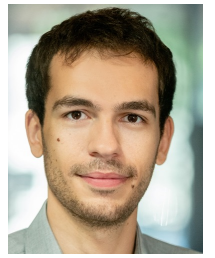
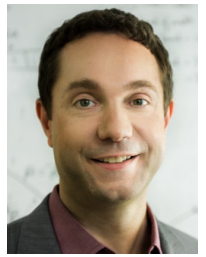




# Toward Responsive DBMS: Optimal Join Algorithms, Enumeration, Factorization, Ranking, and Dynamic Programming

Nikolaos Tziavelis, Wolfgang Gatterbauer, Mirek Riedewald

Northeastern University, Boston

## Part 3: Acyclic queries & Enumeration

Northeastern University
Khoury College of Computer Sciences

Slides: https://northeastern-datalab.github.io/responsive-dbms-tutorial
DOI: https://doi.org/10.1109/ICDE53745.2022.00299
Data Lab: https://db.khoury.northeastern.edu

DATALAB @Northeastern

# Outline tutorial

1: Introduction (Nikos) ~40min
2: Tree Decompositions (Mirek) ~20min
3: Acyclic Queries & Enumeration (Wolfgang) ~25min

BREAK

4: Factorization (Nikos) ~10min
5: Dynamic Programming & Semirings (Wolfgang) ~20min
6: Any-$k$ or Ranked Enumeration (Nikos) ~35min
7. Decomposition of Comparison Predicates (Mirek) ~10min
8. Conclusion (Mirek) ~10min

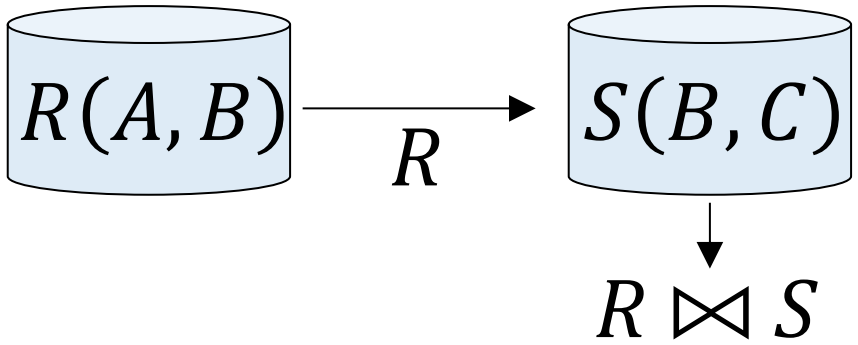
# Outline Part 3

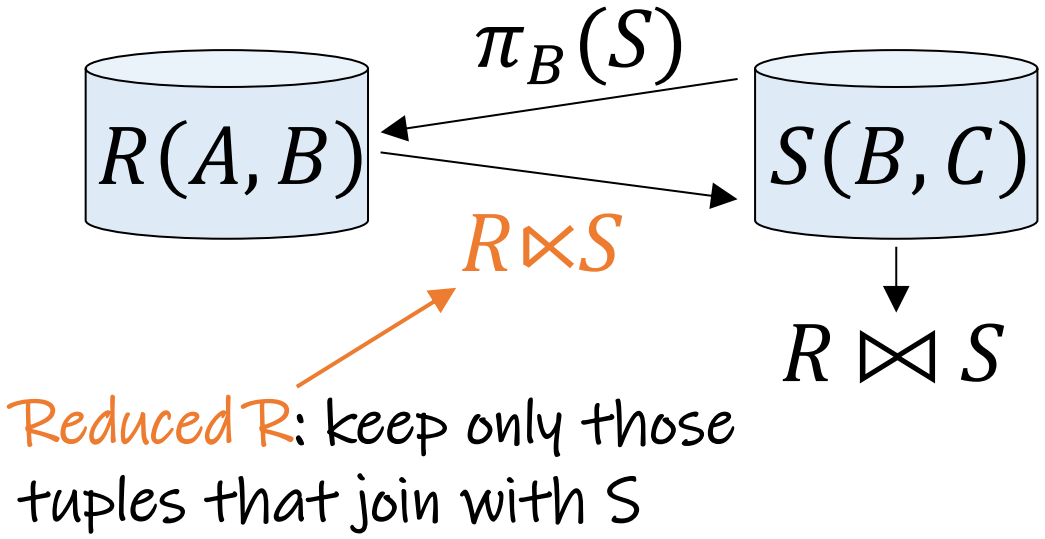
Part 3: Acyclic Queries & Enumeration (Wolfgang) ~25min

- The power of semi-join reductions
- Hypergraphs, GYO, join trees
- Yannakakis = acyclic query evaluation
- Enumerating answers
- Projections

# Semijoins as Message Passing

- Semijoins can reduce network use for equijoins in distributed databases



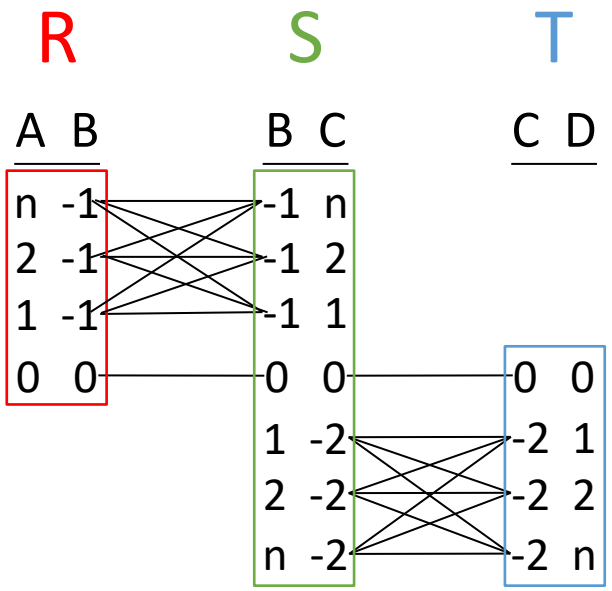




Effective if 1) the size of join attribute B (or number of distinct values) is smaller than A and C, and 2) few tuples from R participate in the join

# Semi-joins can also help if data are local



$$P_3(x,y,z,x) :- R(x,y), S(y,z), T(z,w)$$




Only 1 single output tuple!

```
-------------------------
-- Query 1
-------------------------
select *
into record1
from R natural join S natural join T;
```

```
-------------------------
-- Query 2
-------------------------
With S2 as
     (SELECT *
     FROM S
     WHERE S.B in
          (SELECT R.B
          FROM R)),
S3 as
     (SELECT *
     FROM S2
     WHERE S2.C in
          (SELECT T.C
          FROM T))
select a, b, c, d
into record2
from R natural join S3 natural join T;
```

n=1,000: $t_{Q1}$=1.4 sec

n=2,000: $t_{Q1}$=6.1 sec $O(n^2)$ ☹

$t_{Q2}$=5 msec

$t_{Q2}$=8 msec $O(n)$ ☺

SQL example 601 available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

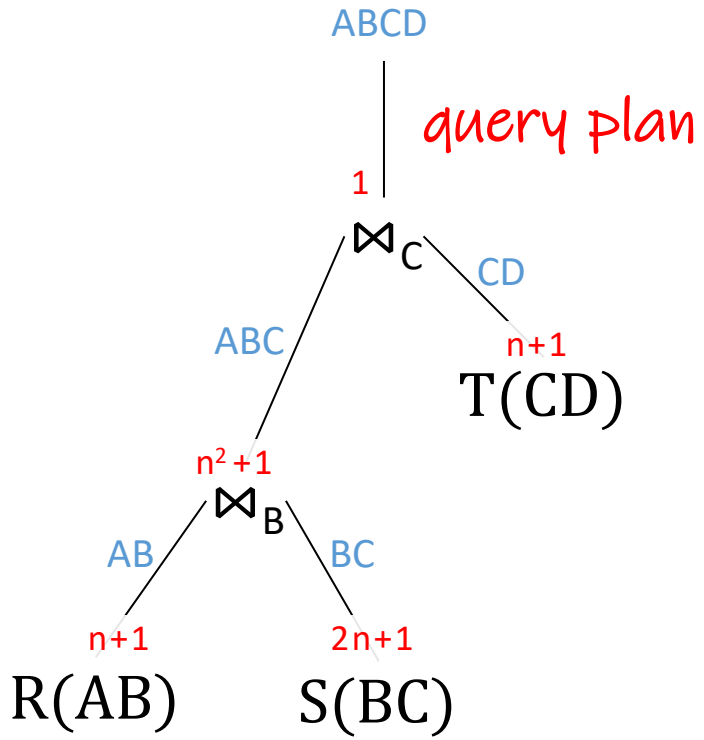
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# The more general idea: "Sideways information passing"

**Sideways information passing**:

- "sending information from one subexpression not simply to its parent expression, but also to some other correlated portion of the query computation, in order to prune irrelevant results" [Ives, Taylor 08]
- includes techniques like two-way semijoins [Bernstein, Goodman 81] and magic sets [Beeri, Ramakrishnan 91]

$$Q = (R \bowtie_B S) \bowtie_C T$$





[Bernstein, Goodman 81]. "Using Semi-Joins to Solve Relational Queries", JACM 1981. https://doi.org/10.1145/322234.322238
[Beeri, Ramakrishnan 91]: "On the power of magic", Journal of Logic Programming, 1991. https://doi.org/10.1016/0743-1066(91)90038-Q
Definition from: [Ives, Taylor 08]. "Sideways Information Passing for Push-Style Query Processing", ICDE 2008. https://doi.org/10.1109/ICDE.2008.4497486
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Outline Part 3

Part 3: Acyclic Queries & Enumeration (Wolfgang) ~25min

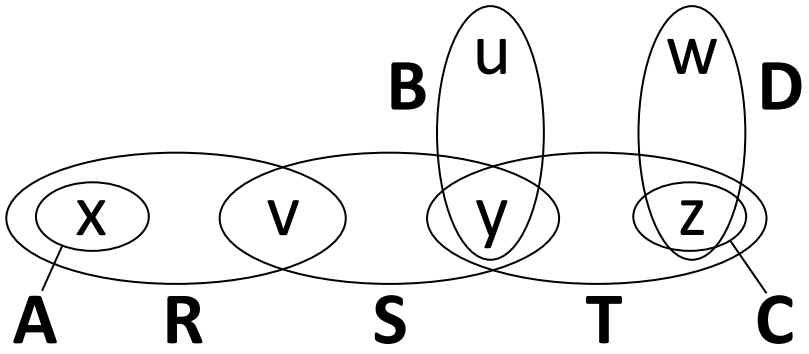
- The power of semi-join reductions
- Hypergraphs, GYO, join trees
- Yannakakis = acyclic query evaluation
- Enumerating answers
- Projections

# Query Hypergraph of a Conjunctive Query

$$Q :- A(x), R(x,v), S(v,y), B(y,u), T(y,z), C(z), D(z,w)$$

## Query hypergraph (nodes = variables)

determines $\alpha$-acyclicity of CQs (that's what we need)



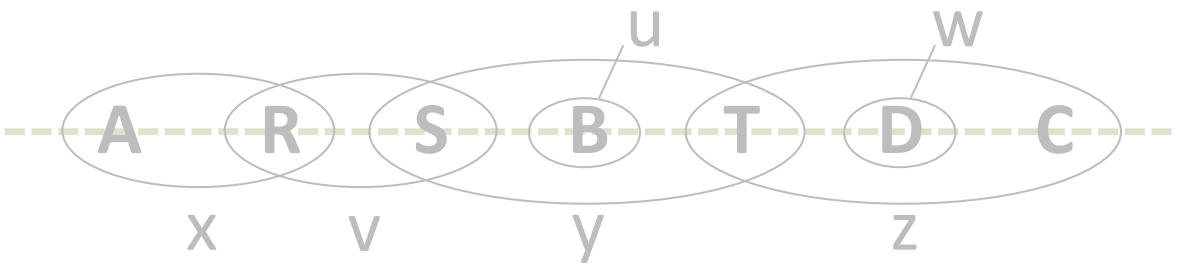

## Incidence matrix

determines minimal query plans [G+'17]
(intuitively, plans with early projections)

| | x | v | y | u | z | w |
|---|---|---|---|---|---|---|
| A | 1 | | | | | |
| R | 1 | 1 | | | | |
| S | | 1 | 1 | | | |
| B | | | 1 | 1 | | |
| T | | | 1 | | 1 | |
| C | | | | | 1 | |
| D | | | | | 1 | 1 |

## Query dual hypergraph (nodes = atoms)

determines complexity of query resilience [Meliou+'10]

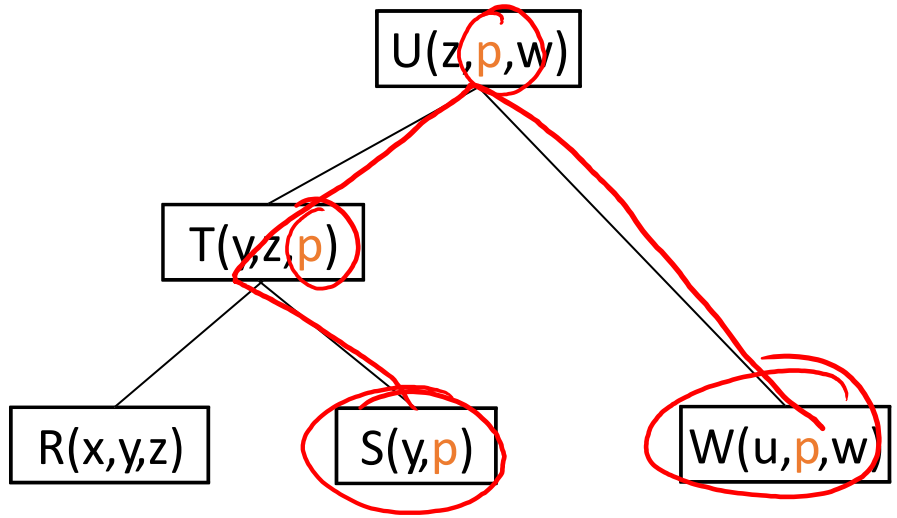




Meliou, Gatterbauer, Moore, Suciu. "The complexity of causality and responsibility for query answers and non-answers", PVLDB 2010. https://doi.org/10.14778/1880172.1880176 /
Gatterbauer, Suciu. "Dissociation and propagation for approximate lifted inference with standard relational database management systems", VLDBJ 2017. https://doi.org/10.1007/s00778-016-0434-5
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# $\alpha$-Acyclic Conjunctive Queries have Join Trees

- Definition: A conjunctive query Q is $\alpha$-acyclic if it has a join tree.
- A join tree for $Q(\mathbf{x}) :- R(\mathbf{z}_1), S(\mathbf{z}_2), T(\mathbf{z}_3), W(\mathbf{z}_4), U(\mathbf{z}_5)$ is a tree T =(V,E) such that:
  - V: The nodes of T are the atoms $R_i(\mathbf{z}_i)$ of Q
  - E: For every variable w occurring in Q, the set of the nodes of T that contain w forms a (connected) subgraph of T (also called running intersection property)
    - in other words, if a variable w occurs in two different atoms $R_j(\mathbf{z}_j)$ and $R_k(\mathbf{z}_k)$ of Q, then it occurs in each atom on the unique path of T joining $R_j(\mathbf{z}_j)$ and $R_k(\mathbf{z}_k)$
- The GYO reduction* helps us find a join tree from the hypergraph iff it is $\alpha$-acyclic




* GYO: This algorithm is named in honor of Marc H. Graham and the team Clement Yu and Meral Ozsoyoglu, who independently came to essentially this algorithm in [YO79] and [Gra79]:
[Gra79] Graham. "On the universal relation." Technical Report, University of Toronto, 1979 / [YO79] Yu, Ozsoyoglu. "An algorithm for tree-query membership of a distributed query." COMPSAC, 1979. https://doi.org/10.1109/CMPSAC.1979.762509

# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

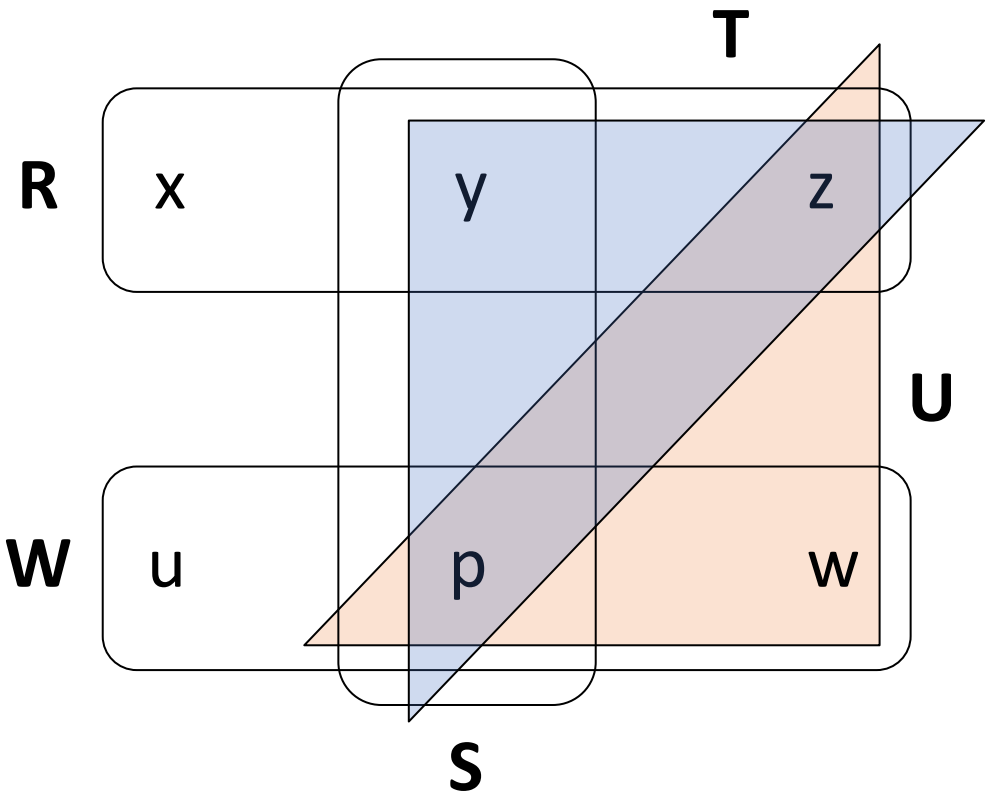
U(z,p,w)

T(y,z,p)

R(x,y,z)

S(y,p)

W(u,p,w)



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

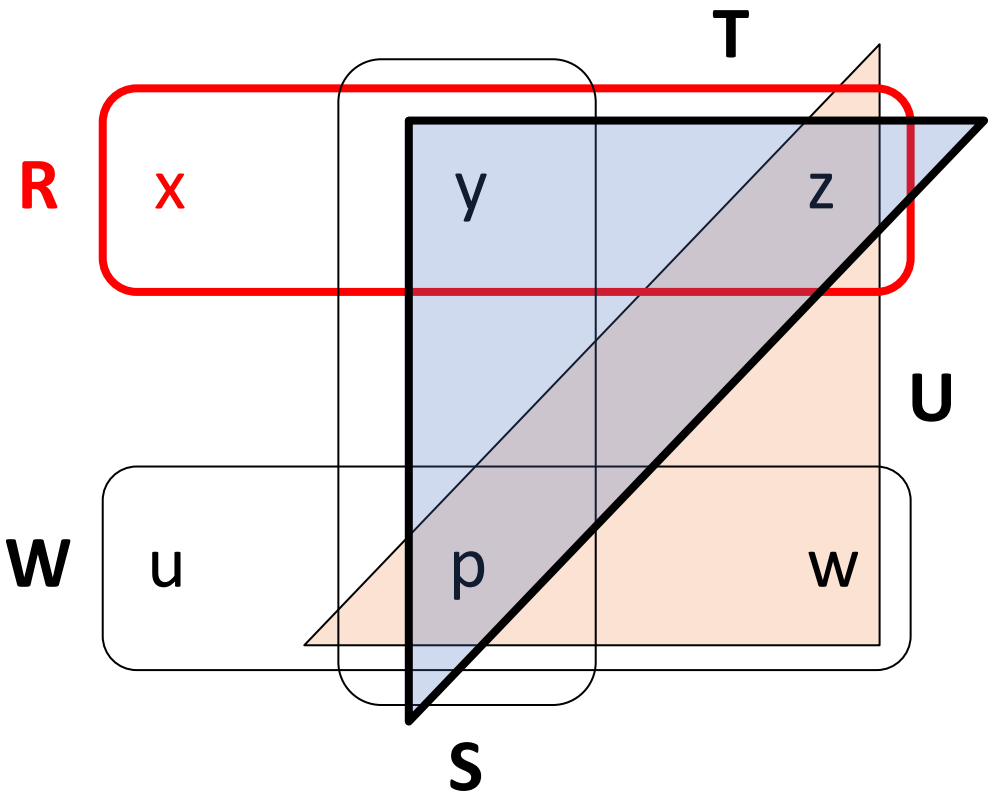
U(z,p,w)

T(y,z,p)

R(x,y,z)

S(y,p)

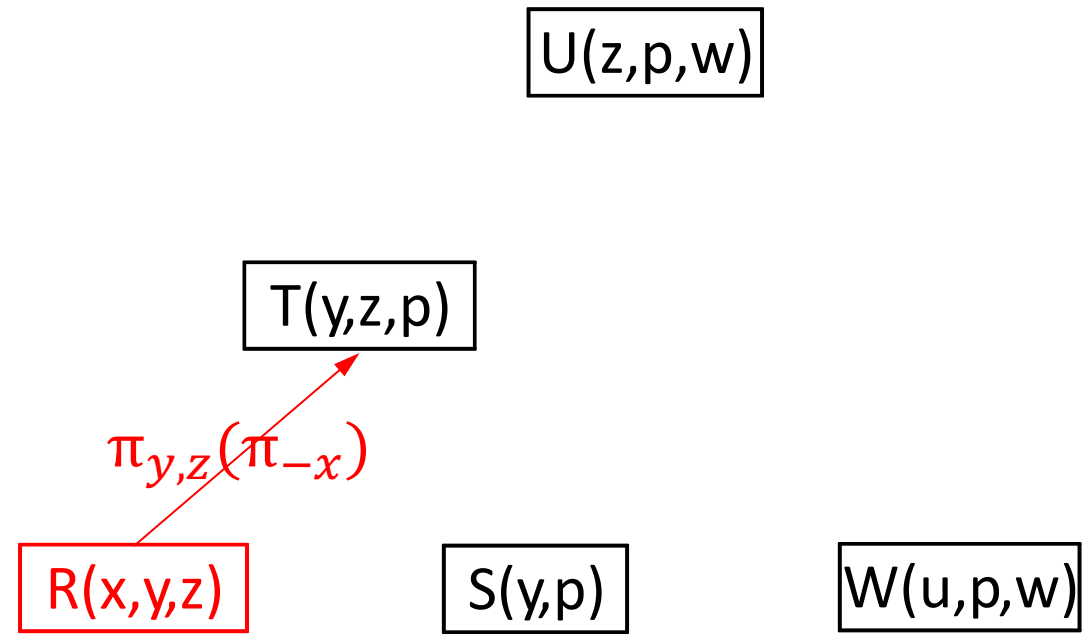
W(u,p,w)

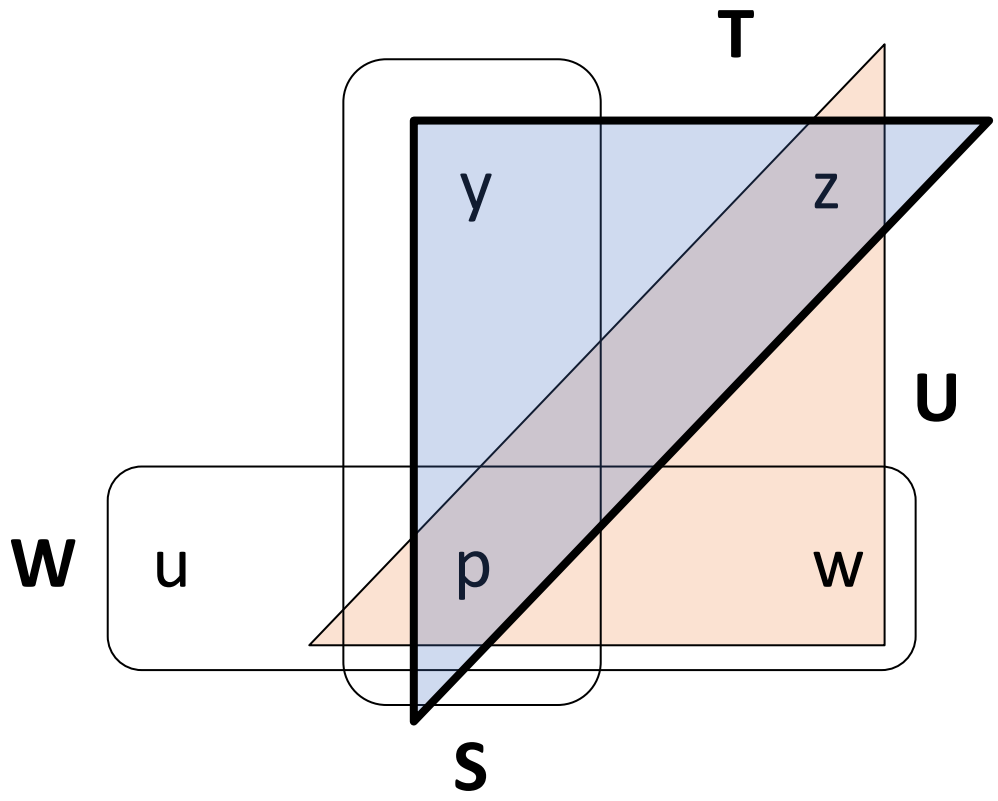


# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

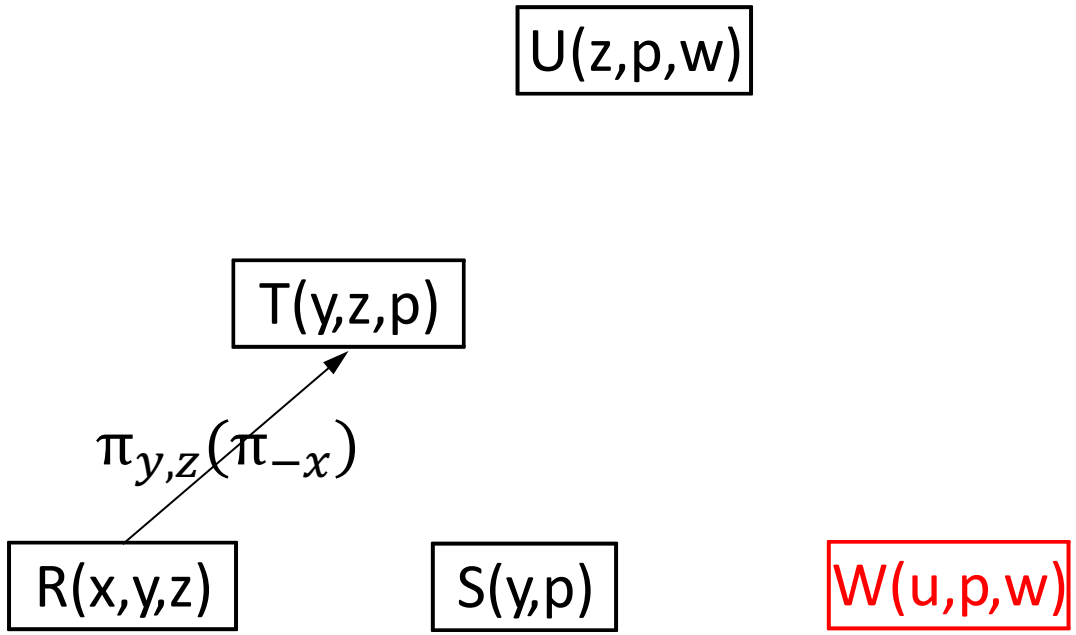


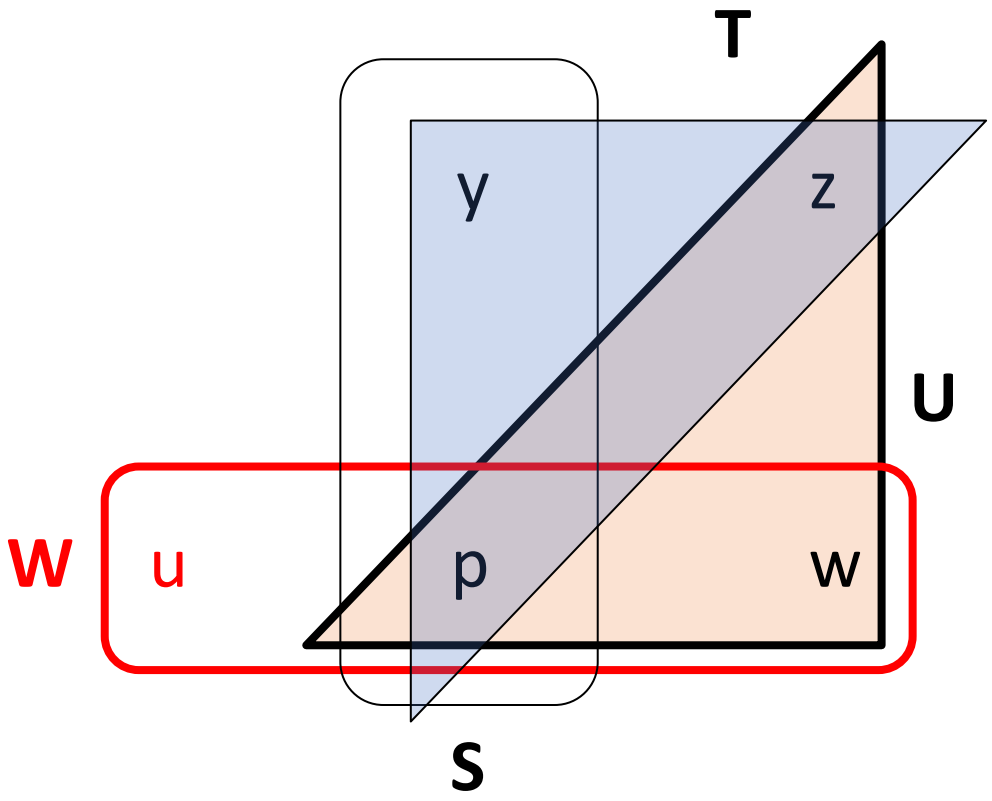



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

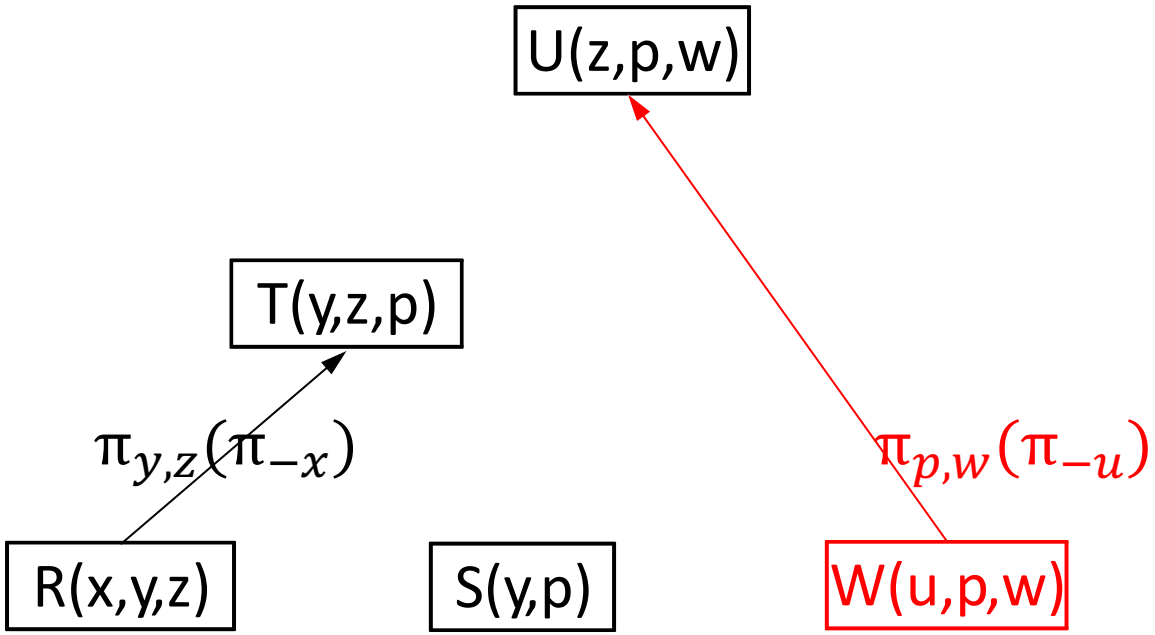


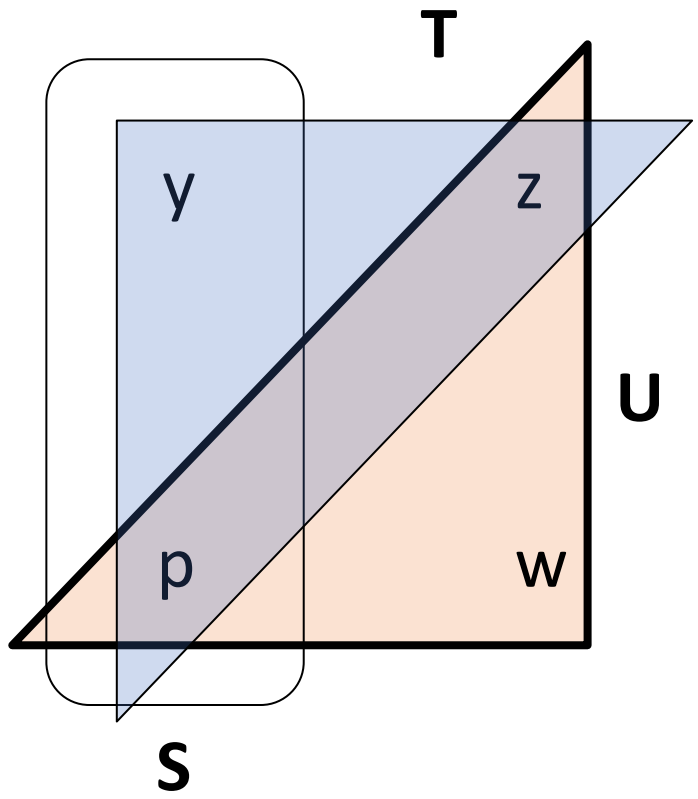



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

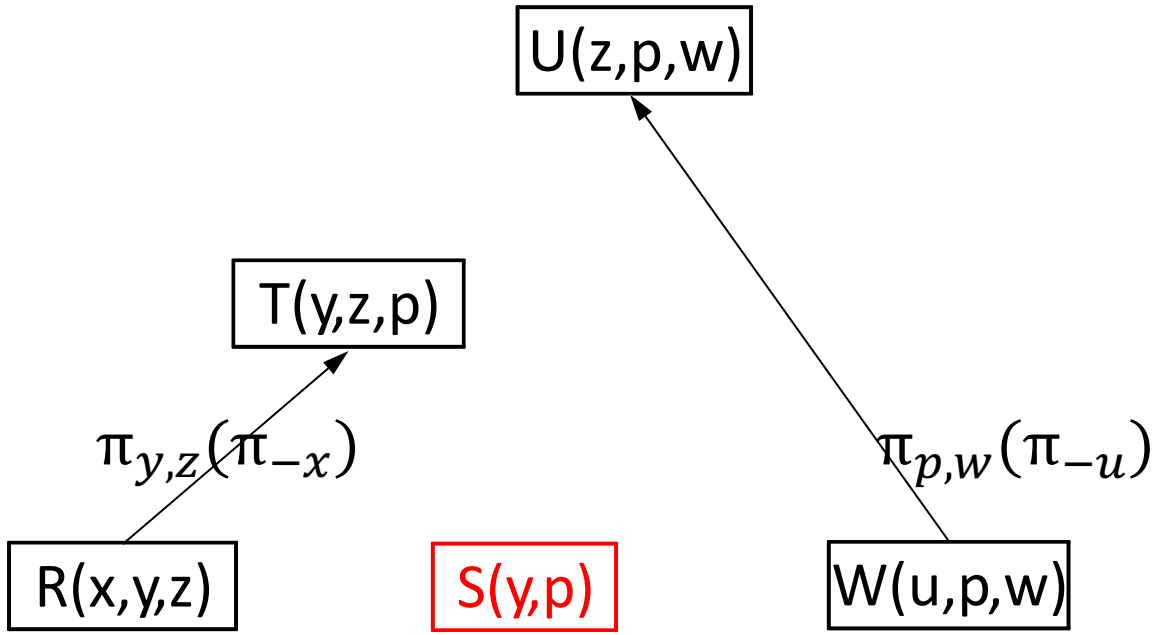


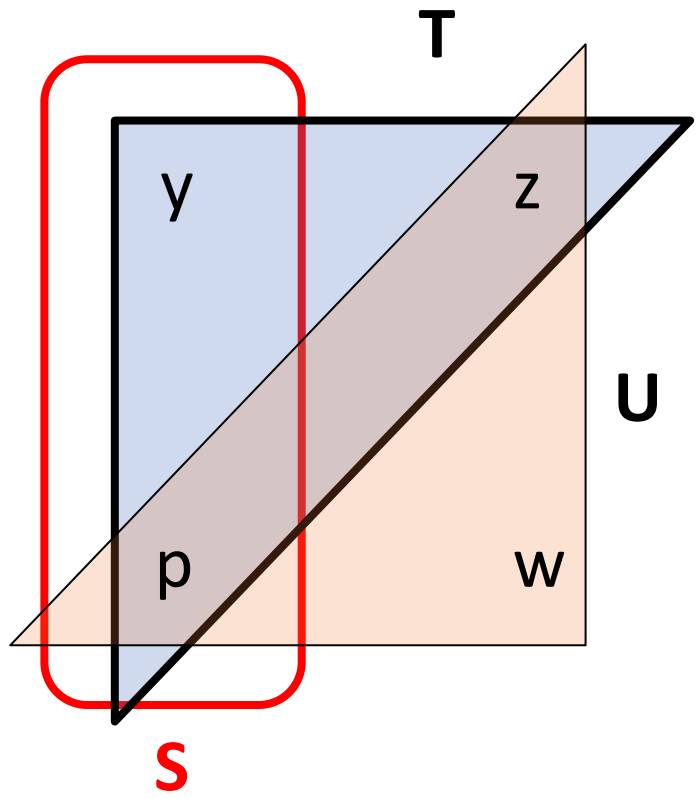



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

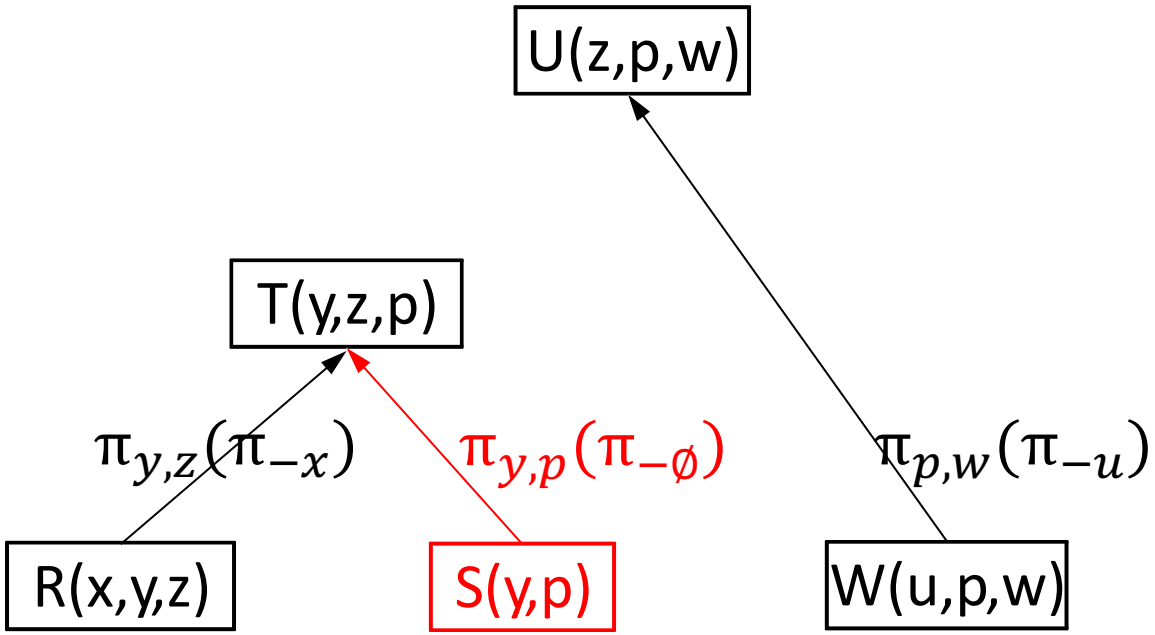


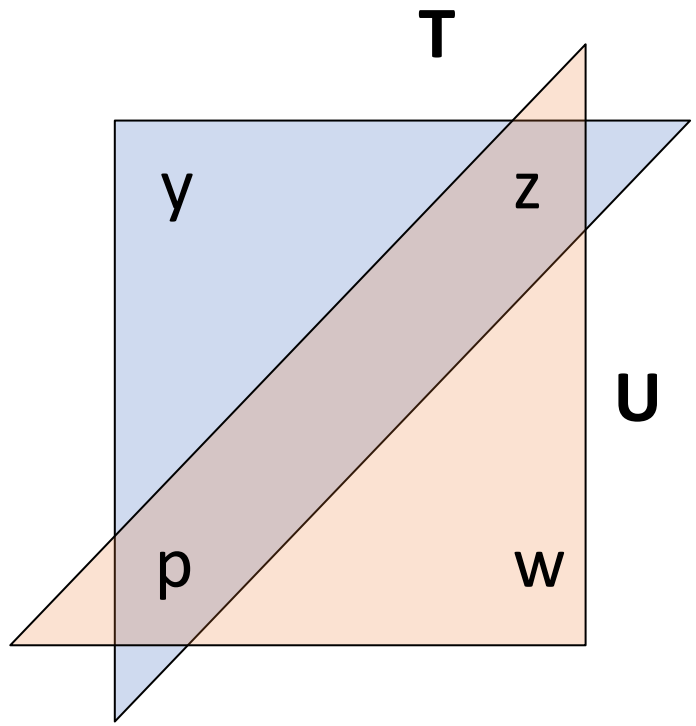



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

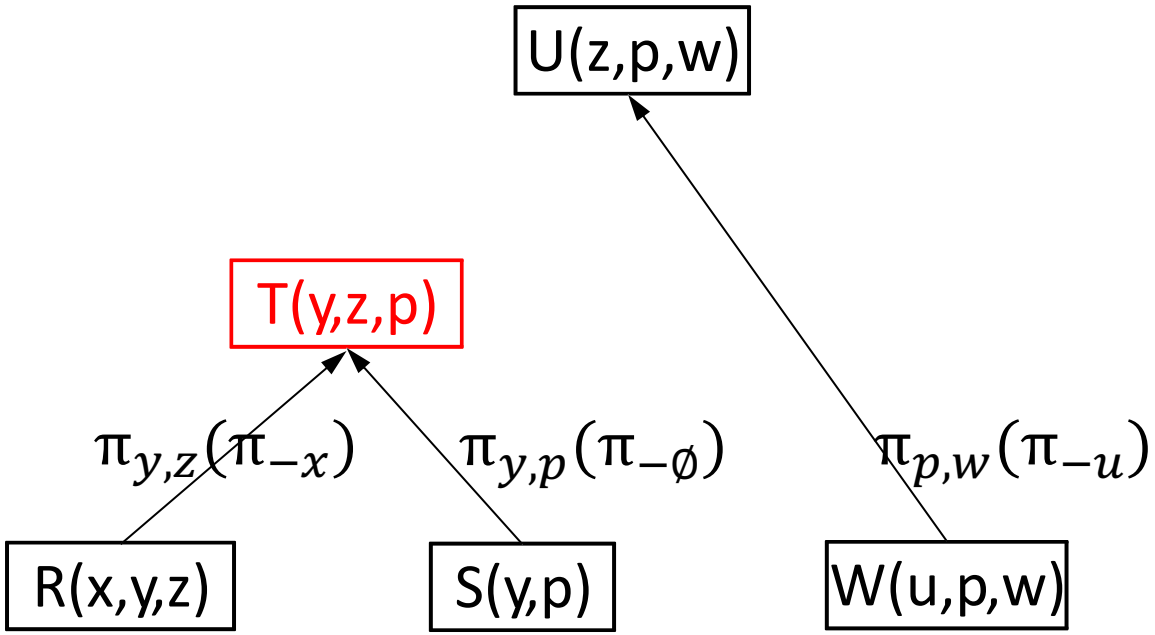


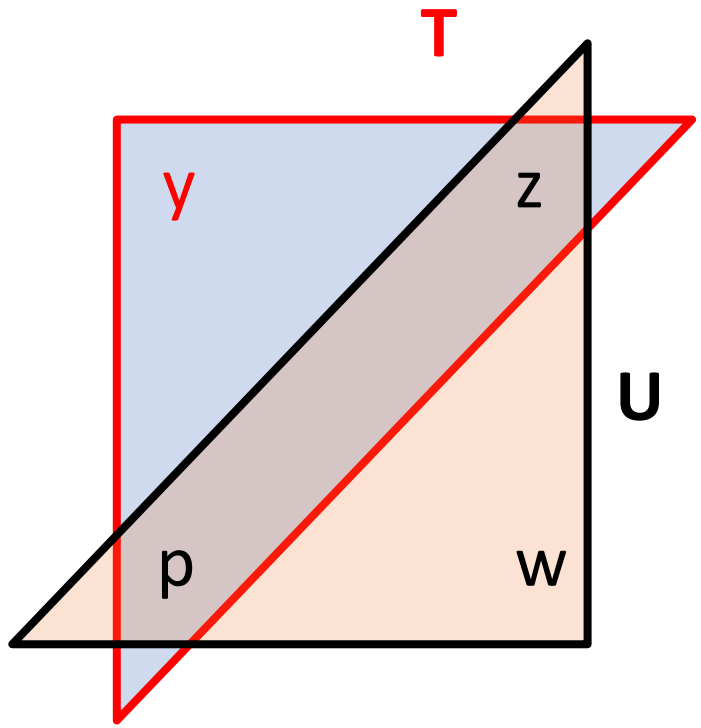



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

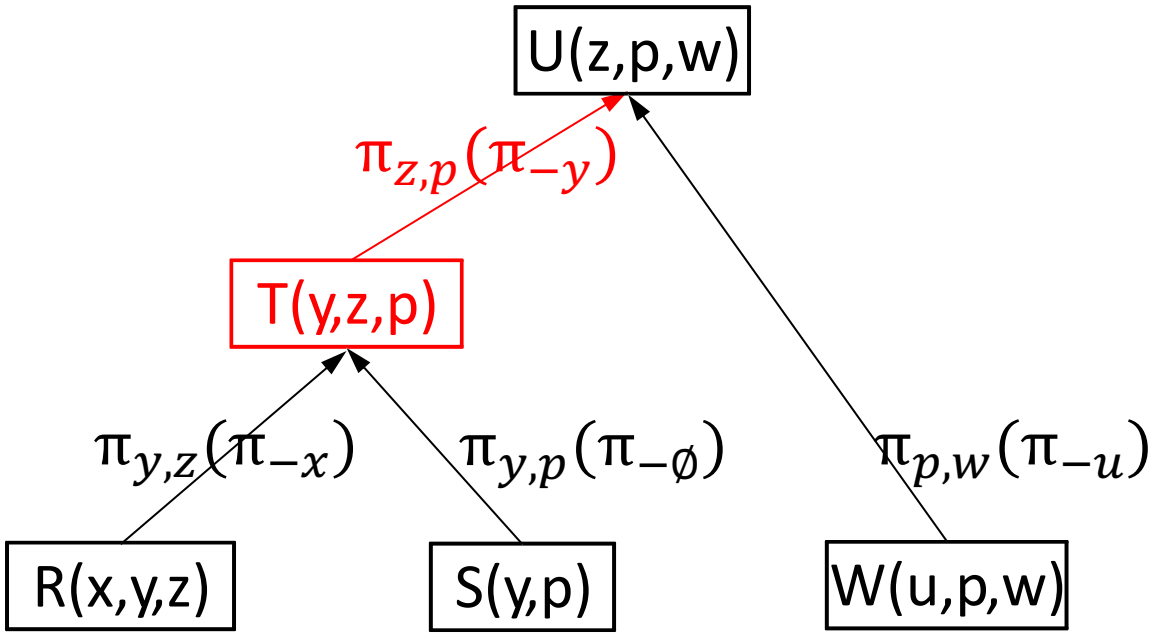


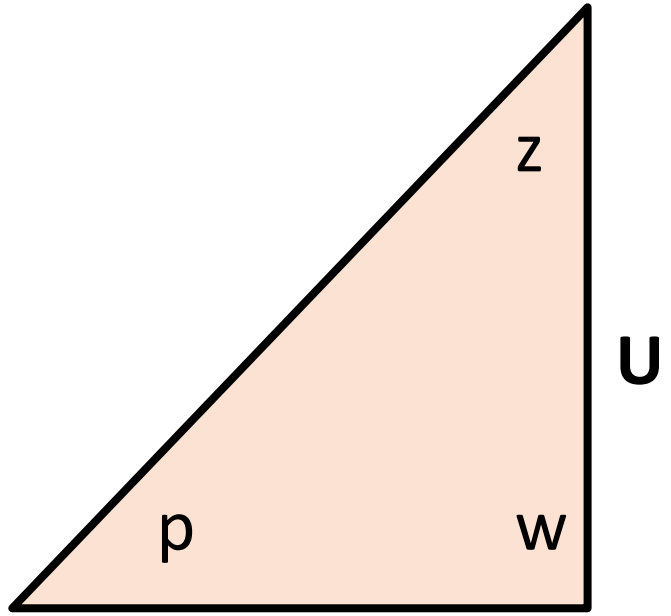



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

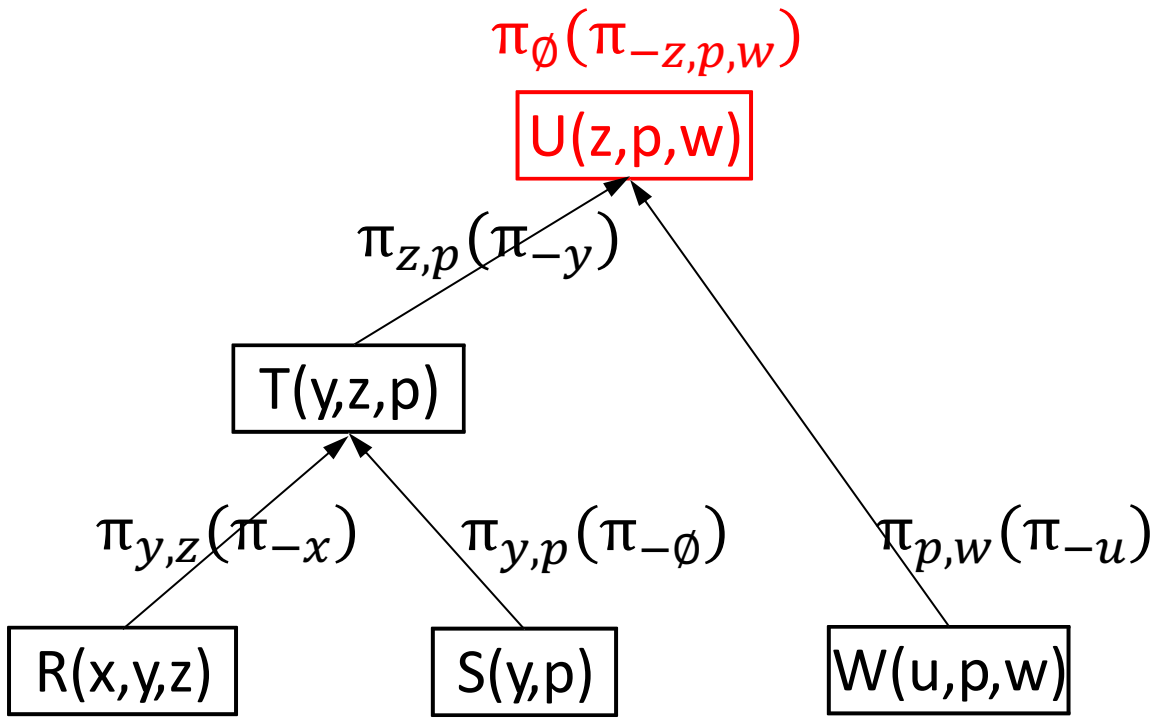


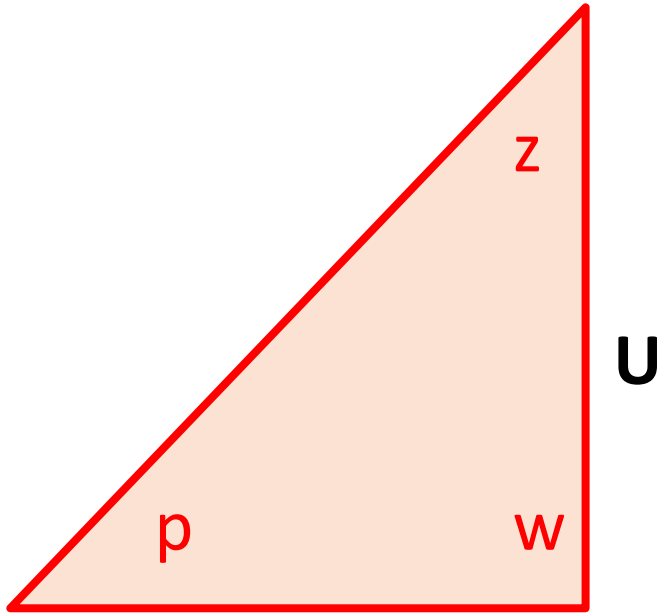



# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges

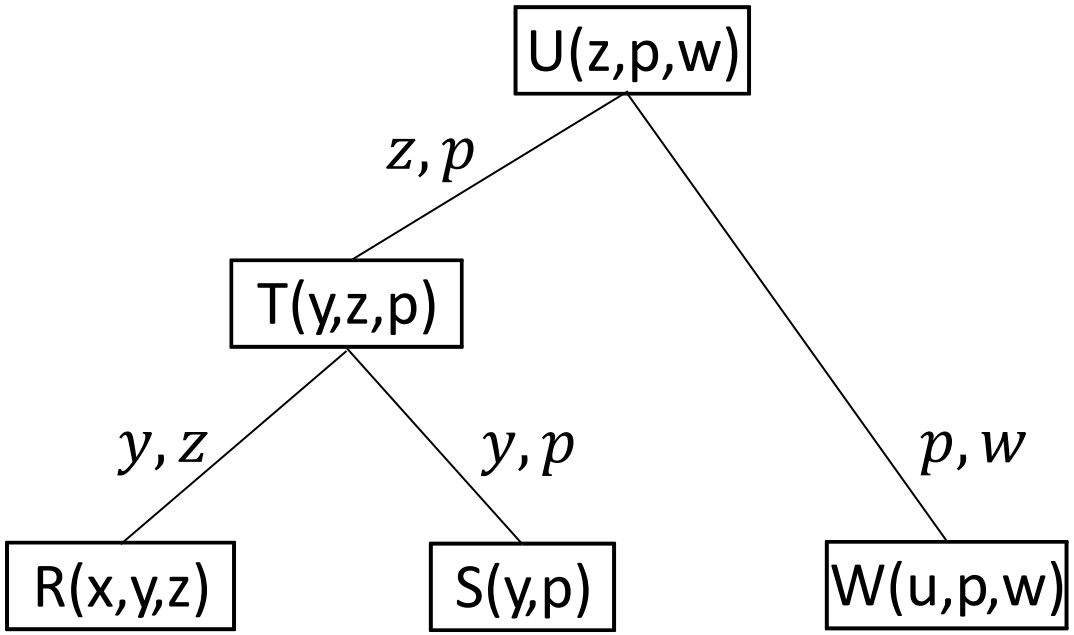


# GYO reduction by Example

Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

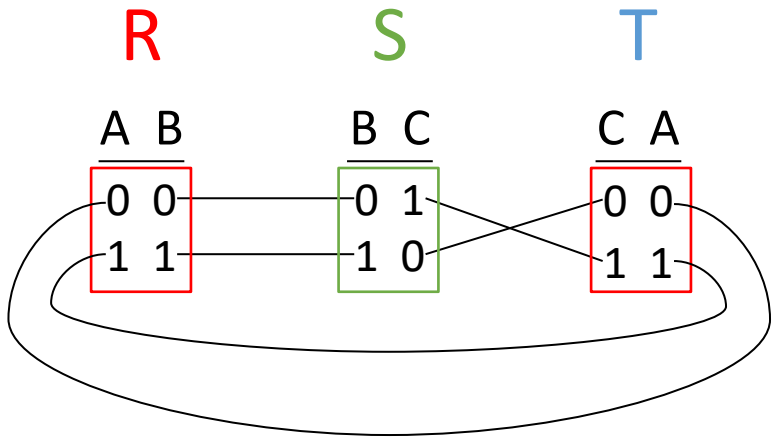
GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges




This is our join tree ☺

# What is so special about acyclic queries?

- "Consistency" = no dangling tuples (recall semi-join)
  - locally join consistent $\pi_{Ri}(R_i \bowtie R_j) = R_i$
  - globally join consistent $\pi_{Ri}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k) = R_i$
- Both implies each other only for acyclic queries
  - For all queries: global $\Rightarrow$ local
  - Acyclic queries: local $\Rightarrow$ global; but not for cyclic queries!

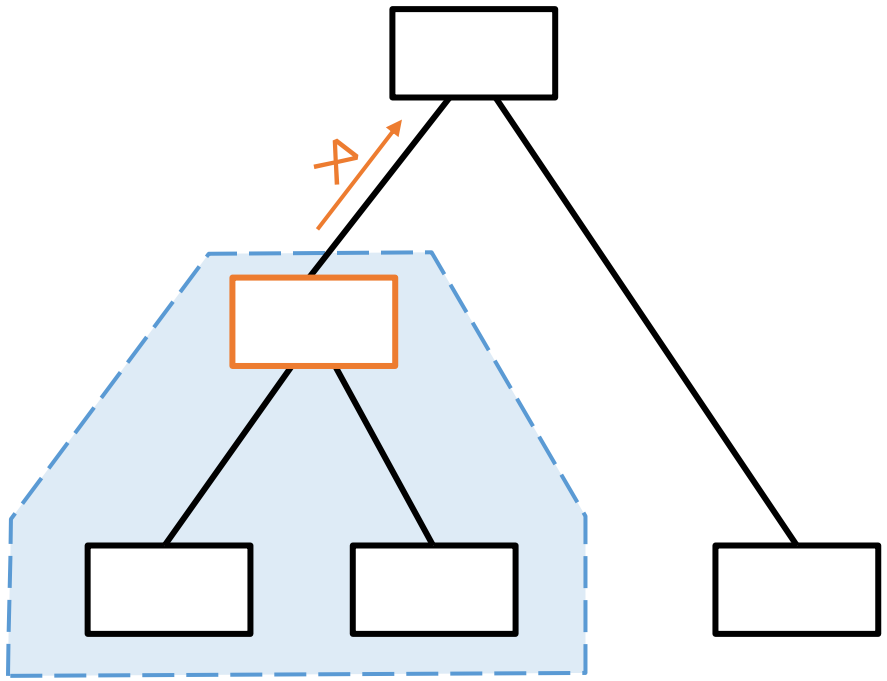



$$Q_\Delta(A,B,C) :- R(A,B), S(B,C), T(C,A)$$

- Any two relations are locally consistent. E.g. $R \bowtie S$ is {(0,0,0),(1,1,1)}, which projected onto $R$ is {(0,0),(1,1)}
- But $R \bowtie S \bowtie T = \emptyset$, so the relations are not globally consistent

Example from Beeri, Fagin, Maier, Yannakakis. On the desirability of acyclic database schemes. JACM 1983. https://doi.org/10.1145/2402.322389
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Semijoin Reduction as Message Passing on Trees



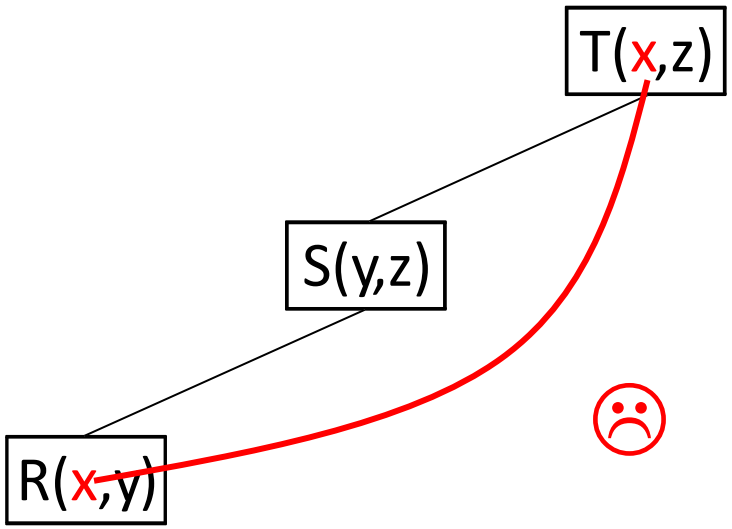
1. A message sent across an edge contains all the information from the subtree rooted at the sender.
   ⇒ Thus the reduction always needs to start at the leaves!

2. Key for acyclic queries: When joining, if there are no dangling tuples (Thus for a "reduced database") every additional join can never decrease the size of the intermediate query results!

# Semi-join reducers fail with the triangle

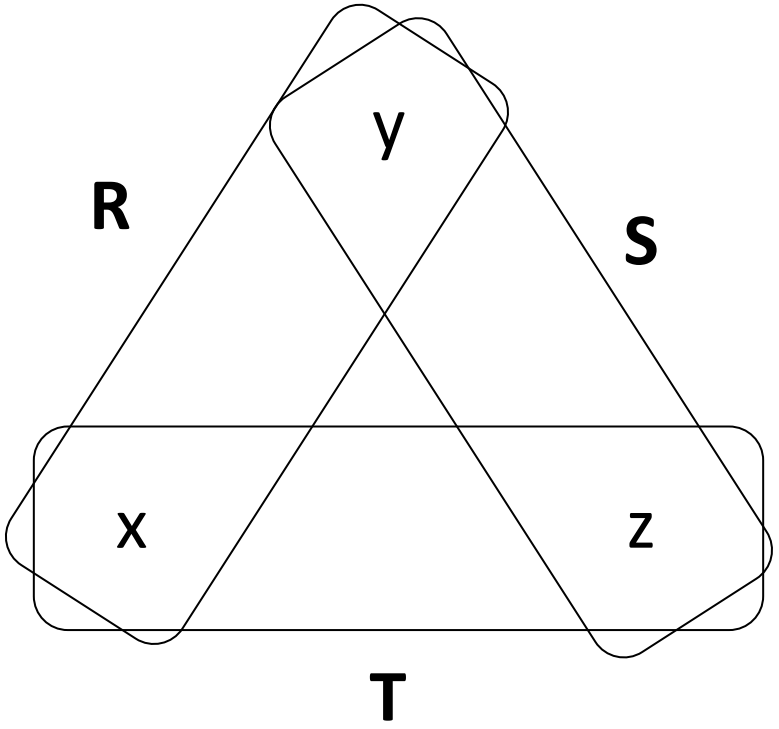
Q(x,y,z) :- R(x,y), S(y,z), T(x,z).

GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges





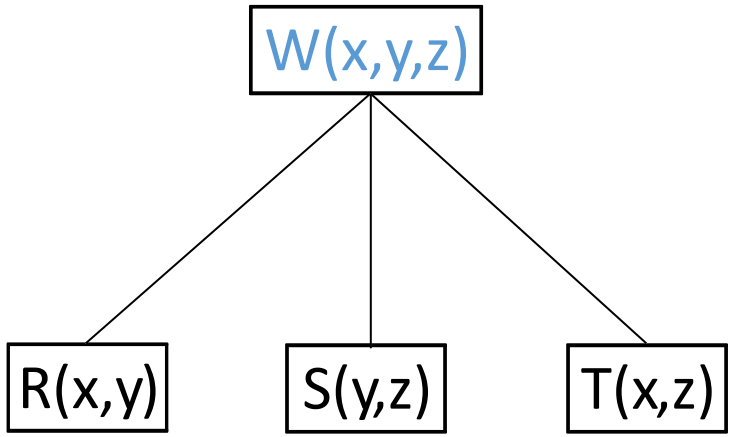

There is no join tree! You can't fulfill the running intersection property...

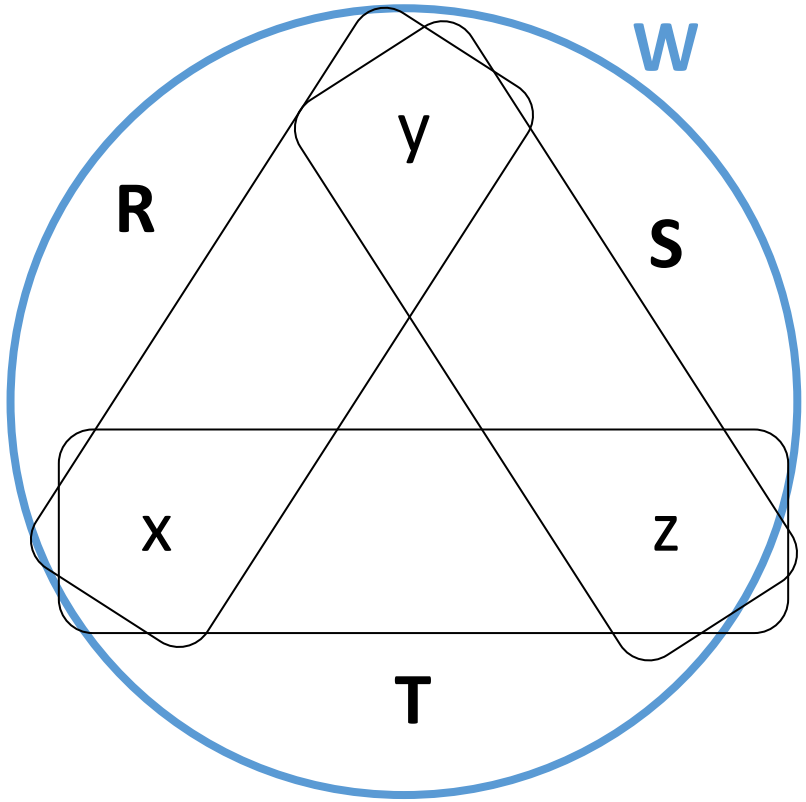


# Semi-join reducers work with the "beta-triangle"

Q(x,y,z) :- R(x,y), S(y,z), T(x,z), W(x,y,z).

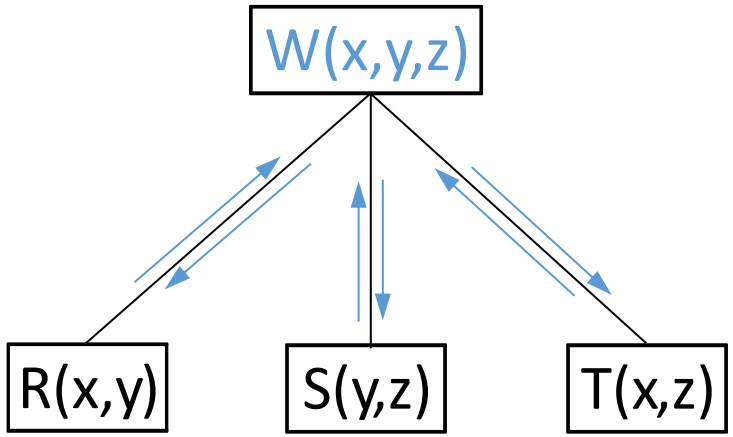
GYO reduction (ear removal)
- remove isolated nodes
- remove consumed or empty edges



# Semi-join reducers work with the "beta-triangle"

Q(x,y,z) :- R(x,y), S(y,z), T(x,z), W(x,y,z).



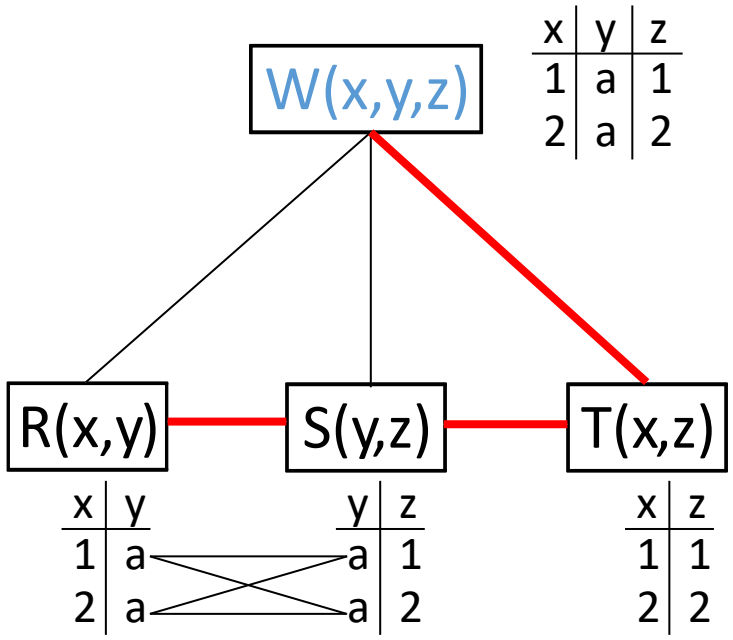

$$W_1(x,y,z) = W(x,y,z) \ltimes R(x,y)$$
$$W_2(x,y,z) = W_1(x,y,z) \ltimes S(y,z)$$
$$W_3(x,y,z) = W_2(x,y,z) \ltimes T(x,z)$$
$$R_1(x,y) = R(x,y) \ltimes W_3(x,y,z)$$
$$S_1(y,z) = S(y,z) \ltimes W_3(x,y,z)$$
$$T_1(x,z) = T(x,z) \ltimes W_3(x,y,z)$$

Q(x,y,z) :- $R_1$(x,y), $S_1$(y,z), $T_1$(x,z), $W_3$(x,y,z).

What can still go wrong after the reduction ?

# … but we still need to follow the join tree!

Q(x,y,z) :- R(x,y), S(y,z), T(x,z), W(x,y,z).




| x | y | z |
|---|---|---|
| 1 | a | 1 |
| 2 | a | 2 |

| x | y |
|---|---|
| 1 | a |
| 2 | a |

| y | z |
|---|---|
| a | 1 |
| a | 2 |

| x | z |
|---|---|
| 1 | 1 |
| 2 | 2 |

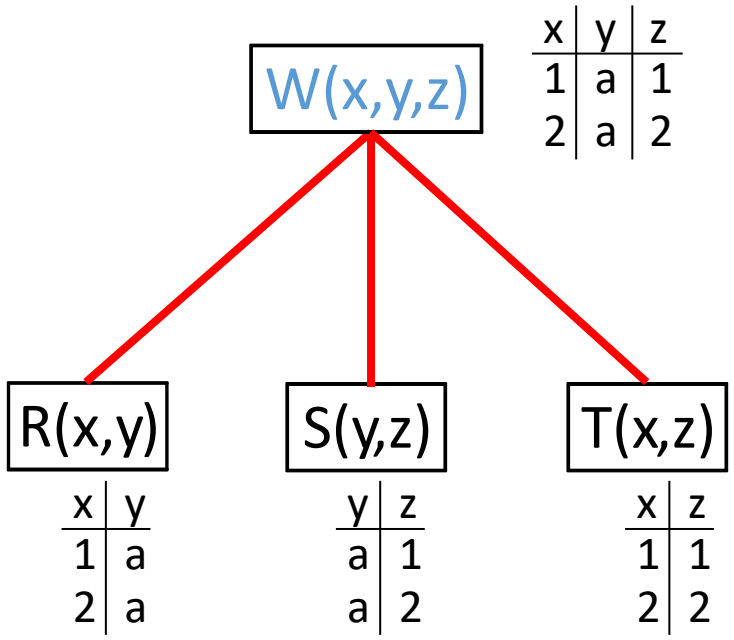
$$W_1(x,y,z) = W(x,y,z) \ltimes R(x,y)$$
$$W_2(x,y,z) = W_1(x,y,z) \ltimes S(y,z)$$
$$W_3(x,y,z) = W_2(x,y,z) \ltimes T(x,z)$$
$$R_1(x,y) = R(x,y) \ltimes W_3(x,y,z)$$
$$S_1(y,z) = S(y,z) \ltimes W_3(x,y,z)$$
$$T_1(x,z) = T(x,z) \ltimes W_3(x,y,z)$$

$$Q(x,y,z) = \left(\left(R_1(x,y) \bowtie S_1(y,z)\right) \bowtie T_1(x,z)\right) \bowtie W_3(x,y,z)$$

| | | |
|---|---|---|
| 1 | a | 1 |
| 2 | a | 2 |
| 1 | a | 1 |
| 2 | a | 2 |

# … but we still need to follow the join tree!

Q(x,y,z) :- R(x,y), S(y,z), T(x,z), W(x,y,z).




| x | y | z |
|---|---|---|
| 1 | a | 1 |
| 2 | a | 2 |

| x | y |
|---|---|
| 1 | a |
| 2 | a |

| y | z |
|---|---|
| a | 1 |
| a | 2 |

| x | z |
|---|---|
| 1 | 1 |
| 2 | 2 |

$$W_1(x,y,z) = W(x,y,z) \ltimes R(x,y)$$
$$W_2(x,y,z) = W_1(x,y,z) \ltimes S(y,z)$$
$$W_3(x,y,z) = W_2(x,y,z) \ltimes T(x,z)$$
$$R_1(x,y) = R(x,y) \ltimes W_3(x,y,z)$$
$$S_1(y,z) = S(y,z) \ltimes W_3(x,y,z)$$
$$T_1(x,z) = T(x,z) \ltimes W_3(x,y,z)$$

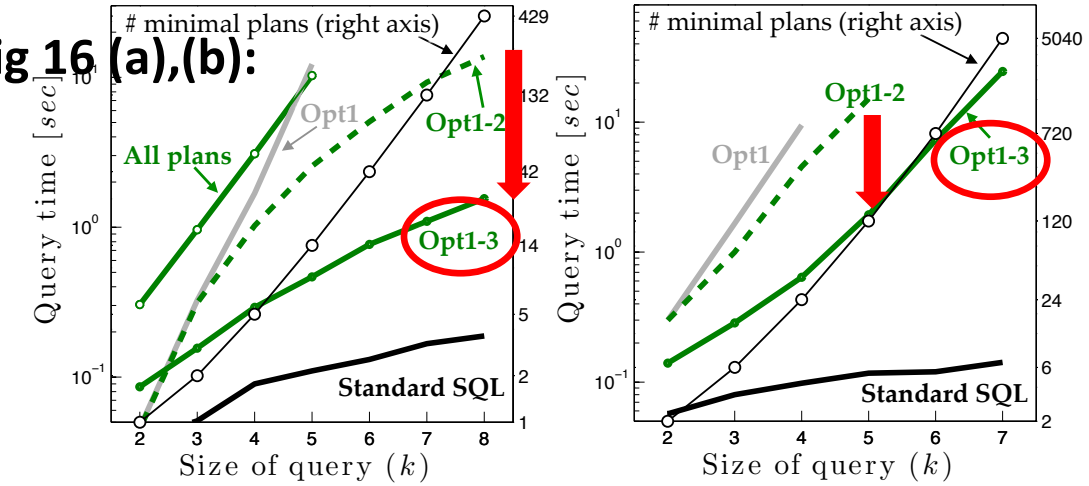
~~$$Q(x,y,z) = ((R_1(x,y) \bowtie S_1(y,z)) \bowtie T_1(x,z)) \bowtie W_3(x,y,z)$$~~

$$Q(x,y,z) = ((R_1(x,y) \bowtie W_3(x,y,z)) \bowtie S_1(y,z)) \bowtie T_1(x,z)$$

We still need to follow the join tree!

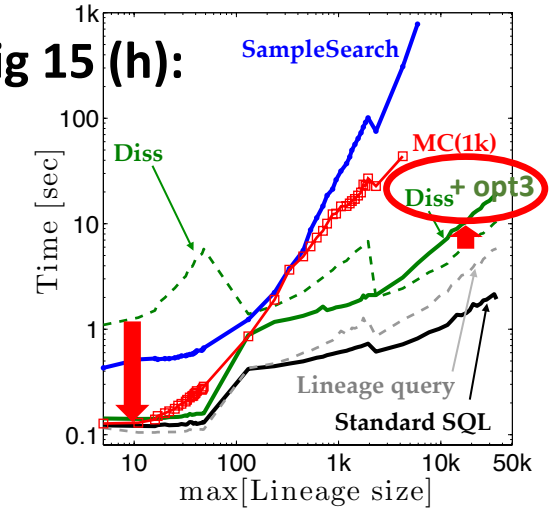
# Semi-join reductions can be extremely powerful

**Fig 16 (a),(b):**




(a) $k$-chain queries (b) $k$-star queries

**Fig 15 (h):**




(h) Combining (a)-(c)

Semi-join reductions can be extremely powerful in different contexts (yet depend on the concrete input to pay off)

6.3 Opt. 3: Deterministic semi-join reduction

The most expensive operations in probabilistic query plans are the group-bys for the probabilistic project operations. These are often applied early in the plans to tuples which are later pruned and do not contribute to the final query result. Our third optimization is to first apply a *full semi-join reduction on the input relations* before starting the probabilistic evaluation from these *reduced input relations*.

We like to draw here an important connection to [54], which introduces the idea of "lazy plans" and shows orders of magnitude performance improvements for safe plans by computing confidences not after each join and projection, but rather at the very end of the plan. We note that our semi-join reduction *serves the same purpose* with similar performance improvements and also apply for safe queries. The advantage of semi-join reductions, however, is that we do not require any modifications to the query engine.

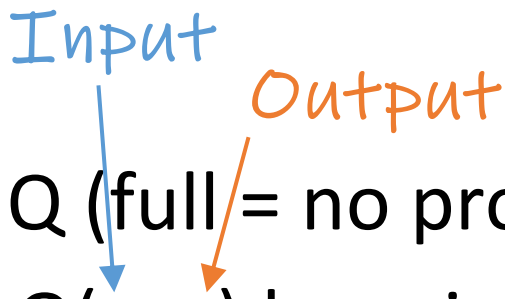
From: Gatterbauer, Suciu. "Dissociation and propagation for approximate lifted inference with standard relational database management systems", VLDBJ 2017. https://arxiv.org/pdf/1310.6257
Reference [54]: Olteanu, Huang, Koch. "Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases", ICDE 2009. https://doi.org/10.1109/ICDE.2009.123
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Outline Part 3

Part 3: Acyclic Queries & Enumeration (Wolfgang) ~25min

- The power of semi-join reductions
- Hypergraphs, GYO, join trees
- Yannakakis = acyclic query evaluation
- Enumerating answers
- Projections

# Yannakakis Algorithm



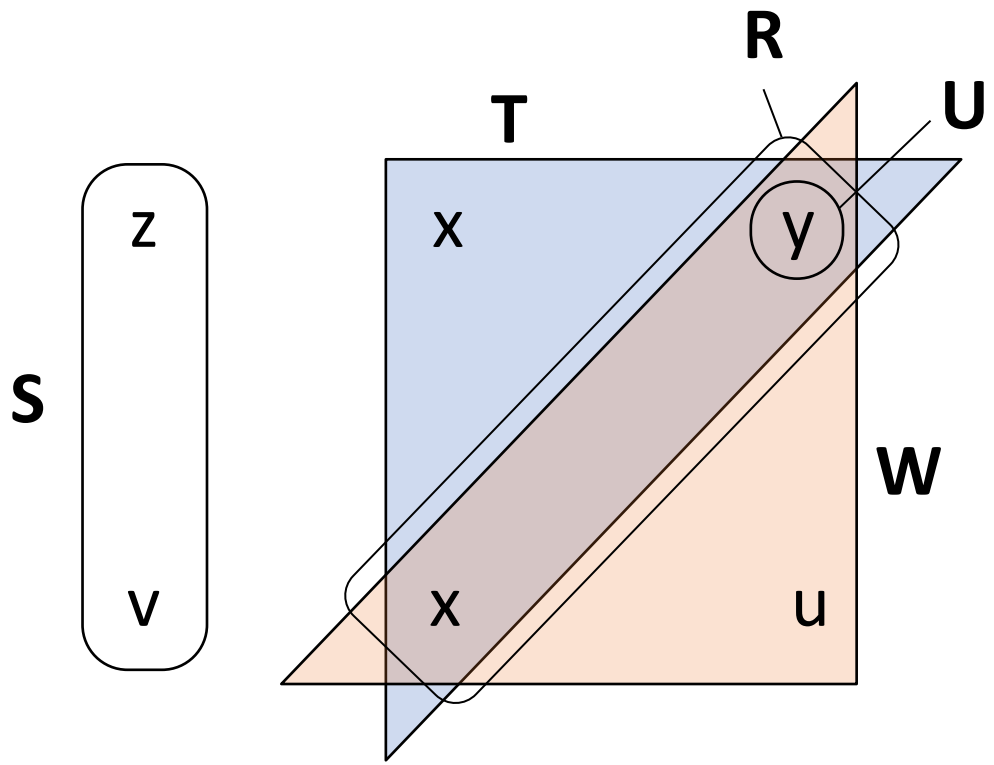

- Given: acyclic full conjunctive query Q (full = no projections)
- Compute Q on any database in time O(n+r) by using the join tree
- Step 1: semi-join reduction (two sweeps)
  - Pick any root node R in the join tree of Q
  - Sweep bottom-up: Do a semi-join reduction from the leaves to R
  - Sweep top-down: Do a semi-join reduction from R to the leaves
- Step 2: use the join tree as query plan: pick any root and join bottom-up or top-down
  - Notice that step 2 can be combined with the top-down SJ-reduction

Yannakakis, "Algorithms for acyclic database schemes", VLDB 1981. https://dl.acm.org/doi/10.5555/1286831.1286840
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**




R(x,y)

| x | y |
|---|---|
| $a_1$ | $b_1$ |
| $a_1$ | $b_2$ |
| $a_4$ | $b_6$ |

one of several possible ones

**Rooted Join tree**

S(z,v)

| z | v |
|---|---|
| $c_1$ | $d_1$ |
| $c_1$ | $d_2$ |
| $c_4$ | $d_6$ |

T(p,x,y)

| p | x | y |
|---|---|---|
| $e_1$ | $a_1$ | $b_1$ |
| $e_1$ | $a_1$ | $b_2$ |
| $e_3$ | $a_3$ | $b_1$ |
| $e_3$ | $a_1$ | $b_4$ |
| $e_2$ | $a_2$ | $b_3$ |

?

U(y)

| y |
|---|
| $b_1$ |
| $b_2$ |
| $b_3$ |

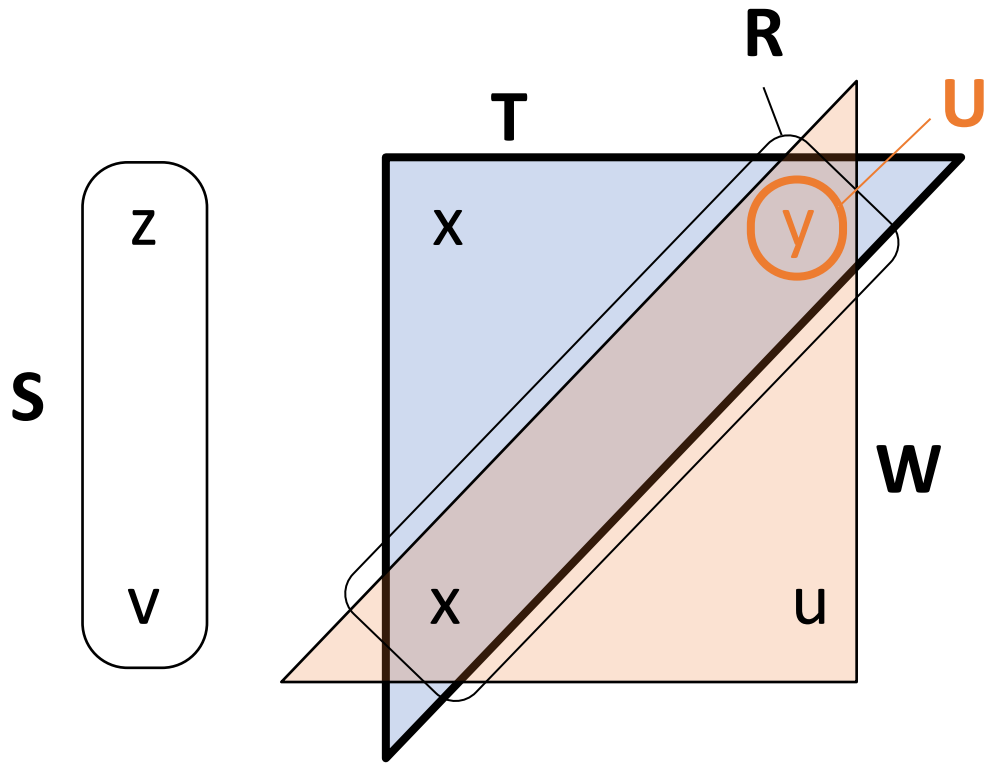
W(u,x,y)

| u | x | y |
|---|---|---|
| $f_1$ | $a_1$ | $b_1$ |
| $f_1$ | $a_1$ | $b_2$ |
| $f_2$ | $a_1$ | $b_2$ |
| $f_2$ | $a_2$ | $b_2$ |

# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**




R(x,y)

| x | y |
|---|---|
| $a_1$ | $b_1$ |
| $a_1$ | $b_2$ |
| $a_4$ | $b_6$ |

**Rooted Join tree**

S(z,v)

| z | v |
|---|---|
| $c_1$ | $d_1$ |
| $c_1$ | $d_2$ |
| $c_4$ | $d_6$ |

T(p,x,y)

| p | x | y |
|---|---|---|
| $e_1$ | $a_1$ | $b_1$ |
| $e_1$ | $a_1$ | $b_2$ |
| $e_3$ | $a_3$ | $b_1$ |
| $e_3$ | $a_1$ | $b_4$ |
| $e_2$ | $a_2$ | $b_3$ |

shared variable y
no isolated variable

y

U(y)

| y |
|---|
| $b_1$ |
| $b_2$ |
| $b_3$ |

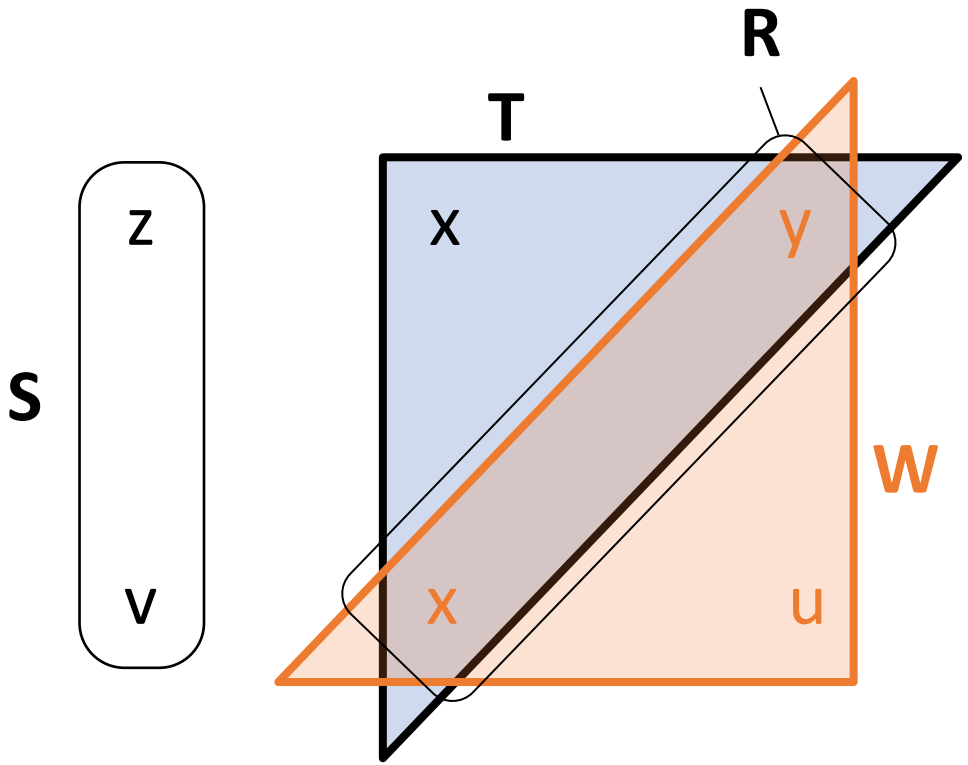
W(u,x,y)

| u | x | y |
|---|---|---|
| $f_1$ | $a_1$ | $b_1$ |
| $f_1$ | $a_1$ | $b_2$ |
| $f_2$ | $a_1$ | $b_2$ |
| $f_2$ | $a_2$ | $b_2$ |

# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**




R(x,y)

| x | y |
|---|---|
| $a_1$ | $b_1$ |
| $a_1$ | $b_2$ |
| $a_4$ | $b_6$ |

**Rooted Join tree**

S(z,v)

| z | v |
|---|---|
| $c_1$ | $d_1$ |
| $c_1$ | $d_2$ |
| $c_4$ | $d_6$ |

T(p,x,y)

| p | x | y |
|---|---|---|
| $e_1$ | $a_1$ | $b_1$ |
| $e_1$ | $a_1$ | $b_2$ |
| $e_3$ | $a_3$ | $b_1$ |
| $e_3$ | $a_1$ | $b_4$ |
| $e_2$ | $a_2$ | $b_3$ |

shared variable y
no isolated variable

shared variables x,y,
isolated variable u
gets projected away


U(y)

| y |
|---|
| $b_1$ |
| $b_2$ |
| $b_3$ |

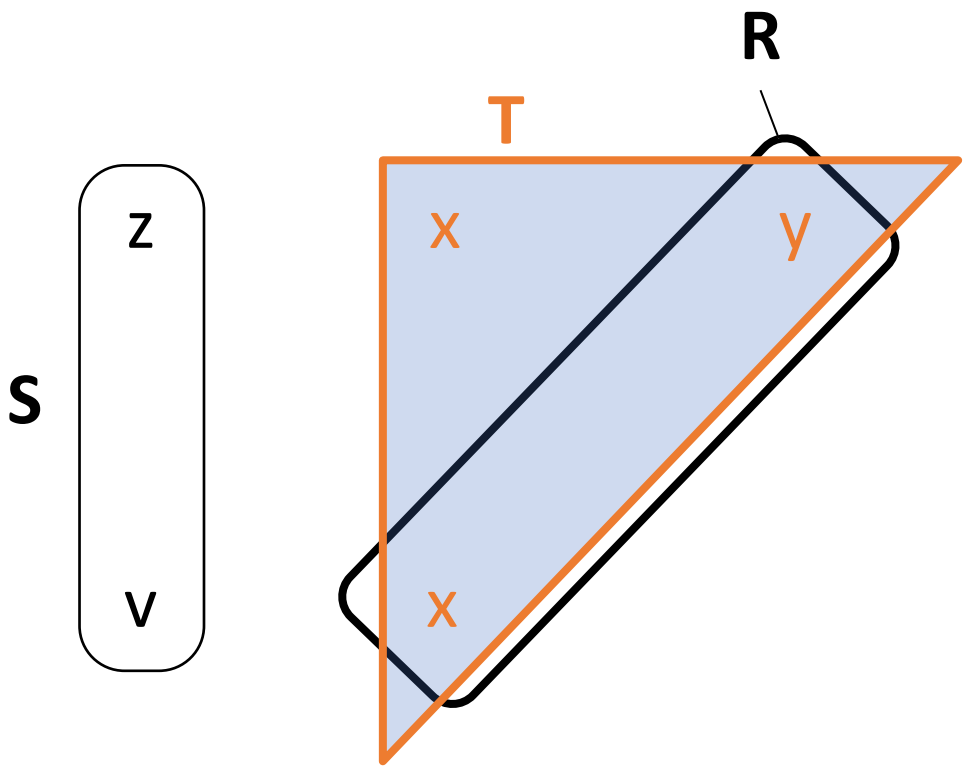
W(u,x,y)

| u | x | y |
|---|---|---|
| $f_1$ | $a_1$ | $b_1$ |
| $f_1$ | $a_1$ | $b_2$ |
| $f_2$ | $a_1$ | $b_2$ |
| $f_2$ | $a_2$ | $b_2$ |

# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**




**Rooted Join tree**

R(x,y)

| x | y |
|---|---|
| $a_1$ | $b_1$ |
| $a_1$ | $b_2$ |
| $a_4$ | $b_6$ |

x,y

S(z,v)

| z | v |
|---|---|
| $c_1$ | $d_1$ |
| $c_1$ | $d_2$ |
| $c_4$ | $d_6$ |

T(p,x,y)

| p | x | y |
|---|---|---|
| $e_1$ | $a_1$ | $b_1$ |
| $e_1$ | $a_1$ | $b_2$ |
| $e_3$ | $a_3$ | $b_1$ |
| $e_3$ | $a_1$ | $b_4$ |
| $e_2$ | $a_2$ | $b_3$ |

shared variables x,y, isolated variable u gets projected away

y x,y

U(y)

| y |
|---|
| $b_1$ |
| $b_2$ |
| $b_3$ |

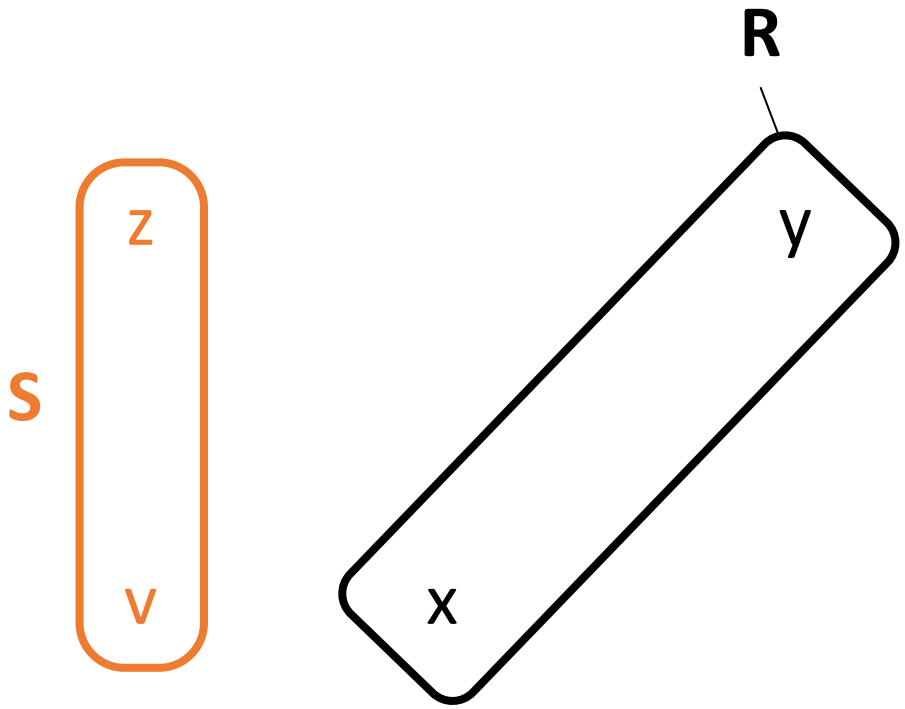
W(u,x,y)

| u | x | y |
|---|---|---|
| $f_1$ | $a_1$ | $b_1$ |
| $f_1$ | $a_1$ | $b_2$ |
| $f_2$ | $a_1$ | $b_2$ |
| $f_2$ | $a_2$ | $b_2$ |

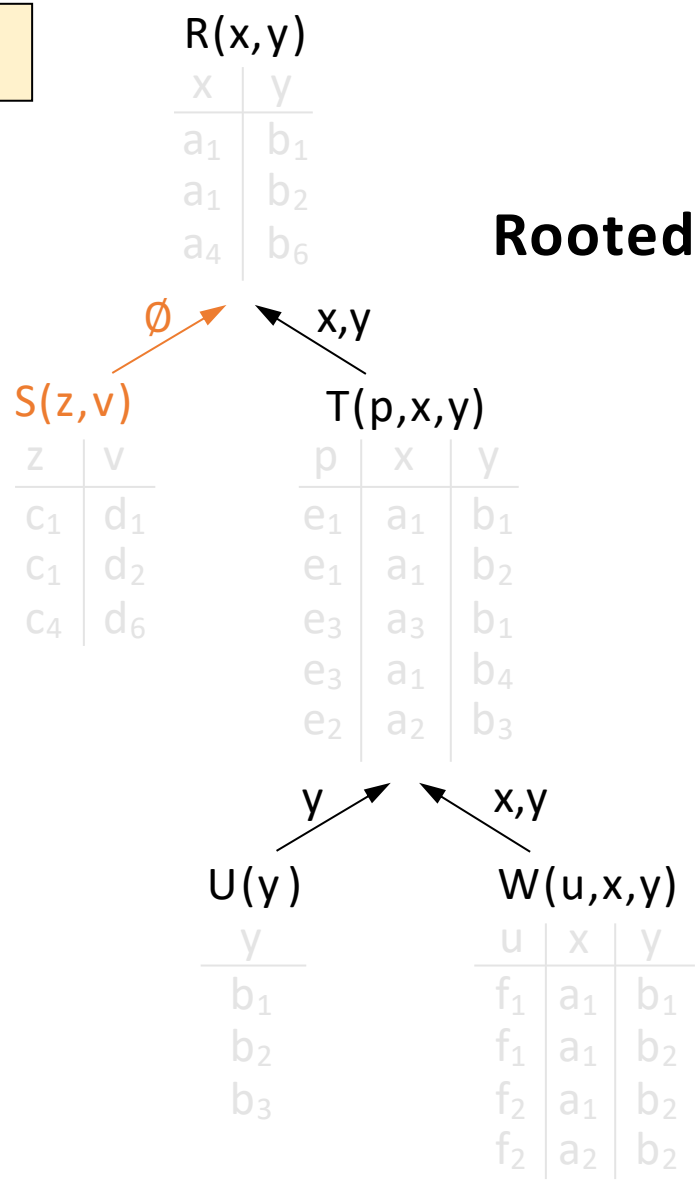
# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**



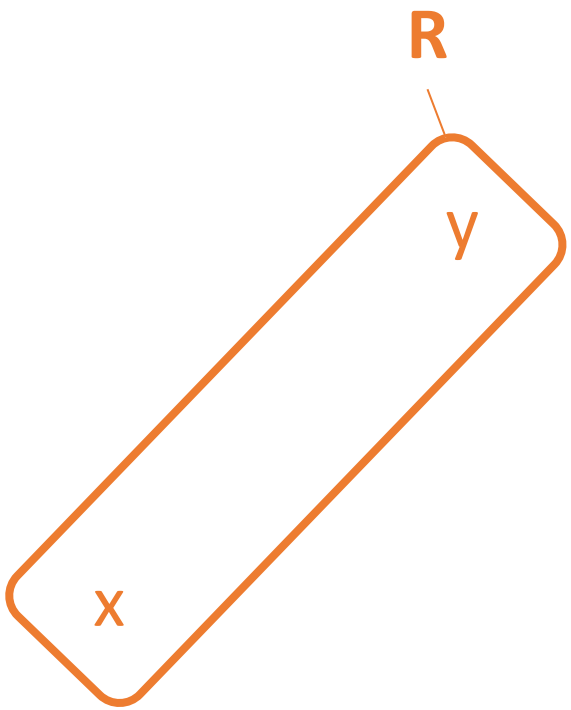

**Rooted Join tree**



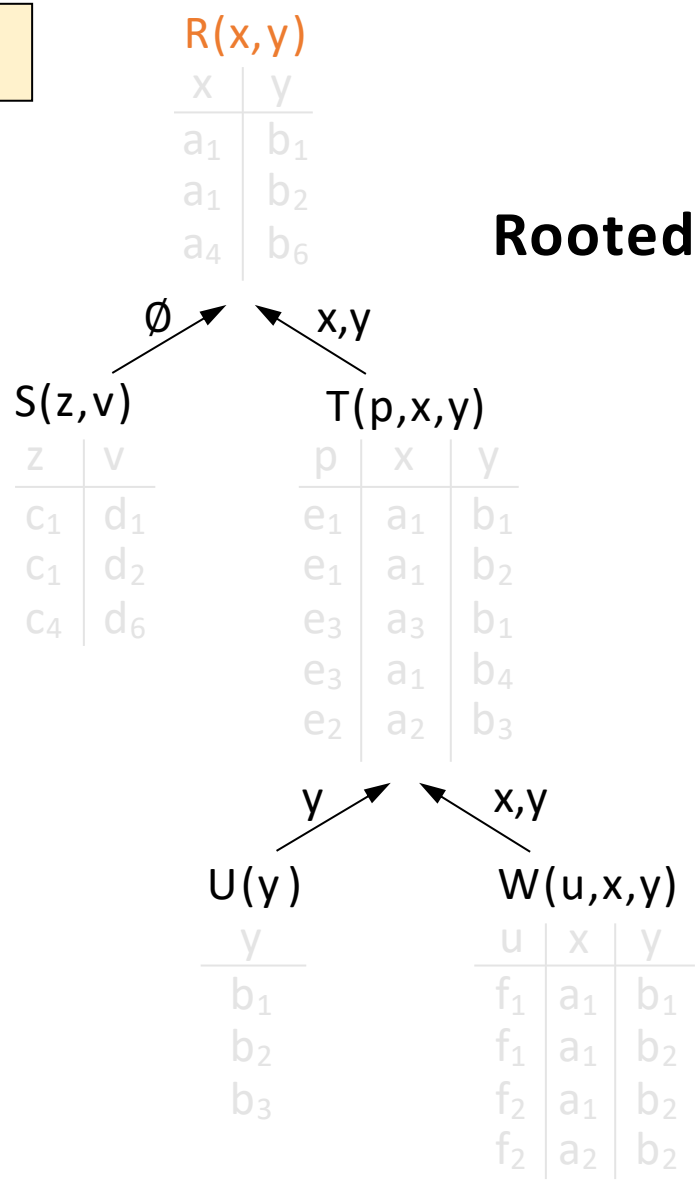
# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**Hypergraph**

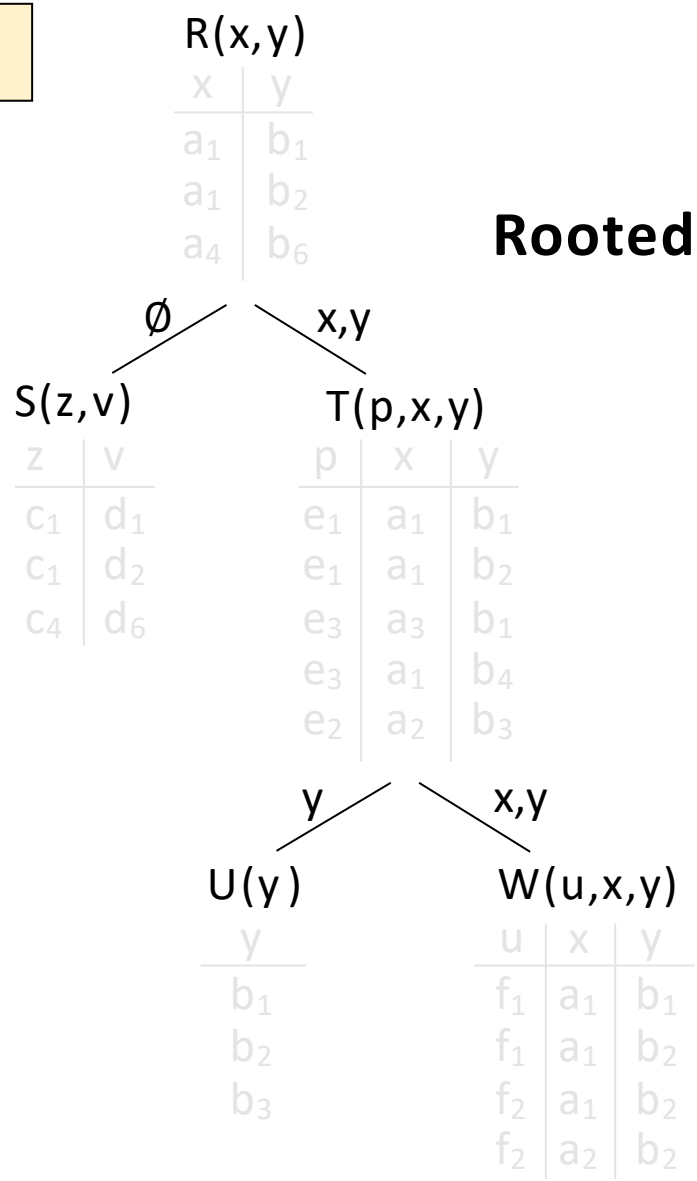



**Rooted Join tree**



# Yannakakis example: first use GYO to get the join tree

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

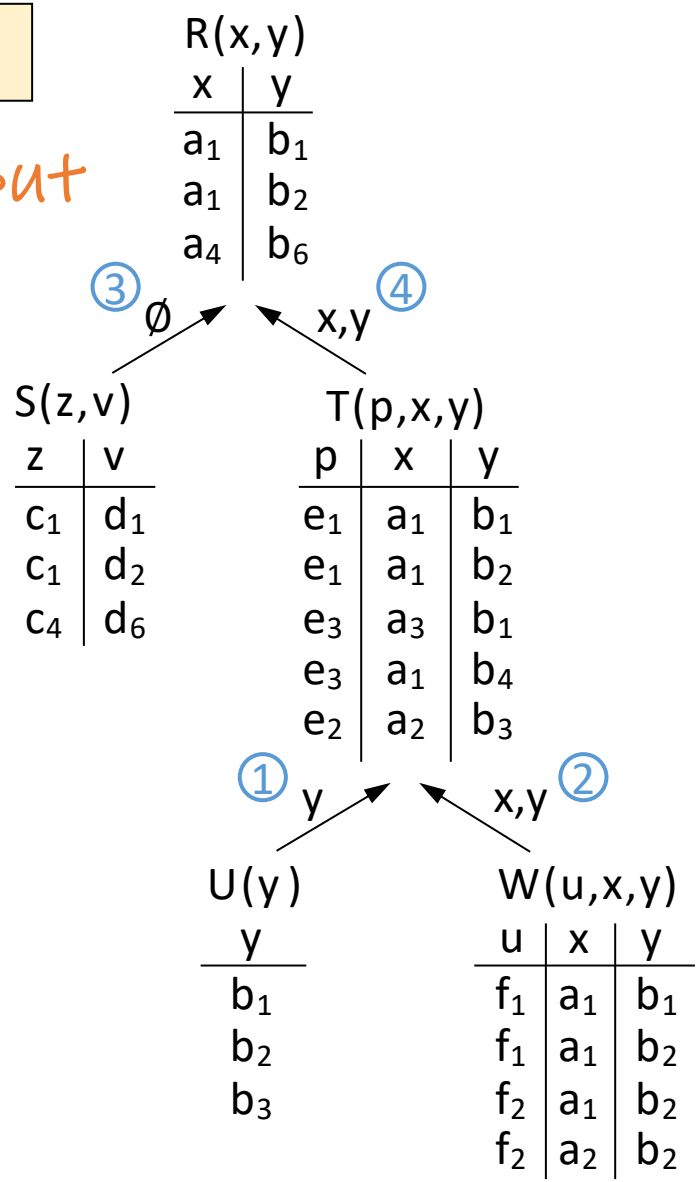


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in $O(n)$ ← Input

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

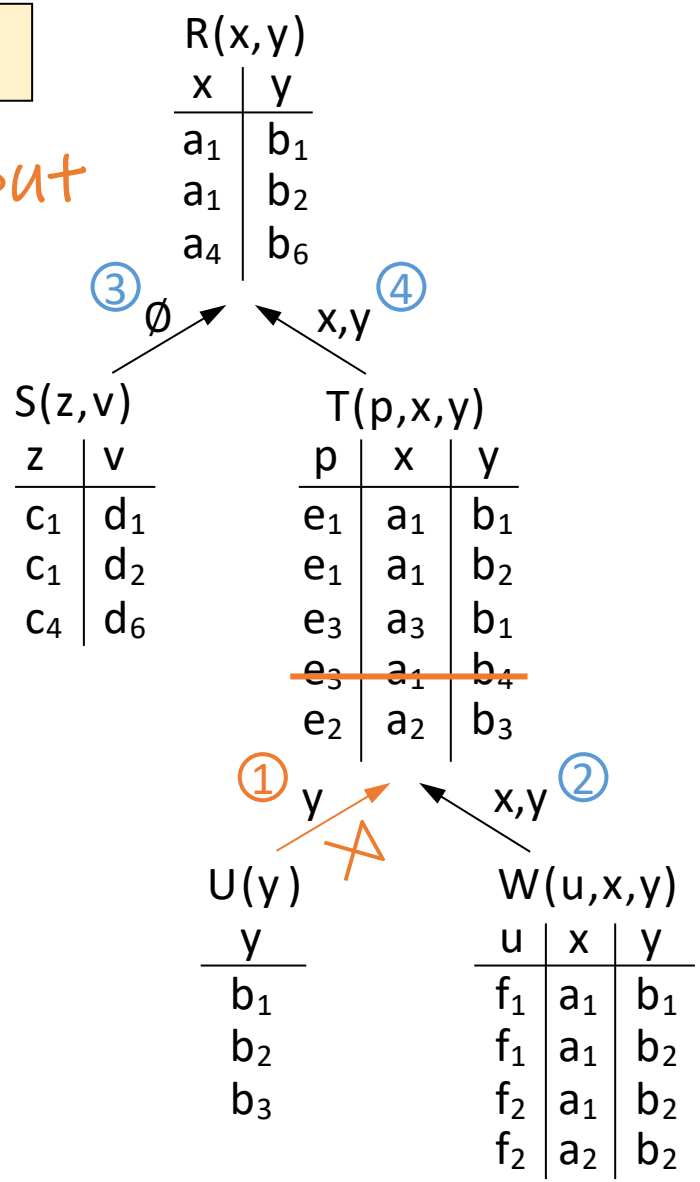


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in $O(n)$ ← Input

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

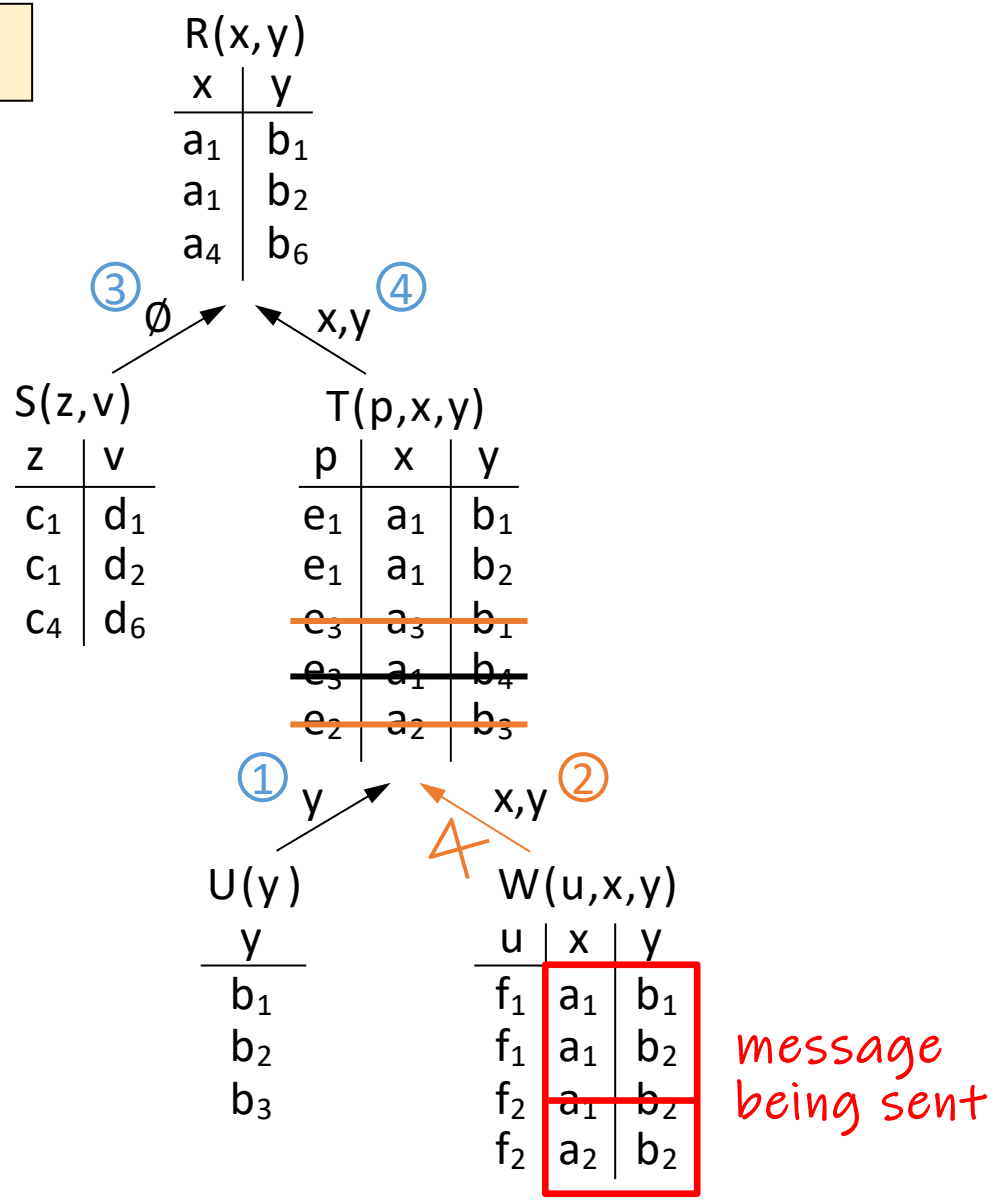


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋉ (remove dangling tuples) in $O(n)$

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

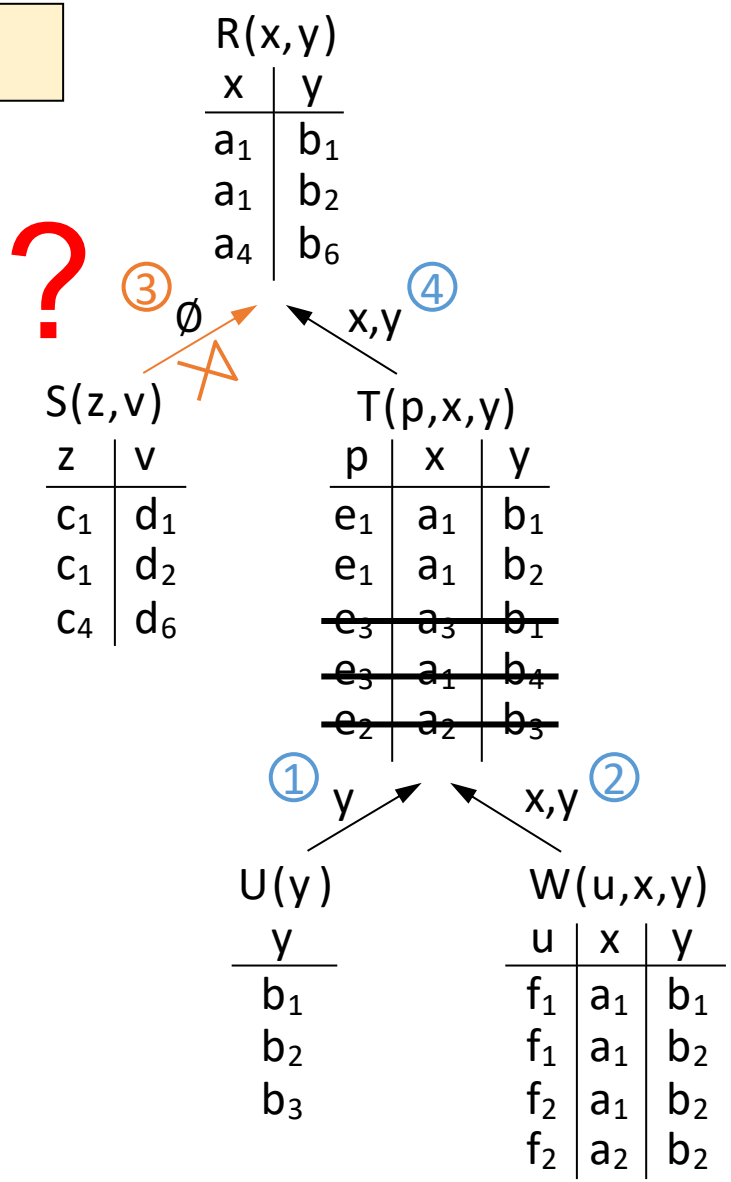


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in *O*(n)

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

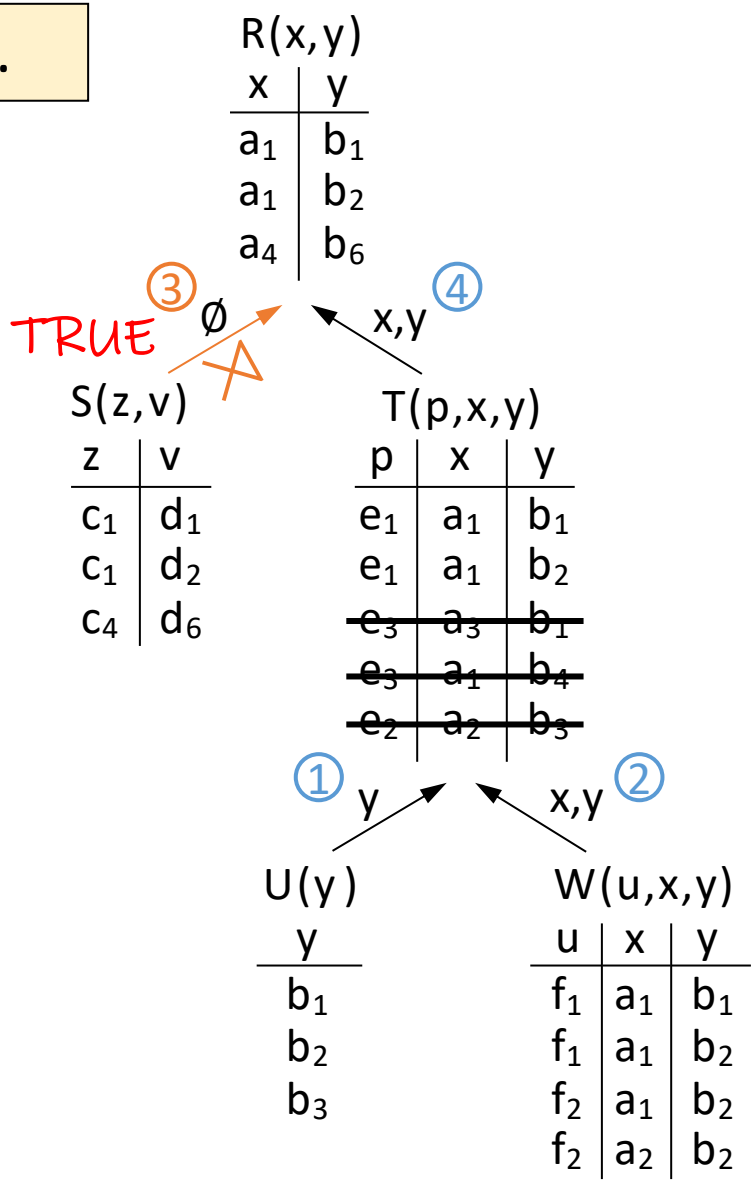


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in $O(n)$

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

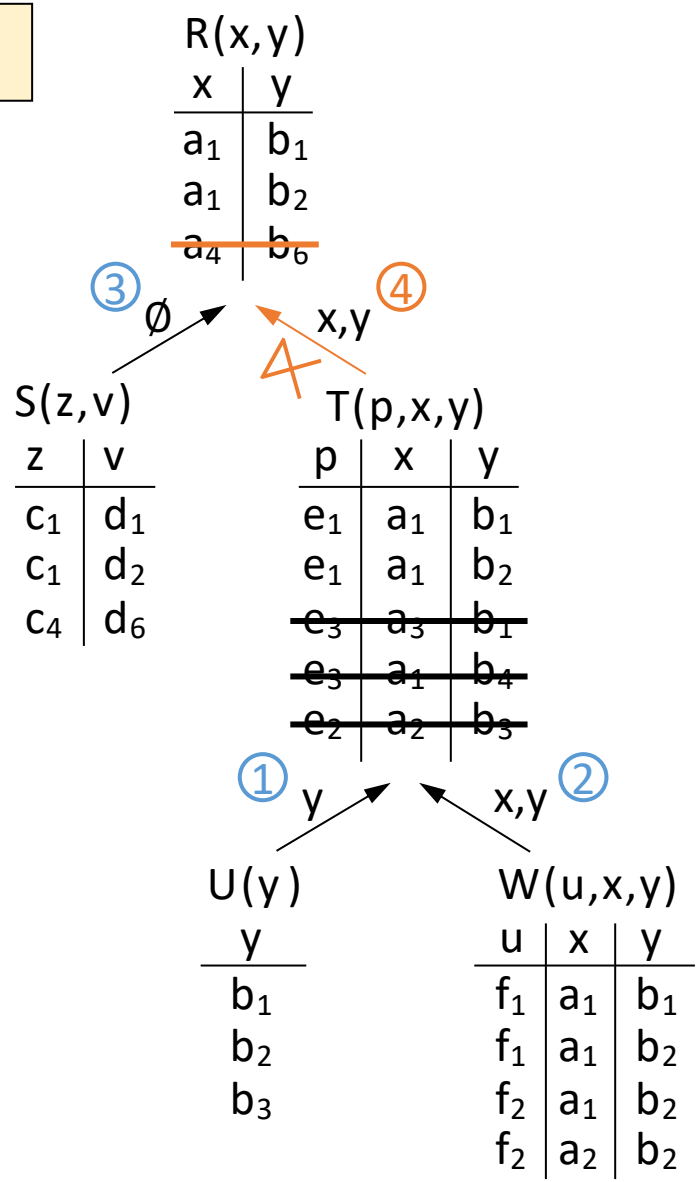


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in $O(n)$

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**

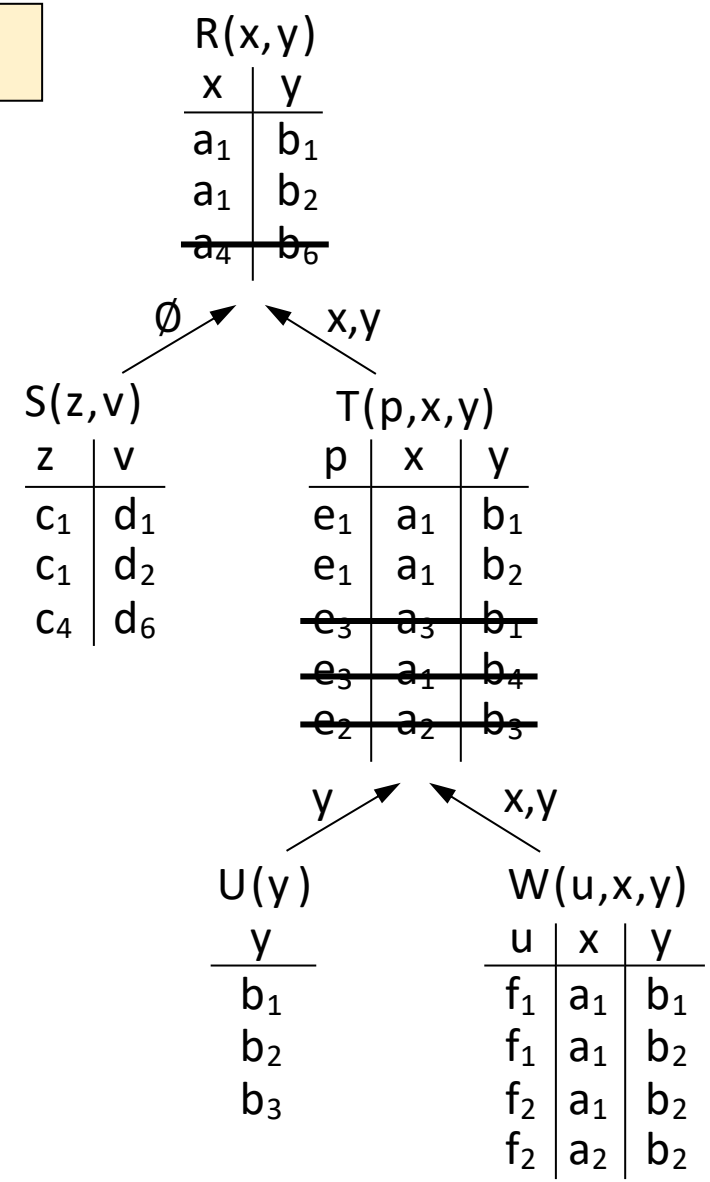


# Yannakakis Algorithm example: 1st pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋉ (remove dangling tuples) in $O(n)$

- **Bottom-up semi-join propagation from leaves to root in some reverse topological order**




Notice that at the end of the 1st pass, the table R at the root does not contain any more dangling tuples; it is completely reduced.

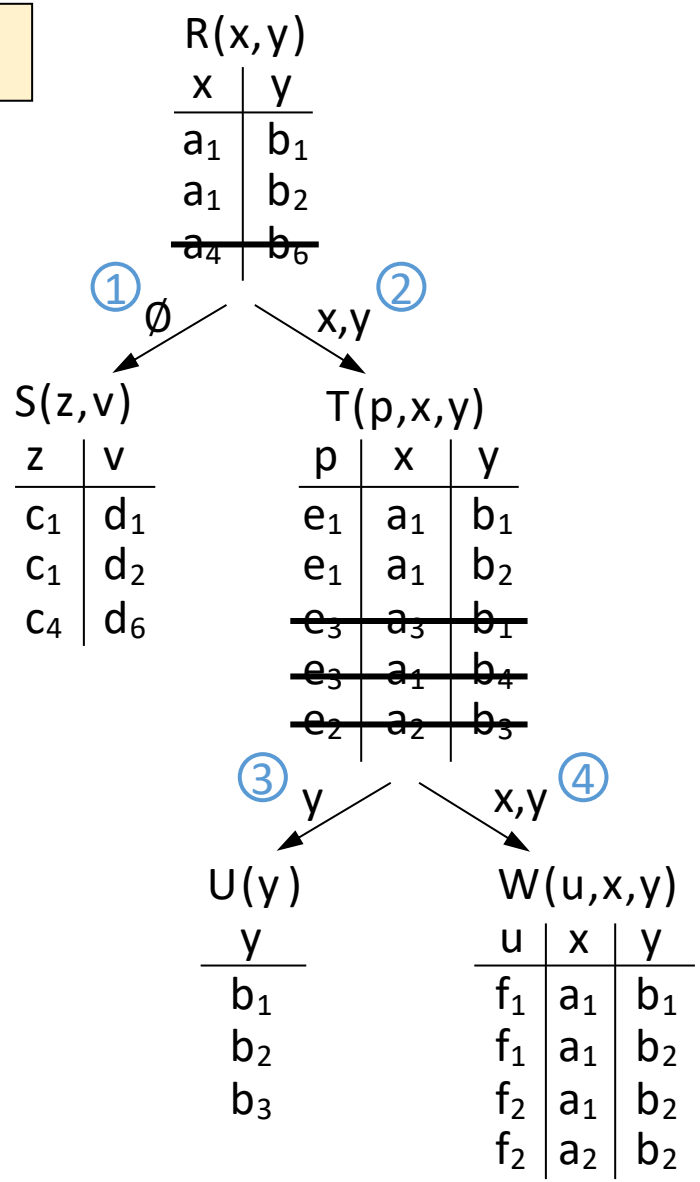
In other words, with a sequence of only local updates, we have accumulated at the root all necessary information to answer the Boolean query.

# Yannakakis Algorithm example: 2nd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**

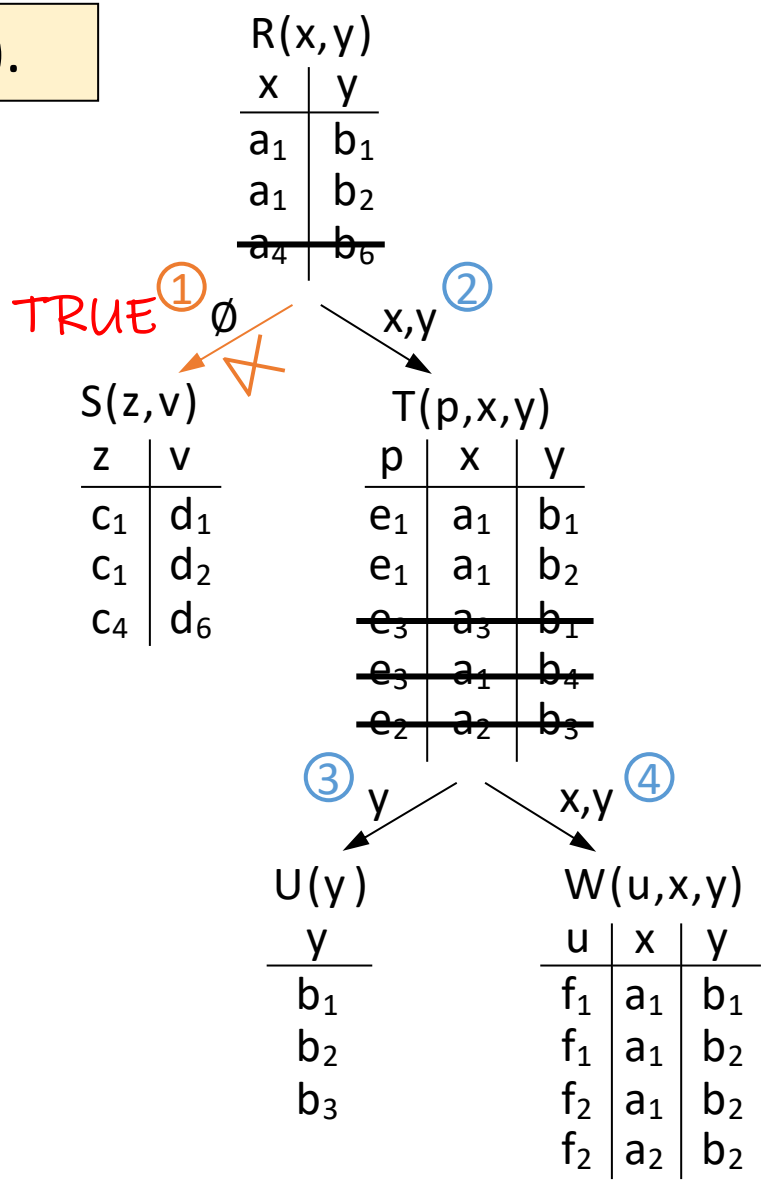


# Yannakakis Algorithm example: 2nd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**

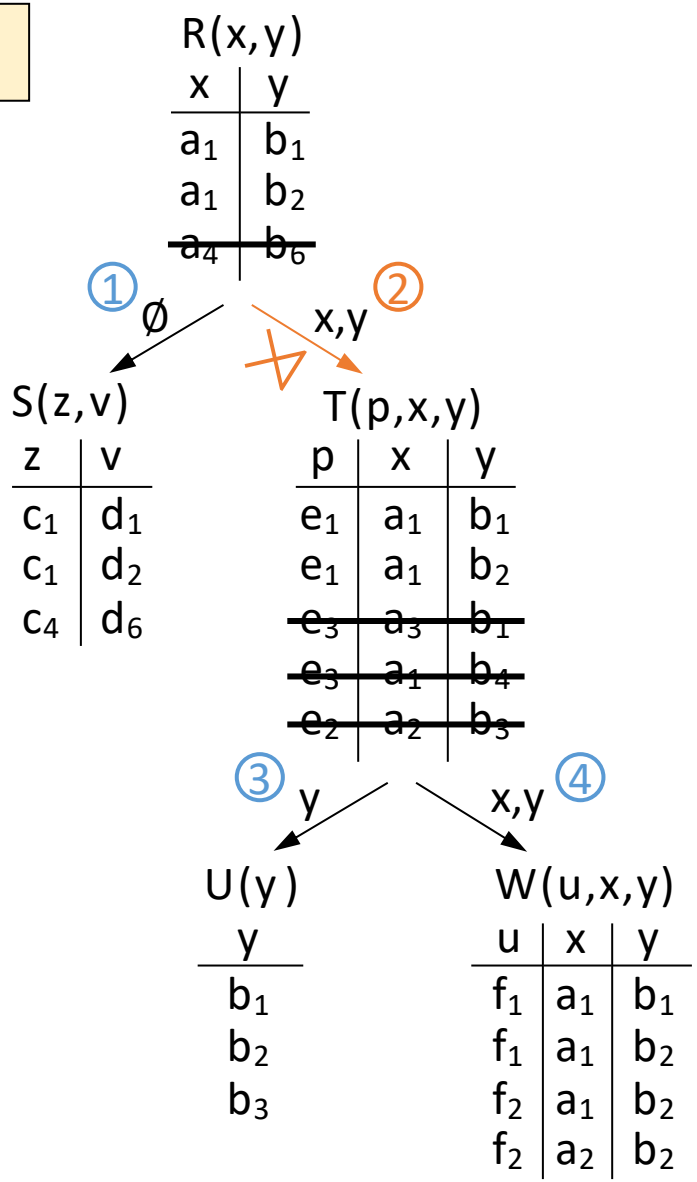


# Yannakakis Algorithm example: 2nd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋉ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**

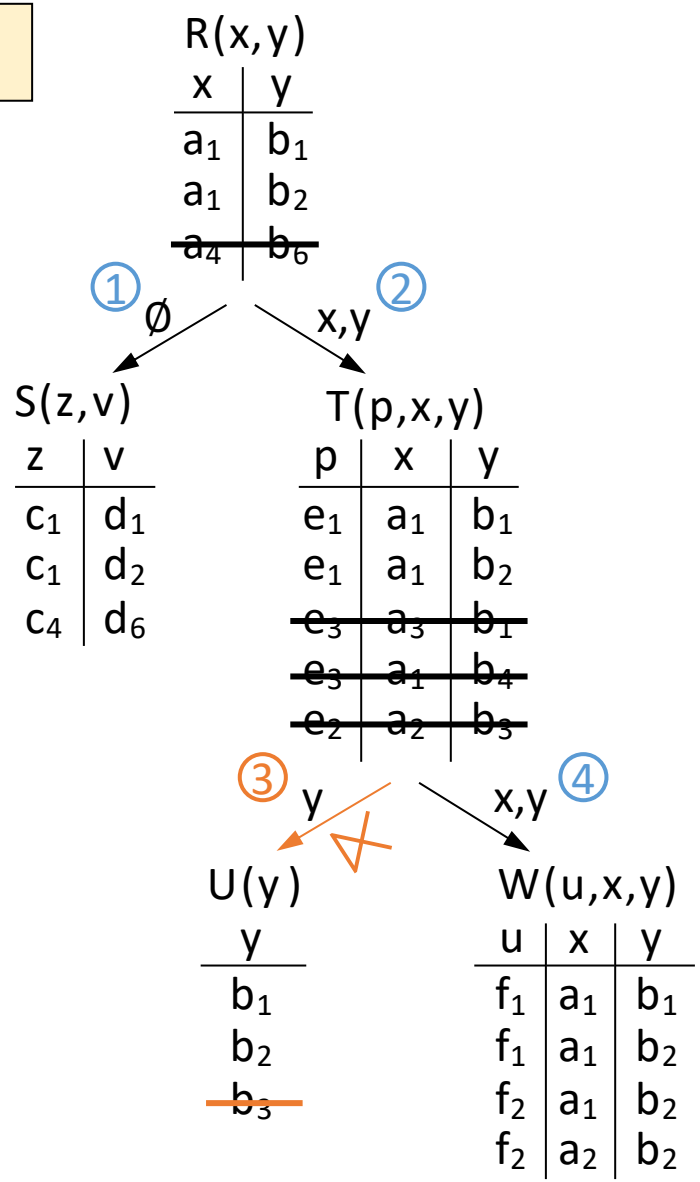


# Yannakakis Algorithm example: 2nd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋉ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**

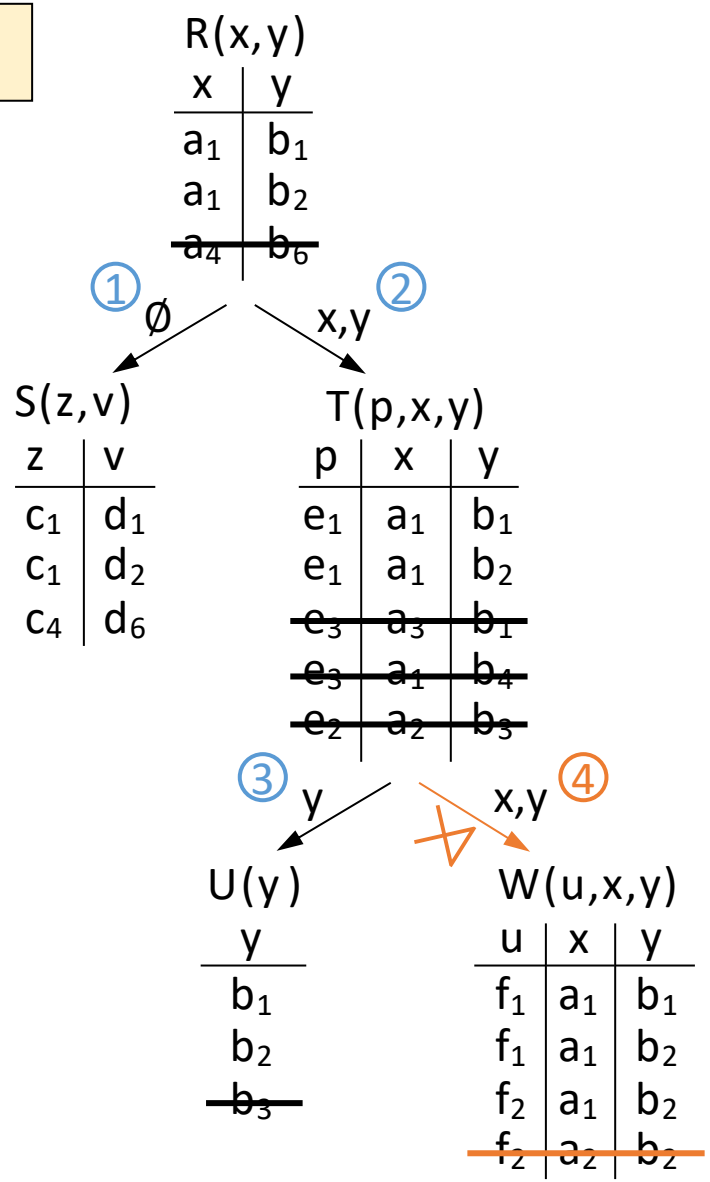


# Yannakakis Algorithm example: 2nd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋊ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**



# Yannakakis Algorithm example: 2nd pass over the data

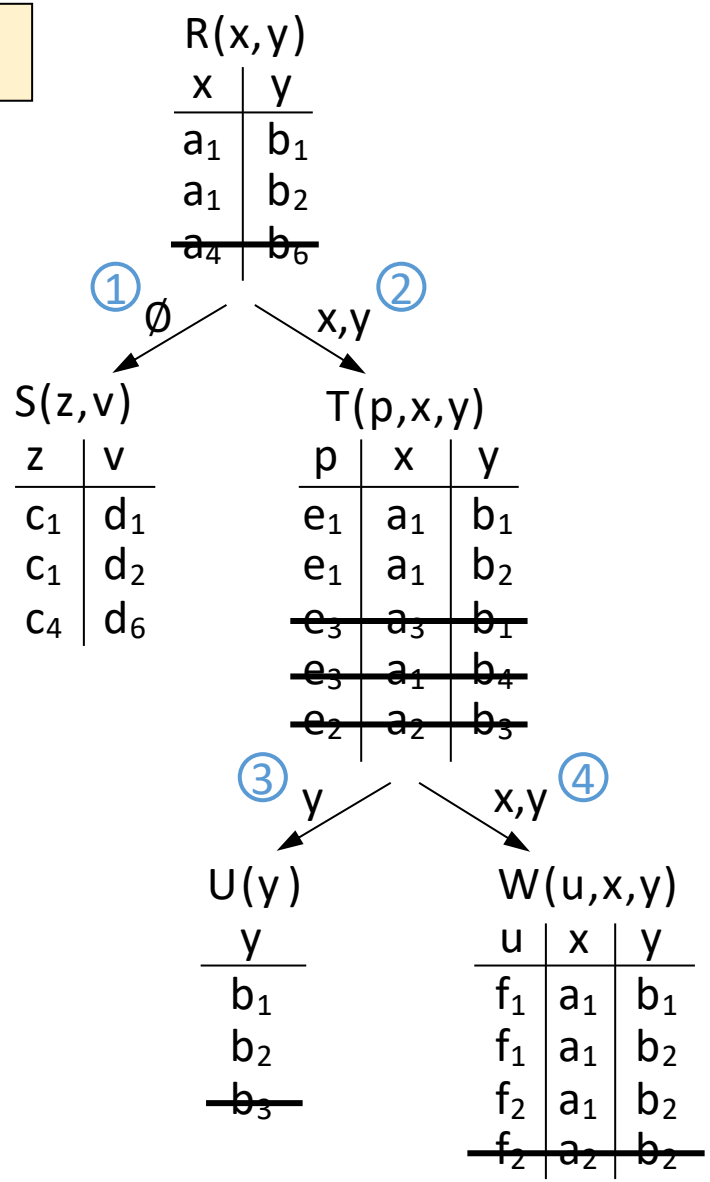
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

**1. Semi-join phase** ⋉ (remove dangling tuples) in *O*(n)

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- **Top-down semi-join propagation from root to leaves in some topological order**

Notice that at the end of the second pass, all tables are reduced; no table contains any more dangling tuples.

In other words, *every* table now "knows" whether the Boolean version of the query is true.



# Yannakakis Algorithm example: 3rd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

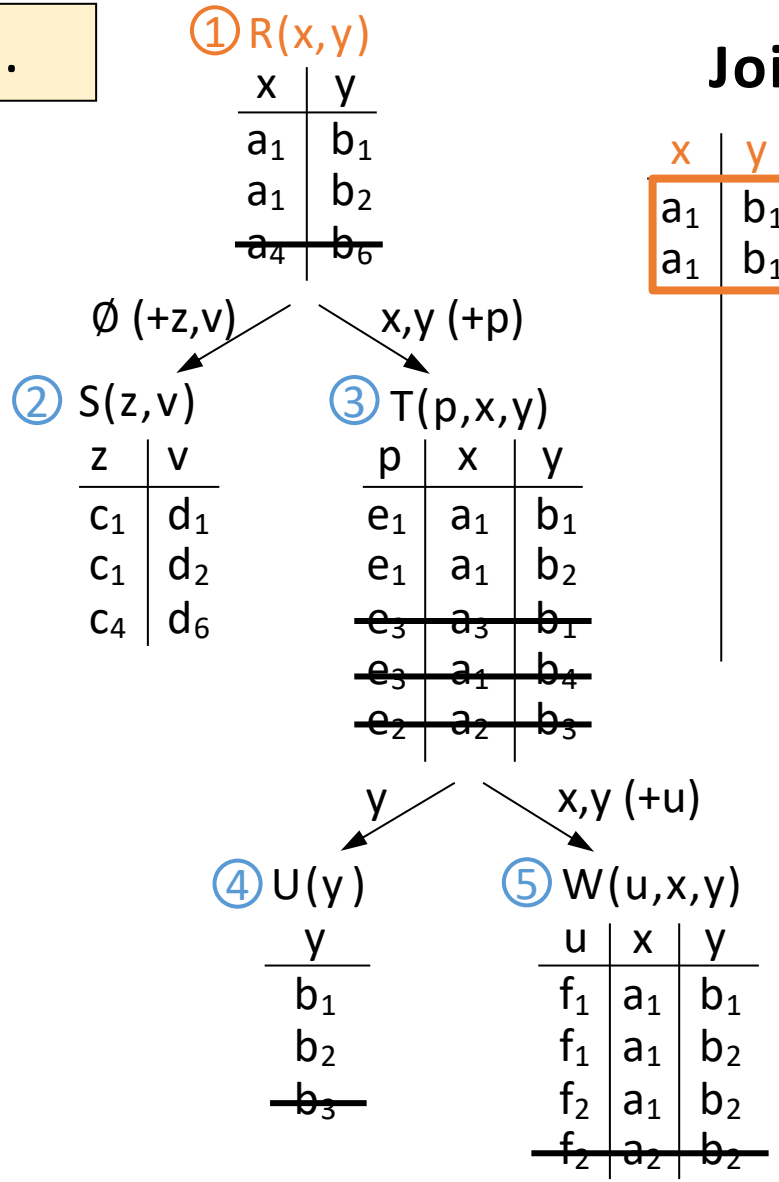
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$ ← Output

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Notice how with every join, the join result can never decrease in size!




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| a1 | b1 | | | | |
| a1 | b1 | | | | |

# Yannakakis Algorithm example: 3rd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

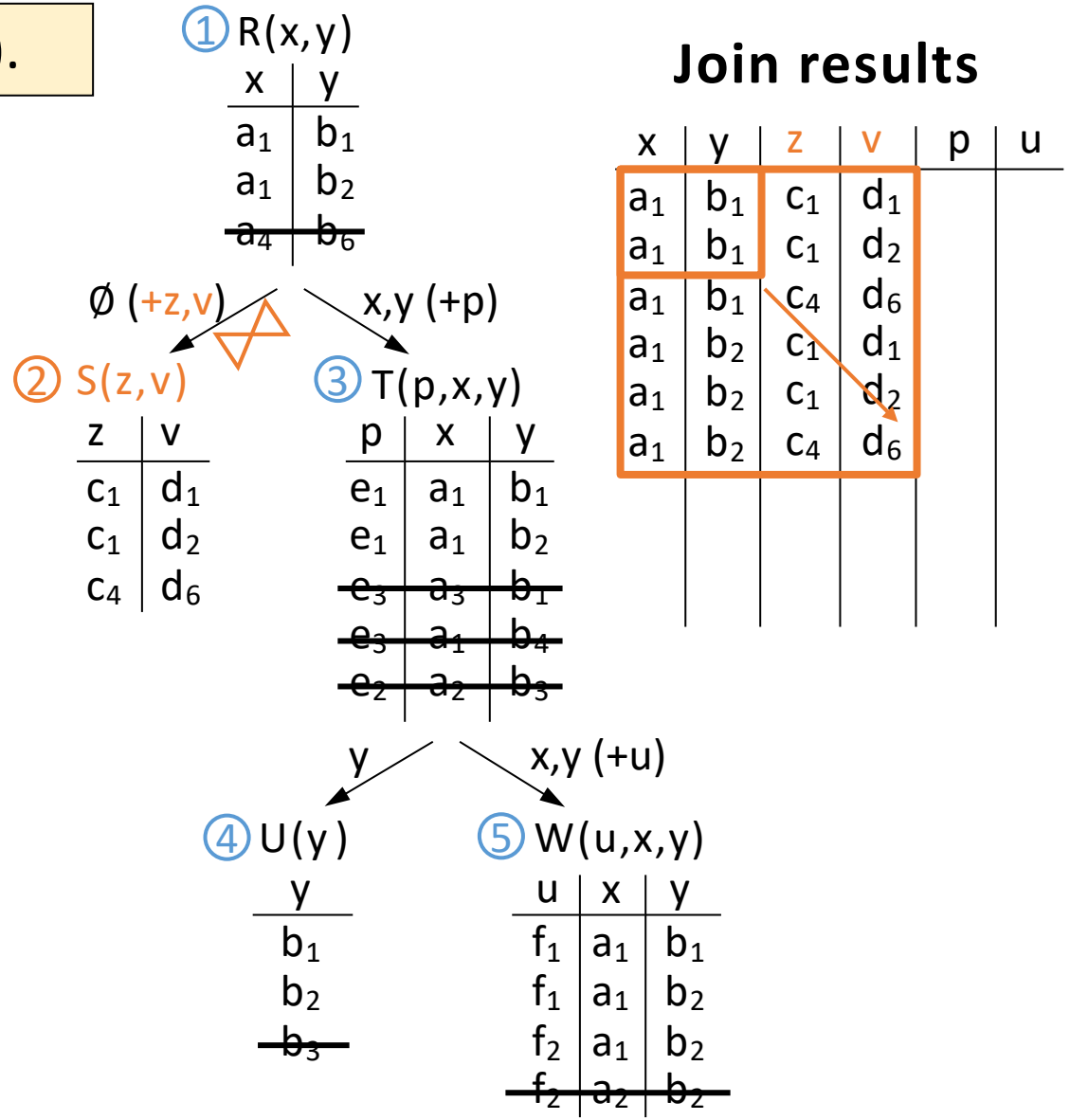
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$ ← Output

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Notice how with every join, the join result can never decrease in size!



# Yannakakis Algorithm example: 3rd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

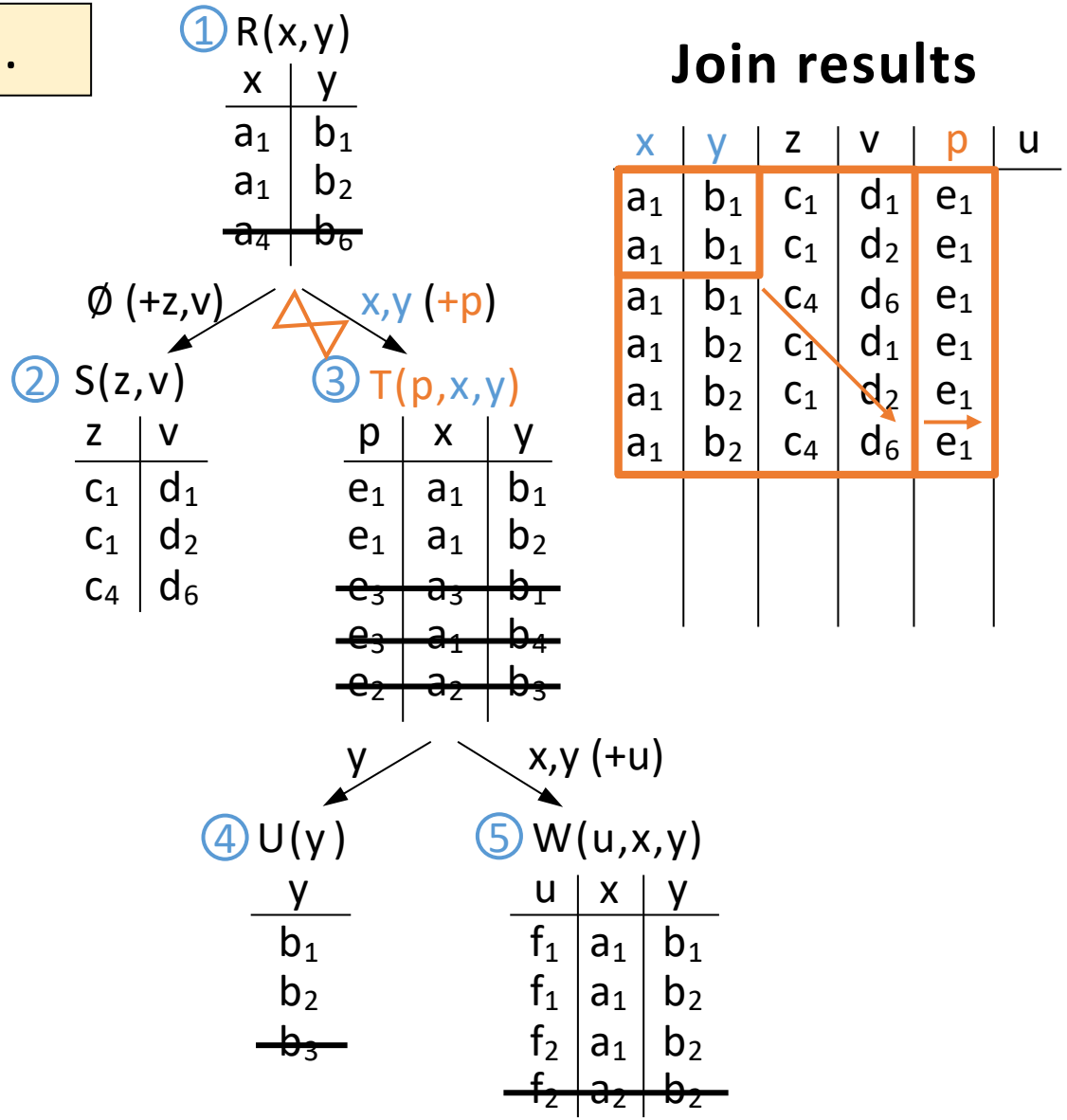
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Notice how with every join, the join result can never decrease in size!



# Yannakakis Algorithm example: 3rd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

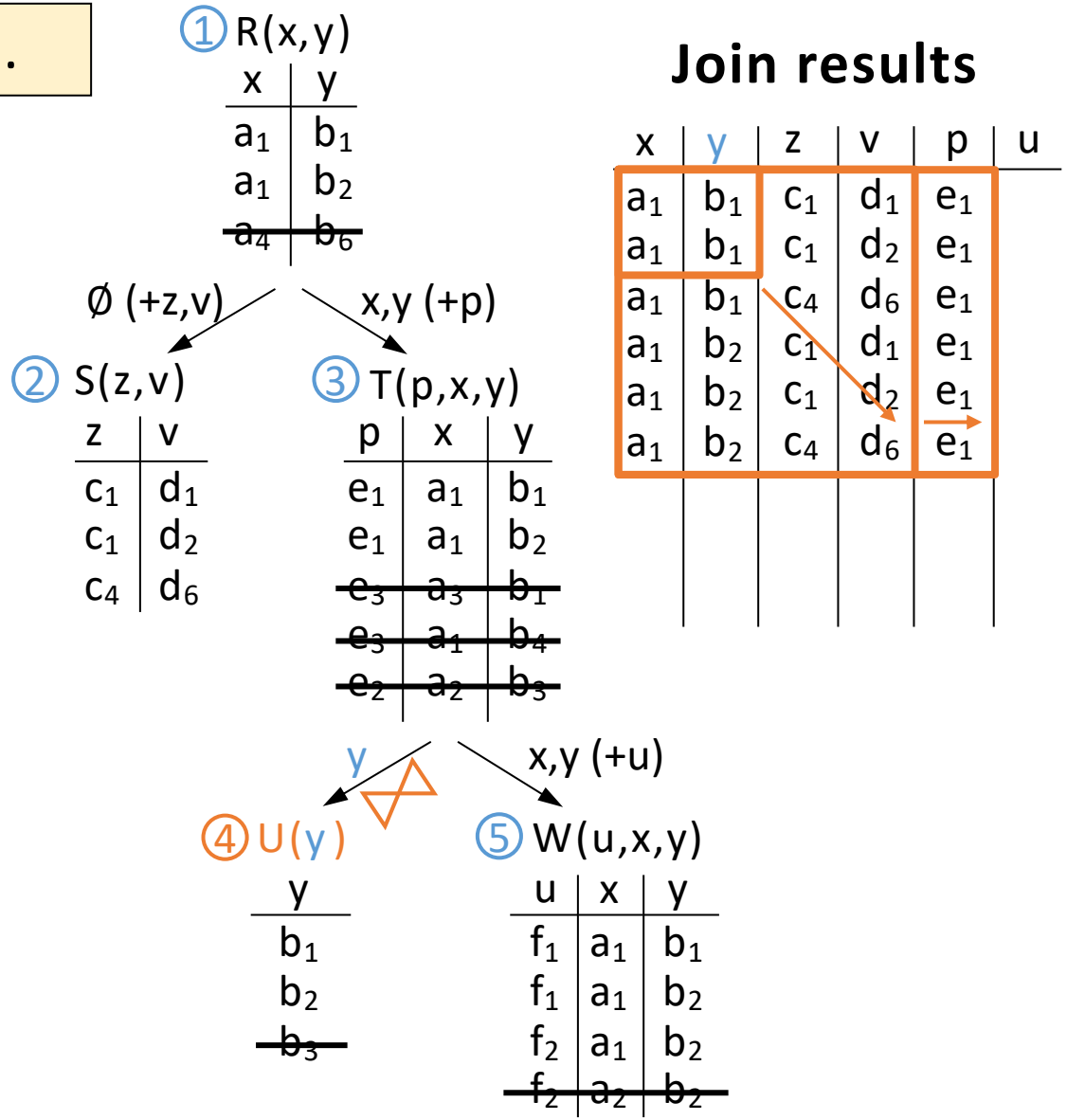
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Notice how with every join, the join result can never decrease in size!



# Yannakakis Algorithm example: 3rd pass over the data

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

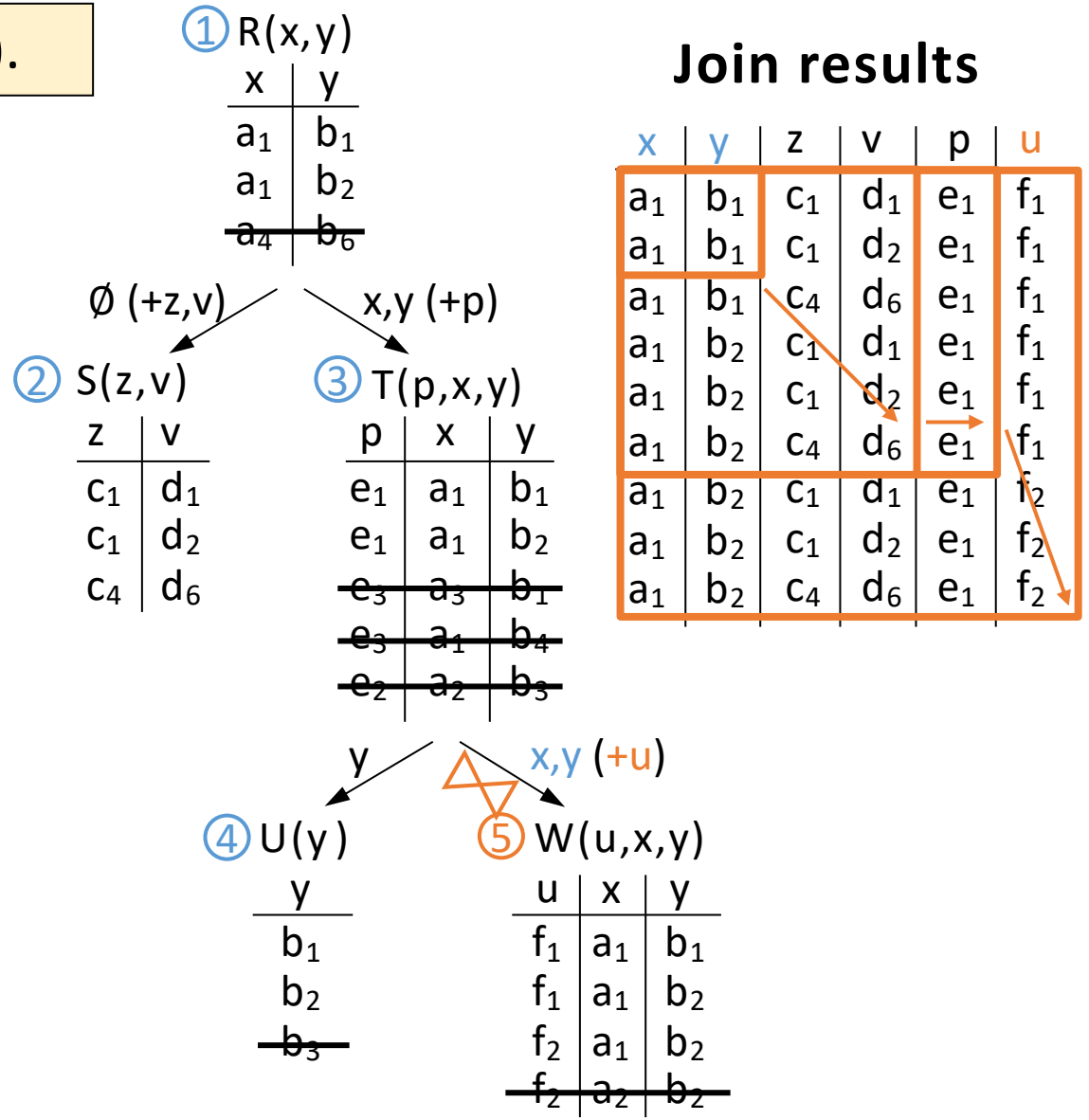
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Notice how with every join, the join result can never decrease in size!



# Yannakakis Algorithm example: summary



Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

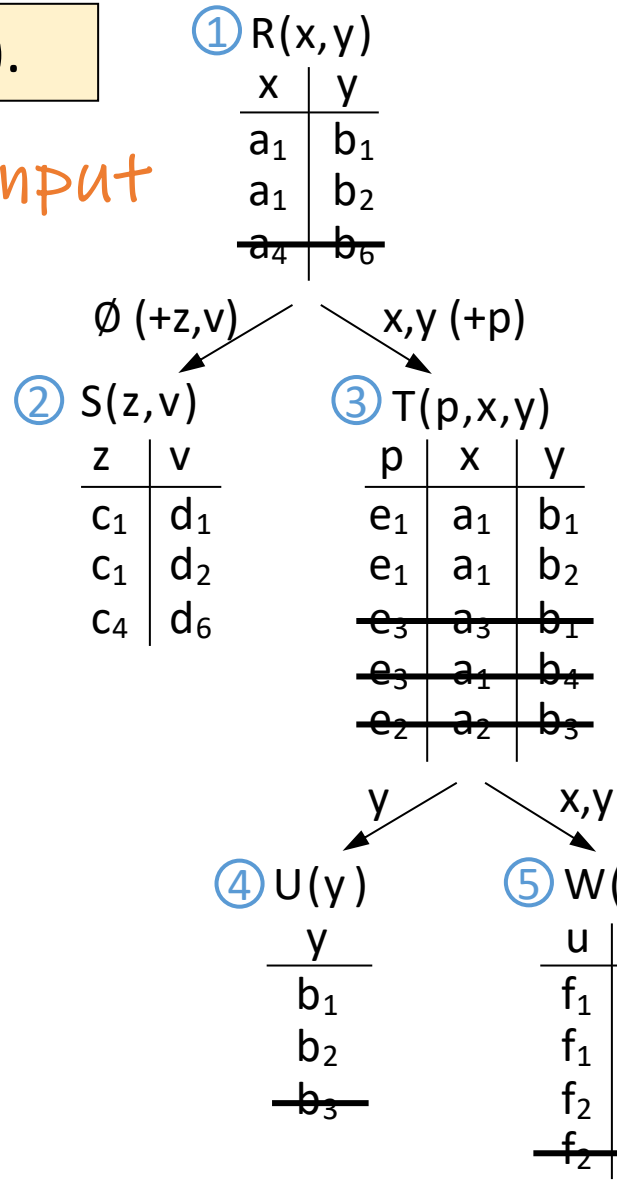
1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$ ← Input

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Join phase ⋈** (compute results) in $O(r)$ ← Output

- **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
  - This step can be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺

Deciding $Q(D)\neq\emptyset$ is in $O(n)$
Computing $Q(D)$ is in $O(n+r)$




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_2$ |

SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

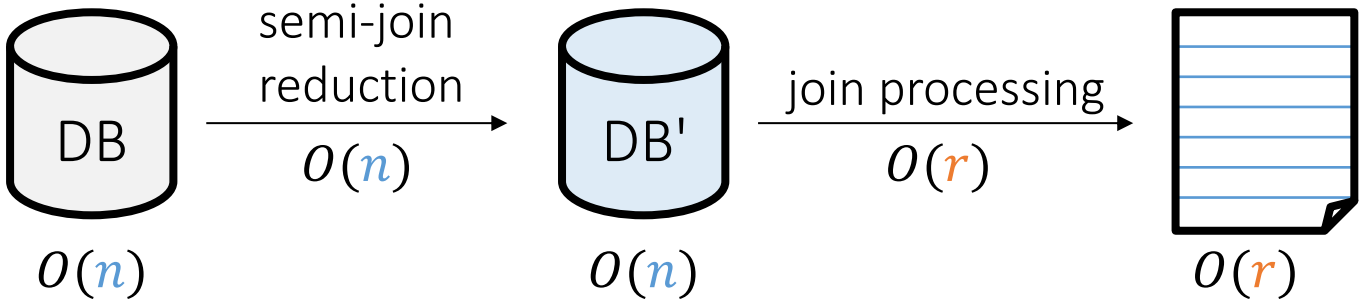
# Outline Part 3

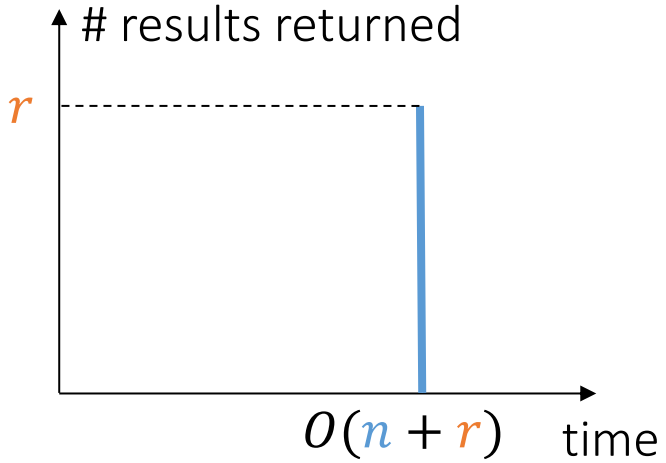
Part 3: Acyclic Queries & Enumeration (Wolfgang) ~25min

- The power of semi-join reductions
- Hypergraphs, GYO, join trees
- Yannakakis = acyclic query evaluation
- Enumerating answers
- Projections

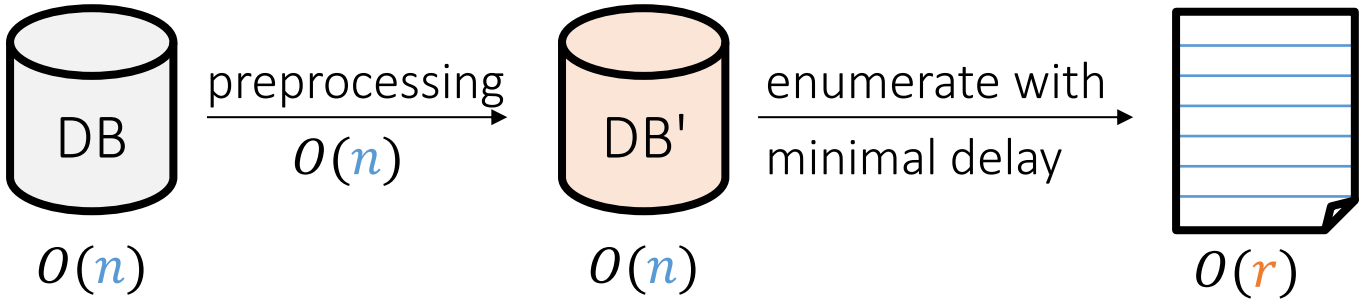
# The enumeration framework

## Standard Yannakakis framework for acyclic join processing

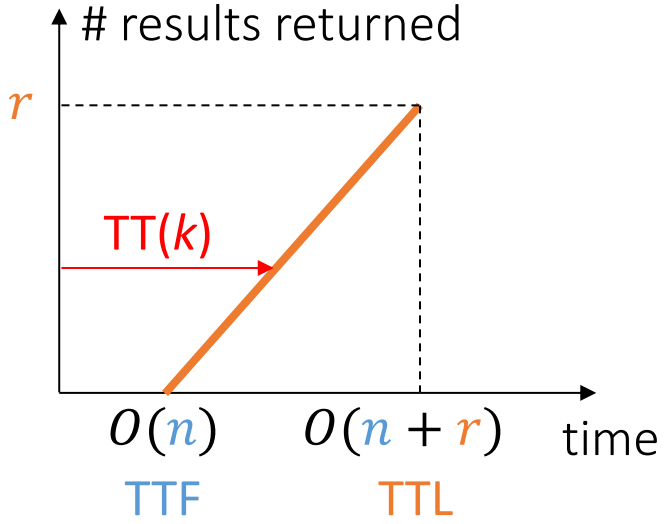



## Enumeration framework for acyclic queries



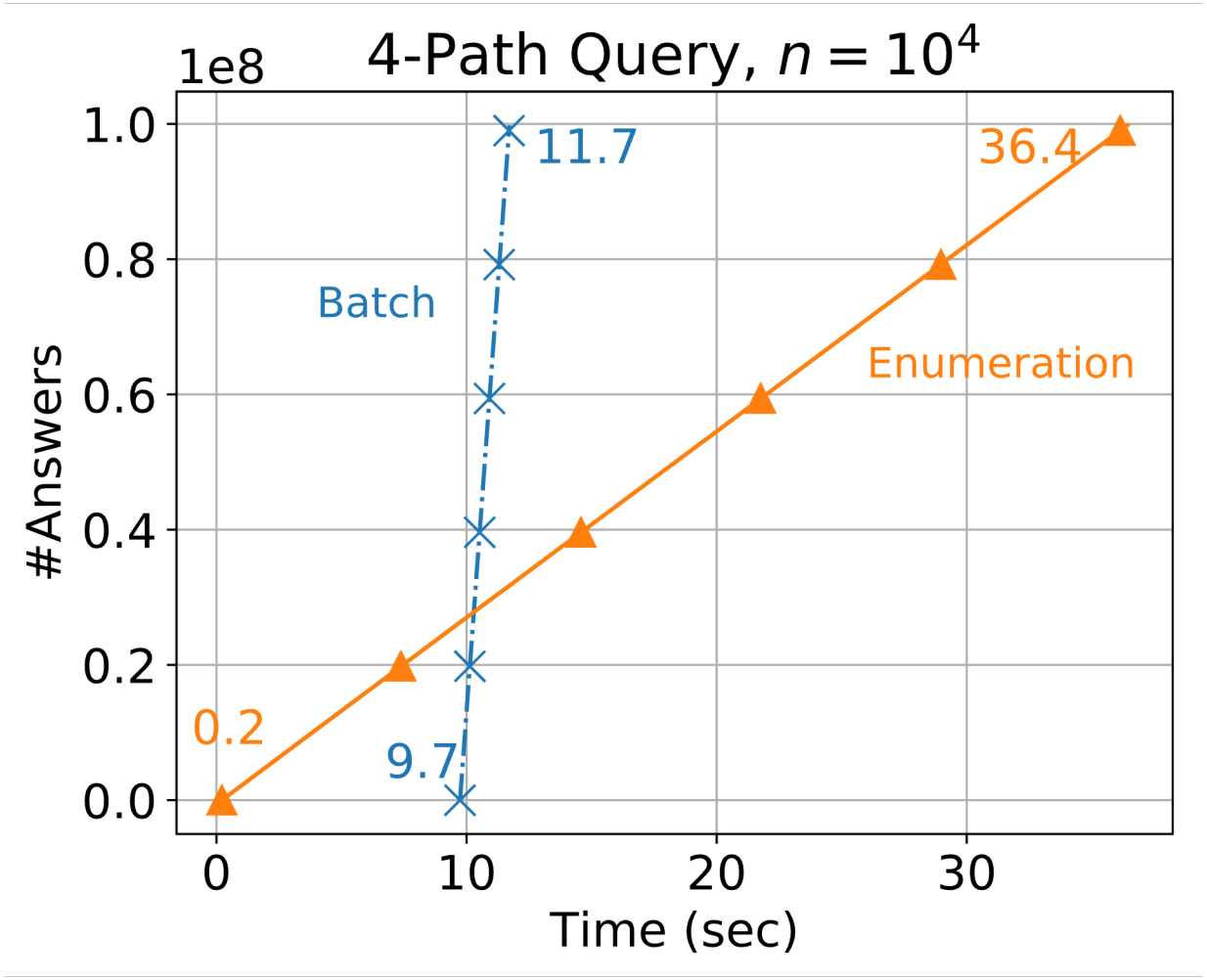

TTF (Time-To-First) TTL (Time-to-last)




$r = O(n^{\rho^*})$ worst-case result size (AGM bound). $\rho^*$ = fractional edge cover

Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Modified Yannakakis for output enumeration



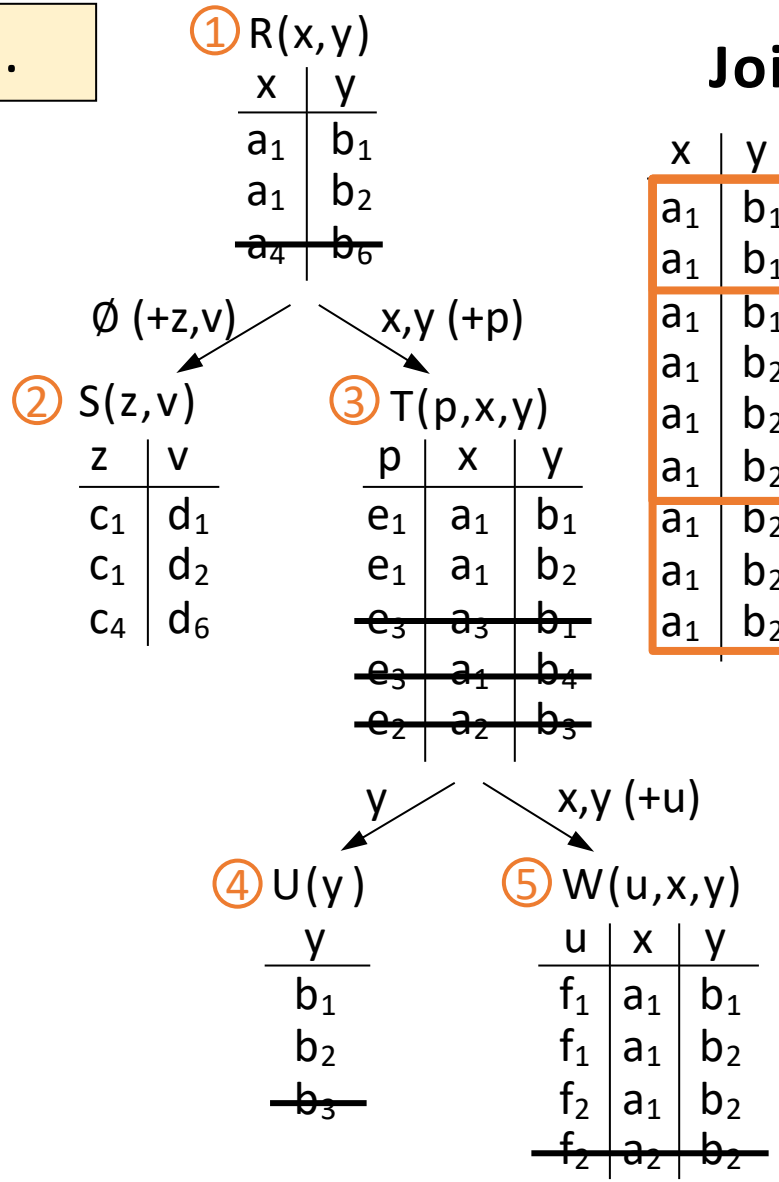

- Standard Yannakakis:
  - Table-at-a-time / Breadth-first
  - After the semi-join reduction, Yannakakis visits each table once top-down, and at each stage increases the size of the answer set
- Enumeration:
  - Tuple-at-a-time / Depth-first
  - By a slight modification of Yannakakis and tuple by tuple

Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Yannakakis Algorithm: example from before

Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. **Semi-join phase** ⋉ (remove dangling tuples) in $O(n)$
   - Bottom-up semi-join propagation from leaves to root in some reverse topological order
   - Top-down semi-join propagation from root to leaves in some topological order
2. **Join phase** ⋈ (compute results) in $O(r)$
   - **Compute the results in a 2nd top-down (or 2nd bottom-up) traversal:**
     - This step can actually be combined with the earlier top-down traversal; thus two total passes (first from leaves, then from root) are actually enough ☺




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_2$ |

# Modified Yannakakis Algorithm: enumeration

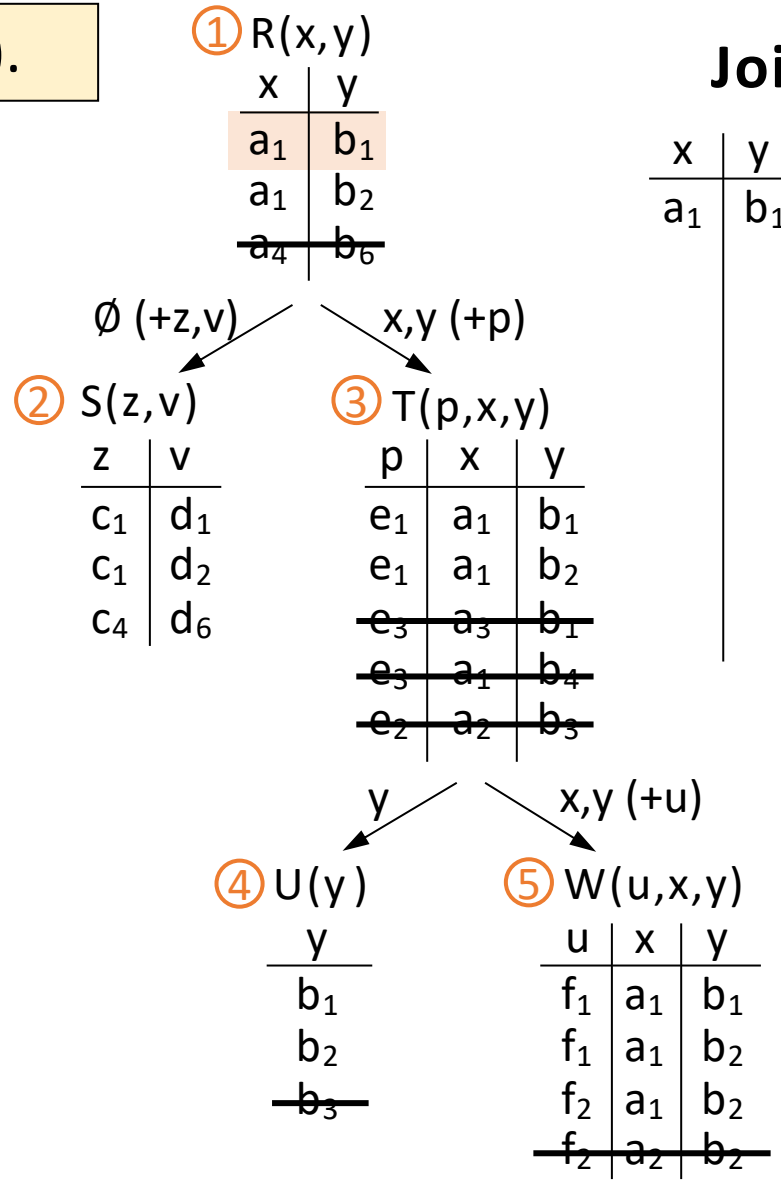
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

2. Enumeration phase ⋈ (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| a1 | b1 | | | | |

?

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

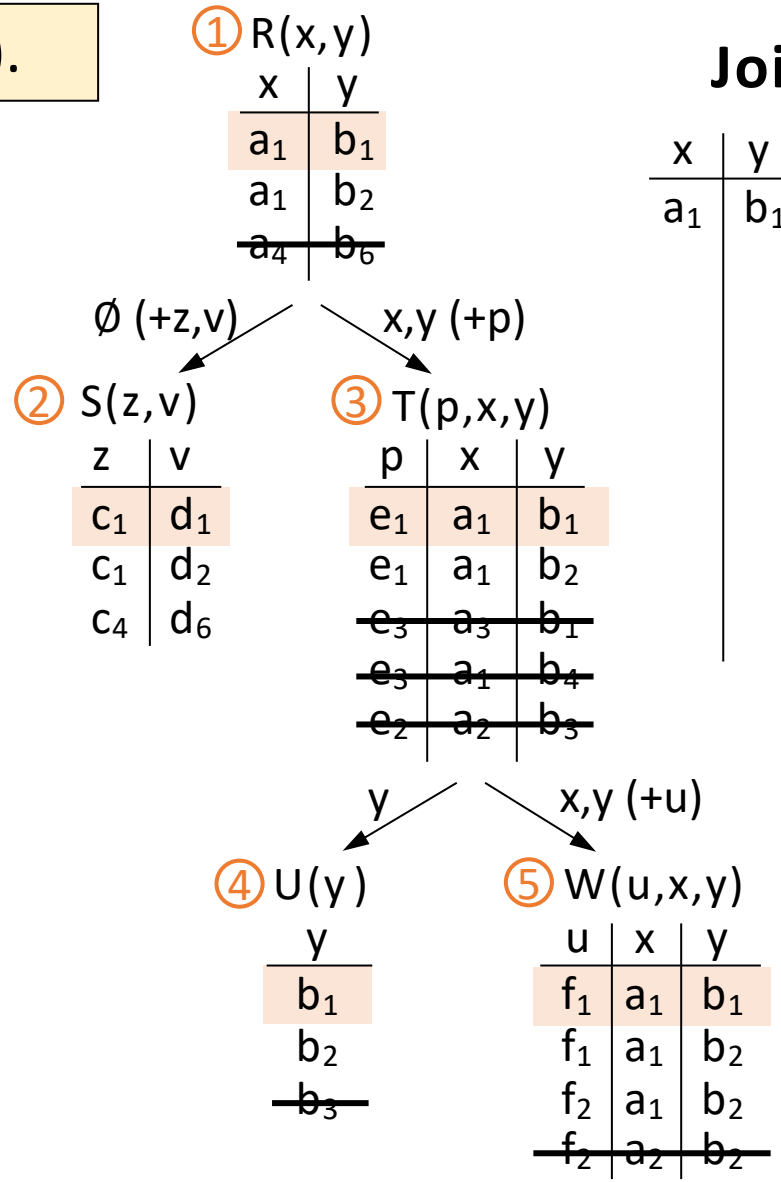
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

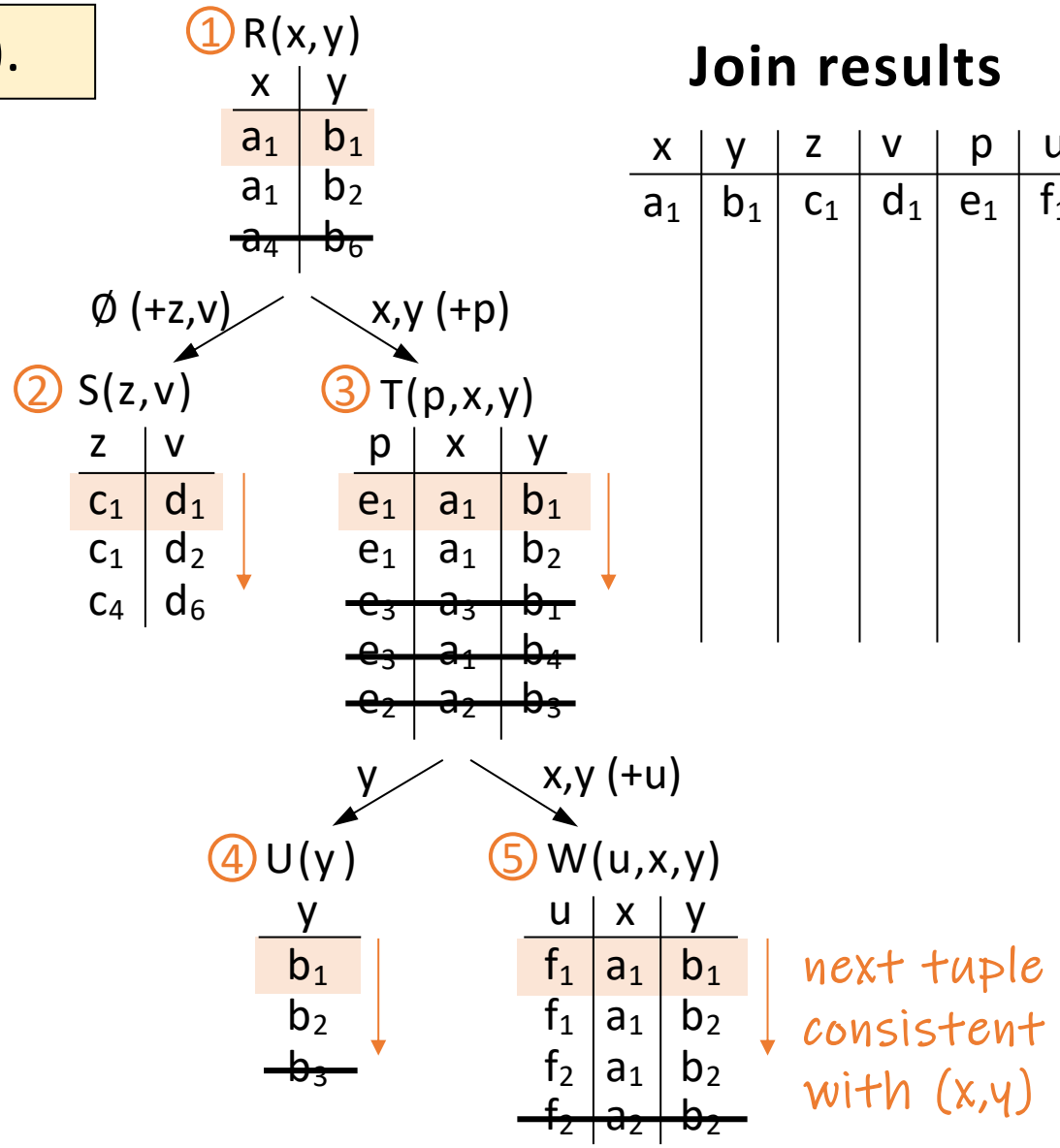
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

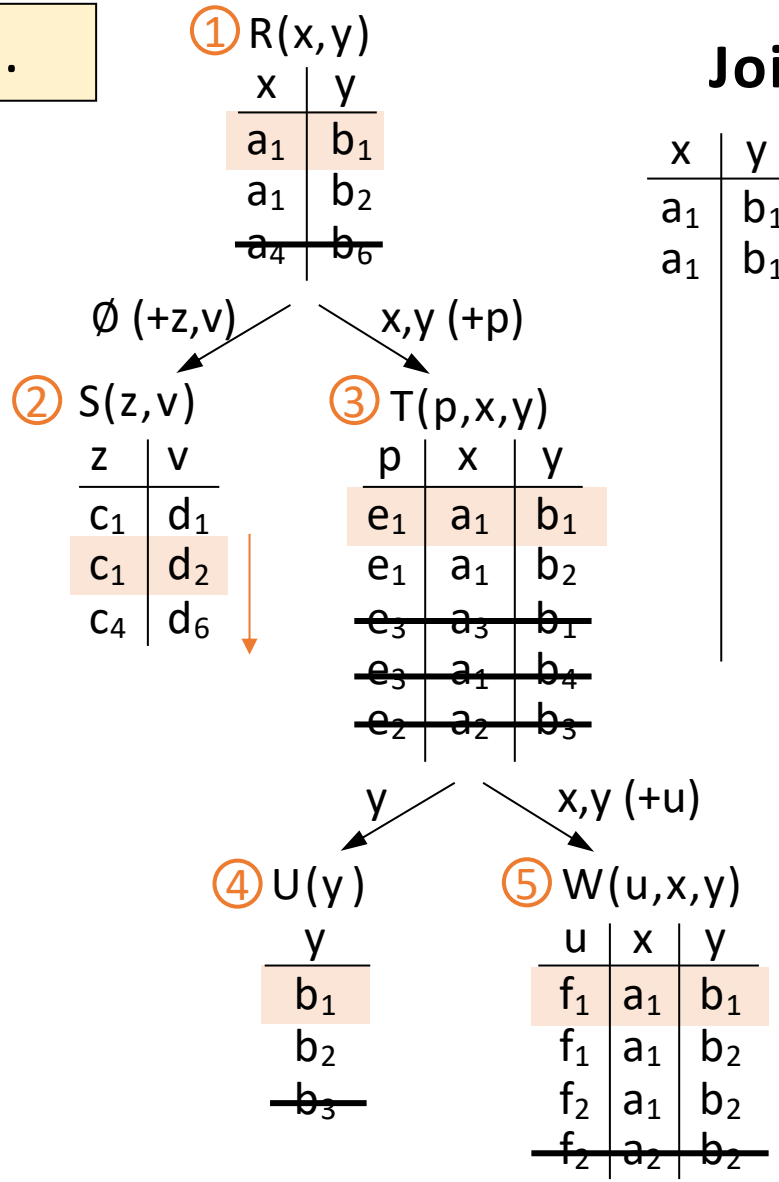
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

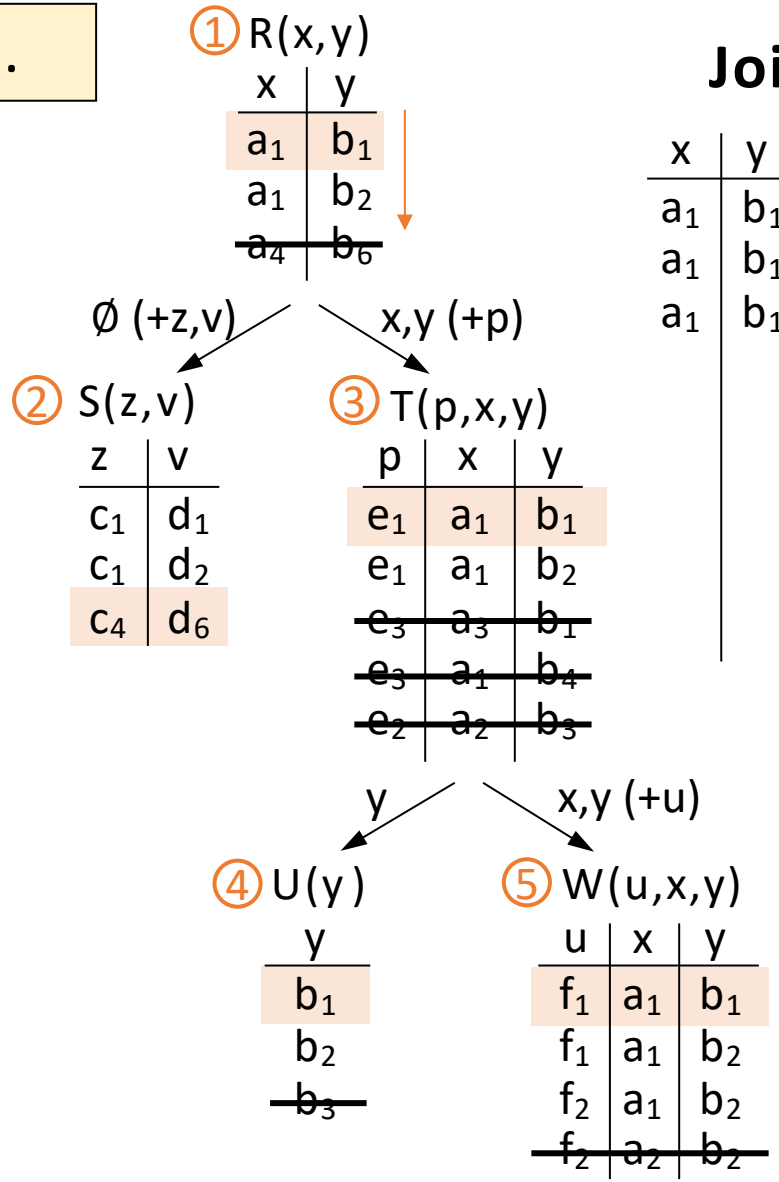
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

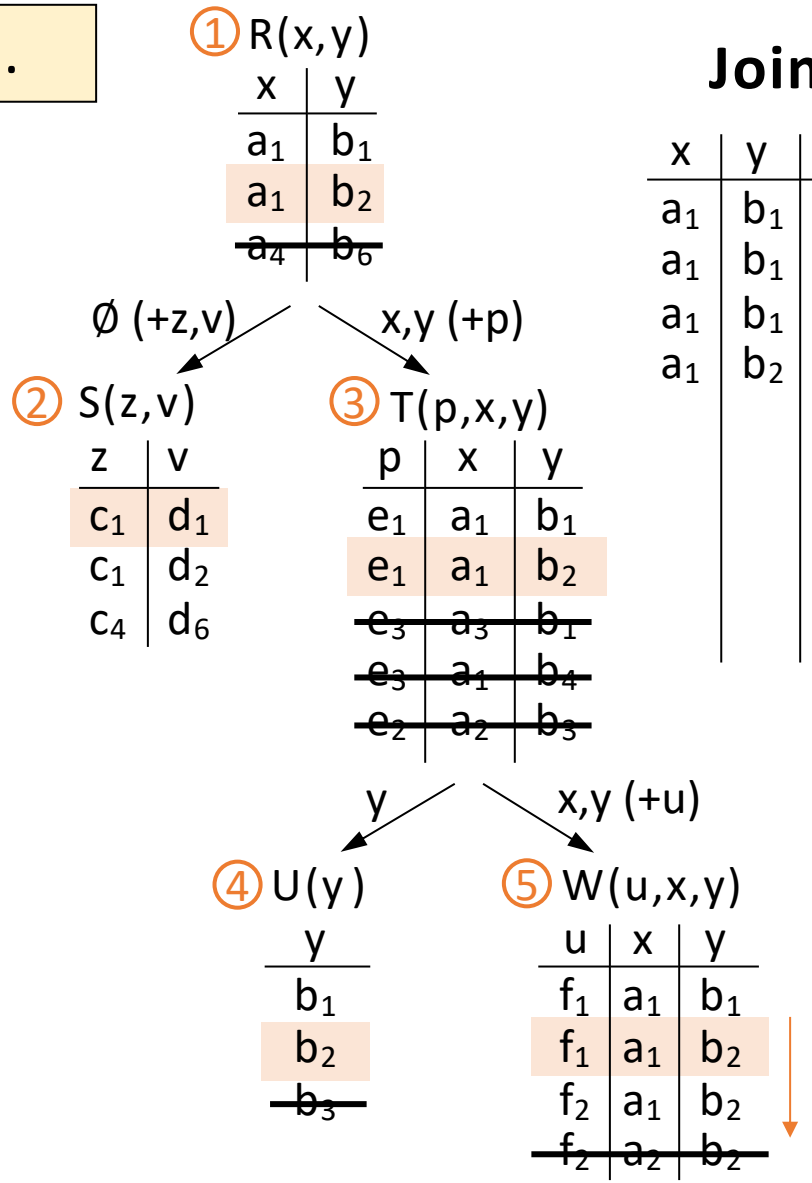
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

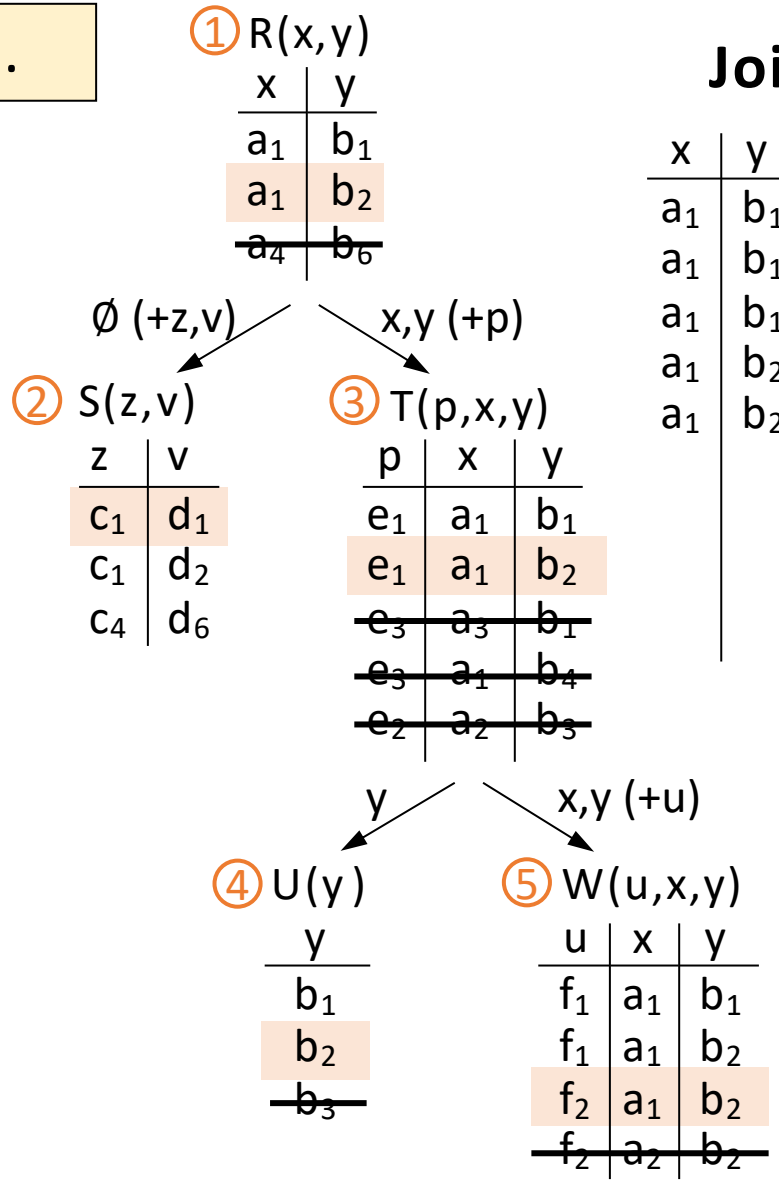
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

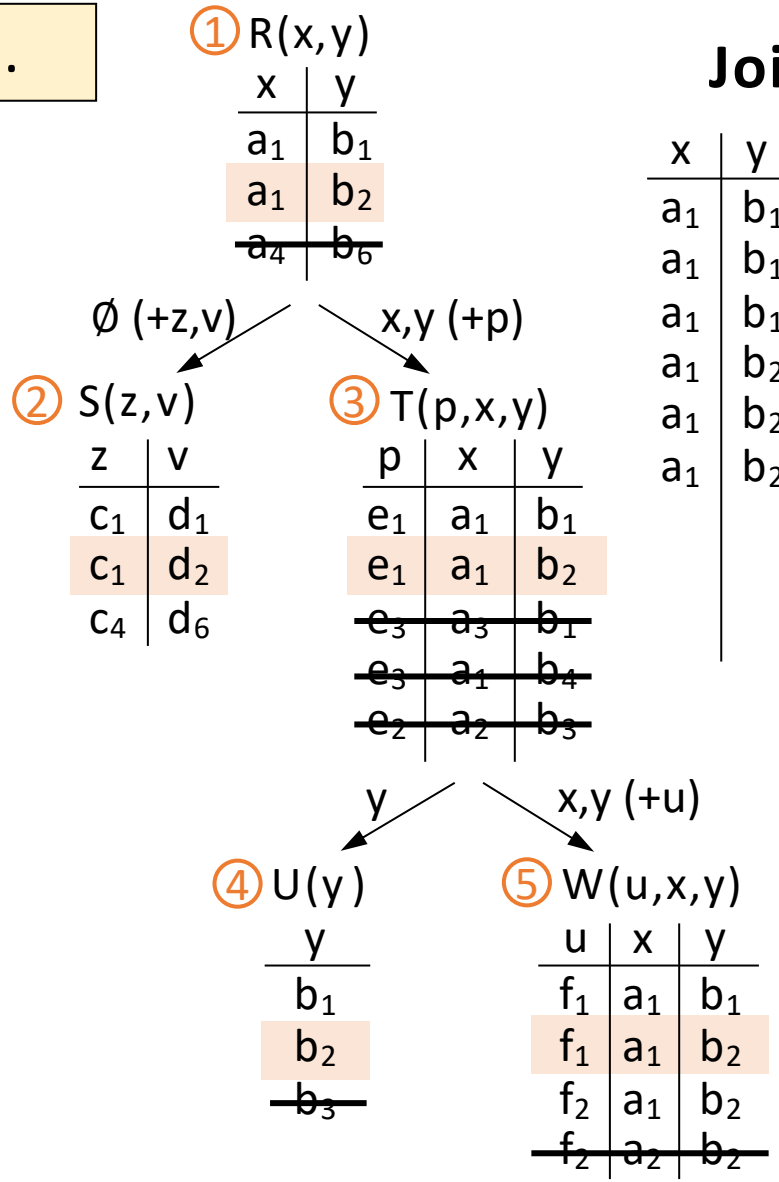
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

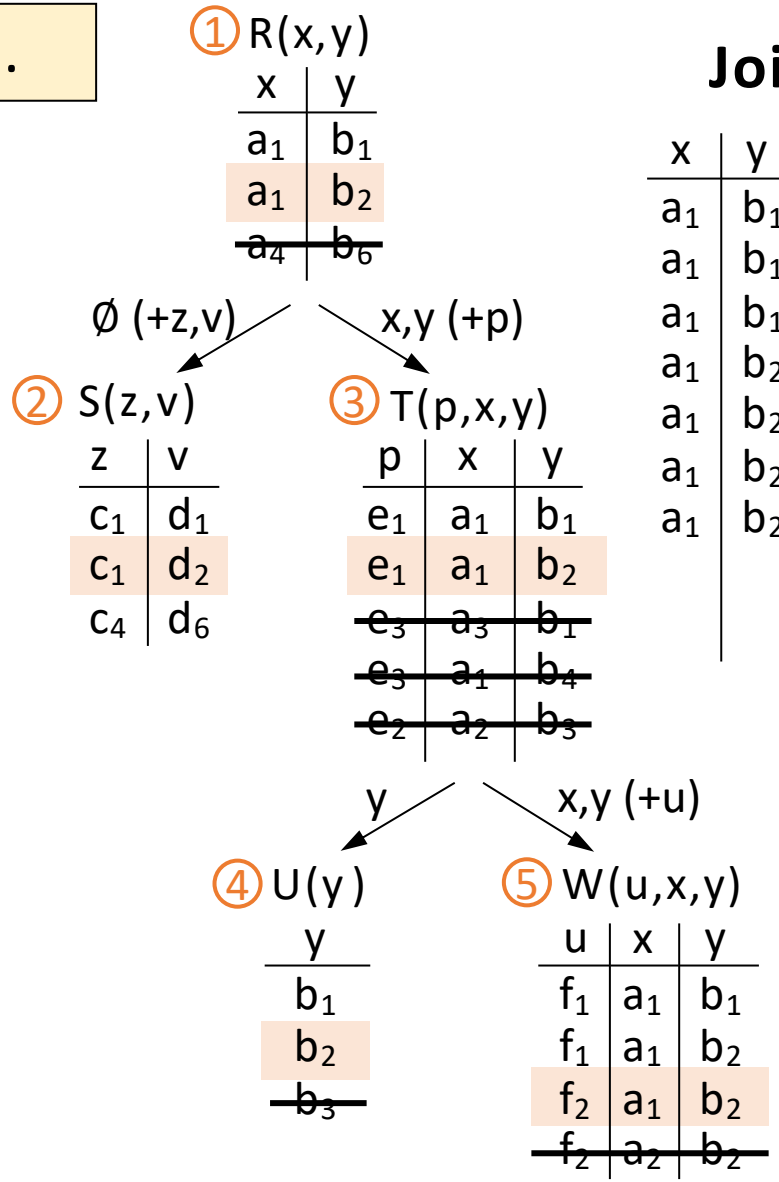
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_2$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

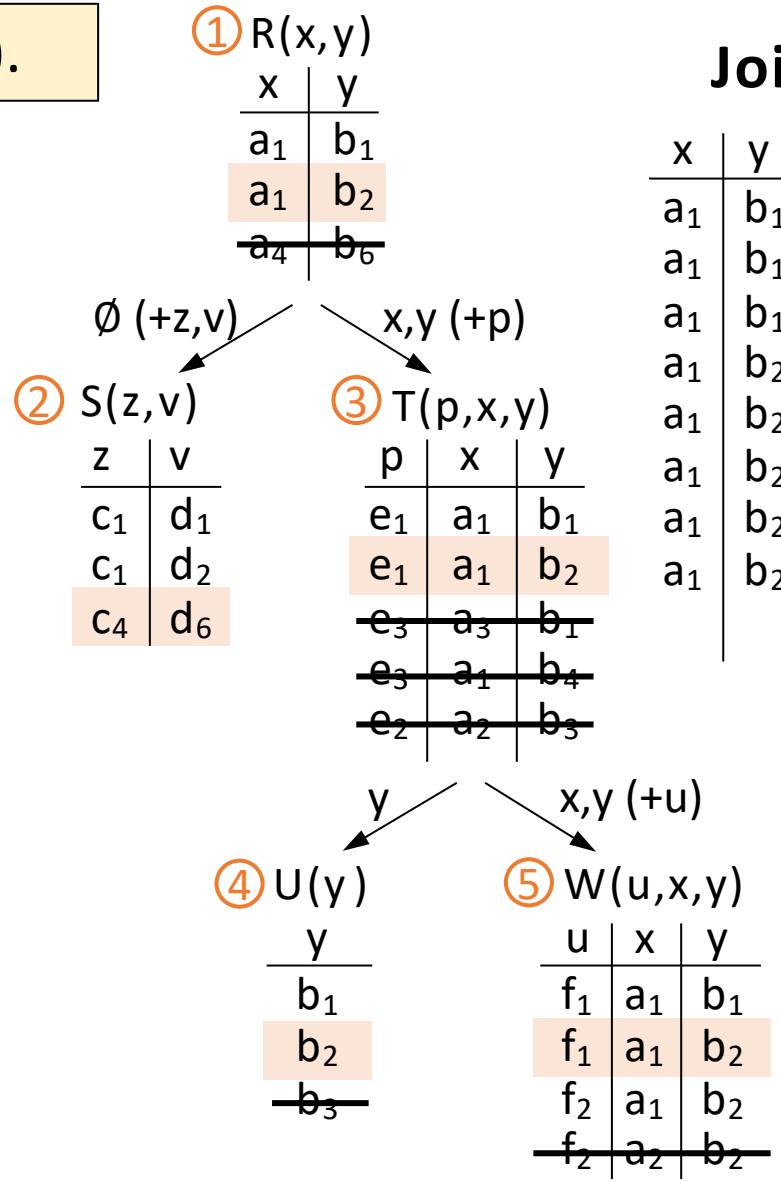
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}




**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis Algorithm: enumeration

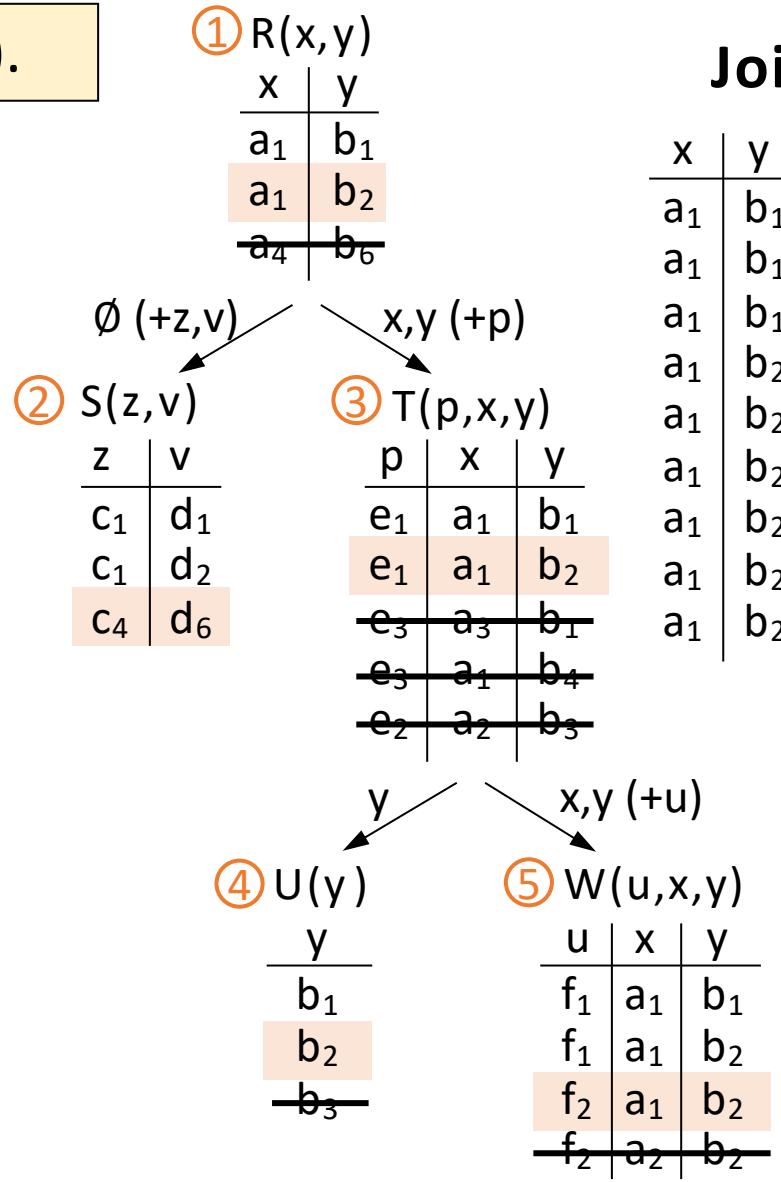
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase ⋉ (remove dangling tuples) in $O(n)$

- Bottom-up semi-join propagation from leaves to root in some reverse topological order
- Top-down semi-join propagation from root to leaves in some topological order

**2. Enumeration phase ⋈** (compute answers with $O(1)$ delay)

- **Compute one result after the other in lexicographic order of the variables (added with the tables ordered in some topological order)**: {x,y}+{z,v}+{p}+∅+{u}

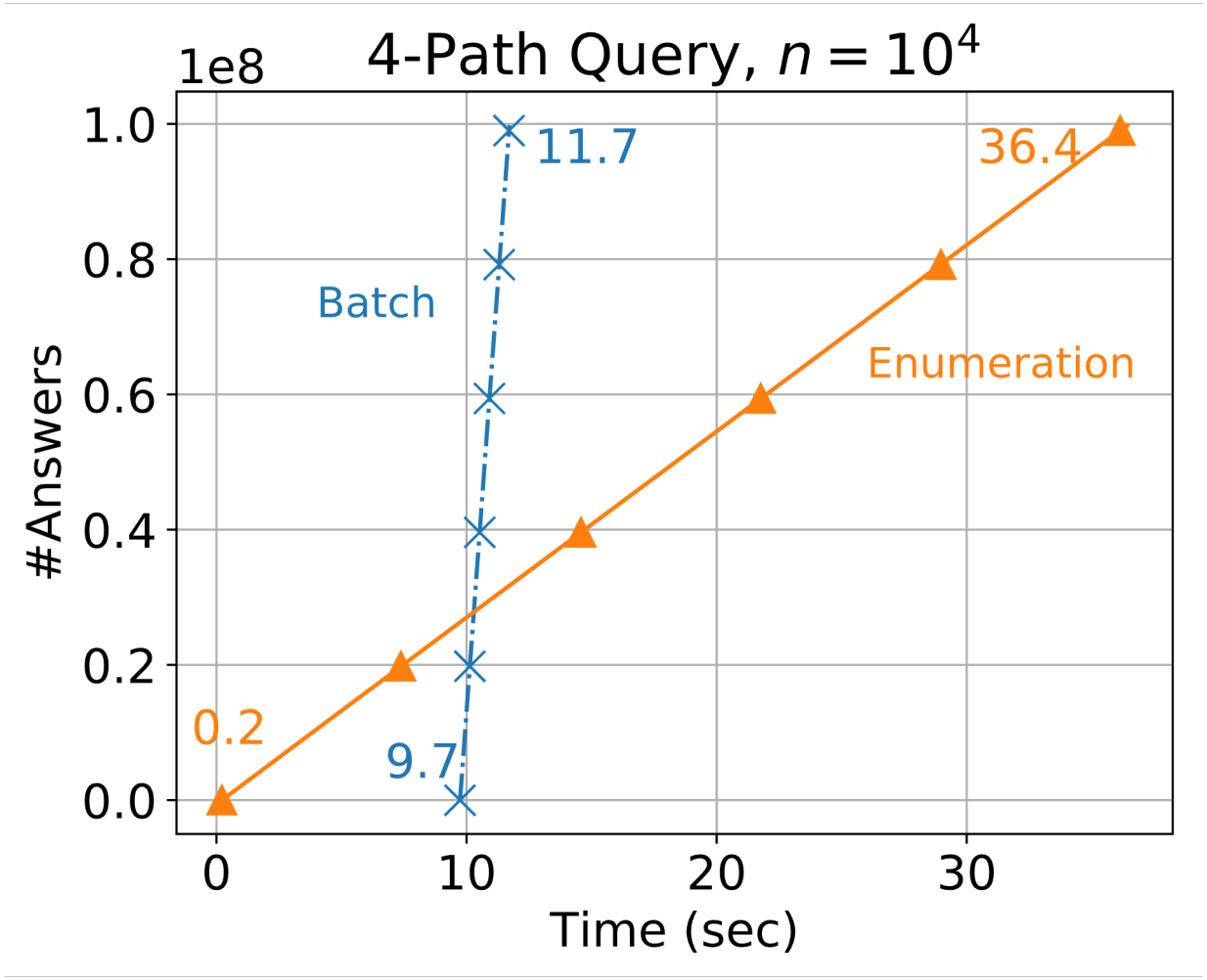



**Join results**

| x | y | z | v | p | u |
|---|---|---|---|---|---|
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_1$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_1$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_1$ | $d_2$ | $e_1$ | $f_2$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_1$ |
| $a_1$ | $b_2$ | $c_4$ | $d_6$ | $e_1$ | $f_2$ |

We start with some tuple in the root and extend it with consistent tuples from each table

# Modified Yannakakis for output enumeration




- Standard Yannakakis:
  - Table-at-a-time / Breadth-first
  - After the semi-join reduction, Yannakakis visits each table once top-down, and at each stage increases the size of the answer set
- Enumeration:
  - Tuple-at-a-time / Depth-first
  - By a slight modification of Yannakakis and tuple by tuple


Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

# Outline Part 3

Part 3: Acyclic Queries & Enumeration (Wolfgang) ~25min

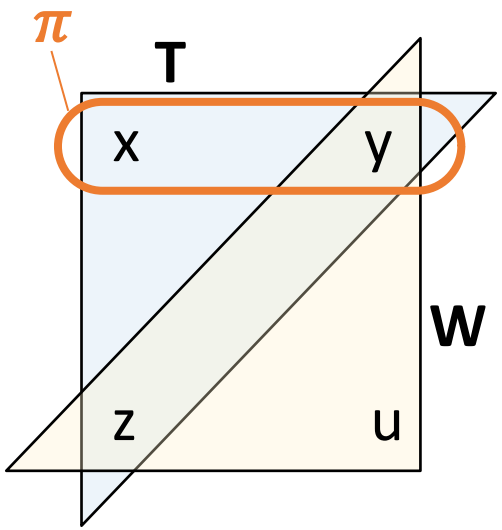
- The power of semi-join reductions
- Hypergraphs, GYO, join trees
- Yannakakis = acyclic query evaluation
- Enumerating answers
- Projections

# For acyclic queries, what changes for projections?

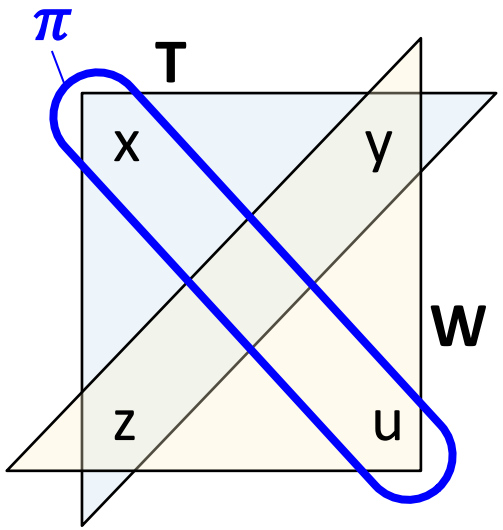
usually written as O(n·r) assuming output size r ≥1

- Yannakakis works in O(n+n·r) for arbitrary projections
  - Then Enumeration works with linear delay and TTL is also O(n+n·r)
- Yannakakis works in O(n+r) for queries that are "free-connex"
  - Then Enumeration works with constant delay and TTL is also O(n+r)

Free-connex: the query remains acyclic after adding an edge for the set of projected variables



free-connex ☺

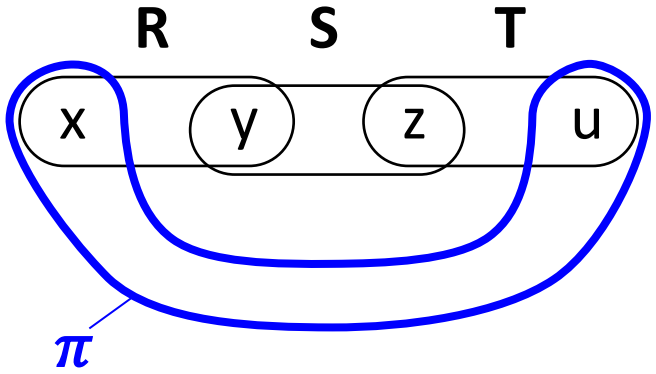


not free-connex ☹


Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18
See recent ICDT'22 tutorial by Nofar Carmeli. Answering Unions of Conjunctive Queries with Ideal Time Guarantees: https://nofar.carme.li/resources/ICDT22TutorialSlides.pdf
Towards Responsive DBMS. ICDE 2022 tutorial: https://northeastern-datalab.github.io/responsive-dbms-tutorial

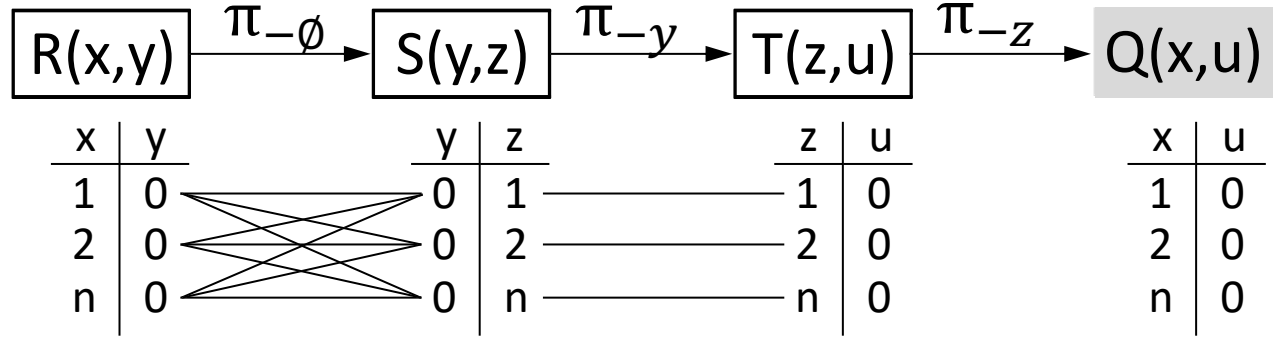
# For acyclic queries, what changes for projections?

Q(x,u) :- R(x,y), S(y,z), T(z,u).




not free-connex ☹

**Join phase:**




| x | y |
|---|---|
| 1 | 0 |
| 2 | 0 |
| n | 0 |

| y | z |
|---|---|
| 0 | 1 |
| 0 | 2 |
| 0 | n |

| z | u |
|---|---|
| 1 | 0 |
| 2 | 0 |
| n | 0 |

| x | u |
|---|---|
| 1 | 0 |
| 2 | 0 |
| n | 0 |

R(x,y)⋈S(y,z)

| x | y | z |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 0 | 1 |
| n | 0 | 1 |
| … | 0 | … |
| n | 0 | 3 |

$O(n^2)$

Output size is $r = O(n)$

Join phase takes $O(n \cdot r) = O(n^2)$

# Summary Acyclic queries

1. Acylic queries imply a join tree can be found by GYO
2. Semi-join reductions work on trees. Can be extremely powerful and get reinvented over and over again
3. Yannakakis works on acyclic queries and gives an optimal time guaranteed of $O(n+r)=O(|INPUT|+|Output|)$ for full CQs
   - $O(n+n \cdot r)$ for arbitrary projections
4. Enumeration slightly modifies Yannakakis to start returning results earlier (depth-first instead of breadth-first)

# Outline tutorial

1: Introduction (Nikos) ~40min
2: Tree Decompositions (Mirek) ~20min
3: Acyclic Queries & Enumeration (Wolfgang) ~25min

BREAK

4: Factorization (Nikos) ~10min
5: Dynamic Programming & Semirings (Wolfgang) ~20min
6: Any-$k$ or Ranked Enumeration (Nikos) ~35min
7. Decomposition of Comparison Predicates (Mirek) ~10min
8. Conclusion (Mirek) ~10min