

Important observation

Let q be a CQ with free variables $\bar{y}$. For any database D, we have $\bar{c} \in q(D)$ ***if and only if*** there is a homomorphism h from $\mathsf{Tbl}^*(q)$ to D such that $h(\bar{y}) = \bar{c}$.

$$q_2 := \quad \{(x, y) \mid \exists wu\ R(y, x) \wedge R(w, x) \wedge R(x, u)\}$$

$(a, b) \in q_2(D)$ means that there is an interpretation I such that

1. $R(I(y), I(x)) \in D$, $R(I(w), I(x)) \in D$, and $R(I(x), I(u)) \in D$.
2. and $I(x) = a$ and $I(y) = b$.

So I is precisely a homomorphism from $\mathsf{Tbl}^*(q_2)$ to D!

Proof Idea

If $q_1 \subseteq q_2$, then $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$

By assumption $q_1(D) \subseteq q_2(D)$. So take $\mathsf{Tbl}^*(q_1)$ as database D. Let $\bar{y}$ be the free variables of q_1. We have that $(x, y) \in q_1(\mathsf{Tbl}^*(q_1))$ since we can just map every variable to itself.

$\mathsf{Tbl}^*(q_1)$ R

1	2
y	x
x	z

$h(x) = x, h(y) = y, h(z) = z$

$\mathsf{Tbl}^*(q_1)$ R

1	2
y	x
x	z

So $(x, y) \in q_1(\mathsf{Tbl}^*(q_1))$

Example queries from before

$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$

$q_2 := \{(x, y) \mid \exists wu\, R(y, x) \wedge R(w, x) \wedge R(x, u)$

Proof Idea

If $q_1 \subseteq q_2$, then $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$

By assumption $q_1(D) \subseteq q_2(D)$. So take $\mathsf{Tbl}^*(q_1)$ as database D. Let $\bar{y}$ be the free variables of q_1. We have that $(x, y) \in q_1(\mathsf{Tbl}^*(q_1))$ since we can just map every variable to itself.

Then also $(x, y) \in q_2(\mathsf{Tbl}^*(q_1))$. By the key observation:

$$\bar{y} \in q_2(\mathsf{Tbl}^*(q_1)) \iff \mathsf{Tbl}^*(q_2) \xrightarrow{hom} \mathsf{Tbl}^*(q_1)$$

That is, the tuple in the **Out** relation of $\mathsf{Tbl}(q_2)$ maps into a tuple of **Out** of $\mathsf{Tbl}(q_1)$.
Furthermore, that homomorphism maps the free variables of q_2 to the free variables of q_1.

We then have $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$.

Proof Idea

Example queries from before

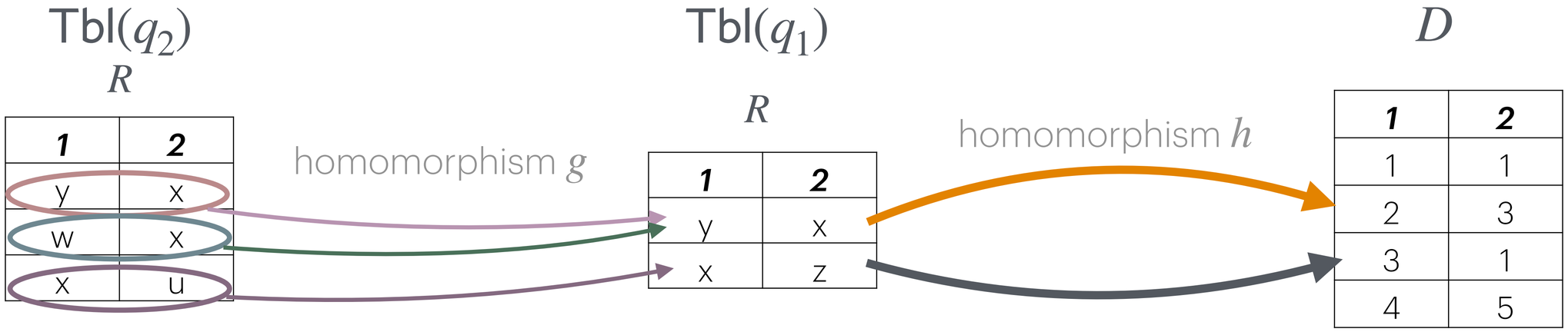
$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$

$q_2 := \{(x, y) \mid \exists wu\, R(y, x) \wedge R(w, x) \wedge R(x, u)\}$

If $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$, then $q_1 \subseteq q_2$

If $\bar{c}$ is an answer of q_1 on some database D then there is a homomorphism $h : \mathsf{Tbl}^*(q_1) \to D$ that maps the output variables of q_1 to $\bar{c}$.

Let g be the homomorphism from $\mathsf{Tbl}(q_2)$ to $\mathsf{Tbl}(q_1)$. It is not hard to see that $h \circ g$ is homomorphism from $\mathsf{Tbl}^*(q_2)$ to D that also maps the output of variables of q_2 to $\bar{c}$.

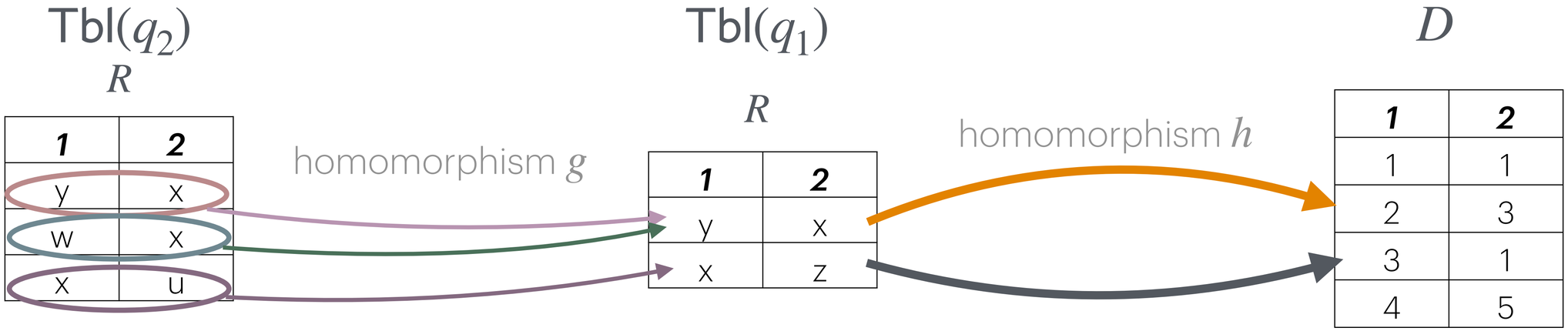


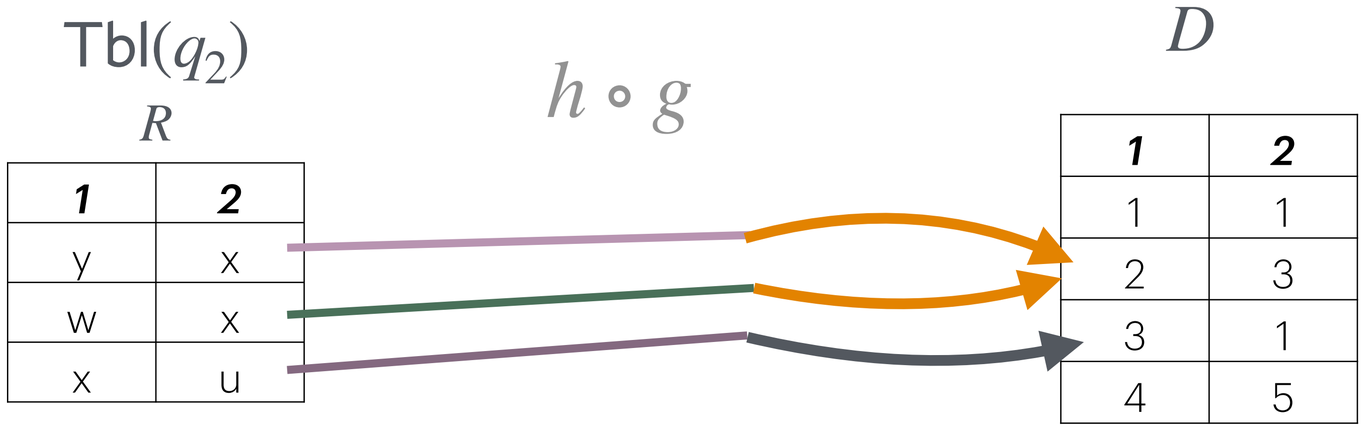
Proof Idea

Example queries from before

$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$

$q_2 := \{(x, y) \mid \exists wu\, R(y, x) \wedge R(w, x) \wedge R(x, u)\}$





Still a homomorphism!

Output variables map to the same values in D!

Proof Idea

Example queries from before

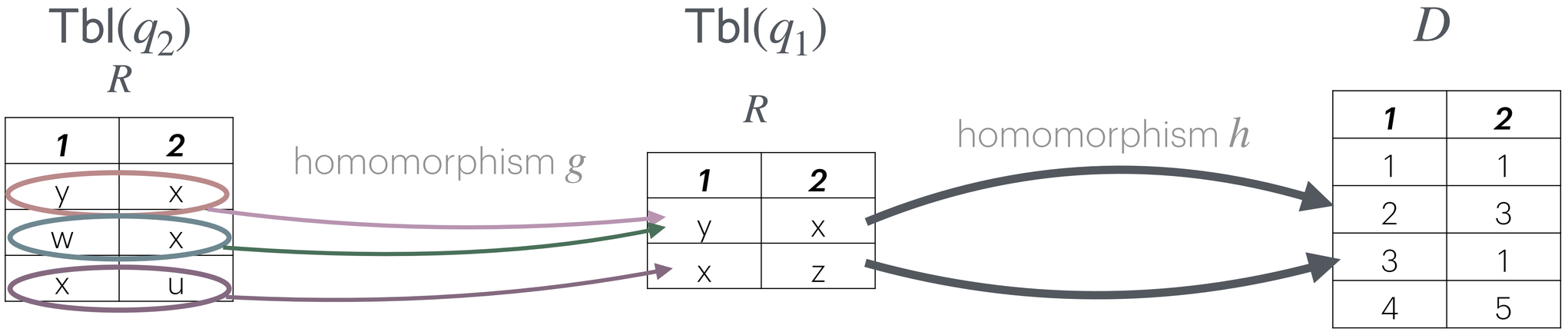
$q_1 := \{(x, y) \mid \exists z\, R(y, x) \wedge R(x, z)\}$

$q_2 := \{(x, y) \mid \exists wu\, R(y, x) \wedge R(w, x) \wedge R(x, u)\}$

If $\mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$, then $q_1 \subseteq q_2$

If $\bar{c}$ is an answer of q_1 on some database D, then there is a homomorphism $h : \mathsf{Tbl}^*(q_1) \to D$ that maps the output variables of q_1 to $\bar{c}$.

Let g be the homomorphism from $\mathsf{Tbl}(q_2)$ to $\mathsf{Tbl}(q_1)$. It is not hard to see that $h \circ g$ is homomorphism from $\mathsf{Tbl}^*(q_2)$ to D that also maps the output of variables of q_2 to $\bar{c}$.



Time for a break.

Turn to your neighbour: briefly discuss the lecture
Stretch, grab water, reset

Query Minimisation

Minimising?

Goal:

Given a CQ q, we want the equivalent CQ q' with the least amount of atoms.

Formally, a CQ q is minimal if there does not exist a CQ q' such that:

a) $q' \equiv q$

b) q' has fewer atoms (=terms in the conjunction) than q

We would like to replace a CQ with its minimal equivalent CQ before evaluating it.

How do we find this minimal equivalent CQ?

To minimise CQ q, it is enough to check only those queries obtained by deleting atoms from q!

Minimisation by Deletion

Assume CQ $q = \{\, \bar{y} \mid \exists \bar{z}\; R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \,\}$.

Furthermore, assume q has an equivalent CQ

$q' = \{\, \bar{y}' \mid \exists \bar{z}'\; S_1(\bar{x}'_1) \wedge S_2(\bar{x}'_2) \wedge \cdots \wedge S_j(\bar{x}'_j) \,\}$ with $j < k$.

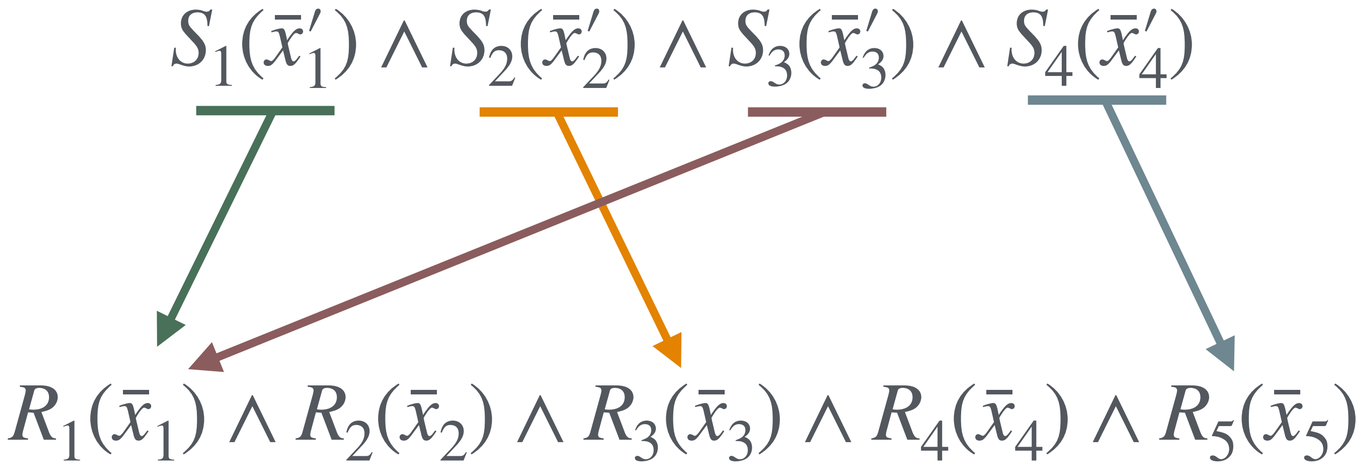
By the Homomorphism Theorem there are homomorphisms:

$$f : \mathsf{Tbl}(q) \to \mathsf{Tbl}(q') \quad \text{and} \quad g : \mathsf{Tbl}(q') \to \mathsf{Tbl}(q)$$

Minimisation by Deletion

We have that $g : \mathsf{Tbl}(q') \to \mathsf{Tbl}(q)$ maps every $S_\alpha(\bar{x}'_\alpha) \in \mathsf{Tbl}(q')$ into some $R_{i_\alpha}(\bar{x}_{i_\alpha}) \in \mathsf{Tbl}(q)$ with $S_\alpha = R_{i_\alpha}$ and $\bar{x}_{i_\alpha} = h(\bar{x}'_\alpha)$.

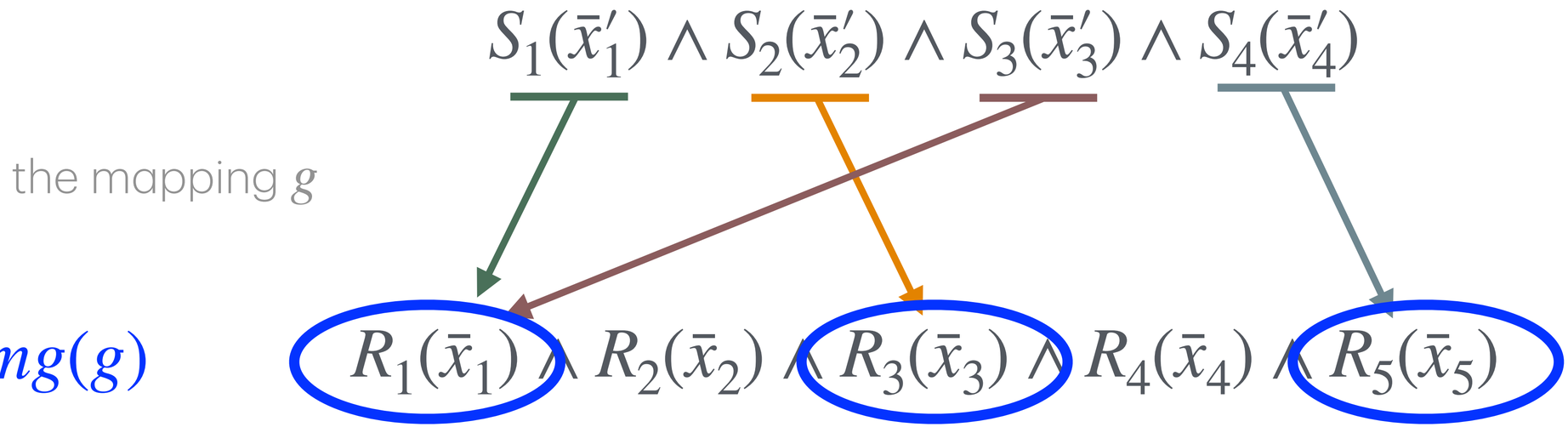
the mapping g



Minimisation by Deletion

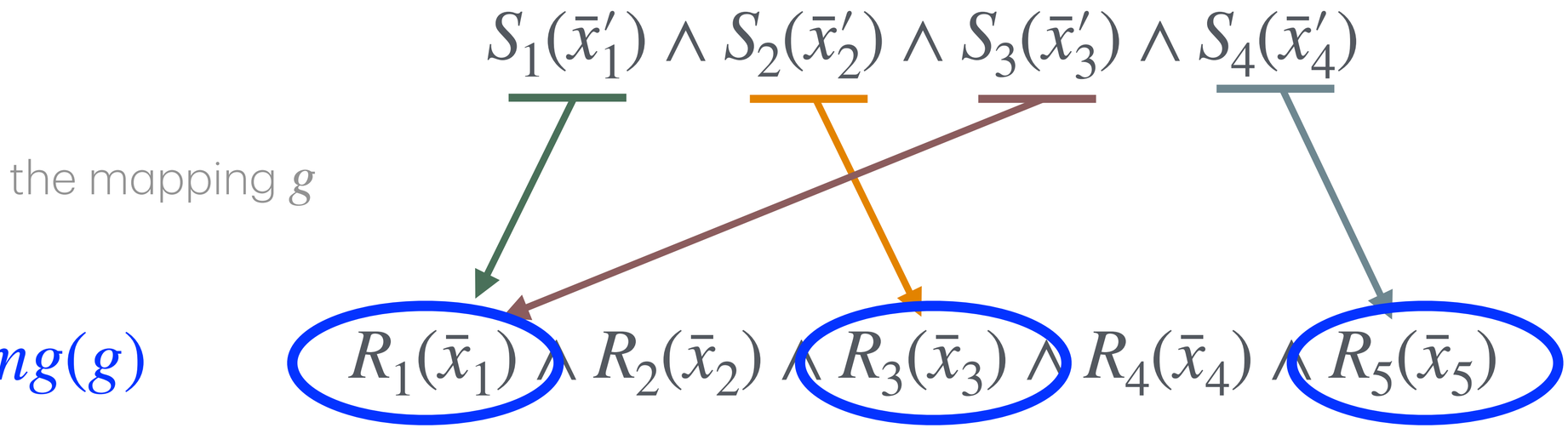
We have that $g : \mathsf{Tbl}(q') \to \mathsf{Tbl}(q)$ maps every $S_\alpha(\bar{x}'_\alpha) \in \mathsf{Tbl}(q')$ into some $R_{i_\alpha}(\bar{x}_{i_\alpha}) \in \mathsf{Tbl}(q)$ with $S_\alpha = R_{i_\alpha}$ and $\bar{x}_{i_\alpha} = h(\bar{x}'_\alpha)$.

Let $Img(g) = \{R_{i_1}(\bar{x}_{i_1}), R_{i_2}(\bar{x}_{i_2}), \ldots, R_{i_\ell}(\bar{x}_{i_\ell})\}$ be the set of all such images of the mapping g applied to the terms of q' and observe that $|Img(g)| \leq j < k$.



Minimisation by Deletion

Let us define the query $q'' = \{ \bar{y} \mid \exists \bar{z}\ R_{i_1}(\bar{x}_{i_1}) \wedge R_{i_2}(\bar{x}_{i_2}) \wedge \cdots \wedge R_{i_\ell}(\bar{x}_{i_\ell}) \}$ consisting of the terms in $Img(g)$. We see that q'' can be obtained by simply deleting some terms from q.



$$q'' = \{\bar{y} \mid R_1(\bar{x}_1) \wedge R_3(\bar{x}_3) \wedge R_5(\bar{x}_5)$$

Minimisation by Deletion

We use the Homomorphism Theorem to show that q'' is also equivalent to q:

- There is a trivial homomorphism $\mathsf{Tbl}(q'') \to \mathsf{Tbl}(q)$:
  simply map every variable to itself.

$$q'' : \quad R_1(\bar{x}_1) \wedge R_3(\bar{x}_3) \wedge R_5(\bar{x}_5)$$

$$q : \quad R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge R_3(\bar{x}_3) \wedge R_4(\bar{x}_4) \wedge R_5(\bar{x}_5)$$

Minimisation by Deletion

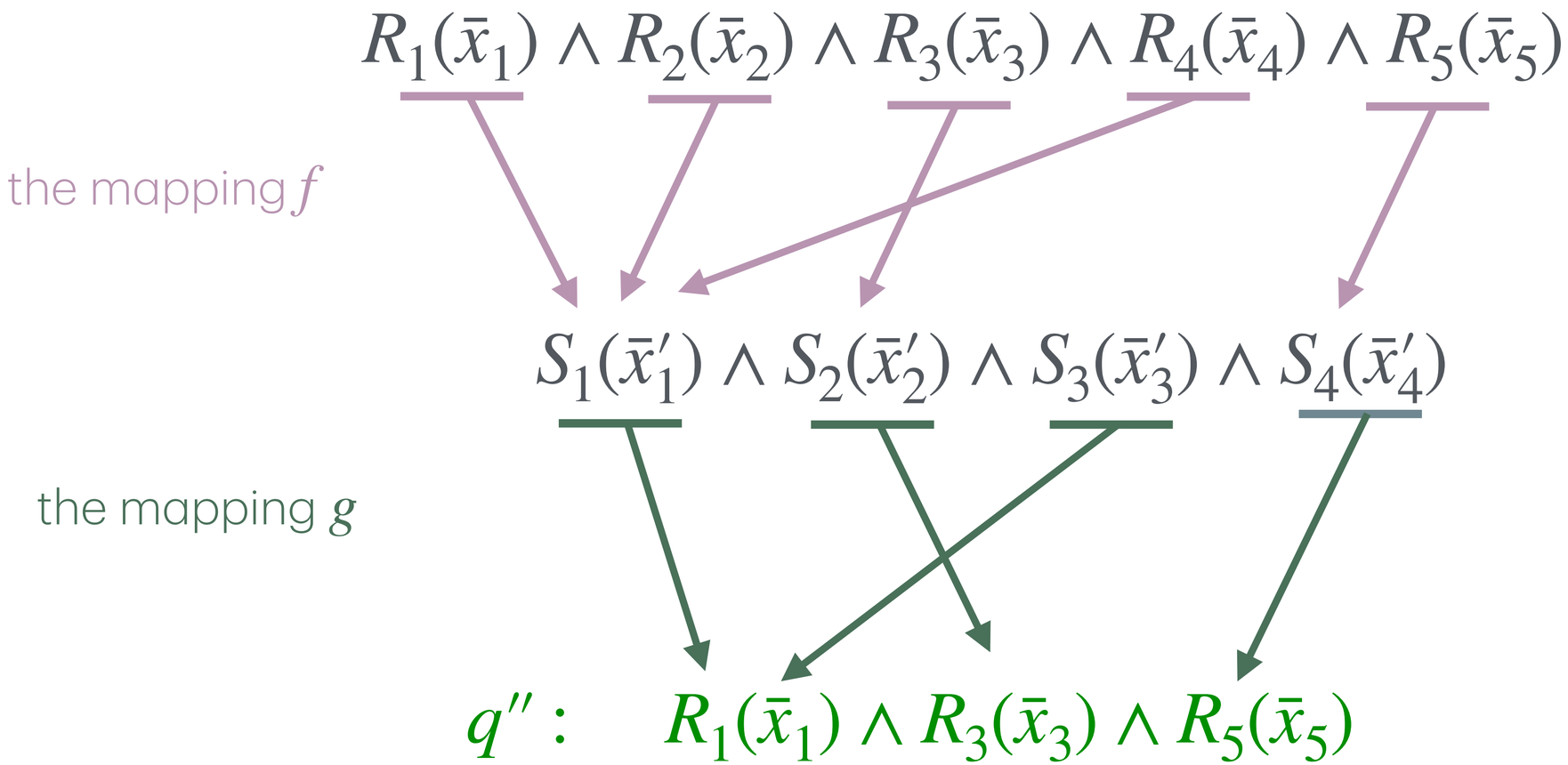
We use the Homomorphism Theorem to show that q'' is also equivalent to q:

- There is a trivial homomorphism $\mathsf{Tbl}(q'') \to \mathsf{Tbl}(q)$:
  simply map every variable to itself.

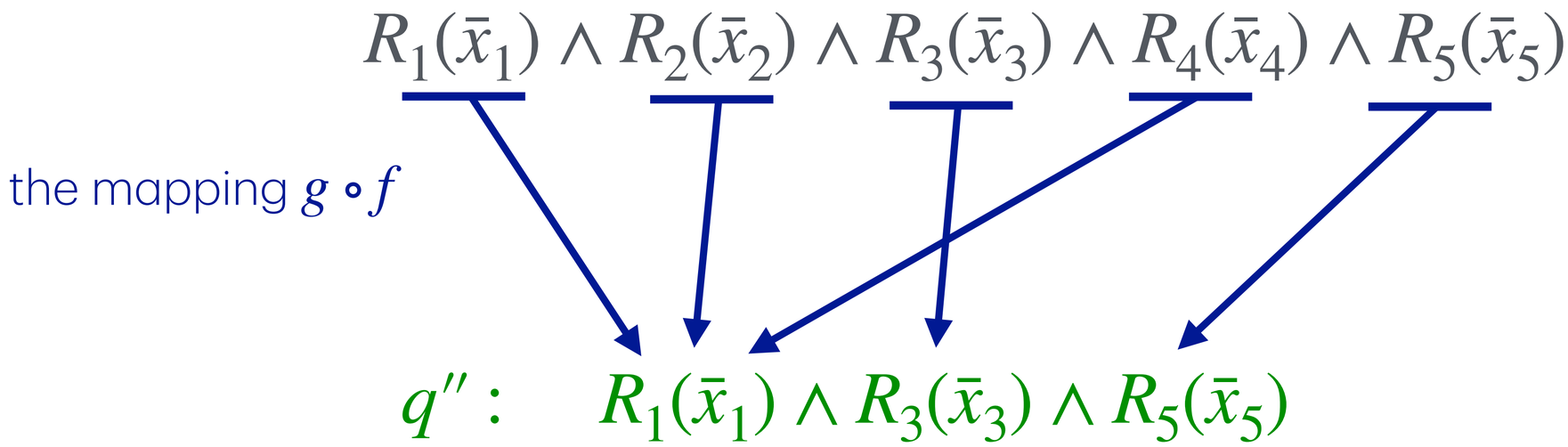
- $g \circ f$ — i.e., the function composition $g(f(\,\cdot\,))$ — is a homomorphism $\mathsf{Tbl}(q) \to \mathsf{Tbl}(q'')$:
  - function f maps every $R_i(\bar{x}_i)$ in q into an $S_\alpha(\bar{x}_\alpha)$ in q' by definition,
  - function g maps every $S_\alpha(\bar{x}_\alpha)$ in q' into an atom of q'' by construction.

✦ $g \circ f$ — i.e., the function composition $g(f(\,\cdot\,))$ — is a homomorphism $\mathsf{Tbl}(q) \to \mathsf{Tbl}(q'')$:

- function f maps every $R_i(\bar{x}_i)$ in q into an $S_\alpha(\bar{x}_\alpha)$ in q' by definition,
- function g maps every $S_\alpha(\bar{x}_\alpha)$ in q' into an atom of q'' by construction.



- $g \circ f$ — i.e., the function composition $g(f(\,\cdot\,))$ — is a homomorphism $\mathsf{Tbl}(q) \to \mathsf{Tbl}(q'')$:
  - function f maps every $R_i(\bar{x}_i)$ in q into an $S_\alpha(\bar{x}_\alpha)$ in q' by definition,
  - function g maps every $S_\alpha(\bar{x}_\alpha)$ in q' into an atom of q'' by construction.



Minimisation by Deletion

Lemma

Assume CQ $q = \{ \bar{y} \mid \exists \bar{z}\ R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \}$.

Furthermore, assume q has a semantically equivalent CQ $q' = \{ \bar{y'} \mid \exists \bar{z'}\ S_1(\bar{x'}_1) \wedge S_1(\bar{x'}_1) \wedge \cdots \wedge S_j(\bar{x'}_j) \}$.

Then q is also semantically equivalent to a CQ q'' that can obtained by deleting atoms from q.

Interesting consequence: there is always a unique minimal equivalent query.

We call this minimal equivalent subquery of q the core of q.

An Algorithm for Minimisation

Algorithm 1: Query Minimisation

Input: Conjunctive query q
Result: A minimal conjunctive query q' with $q \equiv q$
$q' \leftarrow q$
repeat
  for *Term* $R(\bar{x})$ *in* q' **do**
    $q''q'$ without $R(\bar{x})$
    if *there is a homomorphism* $\mathsf{Tbl}(q') \to \mathsf{Tbl}(q'')$ **then**
      $q' \leftarrow q''$
    end
  end
until *no change to* q'
return q'

In plain text

Delete terms from the CQ as long as there is still a homomorphism to the query after deletion.

Once this is no longer possible, the minimum is reached.

CQ Minimisation Example

$$q = \{(x, y, z) \mid \exists \alpha\beta\gamma\ R(x, y, \alpha) \wedge R(\beta, y, \gamma) \wedge R(\beta, y, z)\ \}$$

Tbl(q)

A	*B*	*C*
x	y	α
β	y	ɣ
β	y	z

Homomorphism?

$\longrightarrow$

Tbl(q)$\setminus\{(x, y, \alpha)\}$

A	*B*	*C*
x	y	α
β	y	ɣ
β	y	z

No, because $h(x) = x$, the first row can't be mapped into the right-hand tableau.

CQ Minimisation Example

$$q = \{(x, y, z) \mid \exists \alpha\beta\gamma\ R(x, y, \alpha) \wedge R(\beta, y, \gamma) \wedge R(\beta, y, z)\ \}$$

Tbl(q)

A	*B*	*C*
x	y	α
β	y	ɣ
β	y	z

Homomorphism?

$\longrightarrow$

Tbl(q)$\setminus\{(\beta, y, \gamma)\}$

A	*B*	*C*
x	y	α
β	y	ɣ
β	y	z

Yes! x, y, z, β map to themselves and $h(\gamma) = z$

CQ Minimisation Example

$$q' = \{(x, y, z) \mid \exists \alpha\beta \; R(x, y, \alpha) \wedge R(\beta, y, z) \; \}$$

Tbl(q')

A	*B*	*C*
x	y	α
β	y	z

Homomorphism?

$\longrightarrow$

A	*B*	*C*
x	y	α
β	y	z

or

A	*B*	*C*
x	y	α
β	y	z

Both times no.

Hence, q' is minimal!

Complexity of CQs

Data vs Combined Complexity

Recall, in classical computational complexity theory we study algorithm behavior (time/space/etc.) relative to the size of the input.

In query answering problems there are different variants of this problem:

Eval(q, D)

Input size is the sum of the query size and database size

Matches natural settings such as a DBMS, where queries and data come from user and are arbitrary.

q-Eval(D)

Input size is only the database!

Motivated by the fact that queries are usually much smaller than the databases.

Data vs Combined Complexity

Recall, in classical computational complexity theory we study algorithm behavior (time/space/etc.) relative to the size of the input.

In query answering problems there are different variants of this problem:

Combined Complexity

Input size is the sum of the query size and database size

Matches natural settings such as a DBMS, where queries and data come from user and are arbitrary.

Data Complexity

Input size is only the database!

Motivated by the fact that queries are usually much smaller than the databases.

Our Focus Now

CQ-EVAL

Input: Conjunctive query q, database D (of same schema)

Output: $q(D) \neq \emptyset$

Recall, this corresponds to combined complexity.

NP-Membership

$$\{\, \bar{y} \mid \exists \bar{z}\, R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \,\}$$

When is $q(D) \neq \emptyset$?

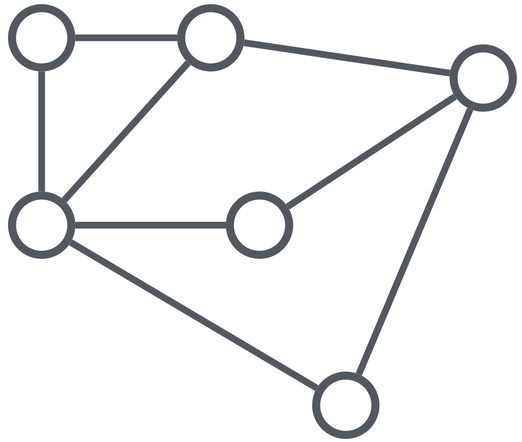
If there is any homomorphism from $\mathsf{Tbl}^*(q)$ to D.

NP-membership is straightforward: guess and check a homomorphism.

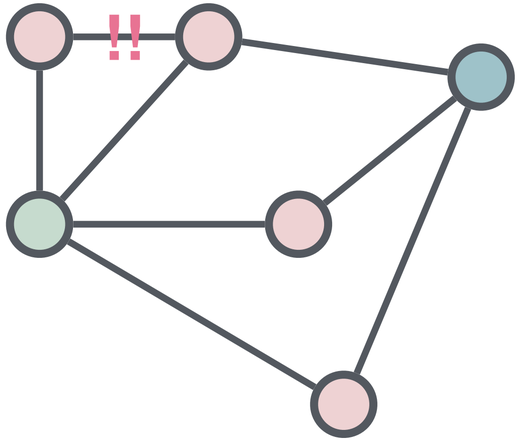
NP-Hardness

- There is an easy reduction from **3-Colourability.**
- **3-Colourability** takes a graph Gas input and decides whether G is 3-colourable.

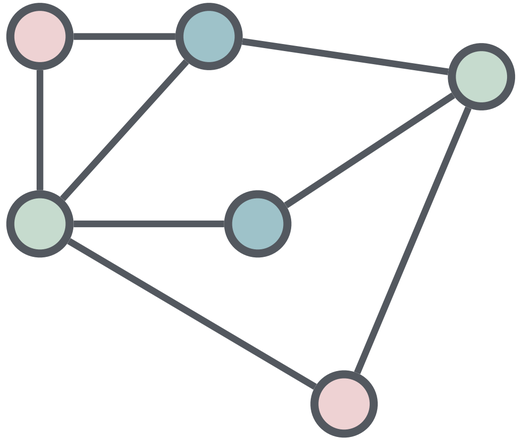
  That is, can we color the vertices of G with red, green, and blue such that no edge is between two vertices of the same colour?



Not a 3-colouring

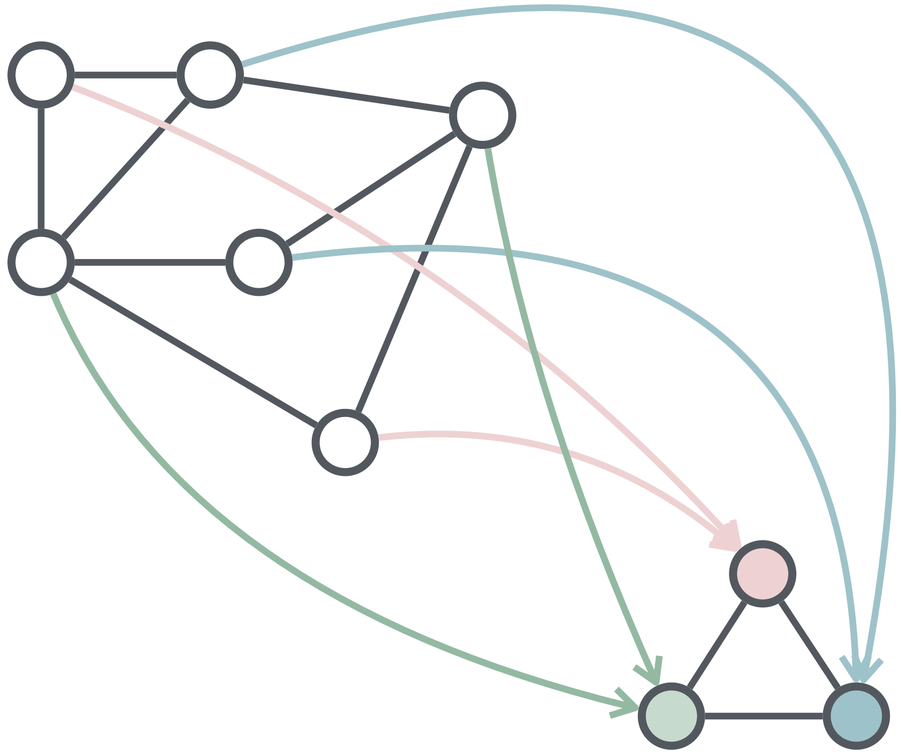


Valid 3-colouring



NP-Hardness

- **3-Colourability** is equivalent to having a homomorphism into the triangle graph.
- The three nodes of the triangle intuitively represent the three colours.
- Note that if there is an edge between v and u, then v, u can't be mapped to the same vertex, i.e., adjacent vertices can't be mapped to the same colour.

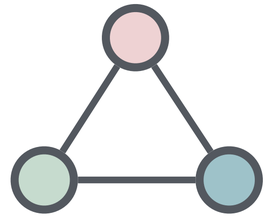


NP-Hardness

This homomorphism into the triangle can be trivially expressed as a conjunctive query.

- Take an input for **3-Colourability**, i.e., a graph G.
- Create a database with relation E for the triangle:

A	*B*
red	green
green	red
red	blue
blue	red
green	blue
blue	green



- Encode the graph as a conjunctive query:

$$q = \{\ () \mid \exists \bar{v} \bigwedge_{\{v_i, v_j\} \in E(G)} E(v_i, v_j) \wedge E(v_j, v_i)\ \}$$

- There is a homomorphism $\mathsf{Tbl}^*(q) \to D$ if and only if G is 3-colourable.

Complexity of CQ -Eval

Theorem

CQ-Eval is **NP**-complete in combined complexity. Moreover, the **NP**-hardness holds already for schemas with a single binary relation symbol.

Complexity of CQ Containment

Recall the Homomorphism Theorem:

$$q_1 \subseteq q_2 \iff \mathsf{Tbl}(q_2) \xrightarrow{hom} \mathsf{Tbl}(q_1)$$

Same reduction applies here too: check whether the query q_1 that represents the triangle is contained in the query q_2 that represents graph G.

Theorem
Deciding CQ Containment is **NP**-complete.

Complexity of CQ Minimisation

Theorem

Checking whether a query q is minimal is **co-NP**-complete.

Intuition: We need to check whether there are *no homomorphisms* into any query obtained by deleting atoms.

Structure of CQs

Structure?

The number of joins is not the only important factor in how difficult it is to evaluate a CQ.

$$q_1 = \{\ (x) \mid\ R(x,y,z) \wedge S(z,w,u) \wedge R(u,v,x)\ \}$$

Which is easier to evaluate ⬆⬇?

$$q_2 = \{\ (x) \mid R(x,y,z) \wedge S(z,w,u) \wedge R(u,v,x) \wedge T(x,y,z,w,u)\ \}$$

q_2 can be solved in time linear in the database

q_1 cannot (under standard assumptions from fine-grained complexity theory).

Structure

To understand why, we need to understand how the structure of CQs is connected to their evaluation.

But what is the structure of a CQ?

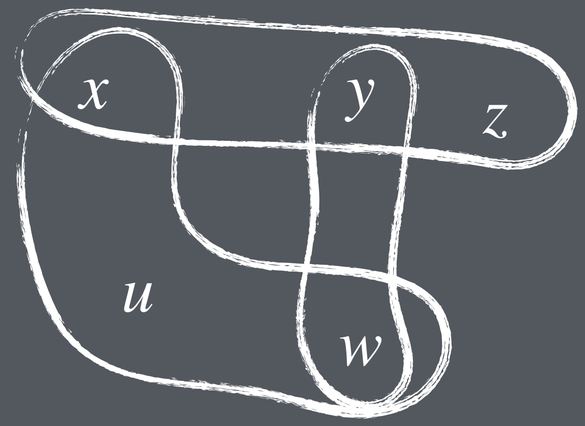
Hypergraphs

- A hypergraph H is a pair (V_H, E_H).
- V_H is the set of vertices, just as in a graph.
- The set of hyperedges $E_H \subseteq 2^{V(H)}$ is some set of sets of vertices.
- Graphs are a special case of hypergraphs where every hyperedge has size exactly 2.

Example hypergraph H

$$V_H = \{x, y, z, u, w\}$$

$$E_H = \{\ \{x, y, z\}, \{y, u\}, \{x, u, w\}\ \}$$



CQs as Hypergraphs

$$q = \{ \bar{y} \mid \exists \bar{z}\, R_1(\bar{x}_1) \wedge R_2(\bar{x}_2) \wedge \cdots \wedge R_k(\bar{x}_k) \}$$

We capture the structure of q as hypergraph H in the following way:

- $V_H = vars(q)$, i.e., we have a vertex for each variable in the query.
- For every atom $R(x_1, \ldots, x_{\#R})$ we add the hyperedge $\{x_1, \ldots, x_{\#R}\}$ to E_H.

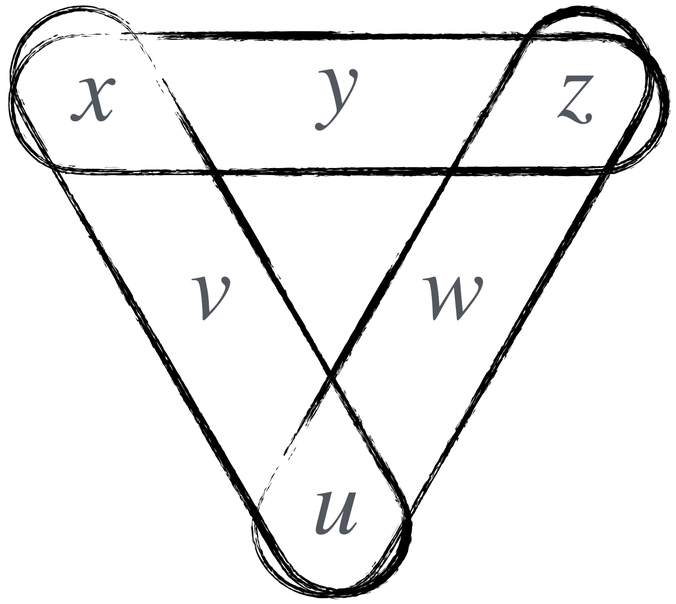
Example

$$q_1 = \{\ (x) \mid\ R(x, y, z) \wedge S(z, w, u) \wedge R(u, v, x)\ \}$$

$V_H = vars(q_1)$

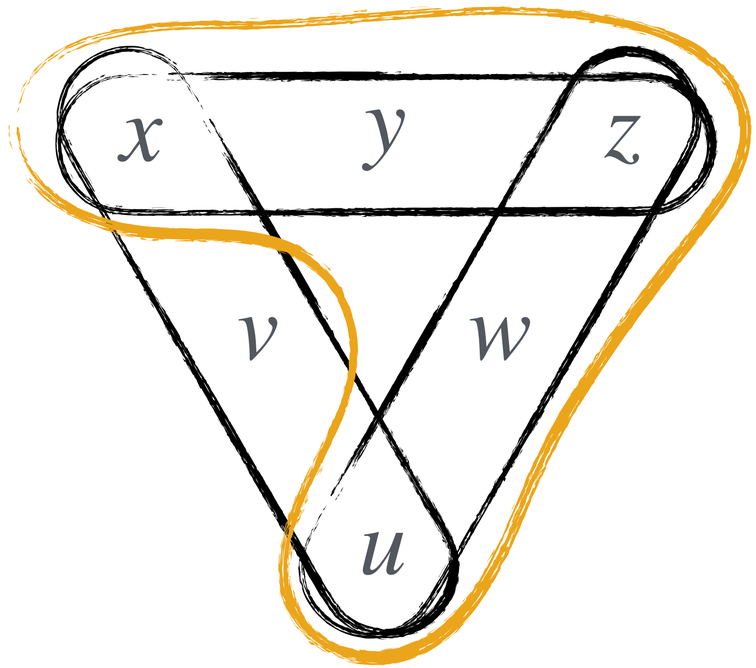
Add edges for each atom.

$R(x, y, z)$ becomes edge $\{x, y, z\}$,
and so on.



Example

$$q_2 = \{\ (x) \mid R(x, y, z) \wedge S(z, w, u) \wedge R(u, v, x) \wedge T(x, y, z, w, u)\ \}$$



When is the structure of a CQ good for evaluation?

Join Trees & Yannakakis Algorithm

Acyclic Hypergraphs

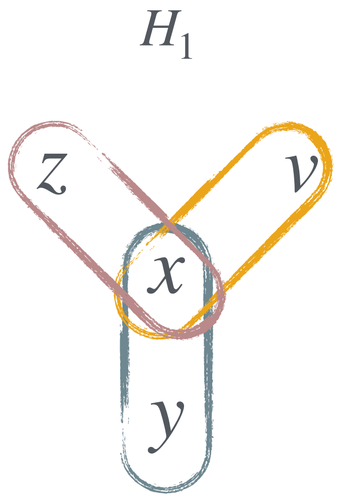
A join tree for a hypergraph H is a tree T with a labelling function $\lambda : V(T) \to E(H)$ that labels every node of the tree with a an edge of H such that:

i) For every $e \in E(H)$ there is *exactly one* node n of the tree T such that $\lambda(n) = e$

ii) For every vertex $v \in V(H)$, the nodes for which $v \in \lambda(n)$ are connected. (connectedness condition)

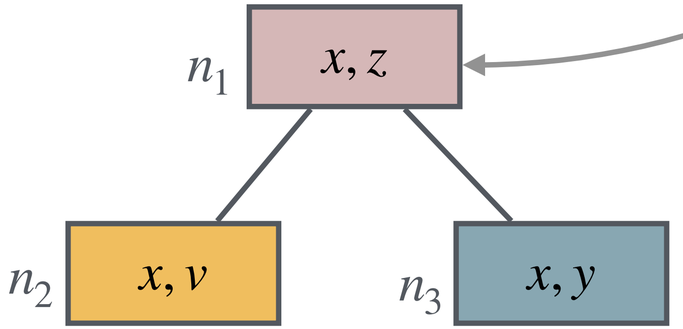
We say that a hypergraph H is α-acyclic if it has a join tree.

Example



Example join tree for H_1

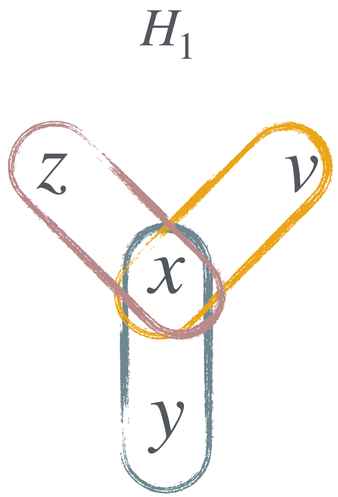
We will denote the λ labels like this



Let's check the two conditions:

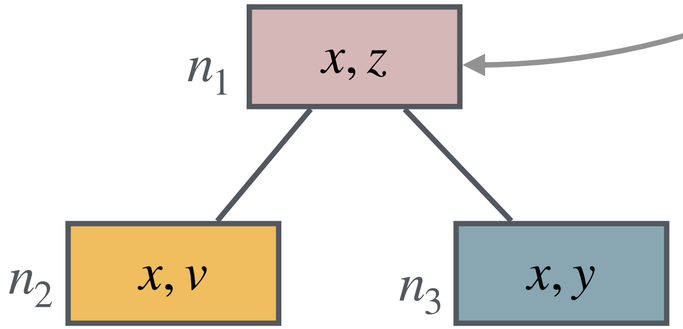
- Every edge is mapped to some label ✔
- Connectedness condition?

Example



Example join tree for H_1

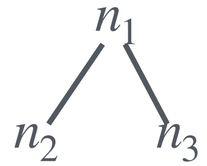
We will denote the λ labels like this



Let's check the two conditions:

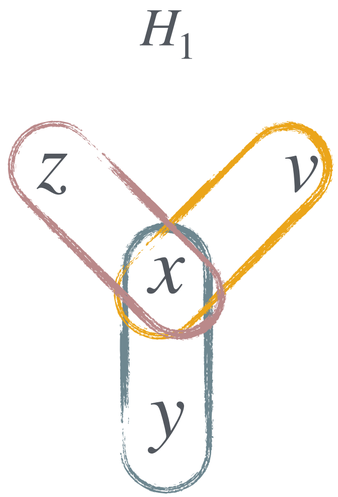
- Every edge is mapped to some label ✔
- Connectedness condition?

Nodes that contain x



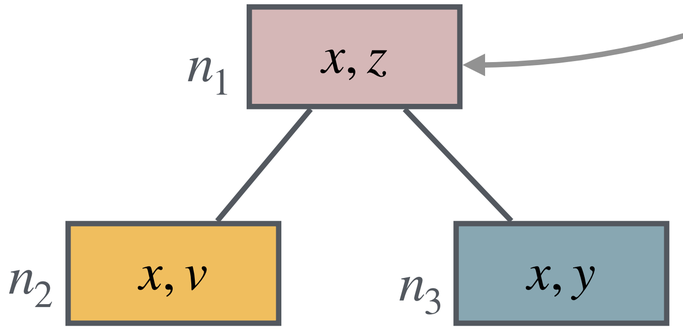
Connected ✓

Example



Example join tree for H_1

We will denote the λ labels like this



Let's check the two conditions:

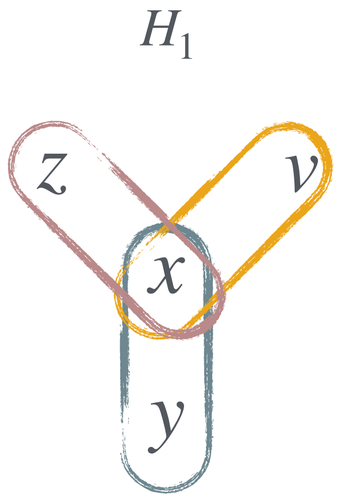
- Every edge is mapped to some label ✔
- Connectedness condition?

Nodes that contain y

n_3 Connected ✓

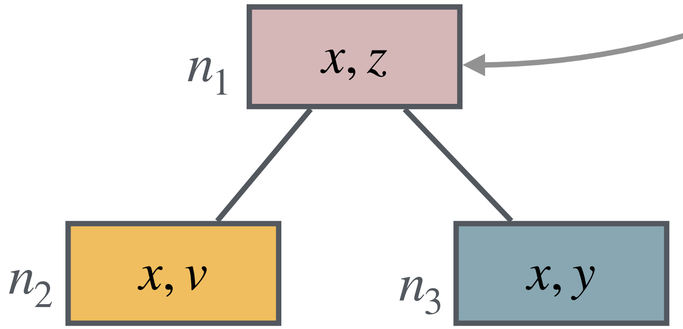
Same for v and z

Example



Example join tree for H_1

We will denote the λ labels like this

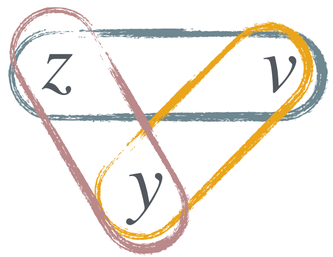


Let's check the two conditions:

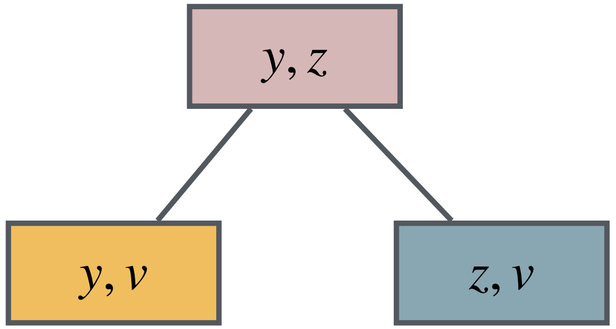
- Every edge is mapped to some label ✔
- Connectedness condition ✔

Example 2

H_2



H_2 has no join tree!

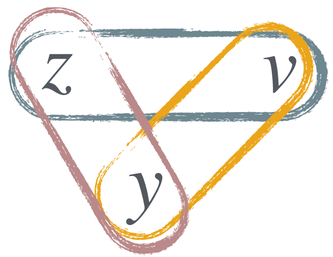


v is in the blue and yellow node but not in the red.

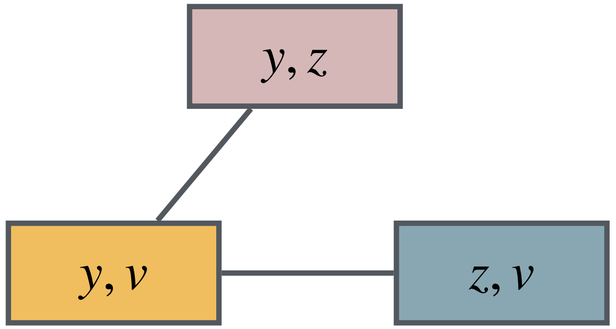
Violates connectedness condition!

Example 2

H_2



H_2 has no join tree!

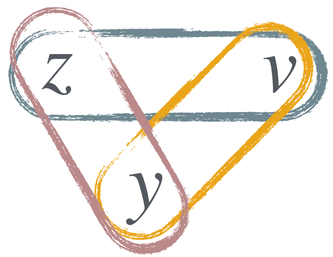


z is in the blue and red node but not in the yellow.

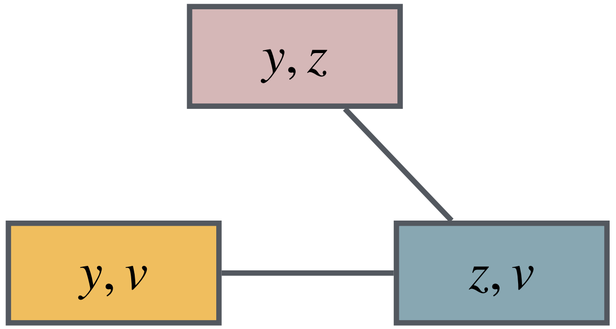
Violates connectedness condition!

Example 2

H_2



H_2 has no join tree!



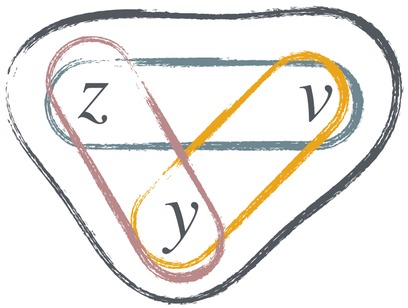
This covers all possible trees on three nodes. We see that none of them is a join tree!

y is in the yellow and red node but not in the blue.

Violates connectedness condition!

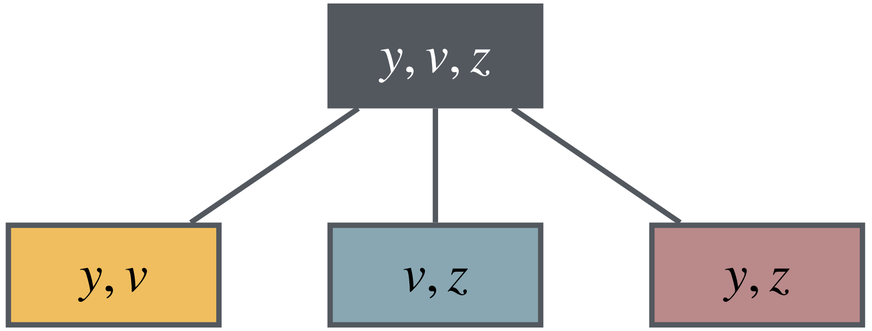
Example 3

H_3



That is, the triangle
+ the edge $\{y, v, z\}$

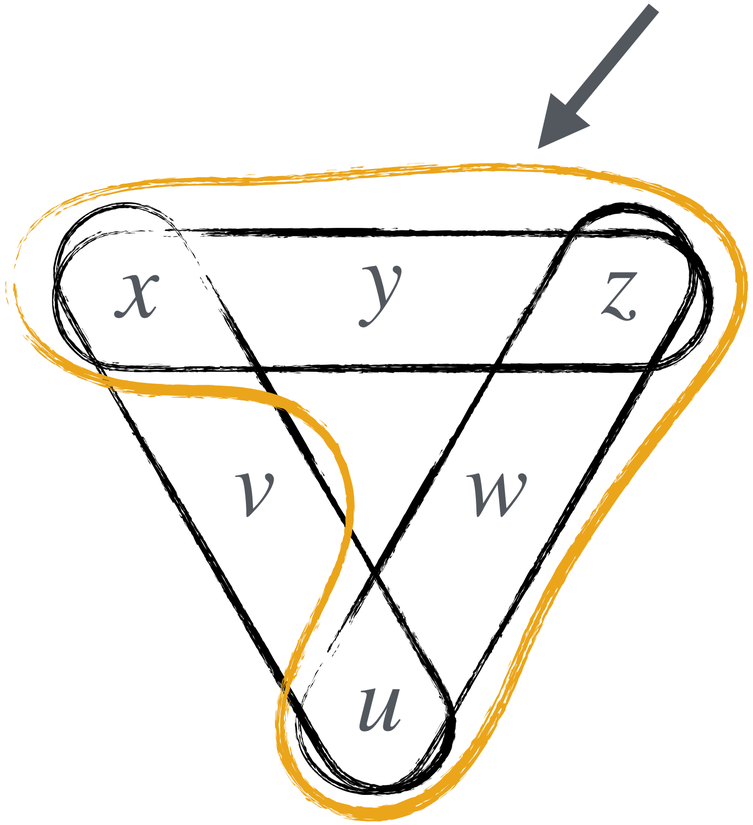
H_3 has a join tree!

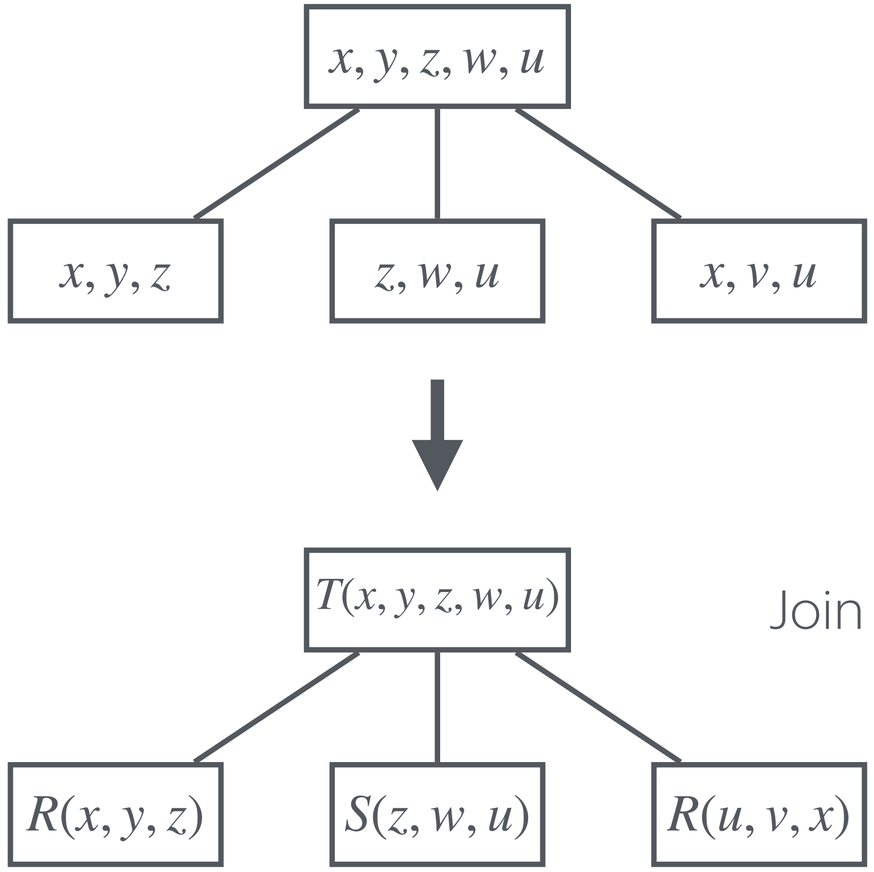


H_3 is α-acyclic even though it contains a cyclic subhypergraph!

Back to Queries

$$q_2 = \{\ (x) \mid R(x, y, z) \wedge S(z, w, u) \wedge R(u, v, x) \wedge T(x, y, z, w, u)\ \}$$





Join tree of CQ q

We can use join trees to design efficient algorithms for CQ evaluation.

The Semi-join Operation $\ltimes$

$$R \ltimes S := \pi_{\text{attr}(R)}\left(R \bowtie S\right)$$

Instead of creating the combined tuples as in a join,
a semi-join only keeps the rows in R that have a join partner in S

A	B	C
3	2	5
6	7	9
5	5	5

$\ltimes$

B	C	D
1	9	1
2	5	3
5	5	1
1	3	2

$=$

A	B	C
3	2	5
6	7	9
5	5	5

The Semi-join Operation $\ltimes$

$$R \ltimes S := \pi_{\text{attr}(R)}\left(R \bowtie S\right)$$

Instead of creating the combined tuples as in a join,
a semi-join only keeps the rows in R that have a join partner in S

Unlike a join, a semi-join cannot create larger relations:

$$|R \ltimes S| \leq |R|$$

In fact, we can compute any semi-join in
$O(n \log n)$ time, where $n = |R| + |S|$.

Show this in a theory exercise!

Yannakakis' Algorithm

Let (T, λ) be a join tree of a CQ q and root it arbitrarily.

Given a database D we can decide $q(D) \neq \emptyset$ as follows:

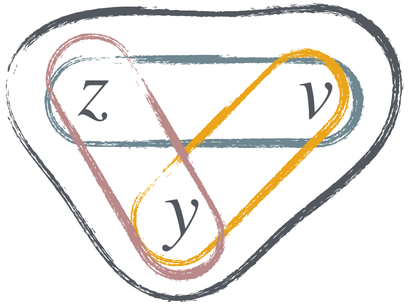
1. Assign to each node labeled with atom $A(\bar{x})$ the relation A^D
   (and rename columns according to the variables in the respective atom).
   We write $rel(N)$ for the relation associated to node n.

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$
   where $c_1, \ldots, c_\ell$ are the children of n

3. At the end, for the root r we have $rel(r) \neq \emptyset$ if and only if $q(D) \neq \emptyset$.

Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z)$$



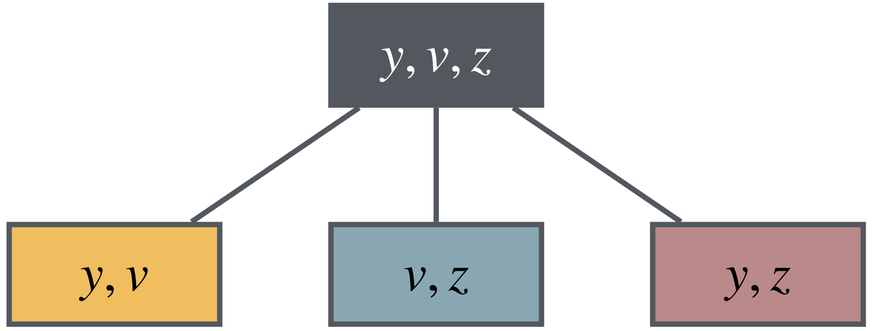


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

1. Assign to each node labeled with atom $A(\bar{x})$ the relation A^D (and rename columns according to the variables in the respective atom).

   We write $rel(N)$ for the relation associated to node n.

$rel(n_1) = T^D$

v	y	z
3	1	2
3	3	4
7	2	1
5	9	0

$rel(n_2) = R^D$

y	v
1	3
3	2
3	3
2	7

$rel(n_3) = R^D$

v	z
1	3
3	2
3	3
2	7

$rel(n_4) = S^D$

z	y
1	2
2	3
4	3
0	9

Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

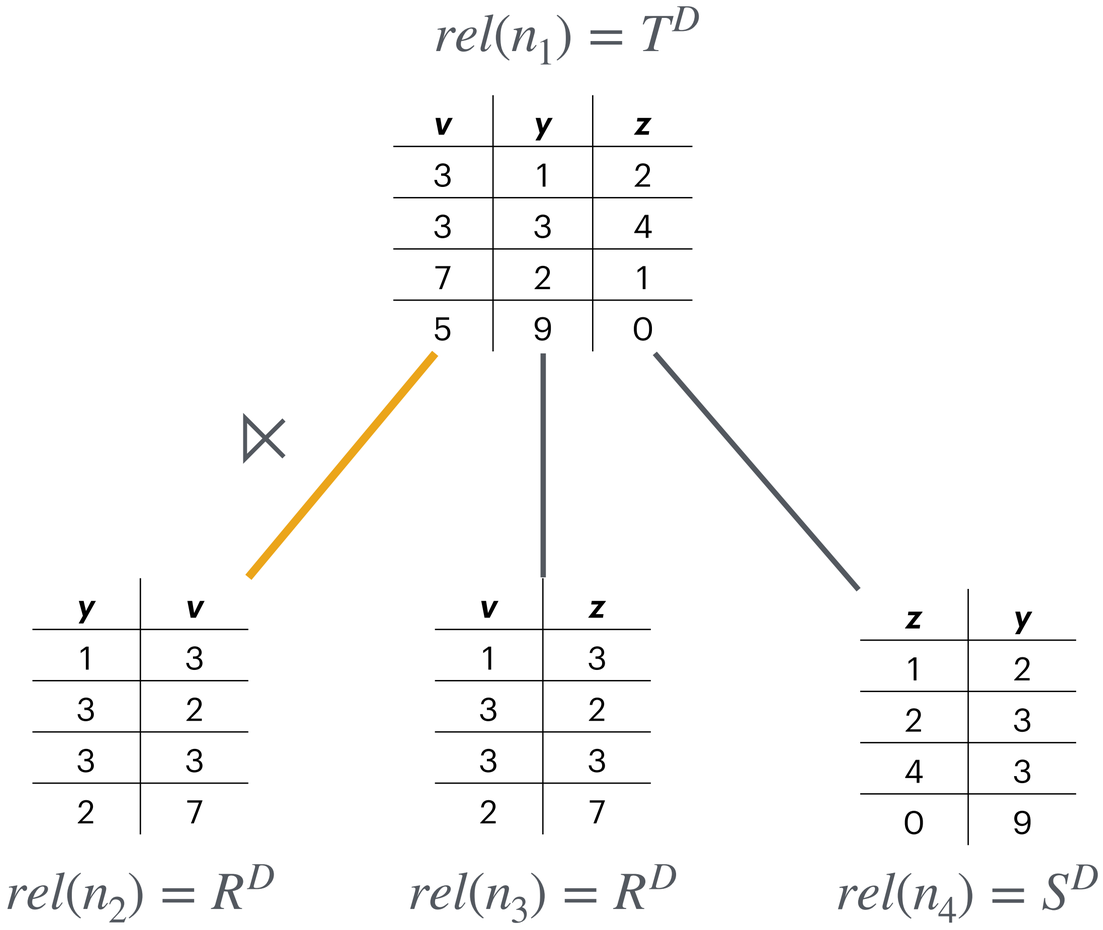


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

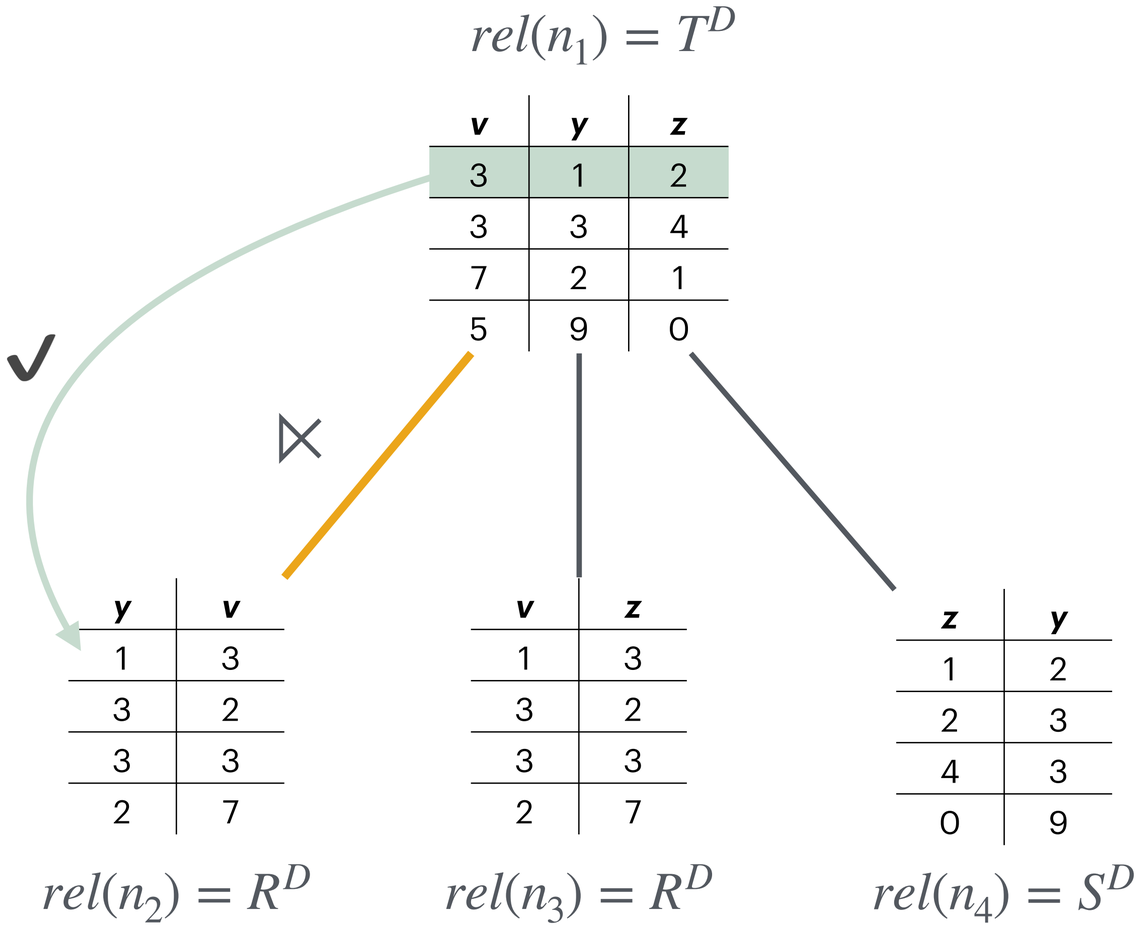


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

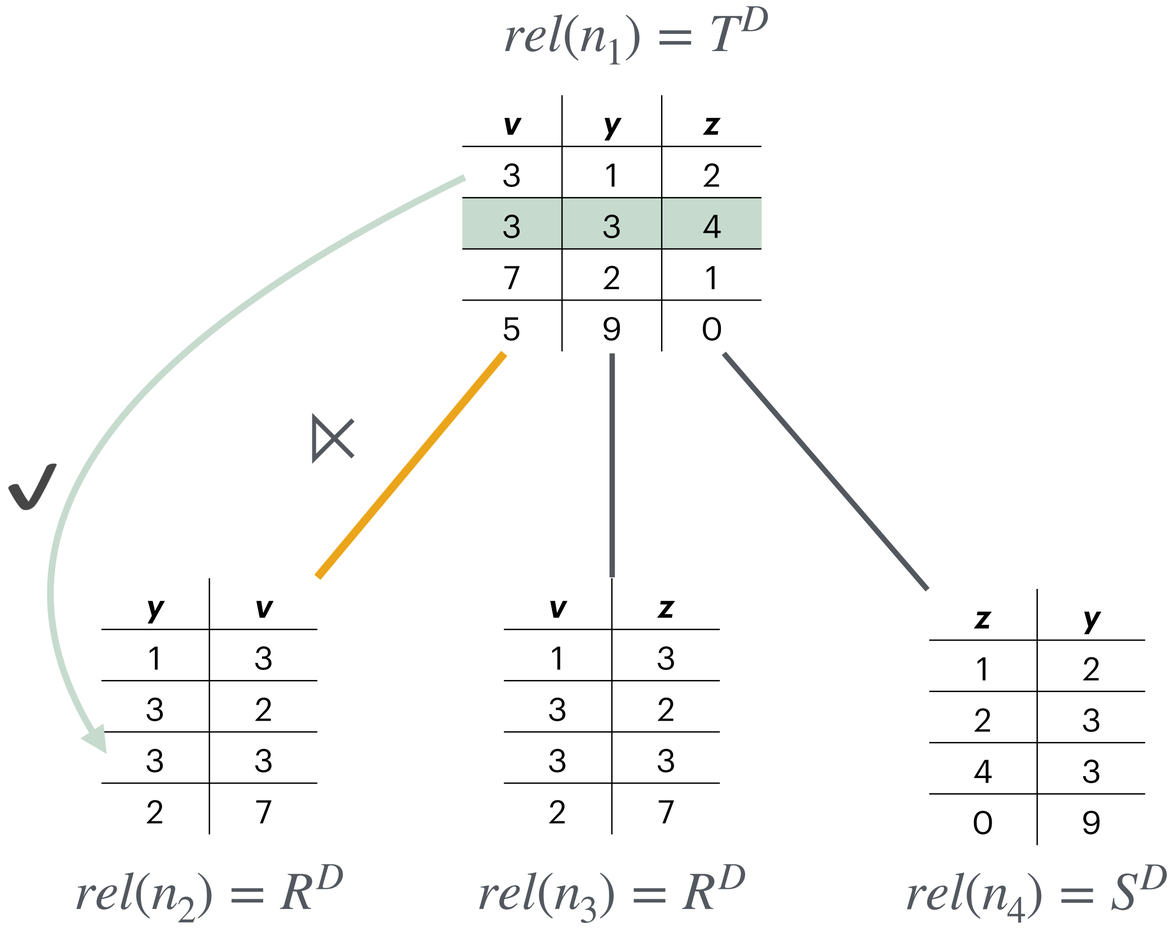


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

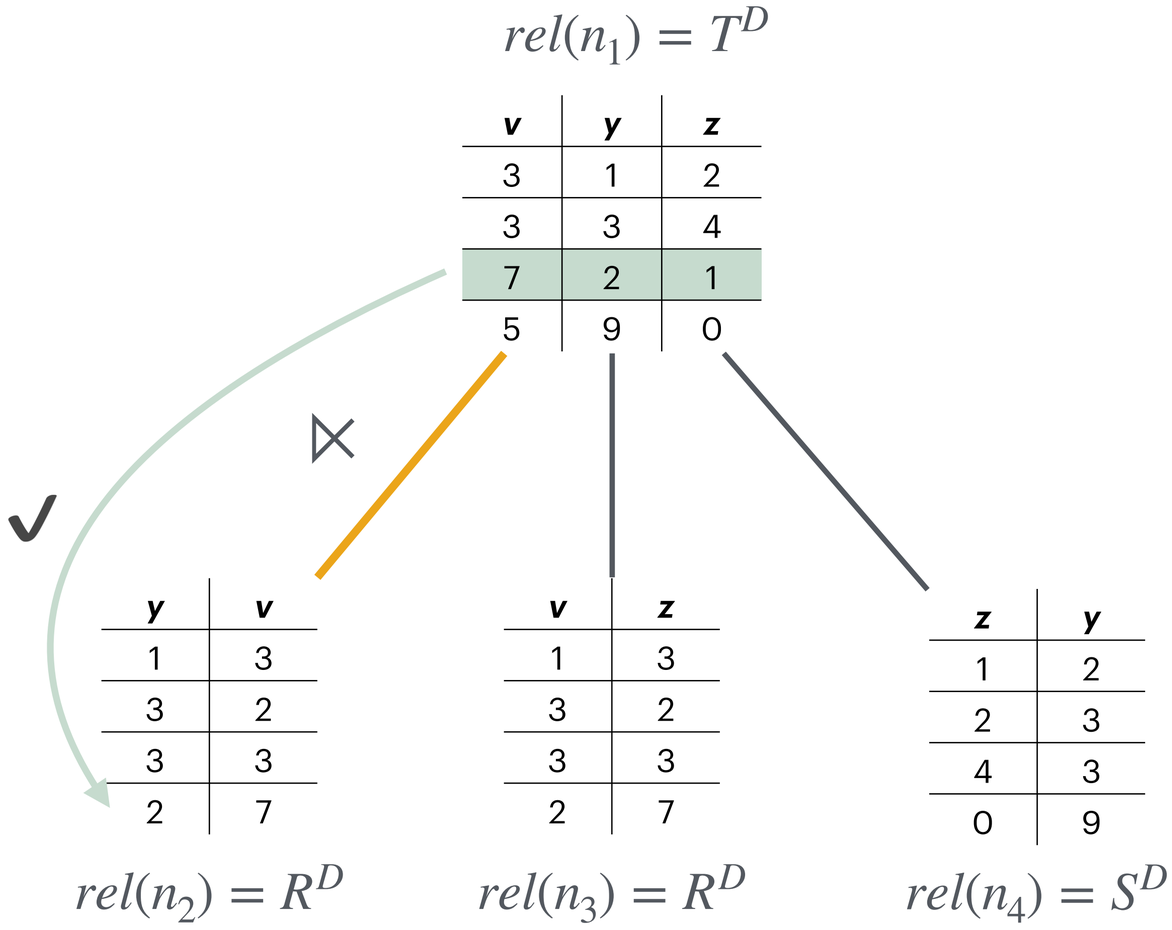
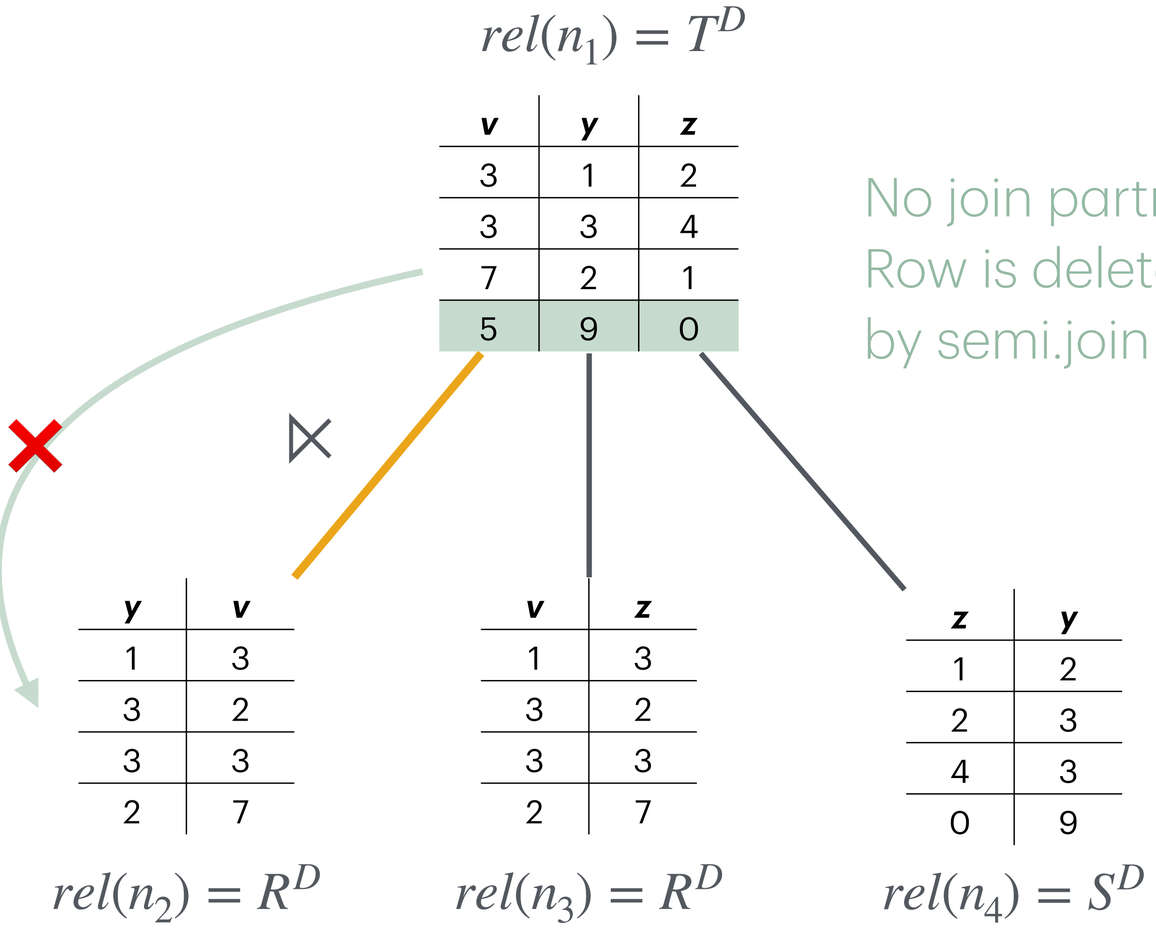


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n



No join partner
Row is deleted
by semi.join

Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

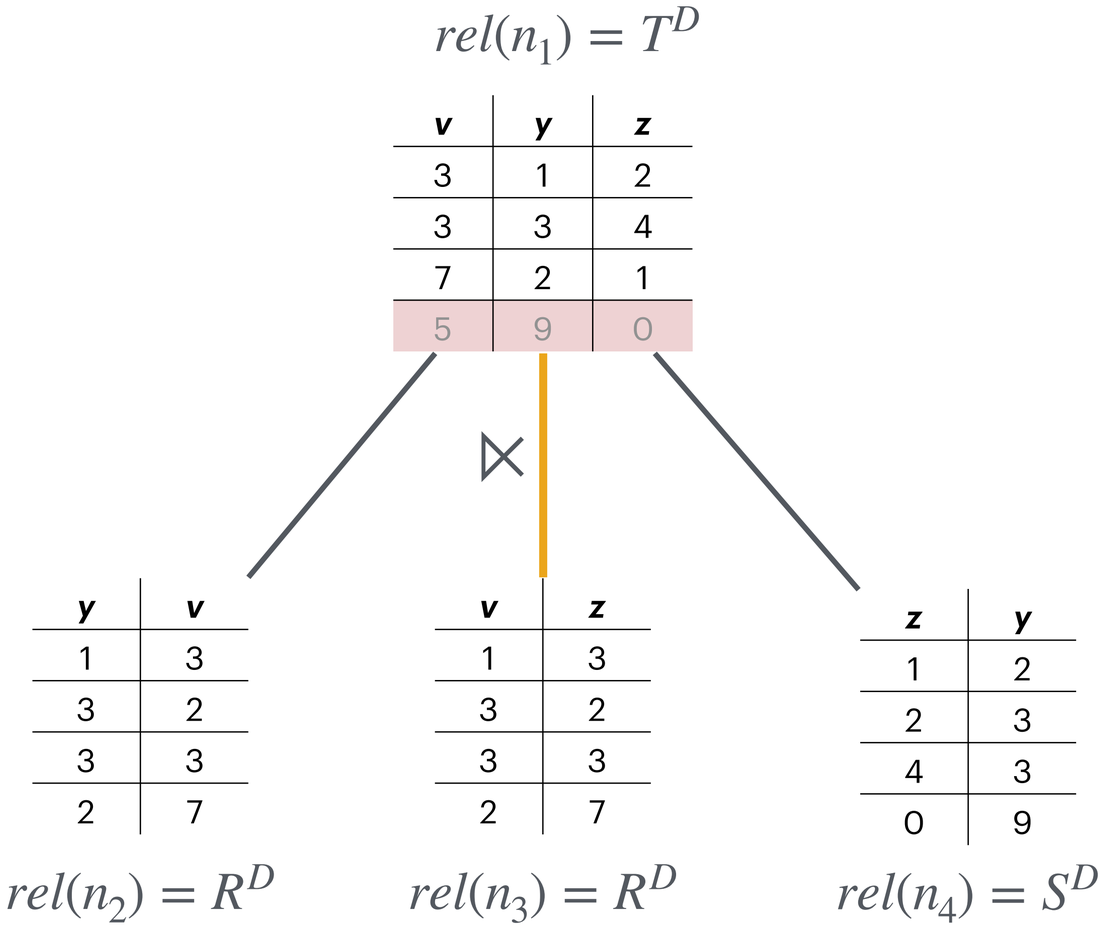


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

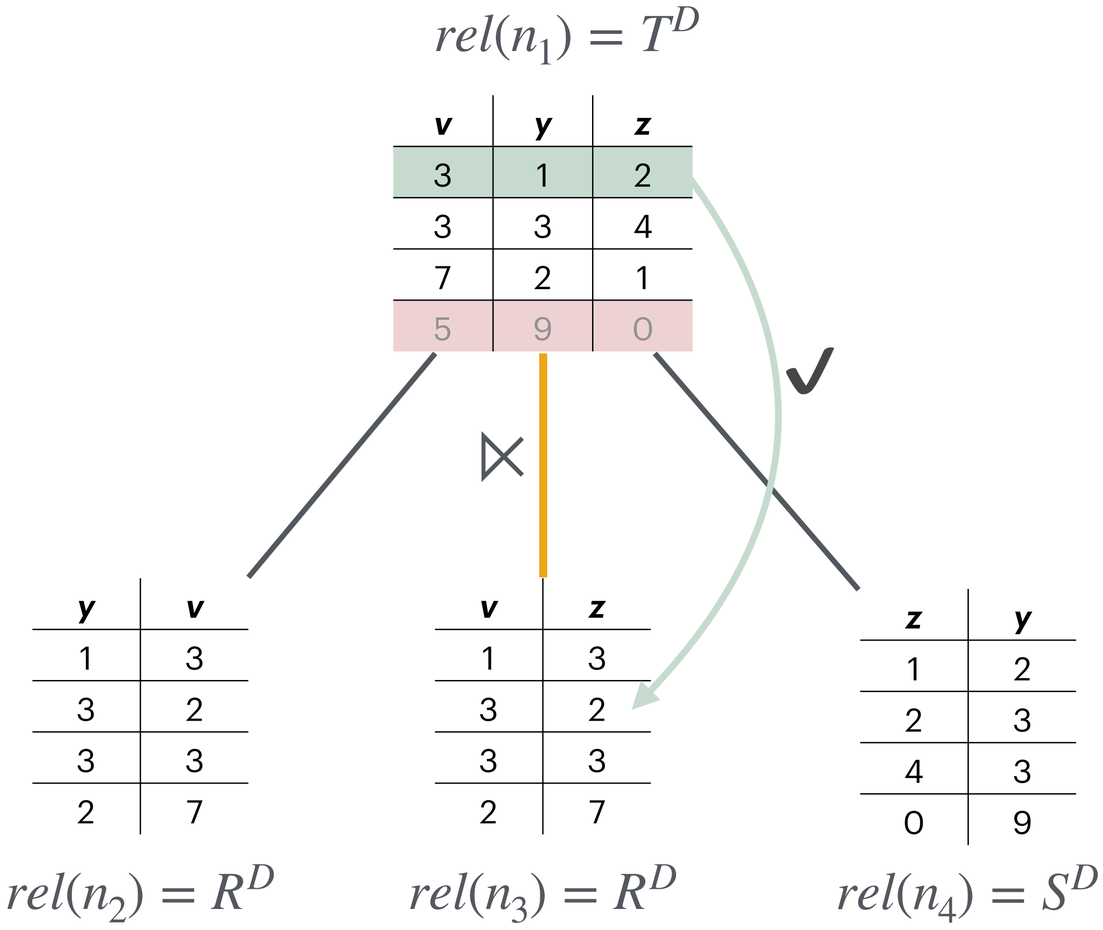


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

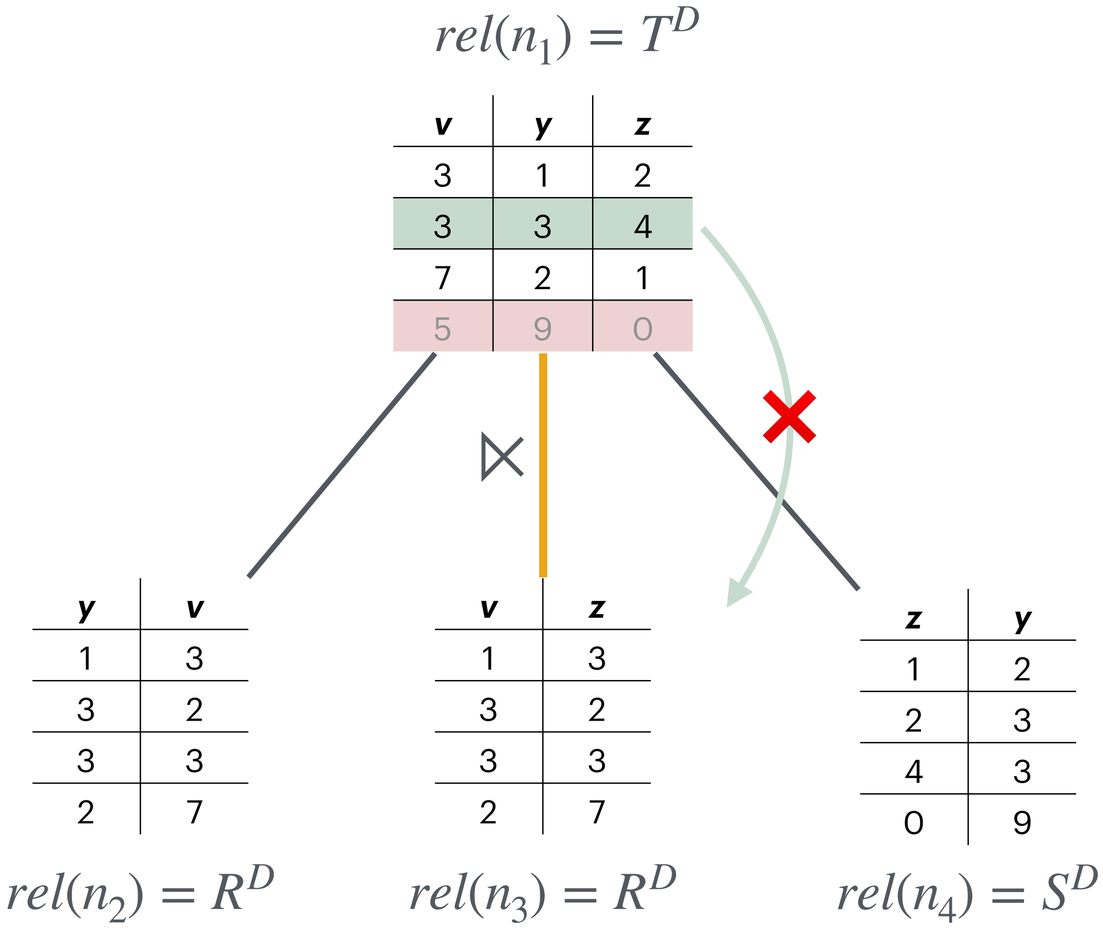


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

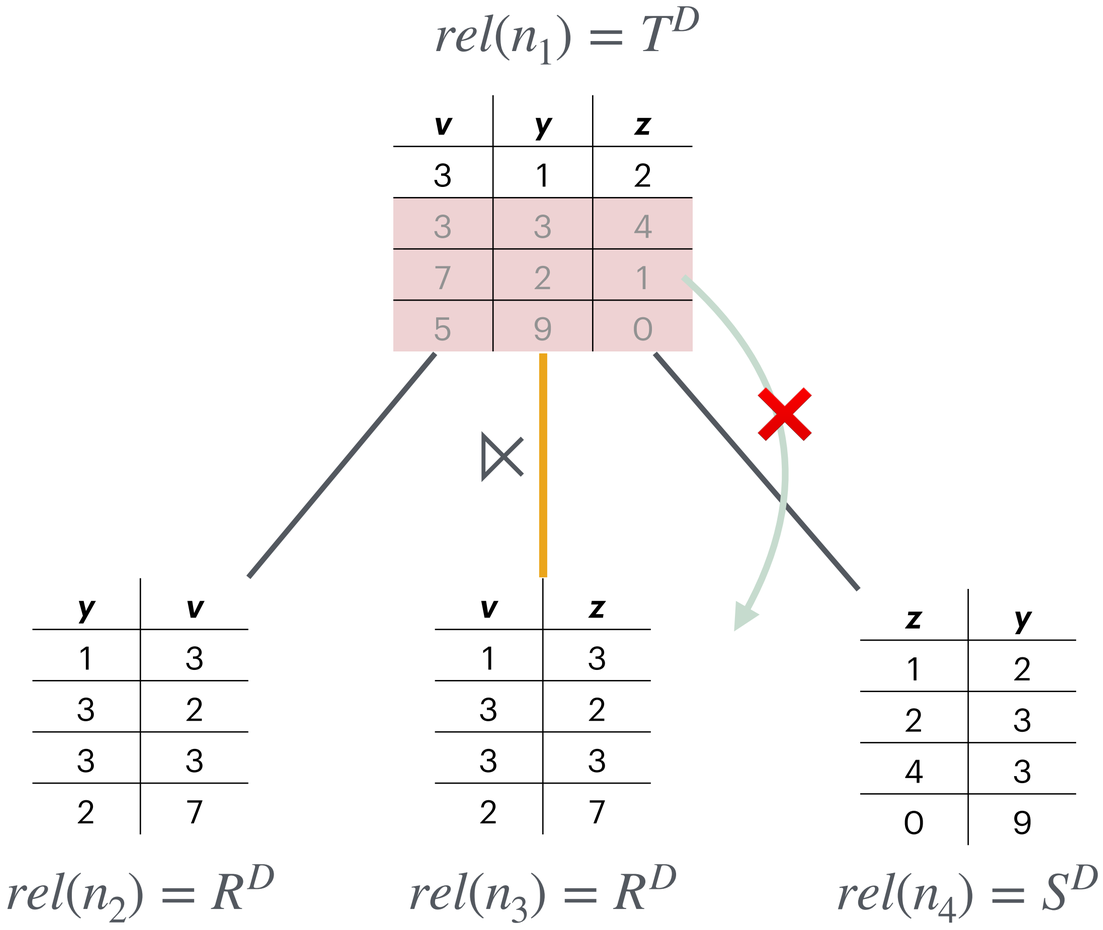


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

2. In a bottom-up traversal: update $rel(n)$ to $\left(rel(n) \ltimes rel(c_1)\right) \ltimes \cdots \ltimes rel(c_\ell)$ where $c_1, \ldots, c_\ell$ are the children of n

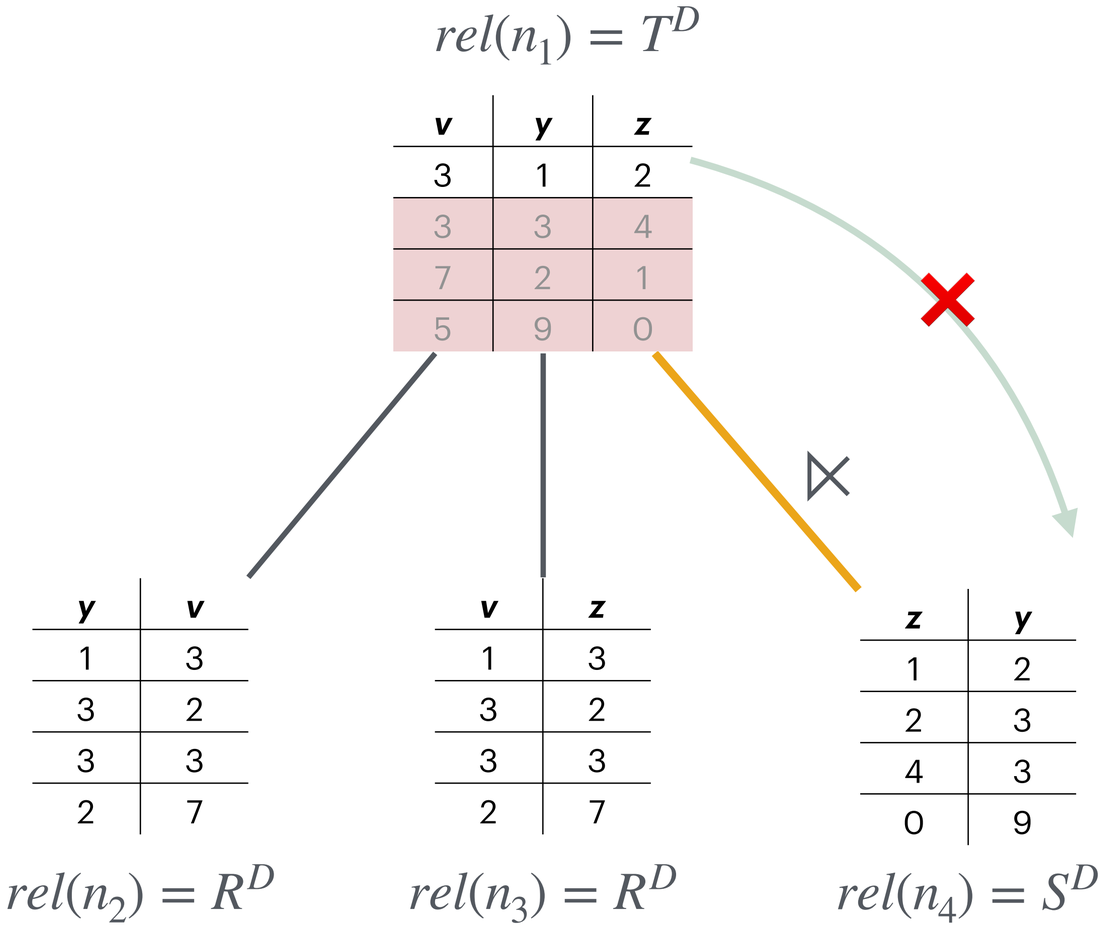
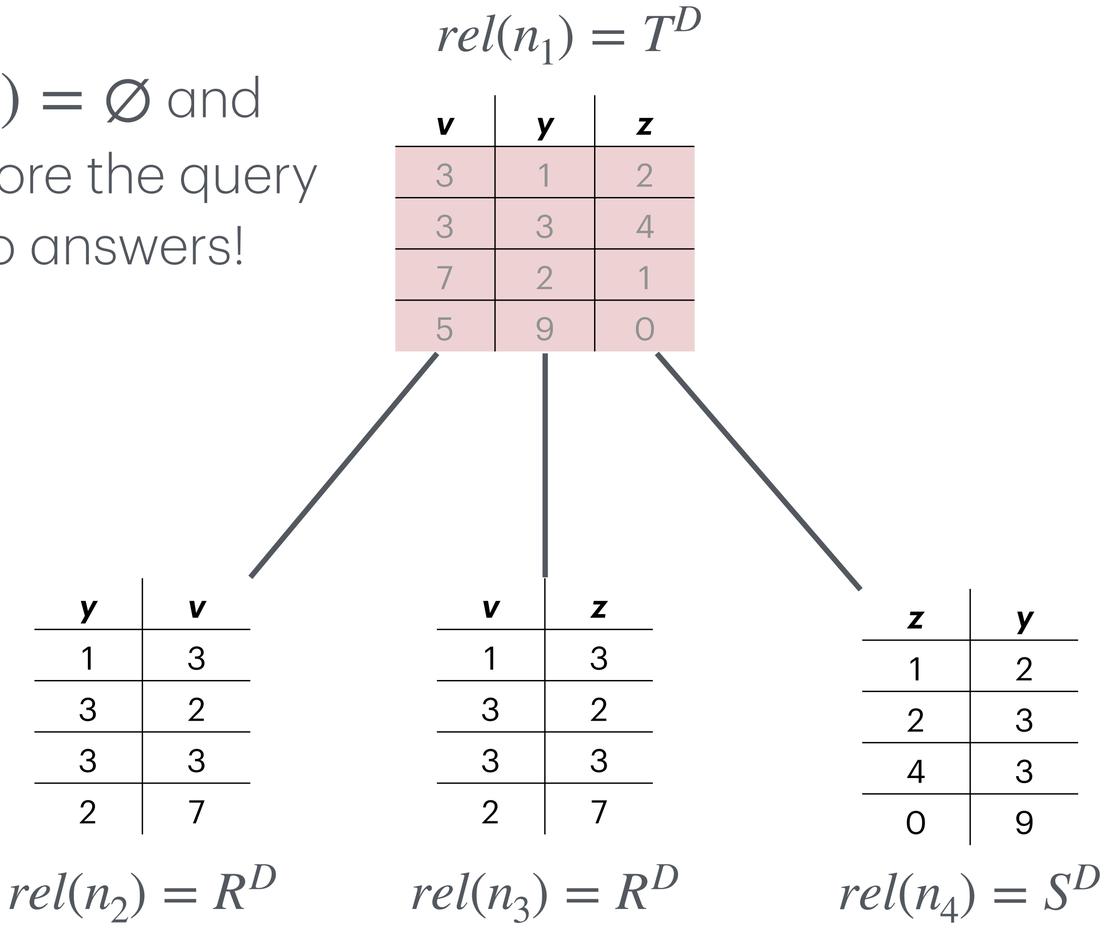


Illustration of Yannakakis' Algorithm

$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$

$rel(n_1) = \emptyset$ and therefore the query has no answers!

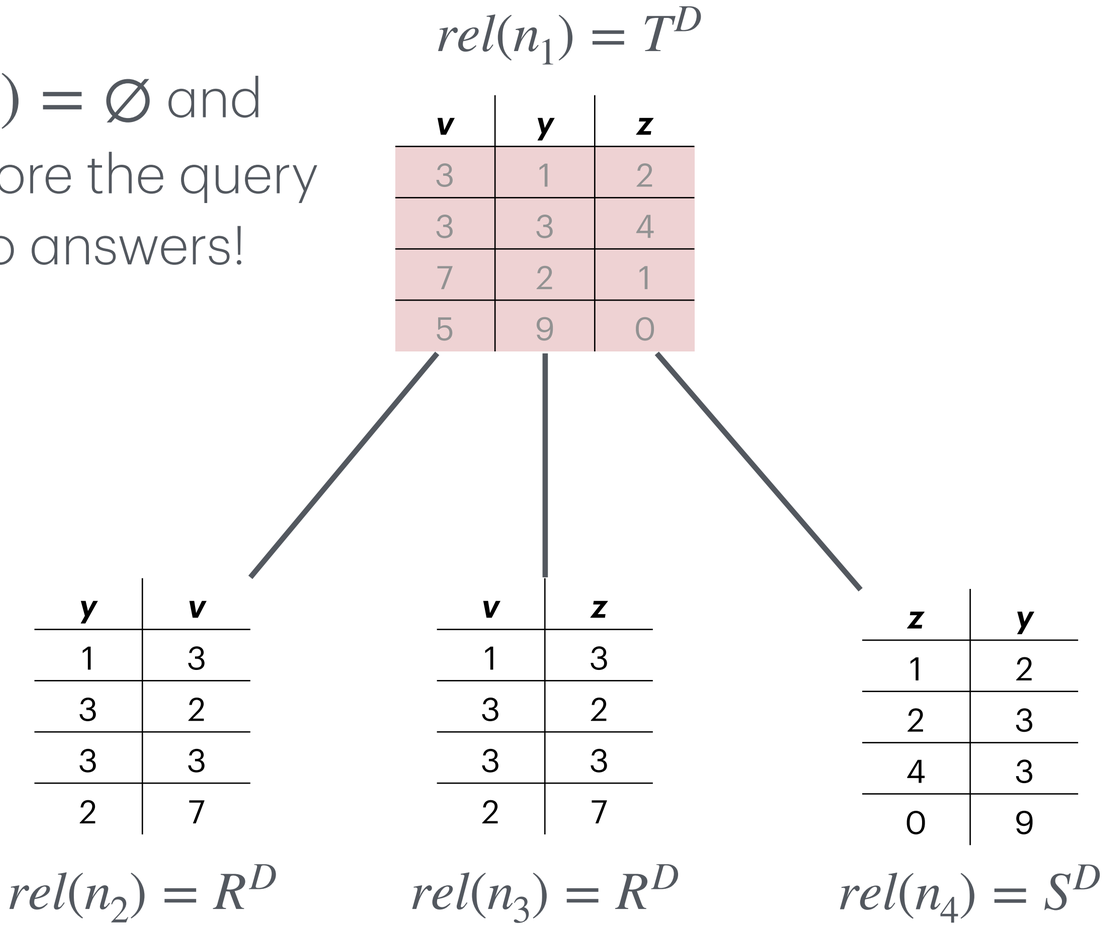


3. At the end, for the root r we have $rel(r) \neq \emptyset$ if and only if $q(D) \neq \emptyset$.

Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z)$$

$rel(n_1) = \emptyset$ and therefore the query has no answers!



How much did it cost to run the algorithm?

$Atoms(q) - 1$ many semi-joins.

Semi-joins each require $O(n \log n)$ time.

In general: $\tilde{O}(|Atoms(q)| \cdot |D|)$. A big improvement over NP.

Beyond α-Acyclicity

In most real-world settings the vast majority of queries are α-acyclic. A large study [1] of over 56mio unique SPARQL queries (a graph query language) found that over 99.9% of queries were acyclic.

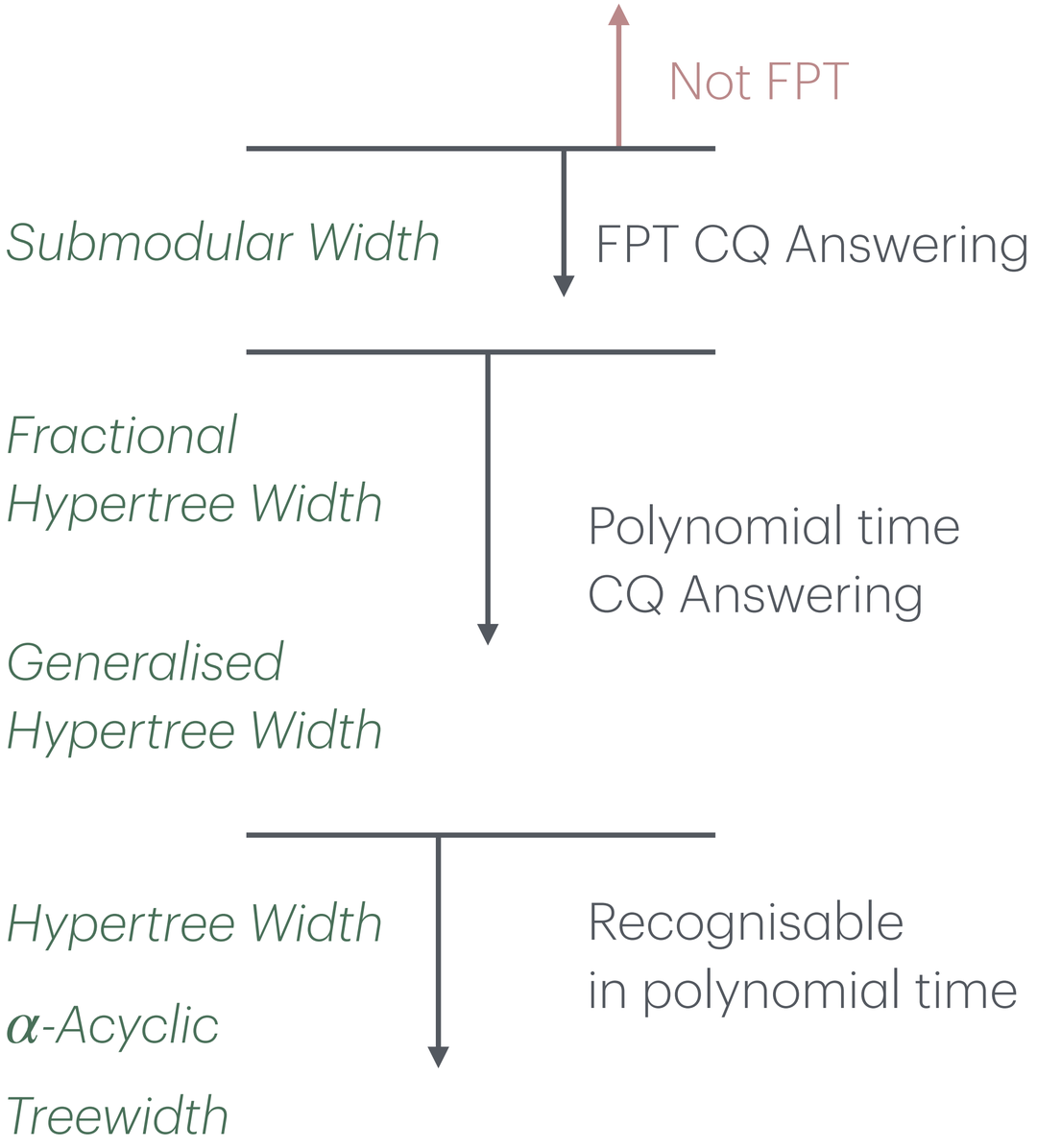
[1] Bonifati, Angela, Wim Martens, and Thomas Timm. "An analytical study of large SPARQL query logs." https://www.vldb.org/pvldb/vol11/p149-bonifati.pdf

However, cyclic queries do come up, and we would like to know if we can do something like Yannakakis' algorithm for those queries. This has lead to the development of *hypertree decompositions* and related notions. They generalize what we've seen here beyond acyclicity!

Explore more in a literature exercise!

A Glimpse Beyond

- The connection between structure and complexity has been an active research topic over the last decades.
- There is a rich theory generalising the concept of α-acyclicity to not only identify linear time queries.



Enumerating Answers

Efficient decision is nice, but what if we want to get the answers?

Enumeration

- For enumeration, quantification of variables will matter again.
- We will focus on the special case of full CQs, i.e., queries where no variables are existentially quantified.

Example full CQ

$$q = \{ (v, x, y, z) \mid R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v) \}$$

Yannakakis' Algorithm — for Enumeration

Let (T, λ) be a join tree of a CQ q and root it arbitrarily.
Given a database D we can decide $q(D) \neq \emptyset$ as follows:

1. Assign to $rel(n)$ to each node n as before.

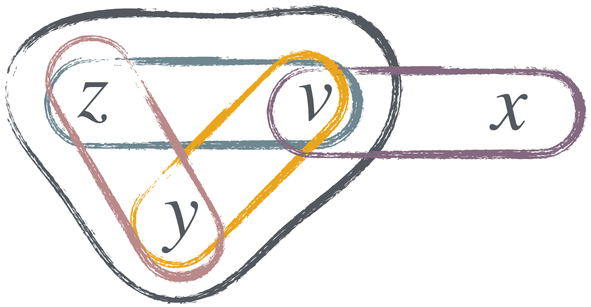
2. In a bottom-up traversal: update $rel(n)$ to $\big(rel(n) \ltimes rel(c_1)\big) \ltimes \cdots \ltimes rel(c_\ell)$
   where $c_1, \ldots, c_\ell$ are the children of n.

3. In a top-down traversal: update $rel(n)$ to $\big(rel(n) \ltimes rel(p)\big)$
   where p is the parent of n.

4. Every tuple left after this process will be part of an output,
   we can collect them one by one in a final bottom-up process.

Example

$$q = \{ (v, x, y, z) \mid R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v) \}$$



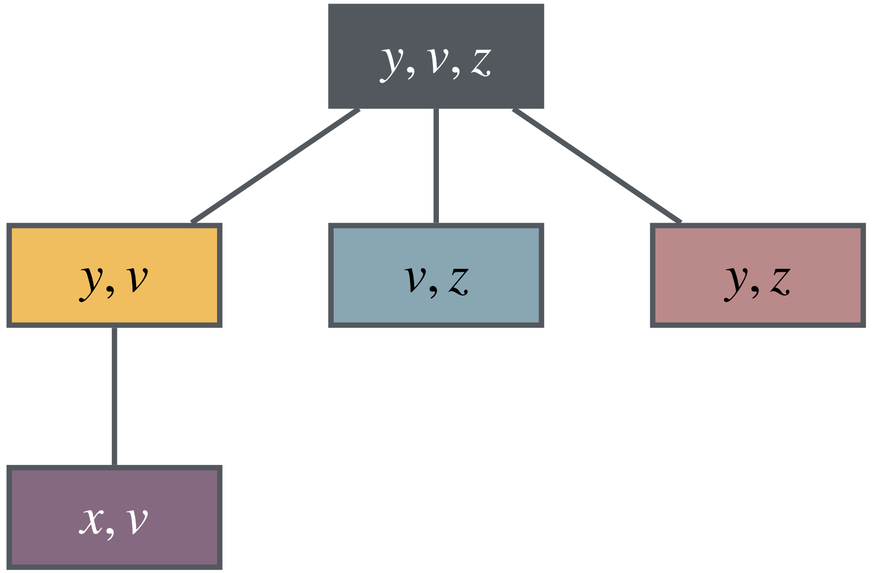


Illustration of Yannakakis' Algorithm

$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$

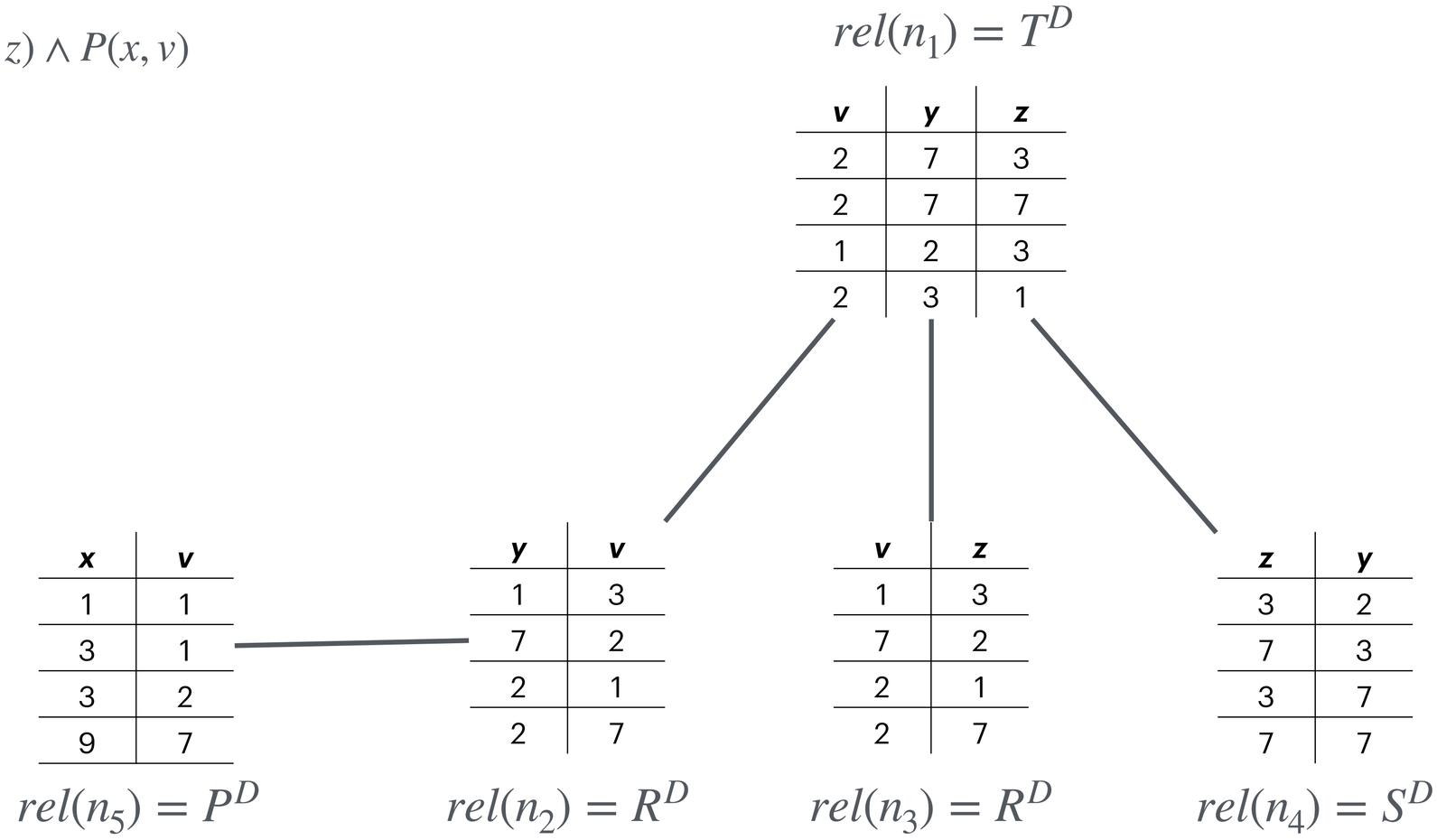


Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

2. The bottom up semi-joins

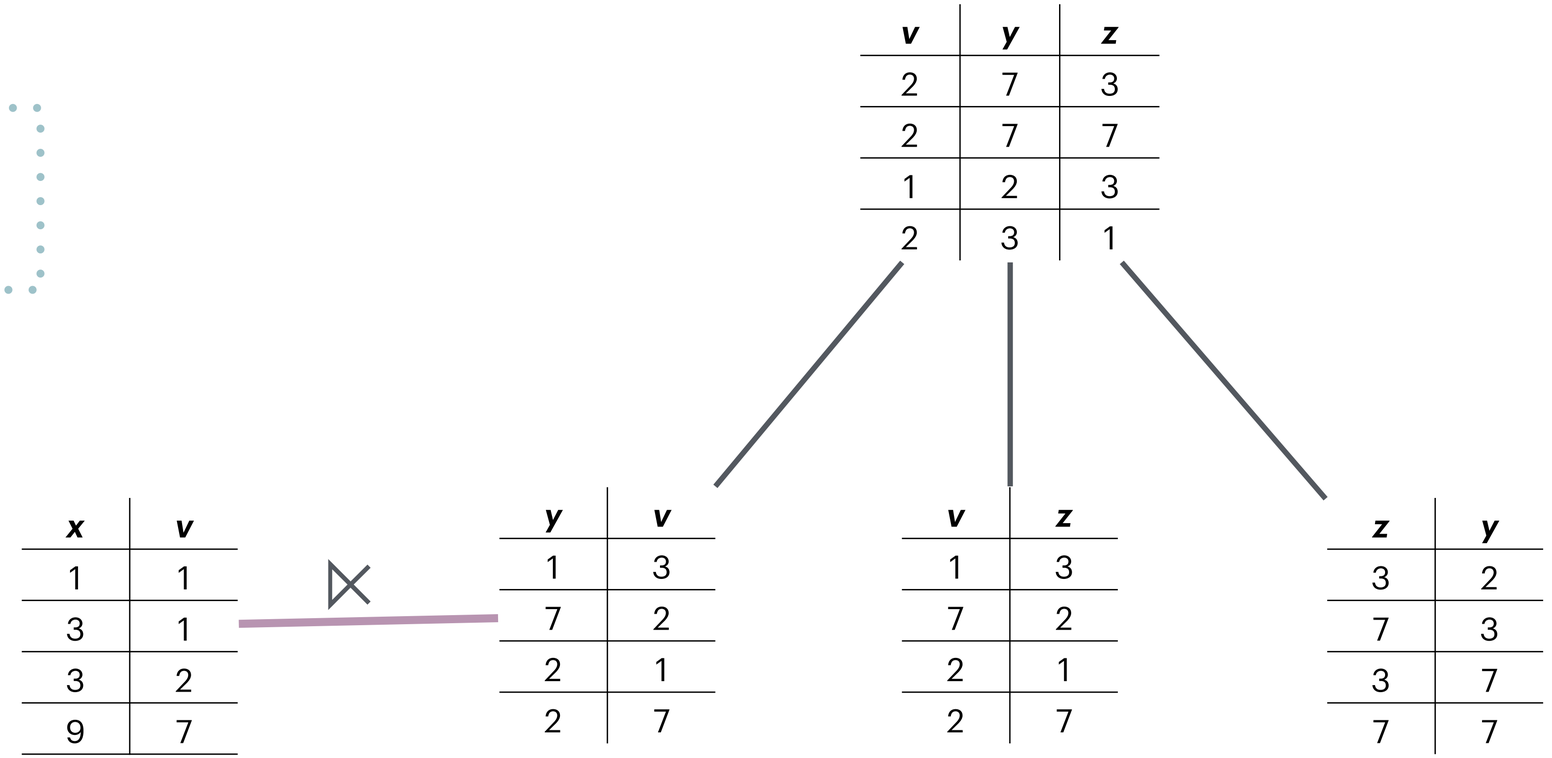


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z) \wedge P(x,v)$$

2. The bottom up semi-joins

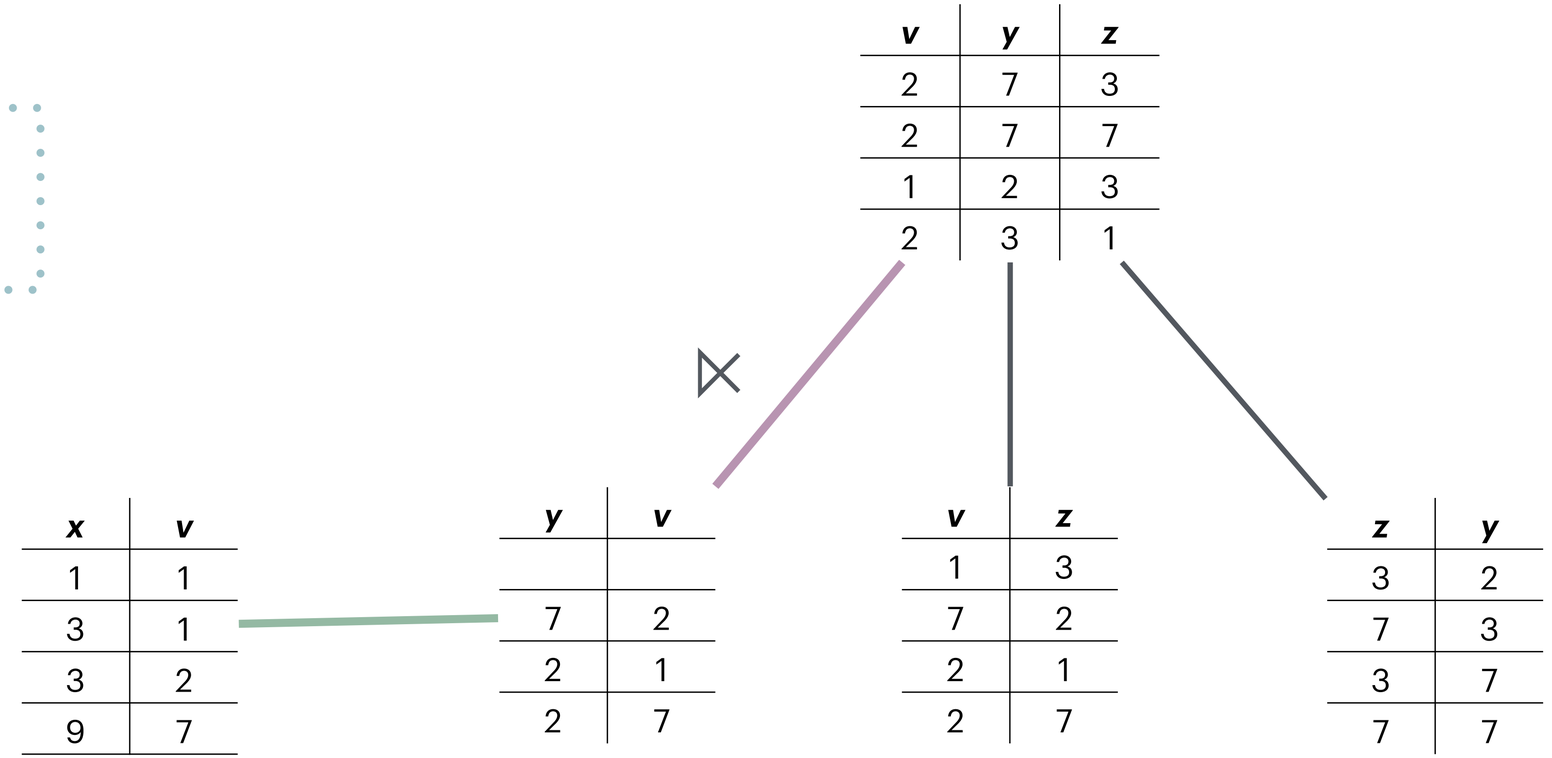


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z) \wedge P(x,v)$$

2. The bottom up semi-joins

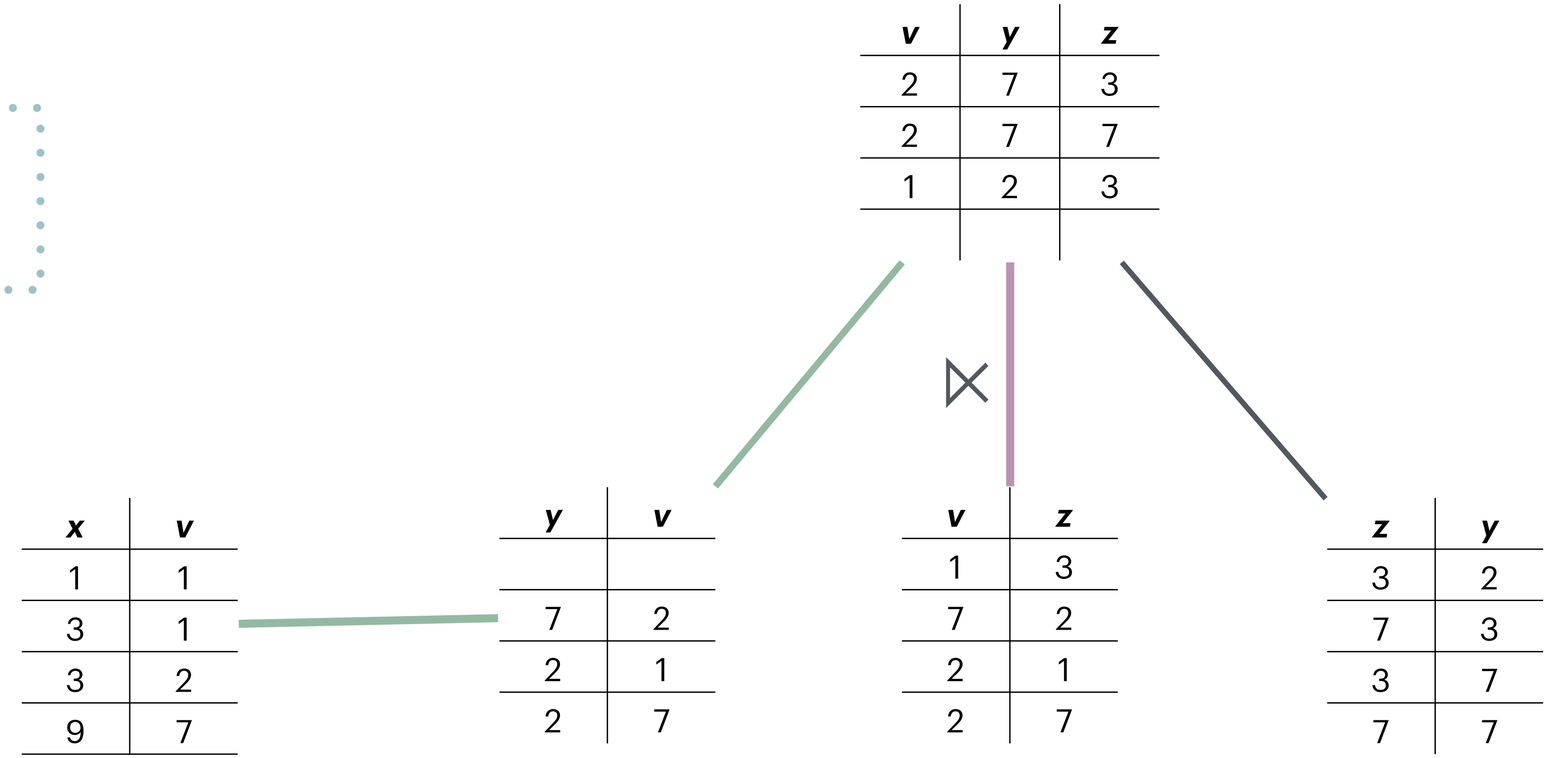


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z) \wedge P(x,v)$$

2. The bottom up semi-joins

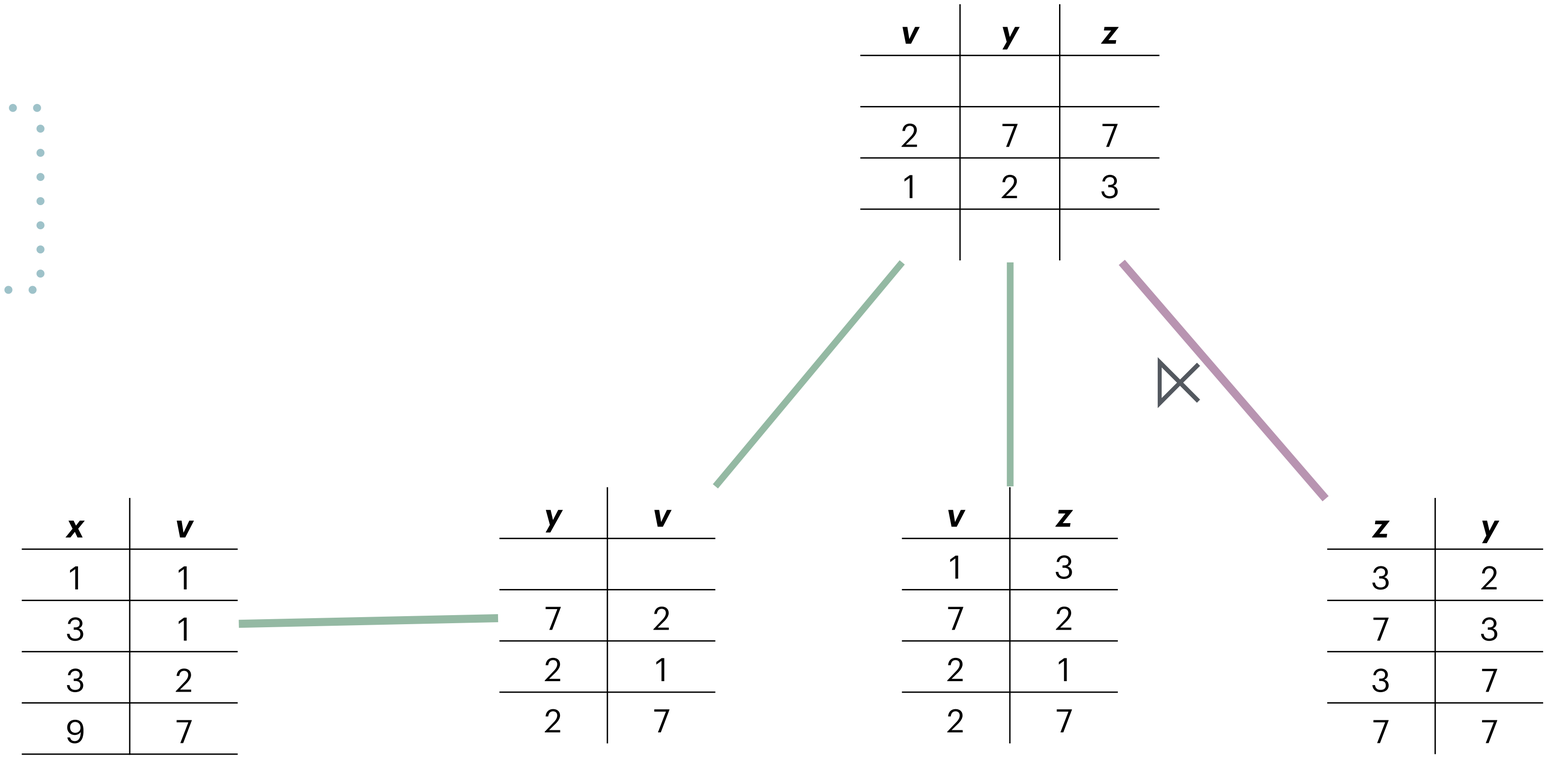


Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

2. The bottom up semi-joins

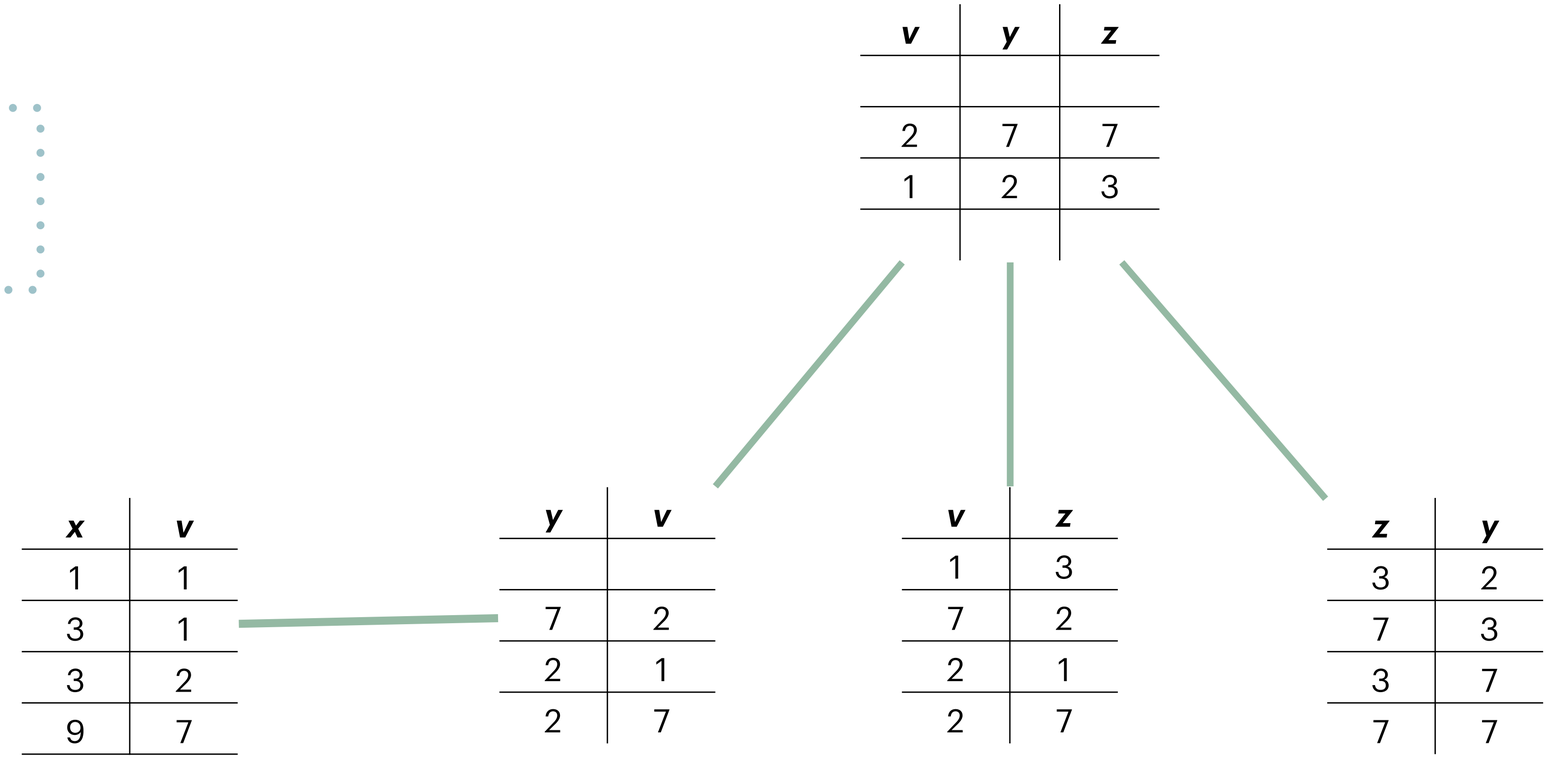
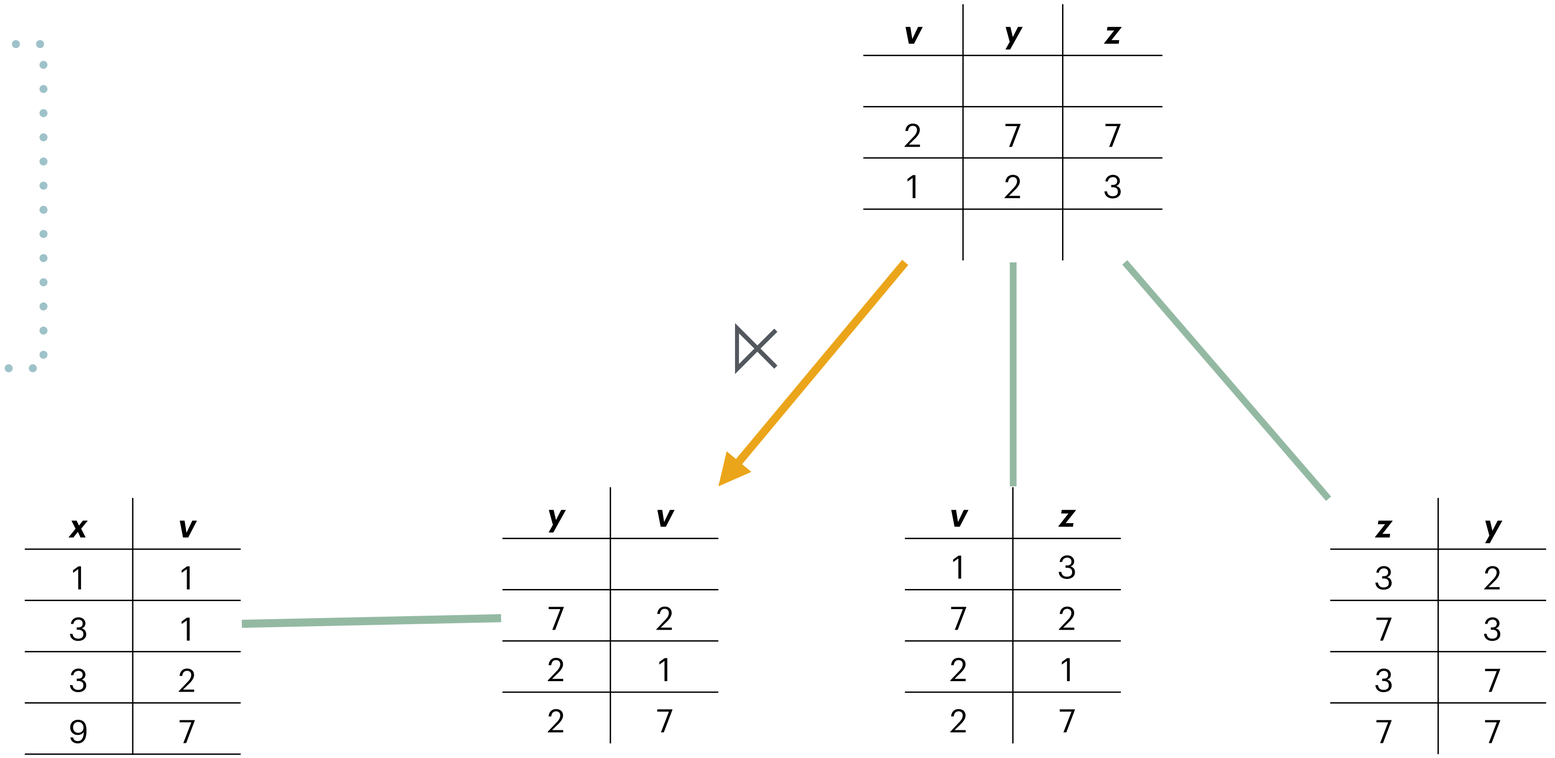


Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

3. The top-down semi-joins. Recall, only from parent to child.



v	y	z
2	7	7
1	2	3

x	v
1	1
3	1
3	2
9	7

y	v
7	2
2	1
2	7

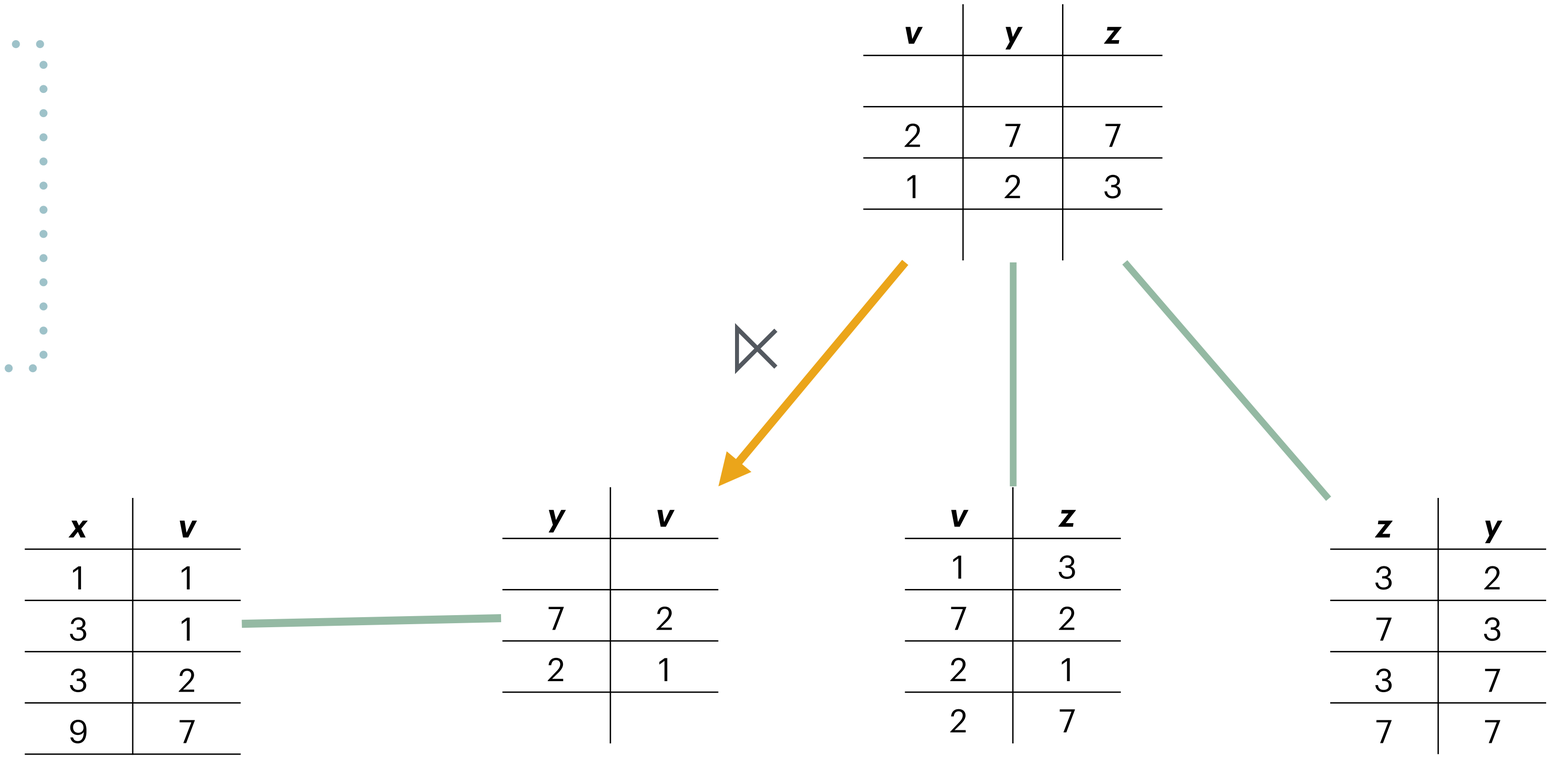
v	z
1	3
7	2
2	1
2	7

z	y
3	2
7	3
3	7
7	7

Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

3. The top-down semi-joins. Recall, only from parent to child.



v	y	z
2	7	7
1	2	3

x	v
1	1
3	1
3	2
9	7

y	v
7	2
2	1

v	z
1	3
7	2
2	1
2	7

z	y
3	2
7	3
3	7
7	7

Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

3. The top-down semi-joins. Recall, only from parent to child.

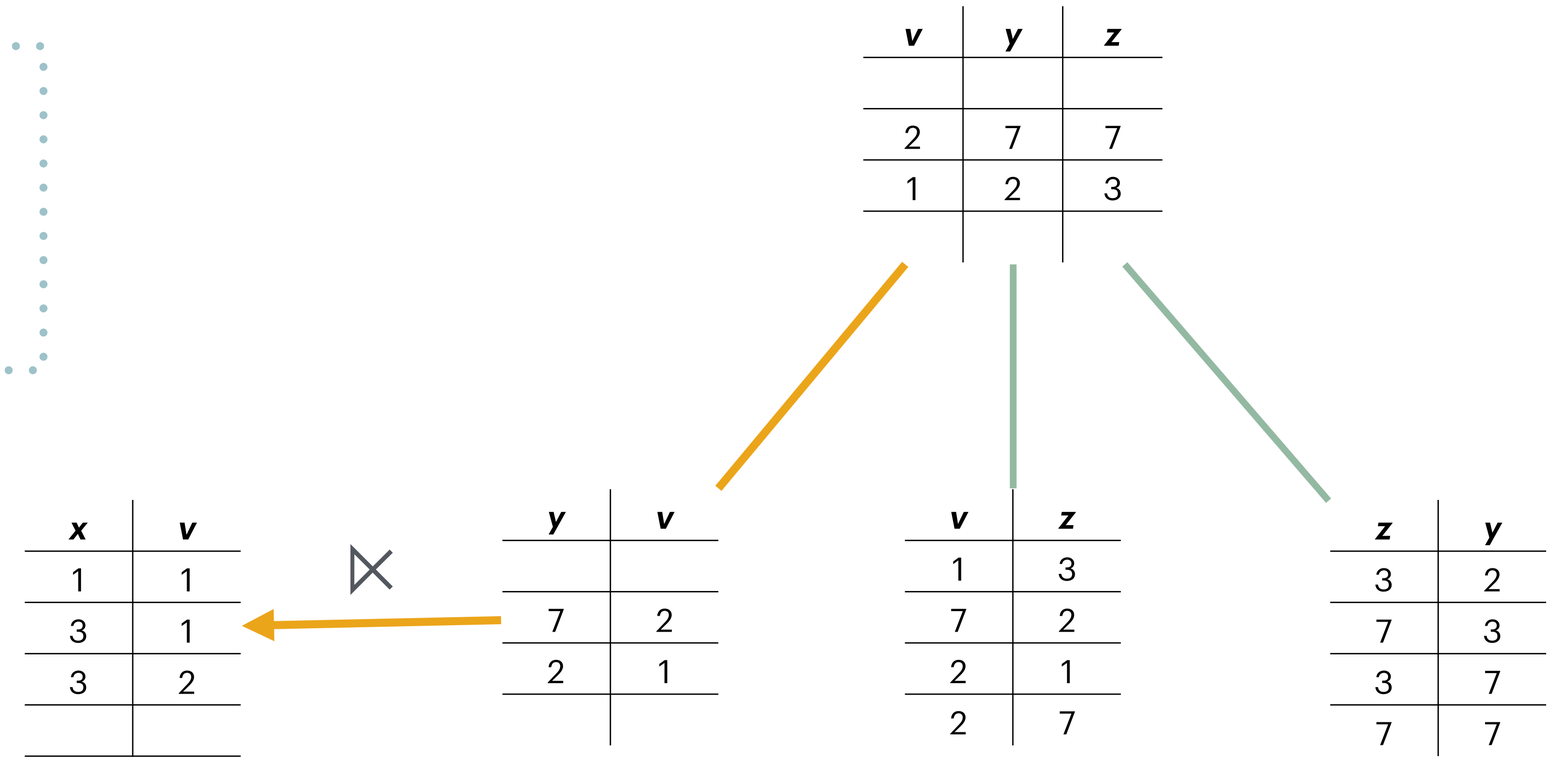


Illustration of Yannakakis' Algorithm

$$R(v,z) \wedge S(z,y) \wedge R(y,v) \wedge T(v,y,z) \wedge P(x,v)$$

3. The top-down semi-joins. Recall, only from parent to child.

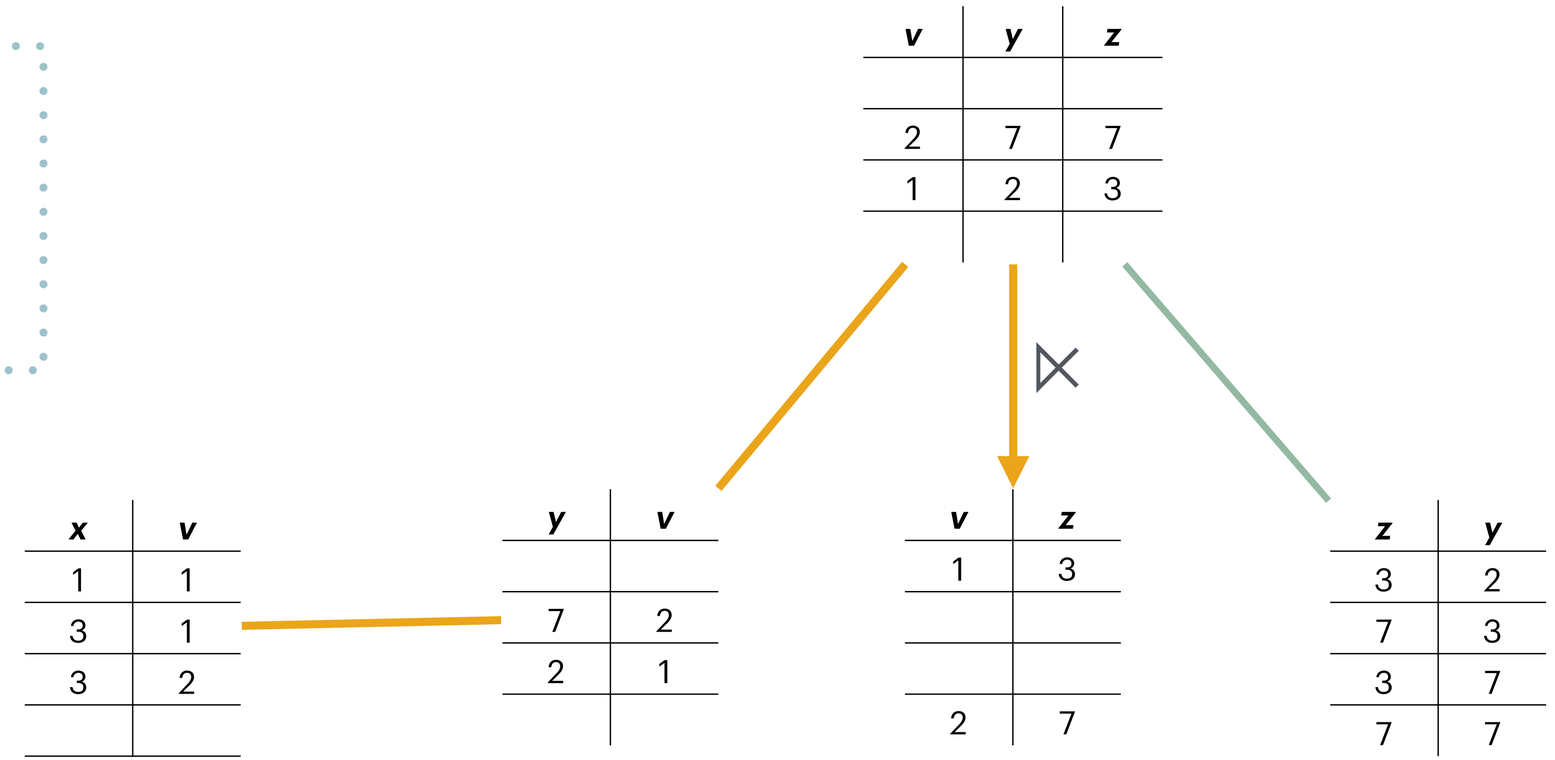


Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

3. The top-down semi-joins. Recall, only from parent to child.

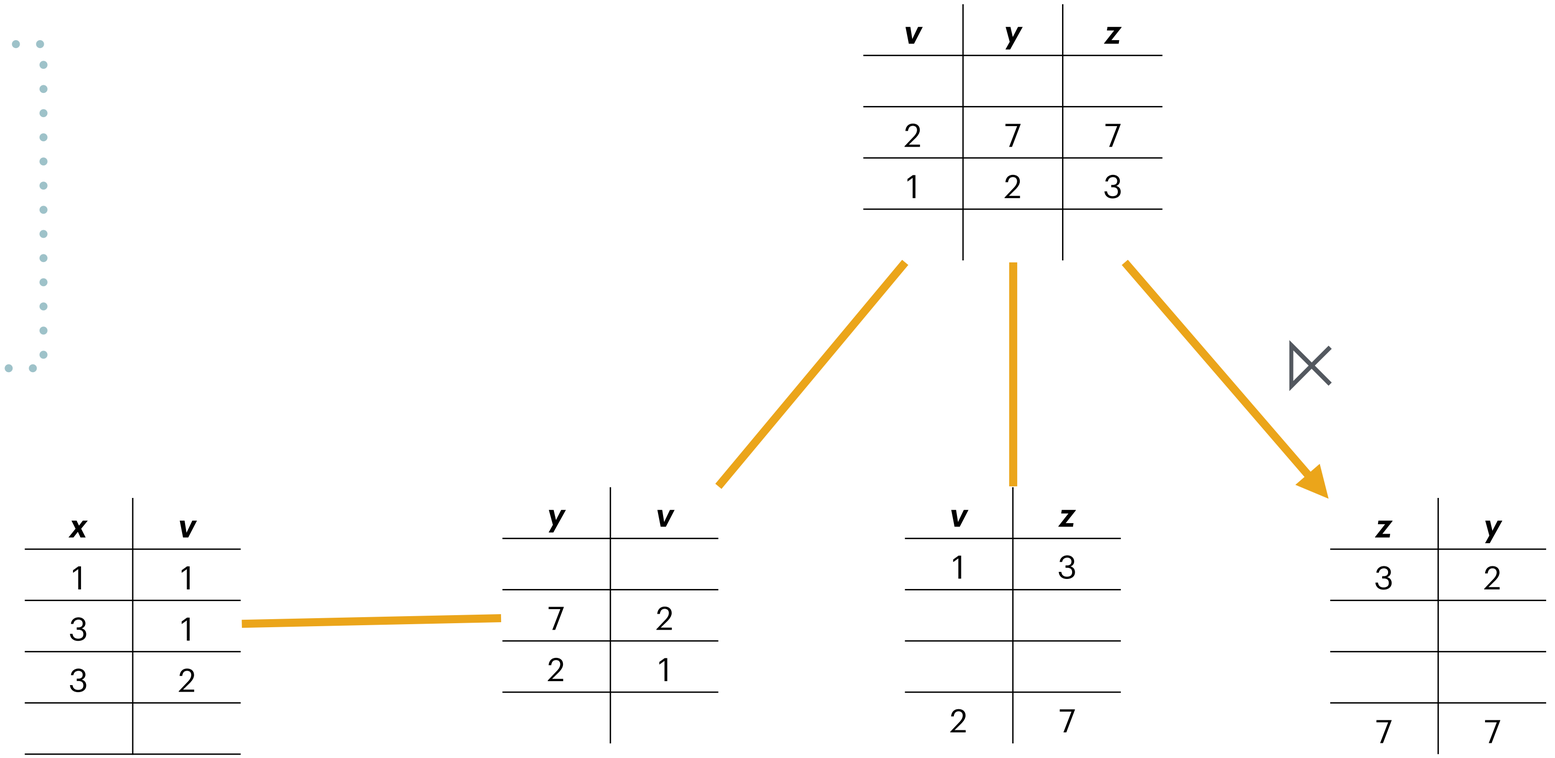


Illustration of Yannakakis' Algorithm

$$R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v)$$

4. All tuples that are left-over are part of a solution. Joining the tables now creates no intermediate results that are not part of the output.

v	*y*	*z*
2	7	7
1	2	3

x	*v*
1	1
3	1
3	2

y	*v*
7	2
2	1

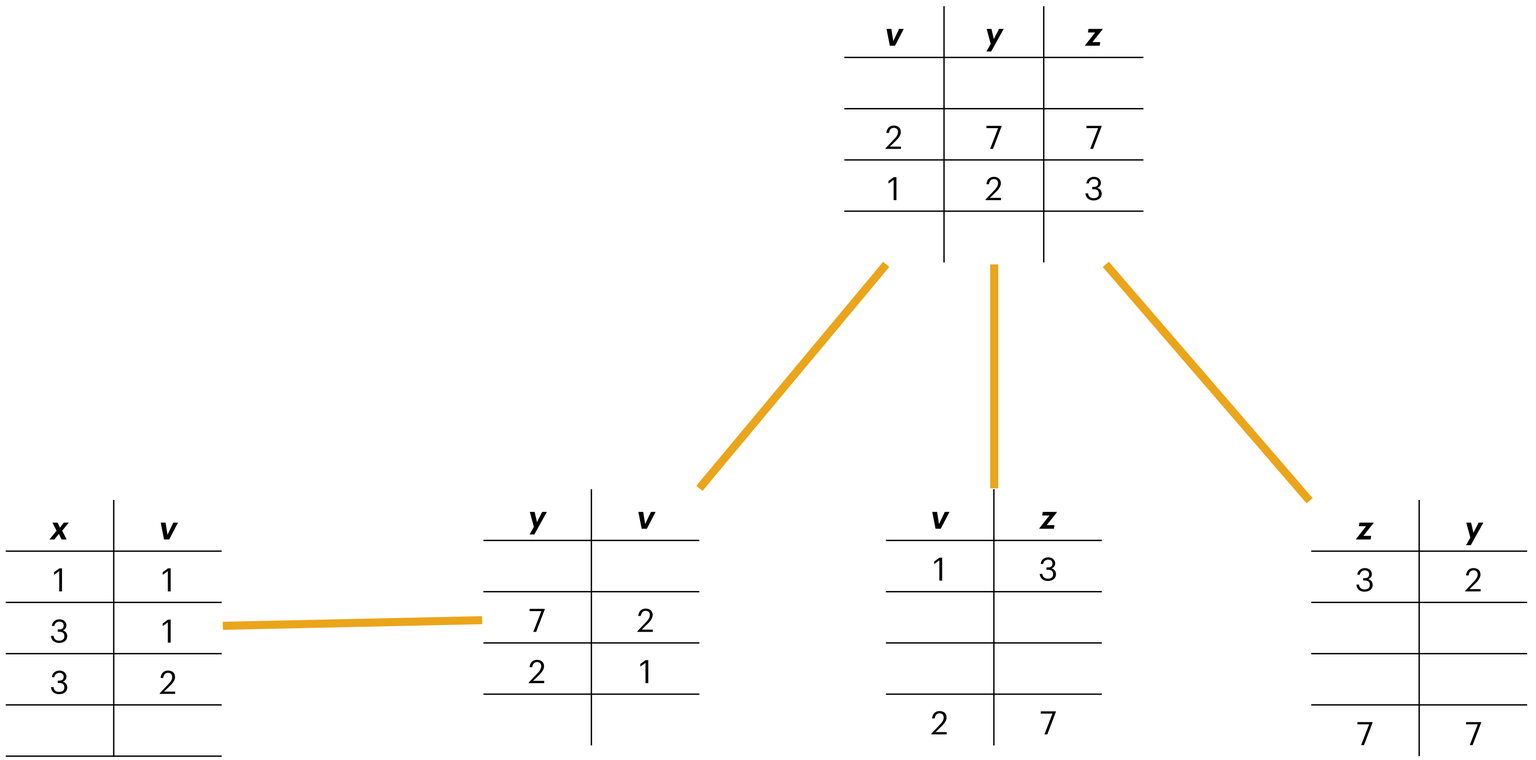
v	*z*
1	3
2	7

z	*y*
3	2
7	7

Illustration of Yannakakis' Algorithm

Answers $q(D)$:

v	*x*	*y*	*z*
2	3	7	7
1	3	2	3
1	1	2	3



How Efficient is This?

Explore more in a literature exercise!

After both semi-join phases of Yannakakis' Algorithm, we are left only with tuples that are part of answers. That means that no unnecessary effort is performed if we now join the left-over tuples to compute the answer in time $\tilde{O}(|D| + |q(D)|)$ for a fixed α-acyclic query.

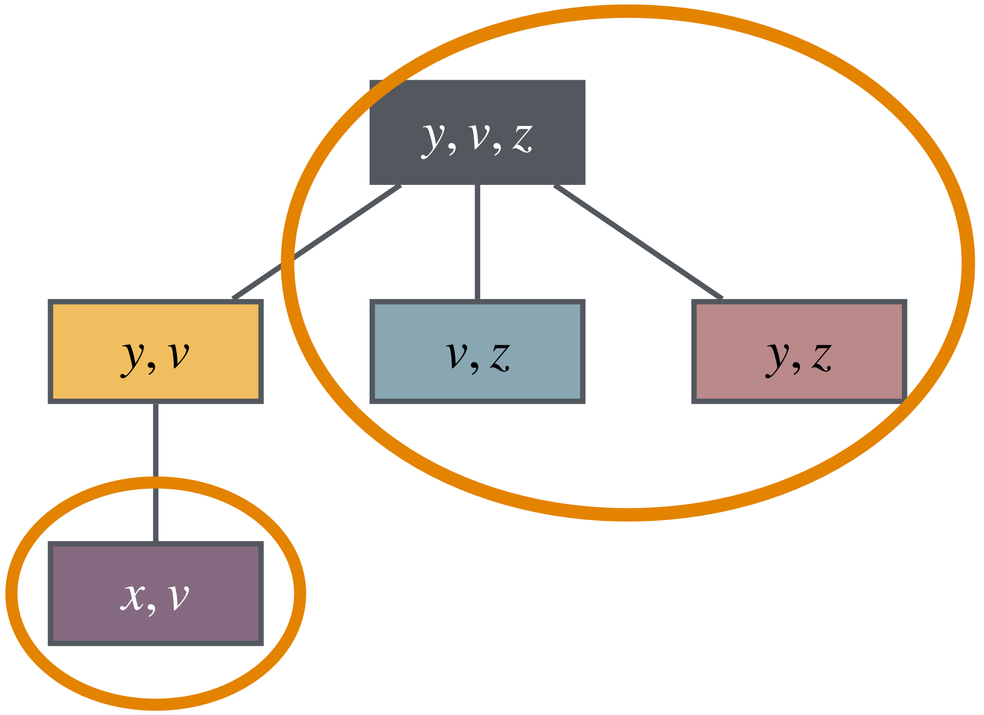
However, with our tree structure we can do something more refined. We can enumerate the answers with **small delay between answers**. That is, the algorithm produces tuples one after another, where the time between outputting each tuple does not depend on the database.

Quantification Matters

Consider a version of our example query with existential quantification:

$$q = \{ (x, z) \mid \exists vy \; R(v, z) \wedge S(z, y) \wedge R(y, v) \wedge T(v, y, z) \wedge P(x, v) \}$$

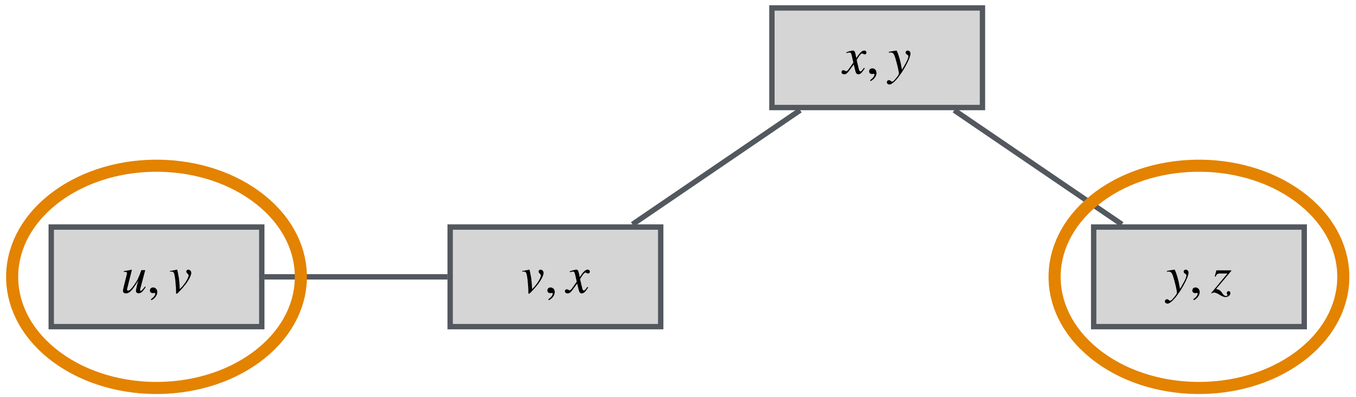
When collecting the final answers,
we need to create intermediate results
that extend to y, v to connect the two parts
of the join tree that contain the output variables.



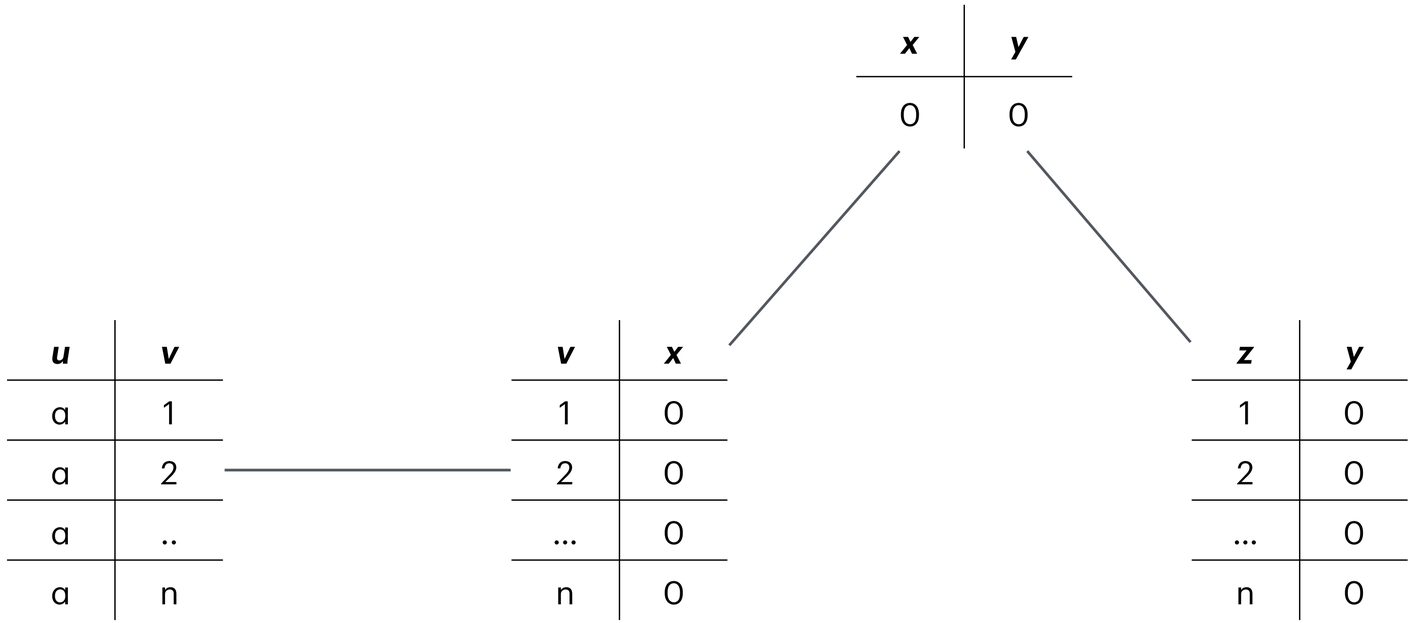
Quantification Matters

Let's simplify our example to see how this can break the linear behaviour of Yannakakis.

$$q = \{ (u, z) \mid \exists vxy \; T(u, v) \wedge R(v, x) \wedge S(x, y) \wedge R(z, y) \}$$



Assume these relations after the semi-join phase



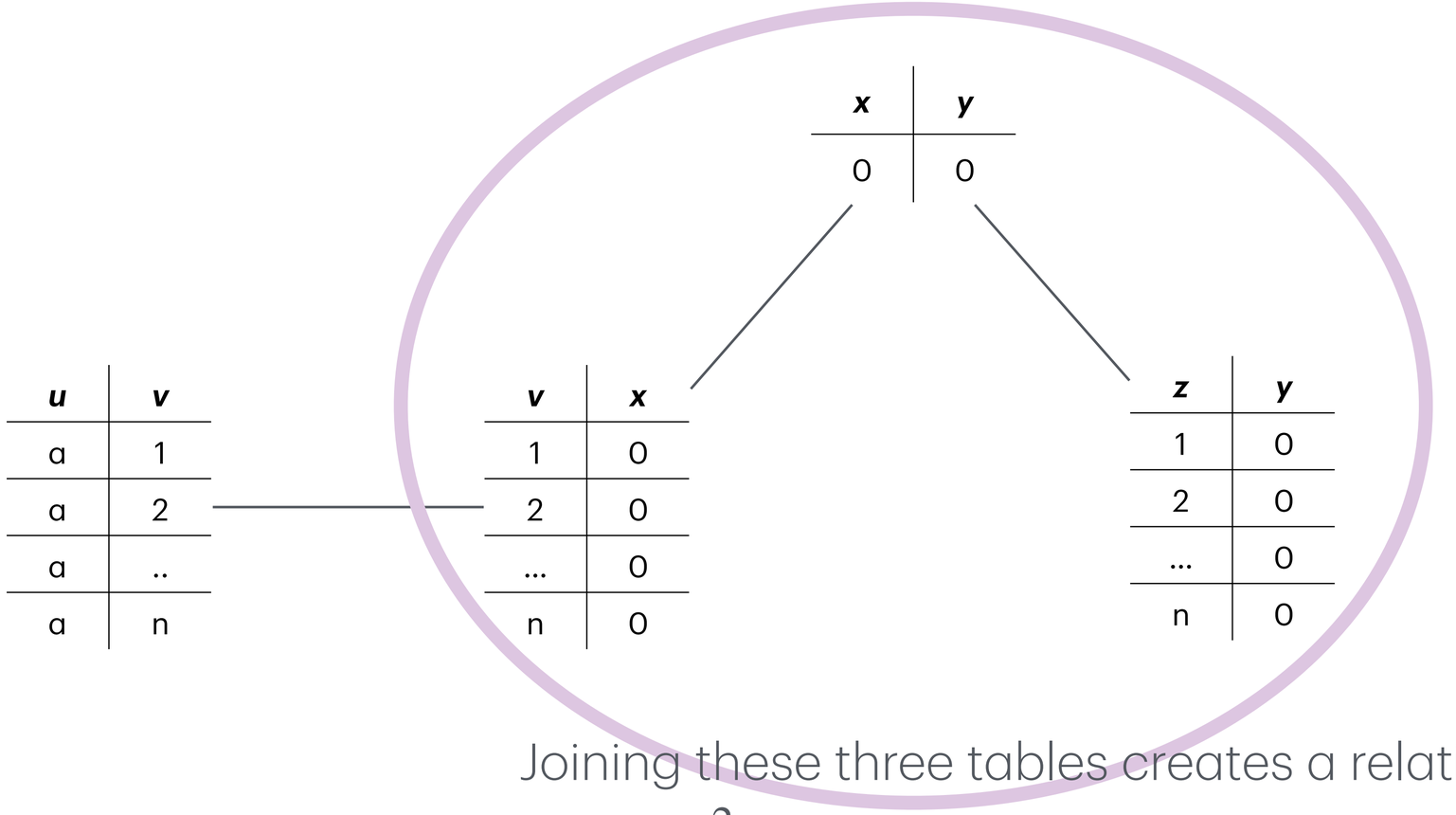
x	*y*
0	0

u	*v*
a	1
a	2
a	..
a	n

v	*x*
1	0
2	0
...	0
n	0

z	*y*
1	0
2	0
...	0
n	0

To see which assignments to u and z work together,
we need to compute joins over variables that are unrelated
to the output.



x	*y*
0	0

u	*v*
a	1
a	2
a	..
a	n

v	*x*
1	0
2	0
...	0
n	0

z	*y*
1	0
2	0
...	0
n	0

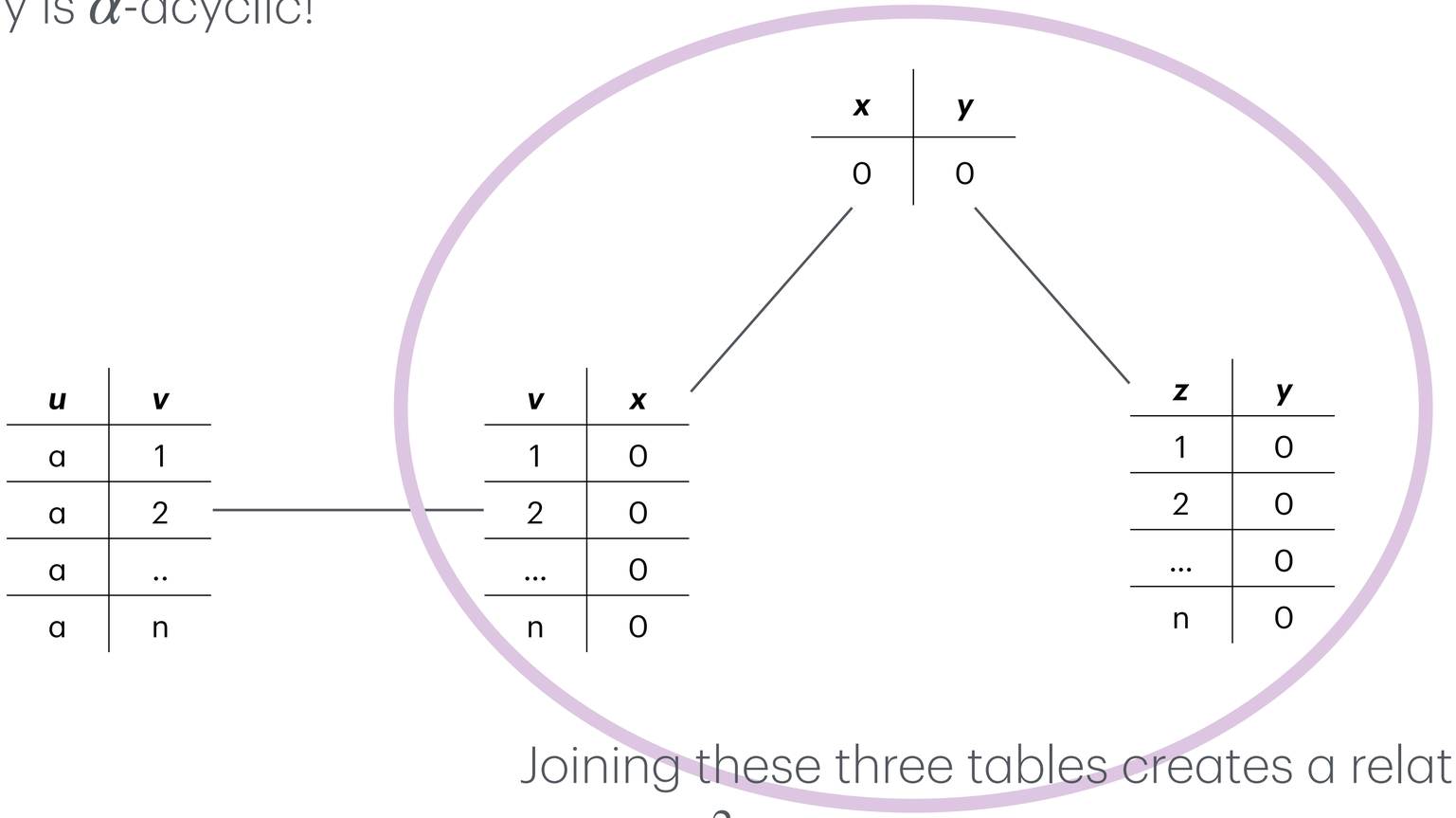
Joining these three tables creates a relation
of size n^2!

The actual result tuples are: $(a,1), (a,2), \ldots, (a,n)$.

Meaning $|q(D)| = n$, but creating the joins requires $\Theta(n^2)$ time!

The time bound $\tilde{O}(|D| + |q(D)|)$ doesn't hold

even though the query is α-acyclic!



u	v
a	1
a	2
a	..
a	n

v	x
1	0
2	0
...	0
n	0

x	y
0	0

z	y
1	0
2	0
...	0
n	0

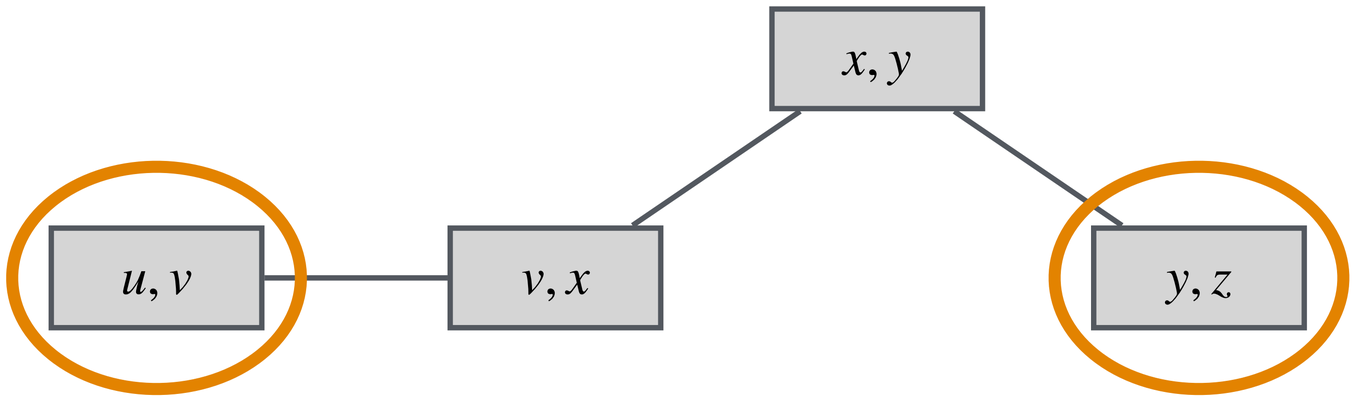
Joining these three tables creates a relation of size n^2!

Quantification Details

- We can obtain the same guarantees (and in particular constant-delay enumeration) only on a limited number of α-acyclic CQs when there is existential quantification!
- Intuitively, this requires the part of the join tree that contains the output variables to be connected:

With u, z as output: No

$$T(u, v) \wedge R(v, x) \wedge S(x, y) \wedge R(z, y)$$

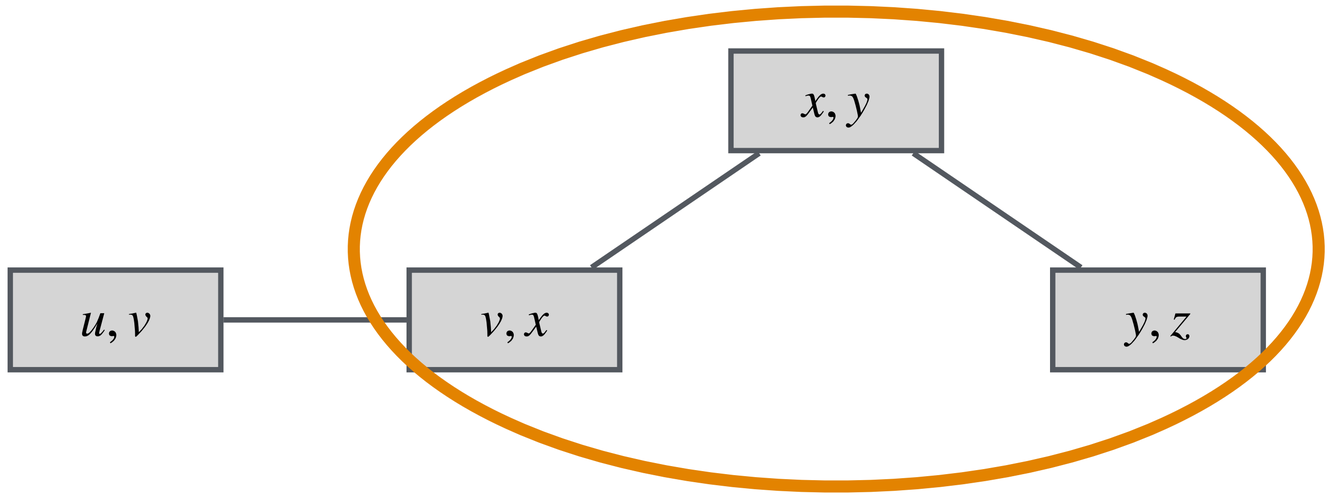


Quantification Details

- We can obtain the same guarantees (and in particular constant-delay enumeration) only on a limited number of α-acyclic CQs when there is existential quantification!
- Intuitively, this requires the part of the join tree that contains the output variables to be connected:

With x, y as output: Yes!

$$T(u, v) \wedge R(v, x) \wedge S(x, y) \wedge R(z, y)$$

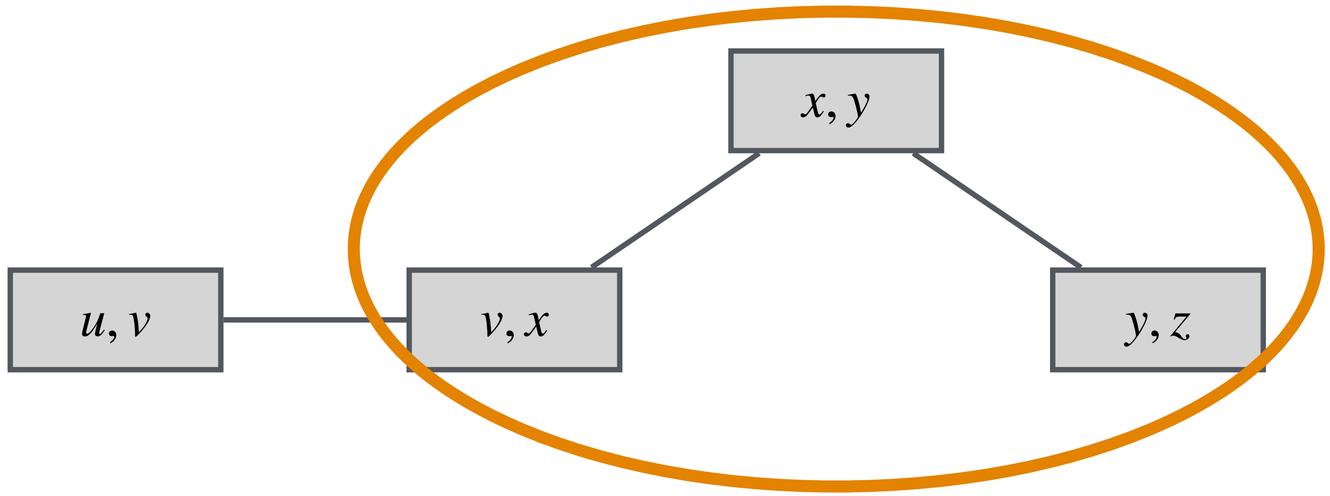


Quantification Details

- We can obtain the same guarantees (and in particular constant-delay enumeration) only on a limited number of α-acyclic CQs when there is existential quantification!
- Intuitively, this requires the part of the join tree that contains the output variables to be connected:

With x, y as output: Yes!

$$T(u, v) \wedge R(v, x) \wedge S(x, y) \wedge R(z, y)$$



Summary

- CQs are the core element of many data retrieval tasks.
- Unlike FO/RA queries, we can decide containment, equivalence and even minimality of CQs.
- Answering CQs is NP-complete, but acyclic queries can always be answered efficiently —> Structure of CQs is a key factor in their complexity.
- Yannakakis' Algorithm!
- Enumeration or counting versions bring their own challenges with them like quantification and more fine-grained notions of complexity.