Updated 3/21/2023

Topic 3: Efficient query evaluation
Unit 1: Acyclic query evaluation
Lecture 19

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp23)

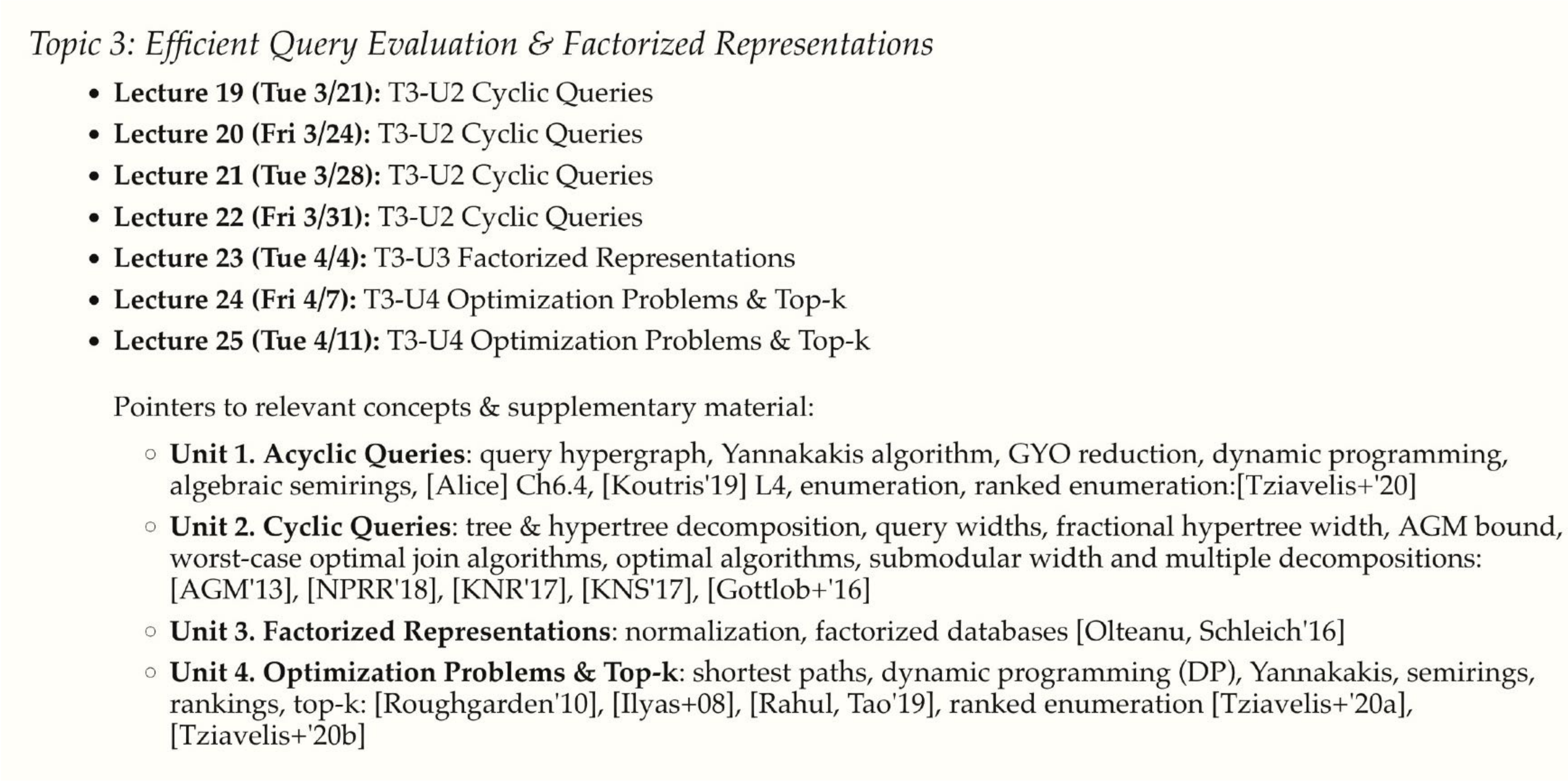
https://northeastern-datalab.github.io/cs7240/sp23/

3/21/2023

Pre-class conversations

- Last class summary
- Project: we can iterate, just approach me (P3: TUE, 3/28)
- Scribes: we are past halftime of the class (0-5 / 7)
  - please see my detailed comments, do also approach me after class with comments / questions / pointers.

- Today:
  - the semi-join reduction as basis for efficient algorithms



Topic 3: Efficient Query Evaluation & Factorized Representations

- **Lecture 19 (Tue 3/21):** T3-U2 Cyclic Queries
- **Lecture 20 (Fri 3/24):** T3-U2 Cyclic Queries
- **Lecture 21 (Tue 3/28):** T3-U2 Cyclic Queries
- **Lecture 22 (Fri 3/31):** T3-U2 Cyclic Queries
- **Lecture 23 (Tue 4/4):** T3-U3 Factorized Representations
- **Lecture 24 (Fri 4/7):** T3-U4 Optimization Problems & Top-k
- **Lecture 25 (Tue 4/11):** T3-U4 Optimization Problems & Top-k

Pointers to relevant concepts & supplementary material:

- ○ **Unit 1. Acyclic Queries**: query hypergraph, Yannakakis algorithm, GYO reduction, dynamic programming, algebraic semirings, [Alice] Ch6.4, [Koutris'19] L4, enumeration, ranked enumeration:[Tziavelis+'20]
- ○ **Unit 2. Cyclic Queries**: tree & hypertree decomposition, query widths, fractional hypertree width, AGM bound, worst-case optimal join algorithms, optimal algorithms, submodular width and multiple decompositions: [AGM'13], [NPRR'18], [KNR'17], [KNS'17], [Gottlob+'16]
- ○ **Unit 3. Factorized Representations**: normalization, factorized databases [Olteanu, Schleich'16]
- ○ **Unit 4. Optimization Problems & Top-k**: shortest paths, dynamic programming (DP), Yannakakis, semirings, rankings, top-k: [Roughgarden'10], [Ilyas+08], [Rahul, Tao'19], ranked enumeration [Tziavelis+'20a], [Tziavelis+'20b]

Islands of Tractability of CQ Evaluation

- Major Research Program: Identify <u>tractable cases</u> of the combined complexity of conjunctive query evaluation.
- Over the years, this program has been pursued by two different research communities:
  - The Database Theory community
  - The Constraint Satisfaction community

$$\begin{tikzcd}
& G \arrow[dl, "h", "\text{CSP}"'] & \\
q_1 \arrow[rr, "h", "q_1 \Leftarrow q_2"'] & & q_2 \arrow[ul, "h", "\text{Query eval.}\ G \vDash q_2"']
\end{tikzcd}$$

- Explanation: Problems in those community are closely related:

Constraint Satisfaction Problem ≡ Homomorphism Problem ≡ CQ evaluation

[Feder, Vardi 1993]　　[Chandra, Merlin 1977]

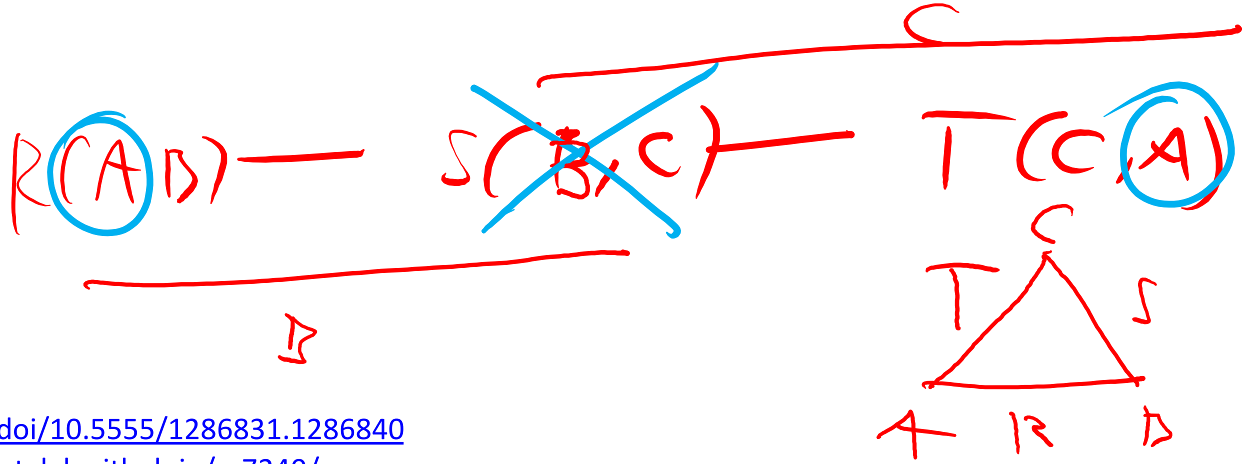
[Kolaitis, Vardi 2000]

Feder, Vardi: Monotone monadic SNP and constraint satisfaction, STOC 1993 https://doi.org/10.1145/167088.167245 / Kolaitis, Vardi: Conjunctive-Query Containment and Constraint Satisfaction, JCSS 2000 https://doi.org/10.1006/jcss.2000.1713 / Chandra, Merlin. "Optimal implementation of conjunctive queries in relational data bases", STOC 1977. https://doi.org/10.1145/800105.803397

Based on Phokion Kolaitis' "Logic and Databases" series at Simons Institute, 2016. https://simons.berkeley.edu/talks/logic-and-databases

Acyclic queries

- Like many areas in computer science, cycles complicate things
  - Same with conjunctive query (CQ) evaluation
- Acyclic CQs are a large and useful tractable case CQs
- A query is acyclic if its relations can be placed in a tree (join tree) s.t.
  - the set of nodes that contain any variable form a connected set
- Yannakakis' algorithm [81]: any acyclic query can be computed in time: O(|Input| + |Output|)



Yannakakis, "Algorithms for acyclic database schemes", VLDB 1981. https://dl.acm.org/doi/10.5555/1286831.1286840

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

Semijoin (⋉): derived RA operator

Actor (aid, name, gender)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$\boxed{R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)}$$ RA

where $A_1, \ldots, A_n$ are the attributes in R

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: ?

Semijoin (⋉): derived RA operator

Actor (aid, name, gender)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$\boxed{R \ltimes S = \pi_{A_1,\dots,A_n}(R \bowtie S)}$$ RA

where $A_1, \dots, A_n$ are the attributes in R

SQL: ?

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: $\text{Actor} \ltimes \text{Play}$

Semijoin (⋉): derived RA operator

Actor (aid, name, age)
Play (aid, mid, role)
Movie(mid, name, year)

- Find tuples in **R** for which there is a matching tuple in **S** that is equal on their common attribute names.

$$\boxed{R \ltimes S = \pi_{A_1,\ldots,A_n}(R \bowtie S)}$$ RA

where $A_1, \ldots, A_n$ are the attributes in R

Intuition: remove "dangling tuples" in R

SQL: Semijoins have no "direct" representation in SQL (just like relational division)

Alternative WHERE EXISTS

```
SELECT DISTINCT *
FROM Actor
WHERE aid IN
      (SELECT aid
       FROM Play)
```

Duplicates in R are preserved!

- Example:

"Find actors who play some role."

RA: $\text{Actor} \ltimes \text{Play}$

Equivalent only if no duplicates in R!

```
SELECT DISTINCT
      A.aid, name, age
FROM Actor A, Play P
WHERE A.aid = P.aid
```

Semijoins in Distributed Databases

- Semijoins are often used to compute equijoins in distributed databases

$R \bowtie S$

Goal: send less data to reduce network bandwidth!

$R(X,Y)$

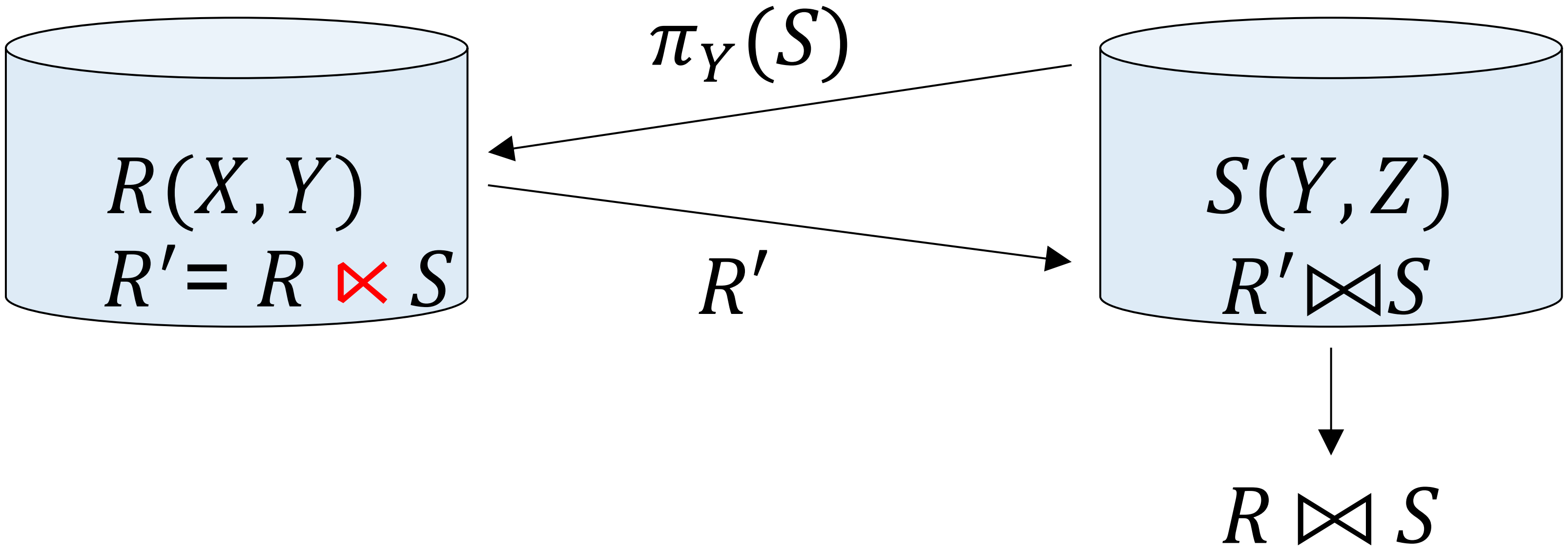
$S(Y,Z)$

Semijoins in Distributed Databases

- Semijoins are often used to compute equijoins in distributed databases

$\mathrm{R} \bowtie \mathrm{S}$

Goal: send less data to reduce network bandwidth!



$\pi_Y(S)$

$R(X,Y)$

$R' = R \ltimes S$

R'

$S(Y,Z)$

$R' \bowtie S$

$R \bowtie S$

$$\mathrm{R} \bowtie \mathrm{S} = \underbrace{(\mathrm{R} \ltimes \mathrm{S})} \bowtie \mathrm{S}$$

law of semijoins

reduced

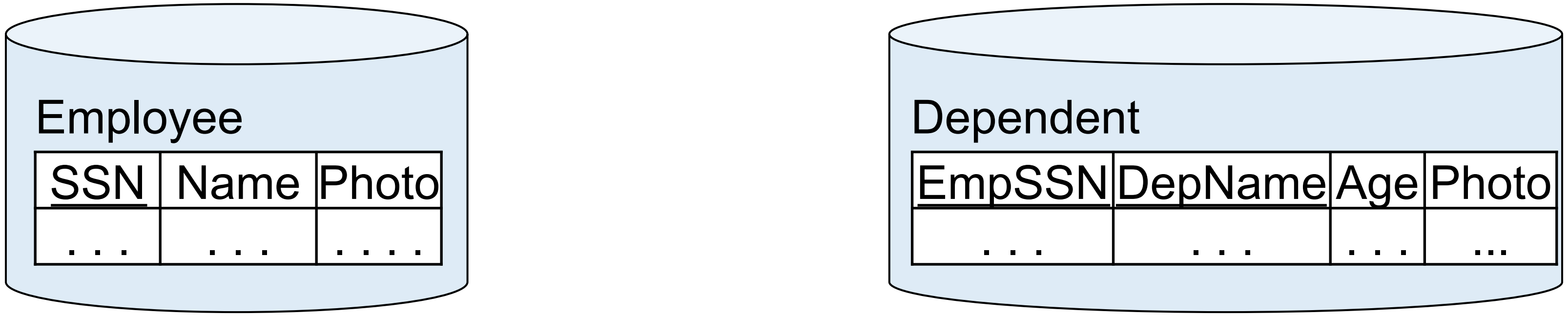
Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71}(\text{Dependent}))$$

Assumptions:

1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. “Photo” is big.



Employee

SSN	Name	Photo
. . .	. . .	. . . .

Dependent

EmpSSN	DepName	Age	Photo
. . .	. . .	. . .	...

?

Based upon an example by Dan Suciu

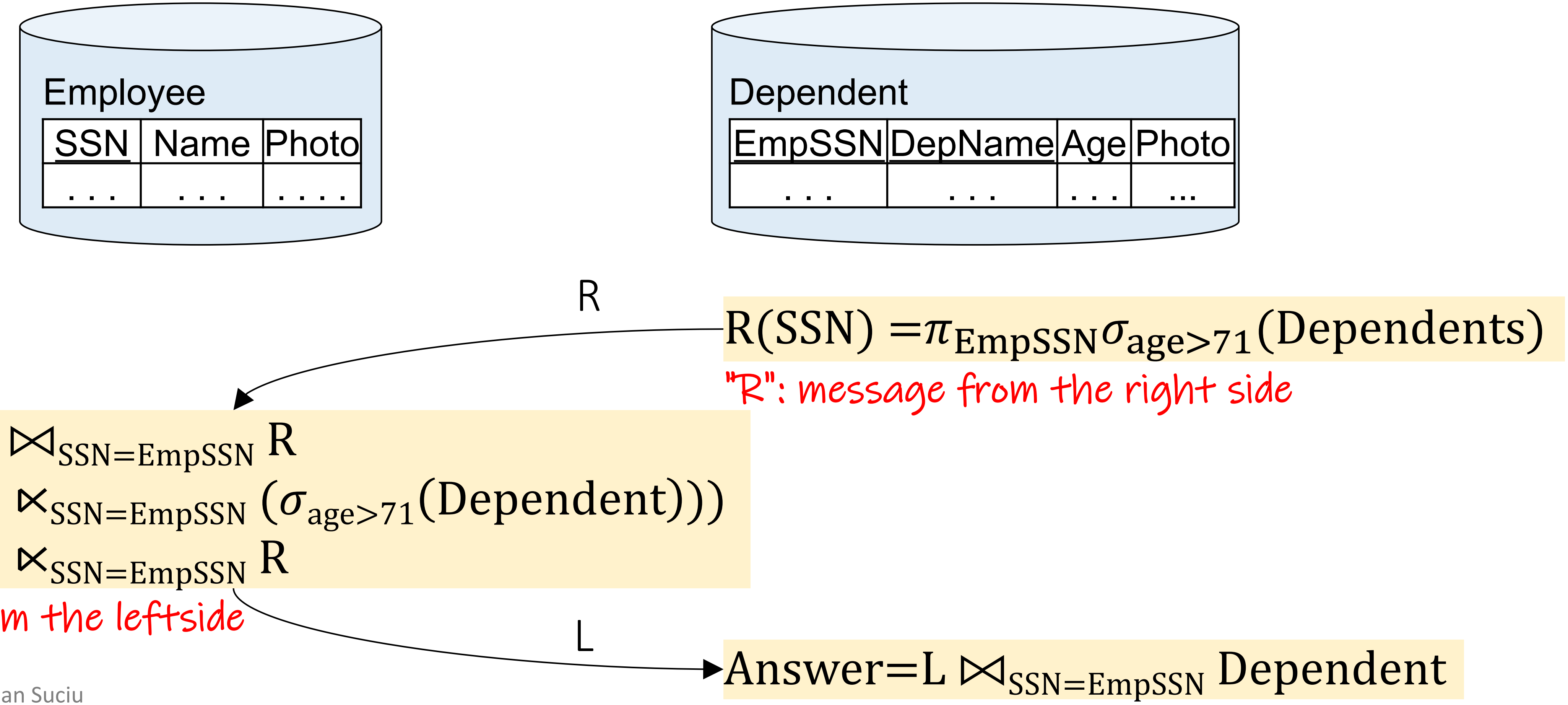
Semijoins in Distributed Databases

Task: compute <u>with minimum data transfer</u>:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent})))$$

Assumptions:

1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. “Photo” is big.



Employee

<u>SSN</u>	Name	Photo
. . .	. . .	. . . .

Dependent

<u>EmpSSN</u>	<u>DepName</u>	Age	Photo
. . .	. . .	. . .	...

$$\text{R(SSN)} = \pi_{\text{EmpSSN}}\sigma_{\text{age>71}}(\text{Dependents})$$

"R": message from the right side

$$\begin{aligned} \text{L} &= \text{Employee} \bowtie_{\text{SSN=EmpSSN}} \text{R} \\ &= \text{Employee} \ltimes_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent}))) \\ &= \text{Employee} \ltimes_{\text{SSN=EmpSSN}} \text{R} \end{aligned}$$

"L": message from the leftside

$$\text{Answer} = \text{L} \bowtie_{\text{SSN=EmpSSN}} \text{Dependent}$$

Based upon an example by Dan Suciu

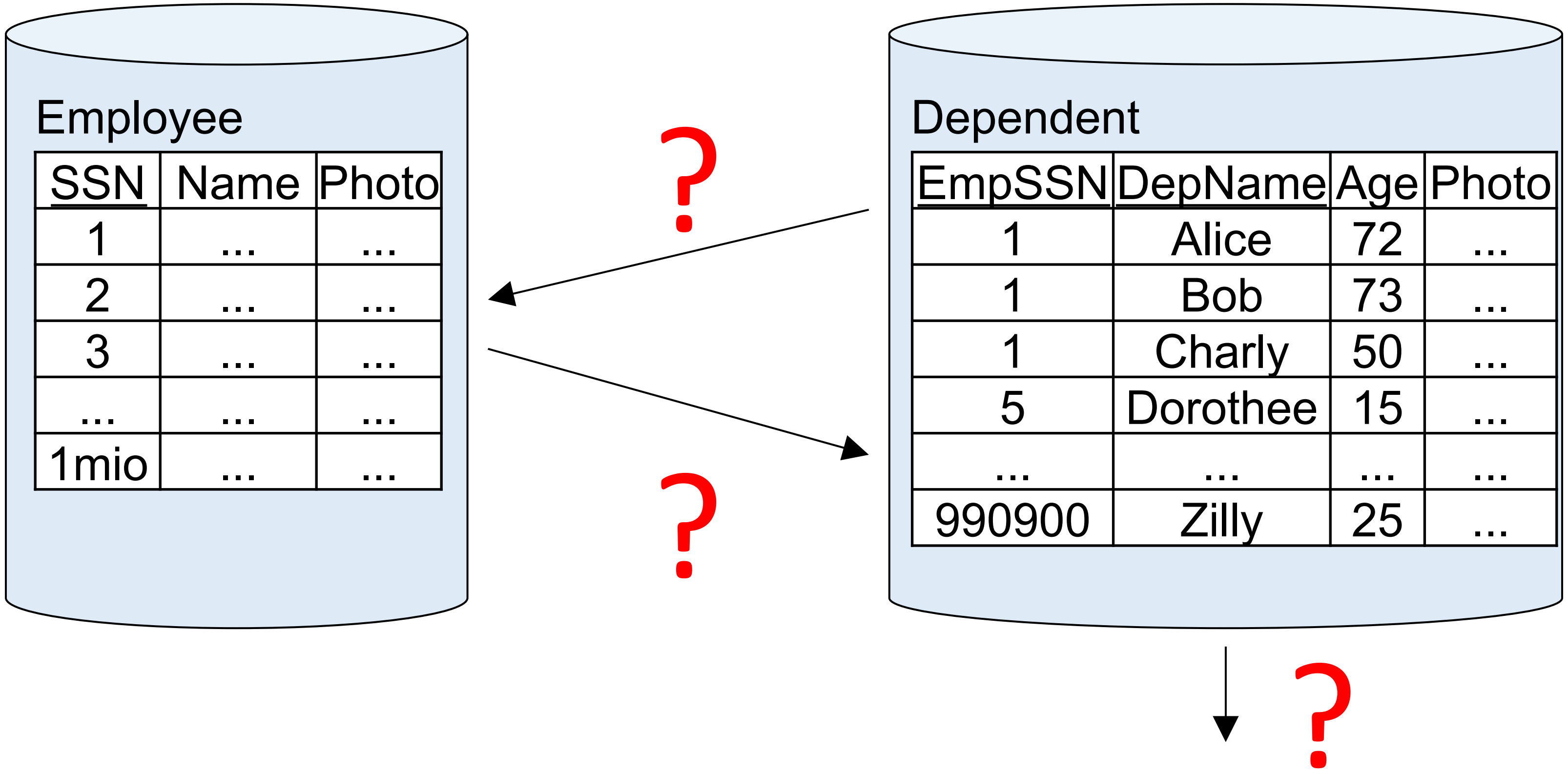
Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent}))$$

Assumptions:

1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. “Photo” is big.



Employee

SSN	Name	Photo
1	...	...
2	...	...
3	...	...
...	...	...
1mio	...	...

Dependent

EmpSSN	DepName	Age	Photo
1	Alice	72	...
1	Bob	73	...
1	Charly	50	...
5	Dorothee	15	...
...	...	...	...
990900	Zilly	25	...

Based upon an example by Dan Suciu

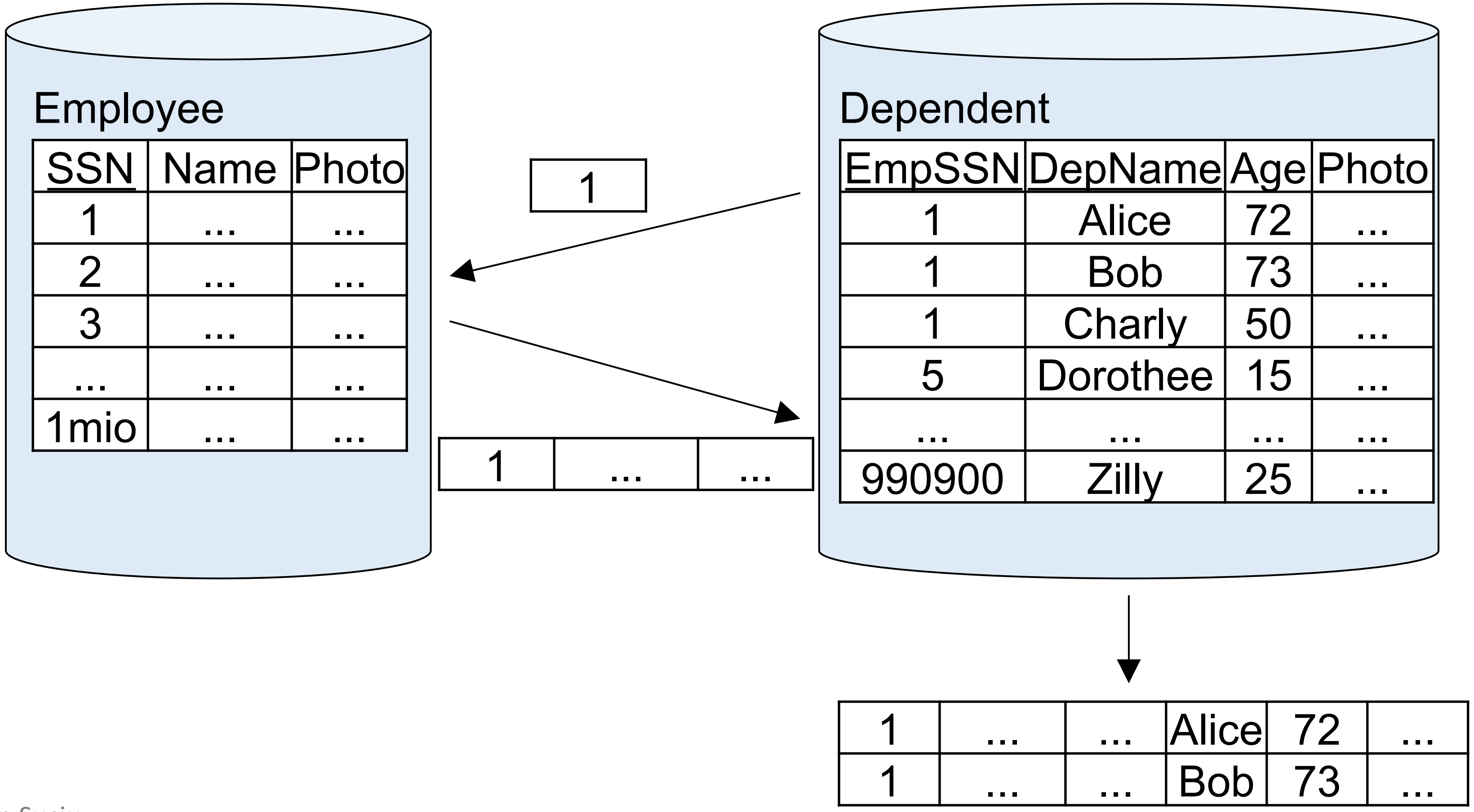
Semijoins in Distributed Databases

Task: compute with minimum data transfer:

$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71}(\text{Dependent})))$$

Assumptions:

1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. “Photo” is big.



Employee

SSN	Name	Photo
1	...	...
2	...	...
3	...	...
...	...	...
1mio	...	...

Dependent

EmpSSN	DepName	Age	Photo
1	Alice	72	...
1	Bob	73	...
1	Charly	50	...
5	Dorothee	15	...
...	...	...	...
990900	Zilly	25	...

1

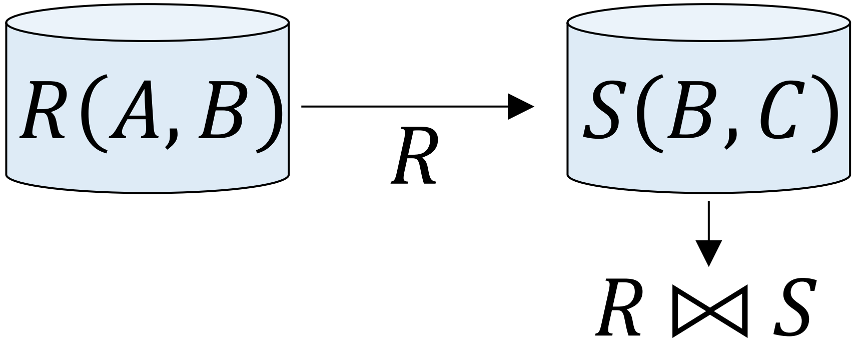
1	...	...

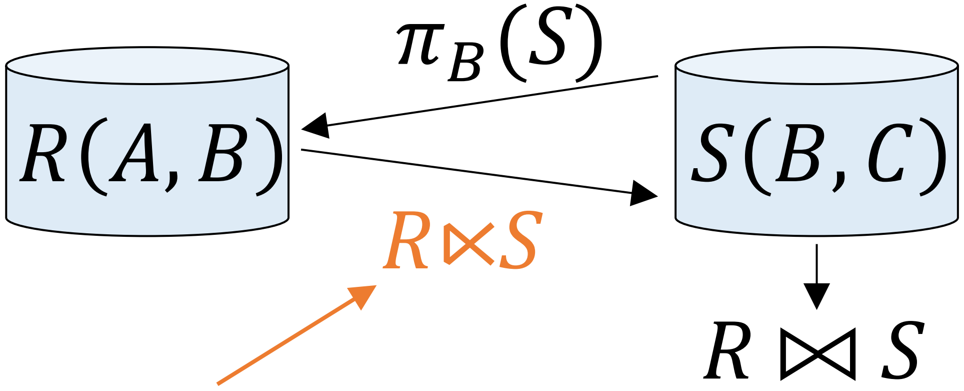
1	...	...	Alice	72	...
1	...	...	Bob	73	...

Based upon an example by Dan Suciu

Semijoins as Message Passing

- Semijoins can reduce network use for equijoins in distributed databases





Reduced R: keep only those tuples that join with S

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

Repetition: Law of Semijoins

- Definition: the semi-join operation is:

$$\boxed{\mathrm{R} \ltimes \mathrm{S} = \pi_{\mathrm{A}_1,\ldots,\mathrm{A}_n}(\mathrm{R} \bowtie \mathrm{S})}$$

  - Formally, R ⋉ S means: retain from R only those tuples that have some matching tuple in S (in bag semantics: duplicates in R are preserved / Duplicates in S don't matter)
  - Data complexity: O(|R| + |S|) ignoring log-factors
  - Input: R(A_1,…,A_n), S(B_1,…,B_m), Output: T(A_1,…,A_n)

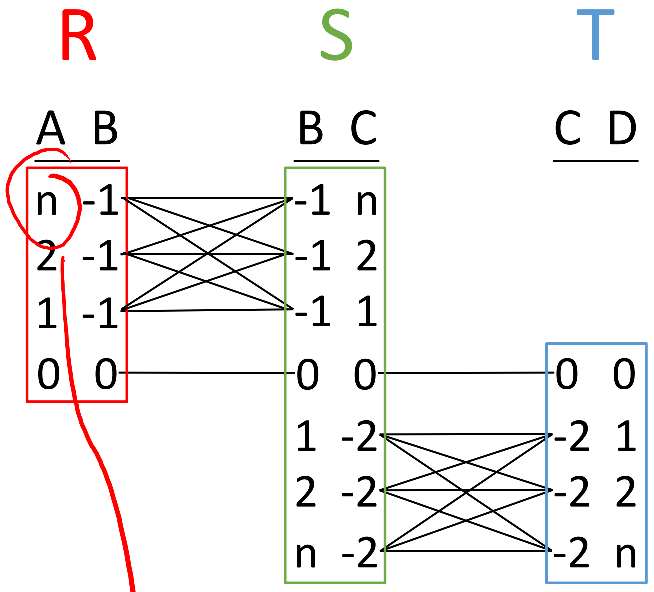
- The law of semijoins is:

$$\boxed{\mathrm{R} \bowtie \mathrm{S} = (\mathrm{R} \ltimes \mathrm{S}) \bowtie \mathrm{S}}$$

  - Thus, removing "dangling tuples" from a table does not change the query result

Semi-joins can also help if data are local

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



R

A	B
n	-1
2	-1
1	-1
0	0

S

B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T

C	D
0	0
-2	1
-2	2
-2	n

3, 5, 100, 1000, 10^4, ..

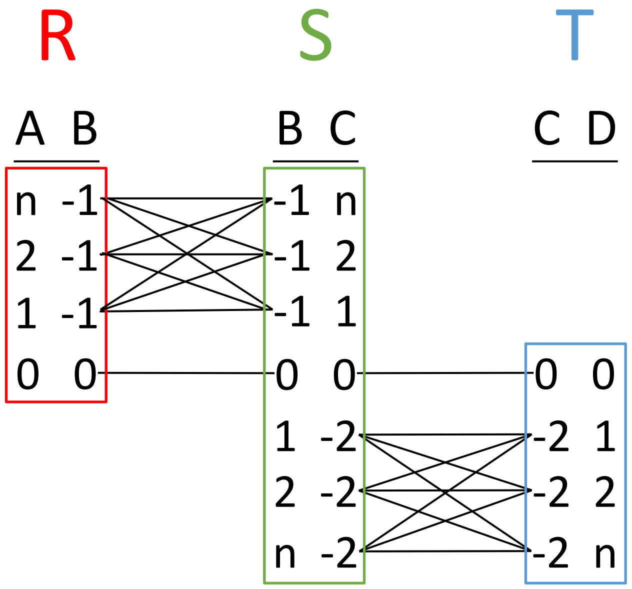
$$Q = (R \bowtie_B S) \bowtie_C T$$

What is the problem?

?

Semi-joins can also help if data are local

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



R

A	B
n	-1
2	-1
1	-1
0	0

S

B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T

C	D
0	0
-2	1
-2	2
-2	n

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

What is the problem?

- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes $O(n^2)$ despite constant output ☹

What is a typical query plan?

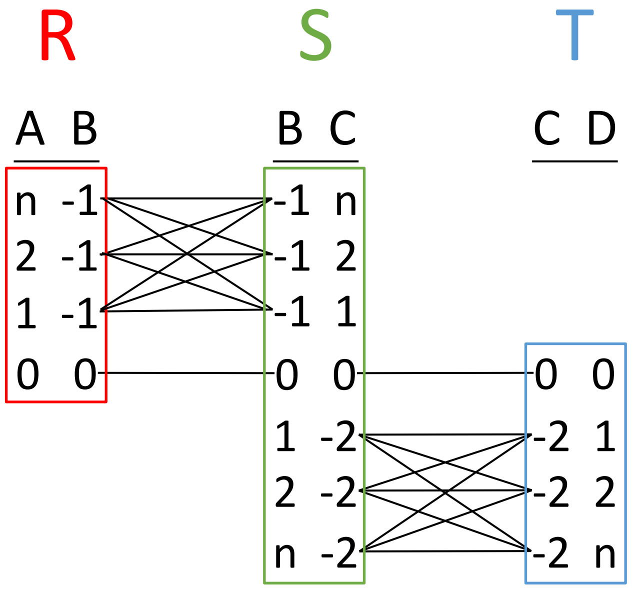
?

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

Semi-joins can also help if data are local

601

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



A	B
n	-1
2	-1
1	-1
0	0

B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

C	D
0	0
-2	1
-2	2
-2	n

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

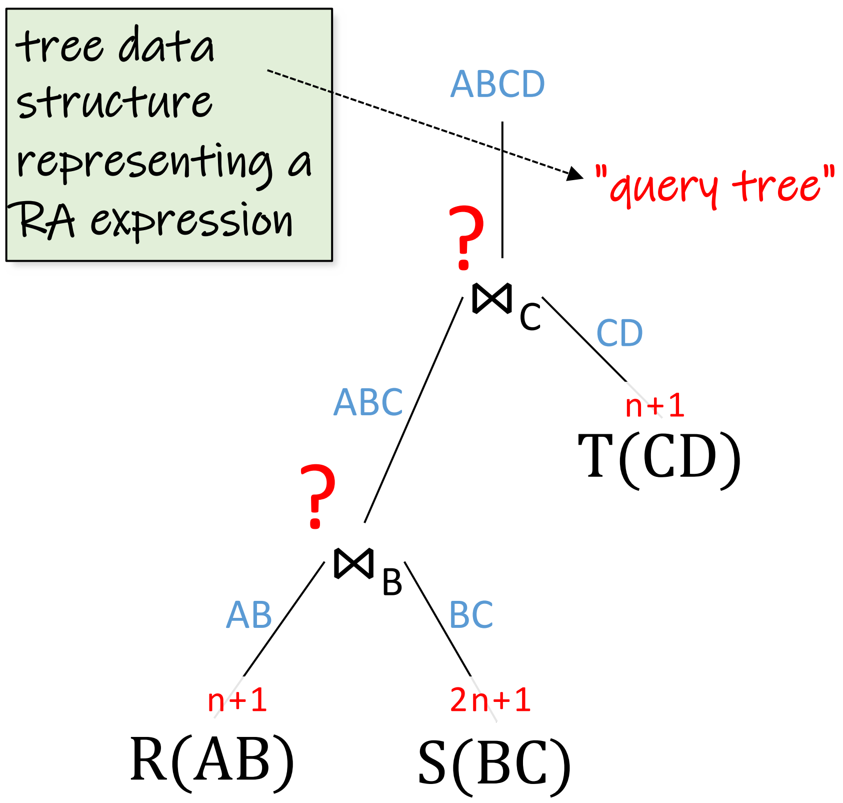
What is the problem?

- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes O(n²) despite constant output ☹

What is a typical query plan?

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


tree data structure representing a RA expression

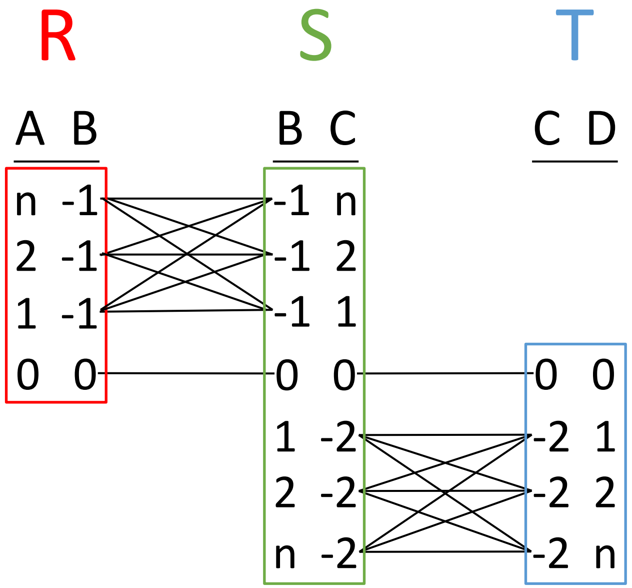
"query tree"

What are the cardinalities at each stage?

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

Semi-joins can also help if data are local

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



R	
A	B
n	-1
2	-1
1	-1
0	0

S	
B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T	
C	D
0	0
-2	1
-2	2
-2	n

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

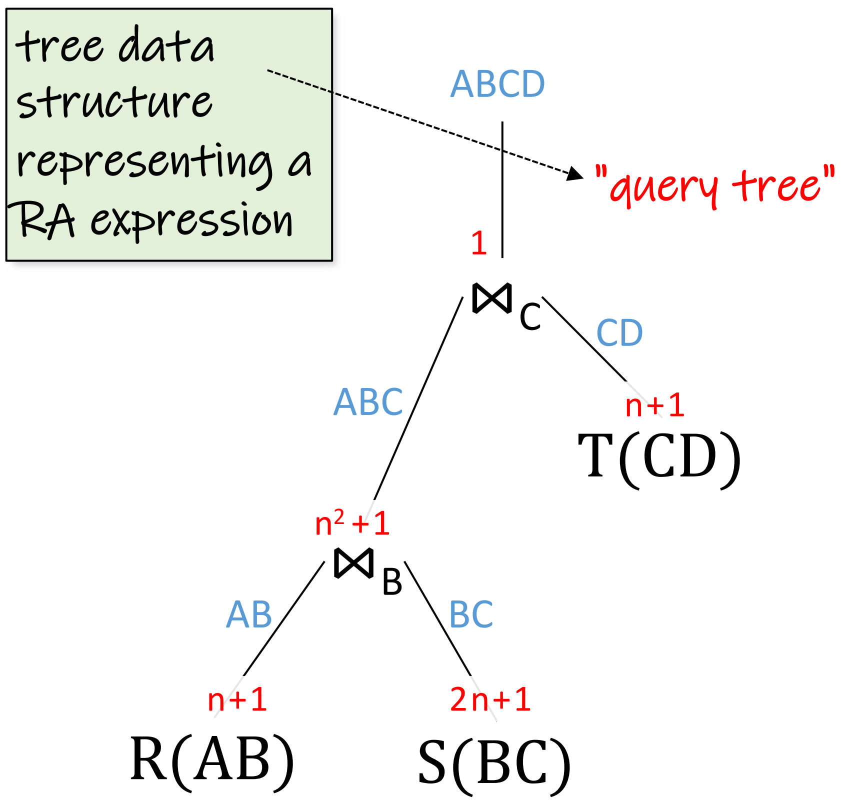
What is the problem?

- Query output cardinality is 1
- But quadratic intermediate result sizes

Thus the query takes O(n²) despite constant output ☹

What is a typical query plan?

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



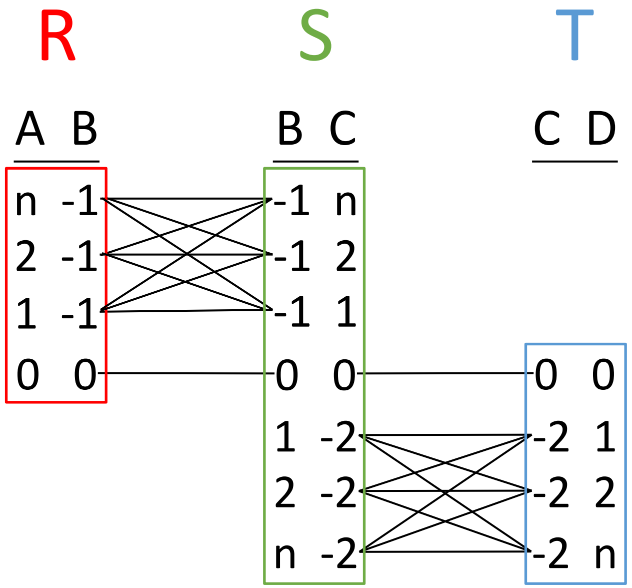
tree data structure representing a RA expression

What are the cardinalities at each stage?

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

Semi-joins can also help if data are local

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



R

A	B
n	-1
2	-1
1	-1
0	0

S

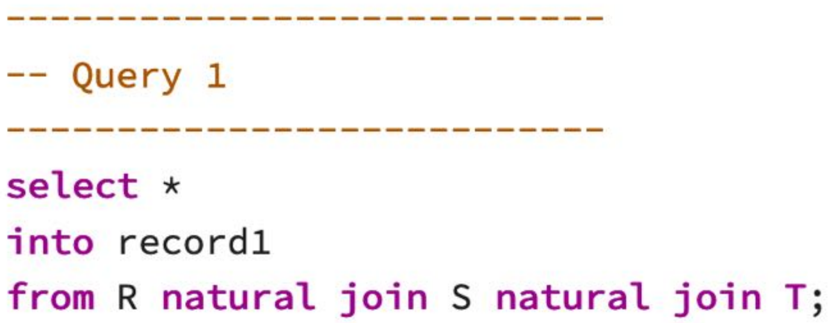
B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T

C	D
0	0
-2	1
-2	2
-2	n

?

Can you think of a faster evaluation plan?



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

n=1,000: t_{Q1}=1.4 sec t_{Q2}=5 msec

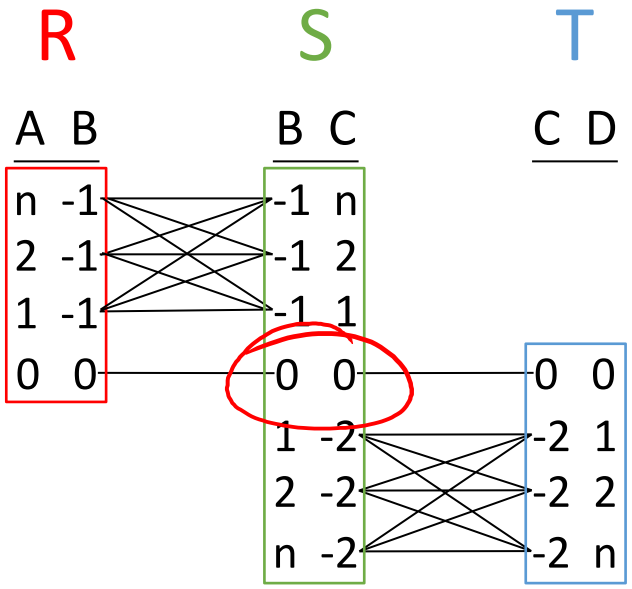
n=2,000: t_{Q1}=6.1 sec t_{Q2}=8 msec

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

Semi-joins can also help if data are local

601

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



R

A	B
n	-1
2	-1
1	-1
0	0

S

B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T

C	D
0	0
-2	1
-2	2
-2	n

WITH clauses (also **CTE** = Common Table Expression) act like temporary views

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;
```

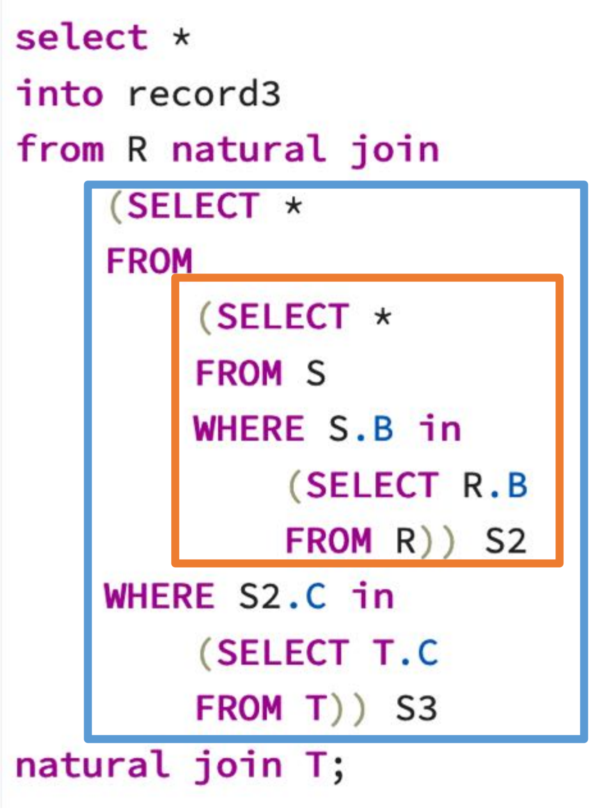
n=1,000: t_{Q1}=1.4 sec t_{Q2}=5 msec

n=2,000: t_{Q1}=6.1 sec $O(n^2)$ ☹ t_{Q2}=8 msec $O(n)$ ☺

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

Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

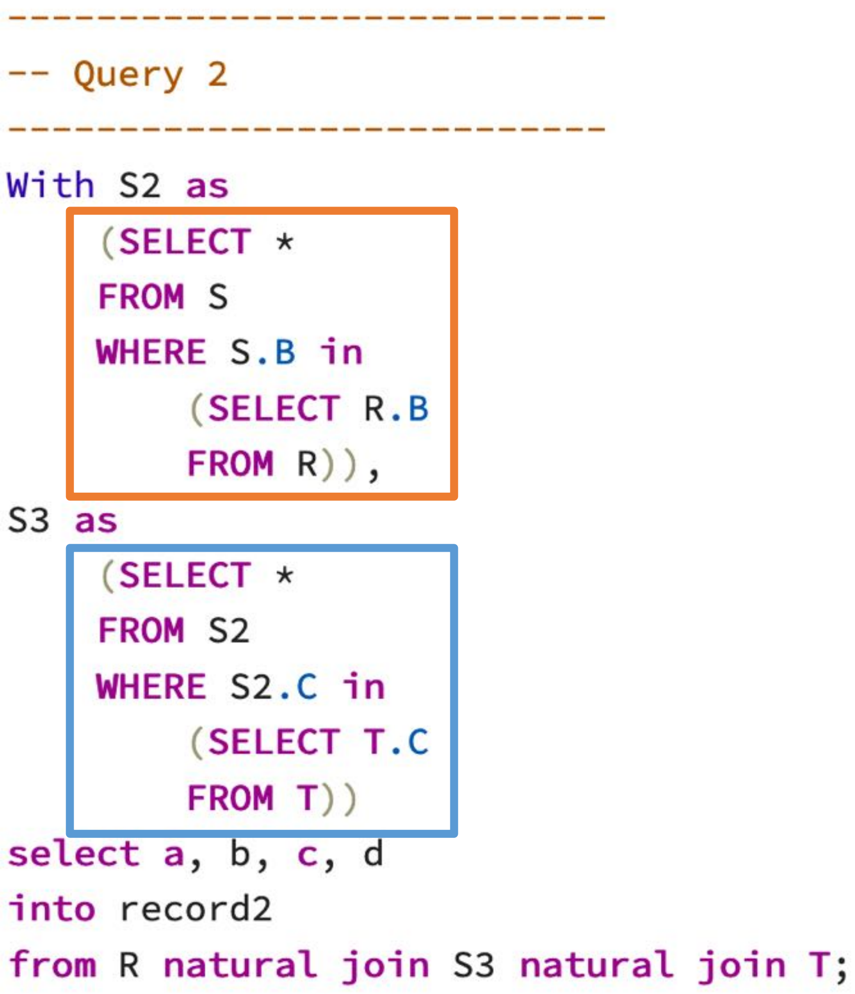
Diversion into CTE's (Common Table Expressions)



```
select *
into record3
from R natural join
    (SELECT *
    FROM
        (SELECT *
        FROM S
        WHERE S.B in
            (SELECT R.B
            FROM R)) S2
    WHERE S2.C in
        (SELECT T.C
        FROM T)) S3
natural join T;
```

WITH clauses (also **CTE** = Common Table Expression) act like temporary views

- Allow to deconstruct more complex queries into simple blocks to be used and reused if necessary.
- Can increase readability by emphasizing a more procedural interpretation of a query in a workflow
- Especially useful if you need to reference a derived table multiple times in a single query or you perform the same calculation multiple times across multiple query components (= memoization)



```
---------------------------
-- 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;
```

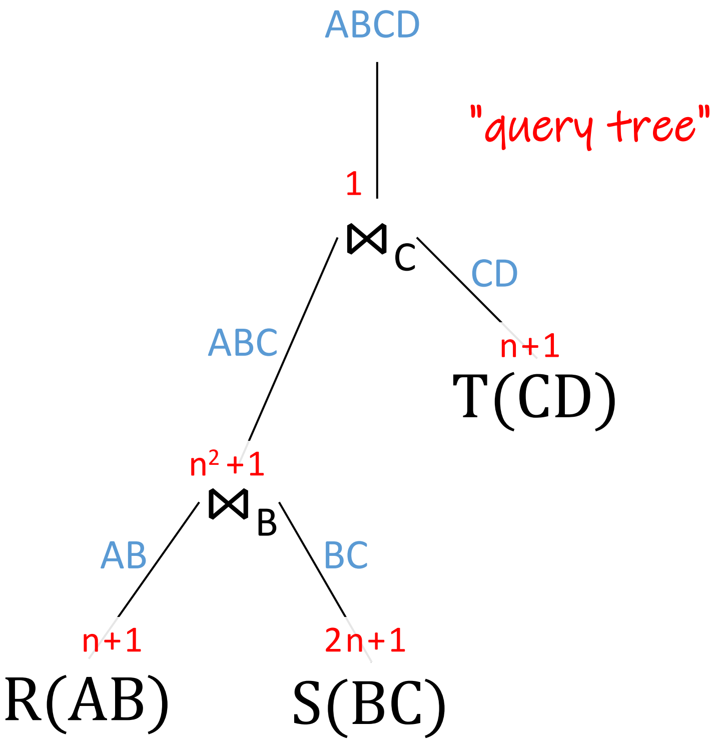
SQL example available at: https://github.com/northeastern-datalab/cs3200-activities/tree/master/sql

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]

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



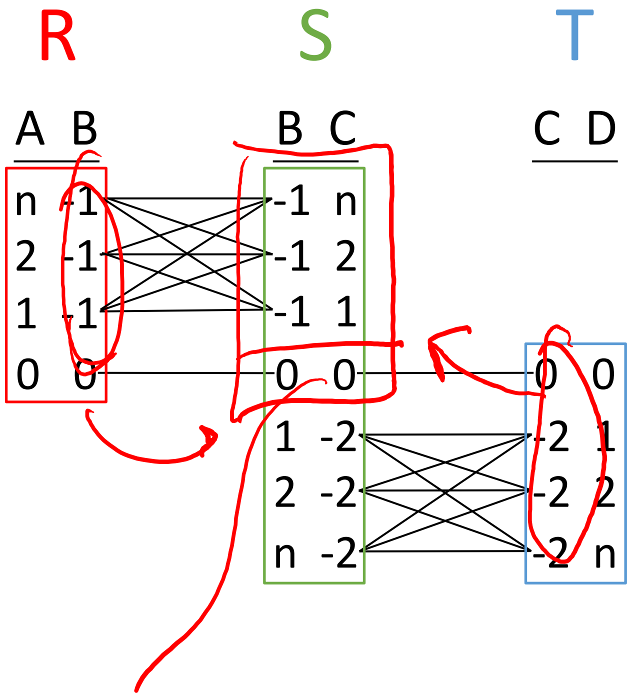
[Bernstein, Goodman 81]. "Using Semi-Joins to Solve Relational Queries", JACM 1981. https://doi.org/10.1145/322234.322238

[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

The more general idea: "Sideways information passing"

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



R

A	B
n	-1
2	-1
1	-1
0	0

S

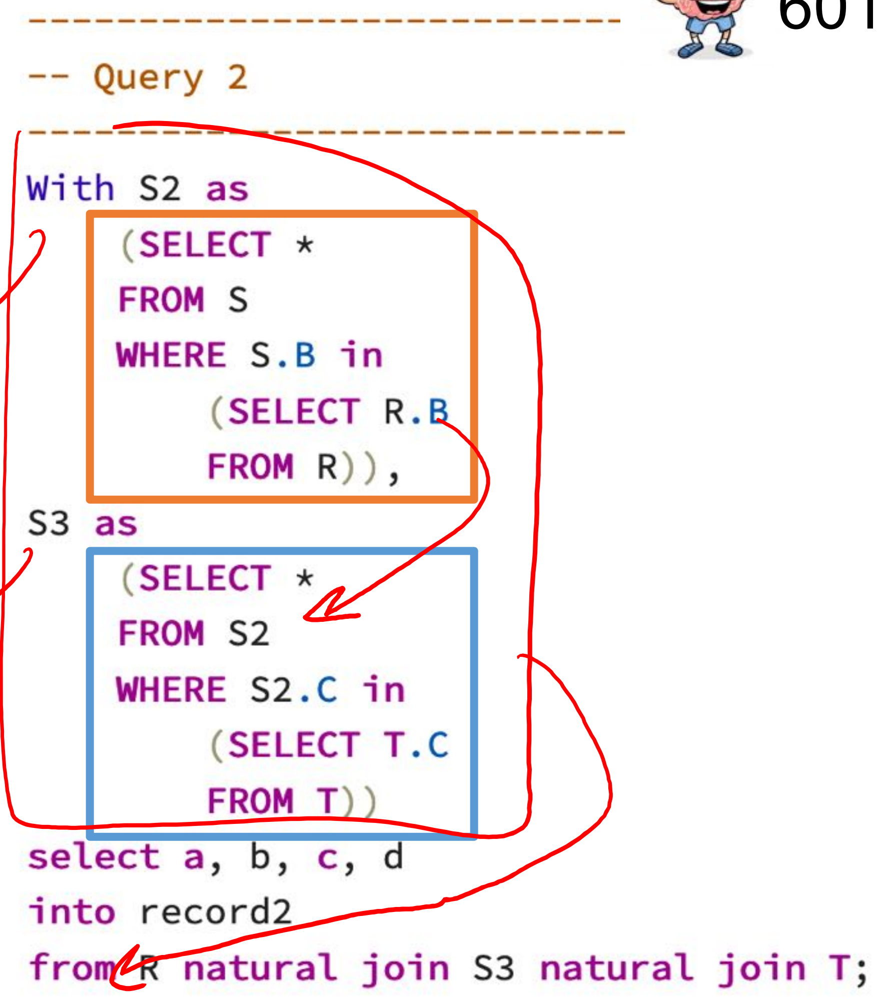
B	C
-1	n
-1	2
-1	1
0	0
1	-2
2	-2
n	-2

T

C	D
0	0
-2	1
-2	2
-2	n

{0, -1}

S(0, 0)



```
------------------------------
-- 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;
```

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

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

Outline journey

- The idea from the previous slides on semijoins can be generalized to more than two joins, as long as the joins are "acyclic"
- We will next define "acyclic queries"
  - It is a bit more complicated for general hypergraphs (with relations of arbitrary arity) than for graphs
- And later see the "semi-join reduction" on general acyclic queries
- Which is basically the Yannakakis algorithm

Query Hypergraph (vs. Dual Hypergraph vs. Incidence Matrix)

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

Query hypergraph (nodes = variables)

?

Incidence matrix

?

Query dual hypergraph (nodes = atoms)

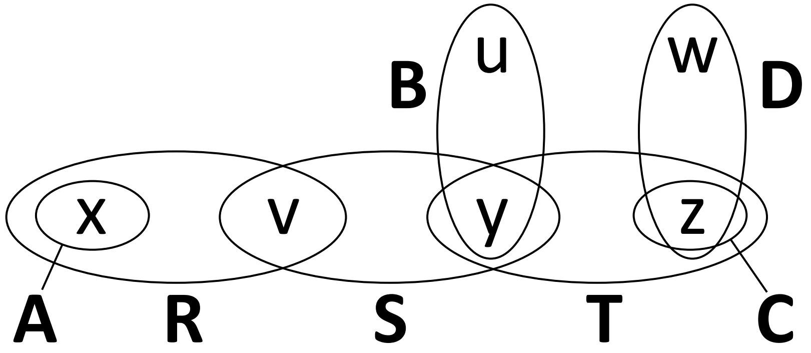
?

Query Hypergraph (vs. Dual Hypergraph vs. Incidence Matrix)

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

Query hypergraph (nodes = variables)

determines α-acyclicity of CQs (that's what we need)



Incidence matrix

?

Query dual hypergraph (nodes = atoms)

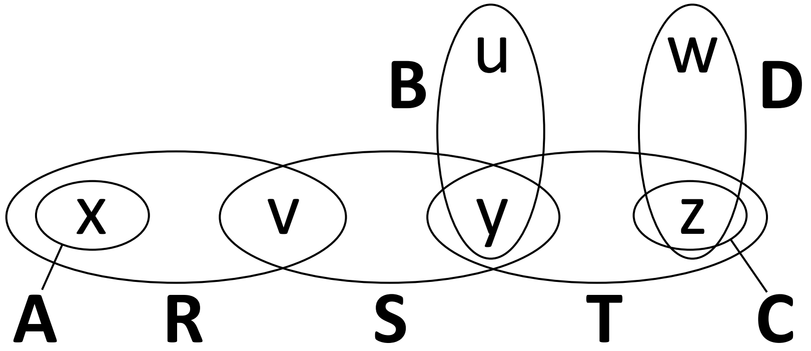
?

Query Hypergraph (vs. Dual Hypergraph vs. Incidence Matrix)

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

Query hypergraph (nodes = variables)

determines α-acyclicity of CQs (that's what we need)

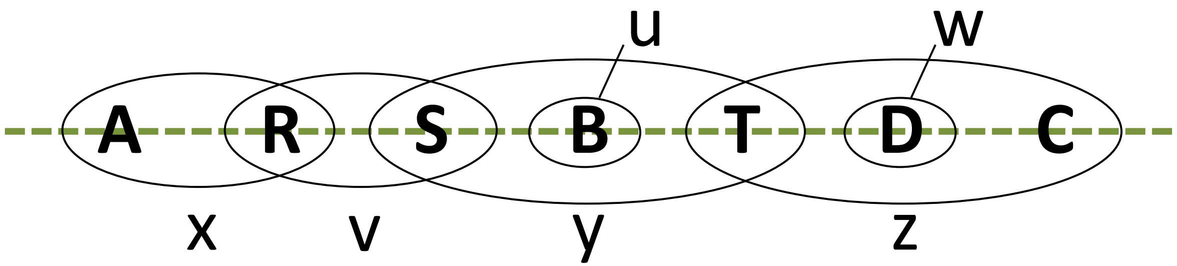


Incidence matrix

?

Query dual hypergraph (nodes = atoms)

determines complexity of query resilience [Meliou+'10]



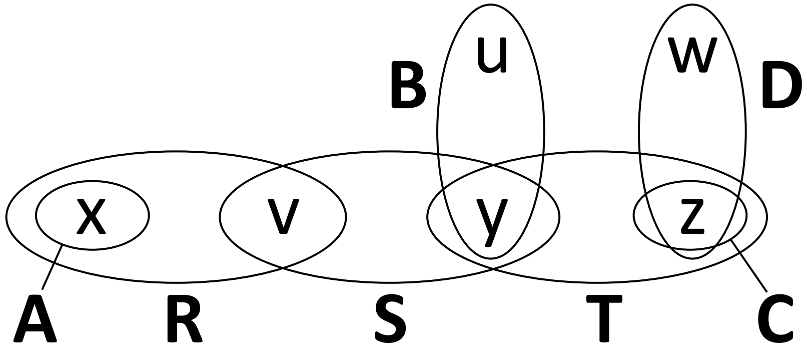
Meliou, Gatterbauer, Moore, Suciu. "The complexity of causality and responsibility for query answers and non-answers", PVLDB 2010. https://doi.org/10.14778/1880172.1880176

Query Hypergraph (vs. Dual Hypergraph vs. Incidence Matrix)

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

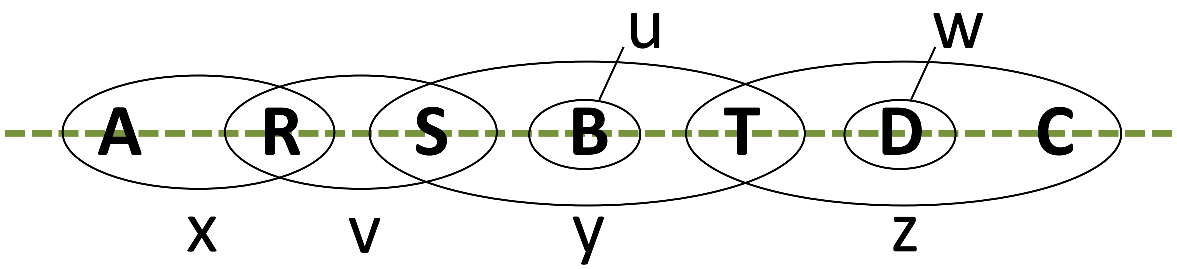
Query hypergraph (nodes = variables)

determines α-acyclicity of CQs (that's what we need)



Query dual hypergraph (nodes = atoms)

determines complexity of query resilience [Meliou+'10]



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

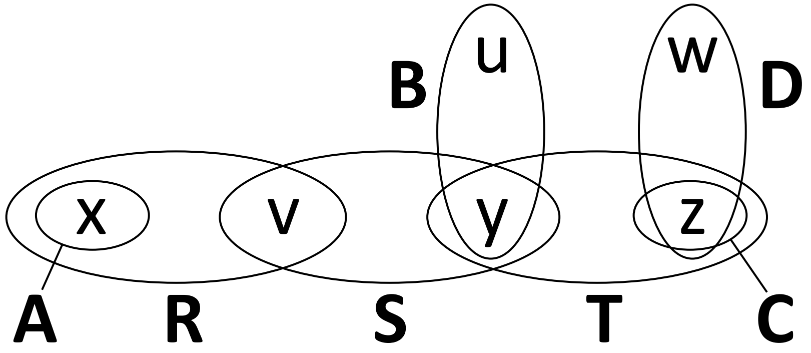
Meliou, Gatterbauer, Moore, Suciu. "The complexity of causality and responsibility for query answers and non-answers", PVLDB 2010. https://doi.org/10.14778/1880172.1880176 /
Gatterbauer, Suciu. "Dissociation and propagation for approximate lifted inference with standard relational database management systems", VLDBJ 2017. https://doi.org/10.1007/s00778-016-0434-5

Our focus is on the Query Hypergraph of a Conjunctive Query

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

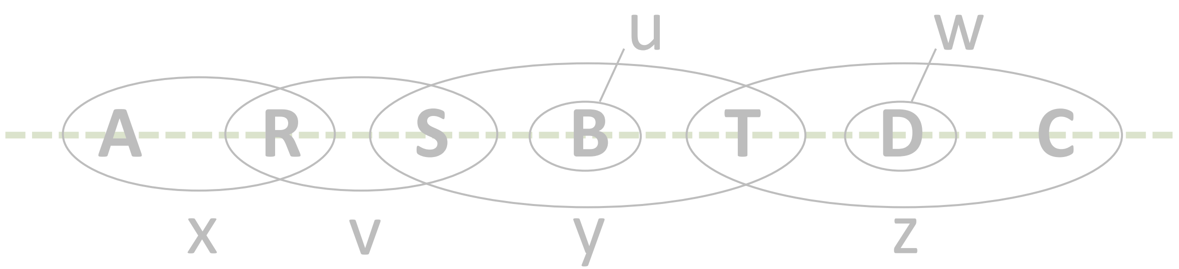
Query hypergraph (nodes = variables)

determines α-acyclicity of CQs (that's what we need)



Query dual hypergraph (nodes = atoms)

determines complexity of query resilience [Meliou+'10]



Incidence matrix

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

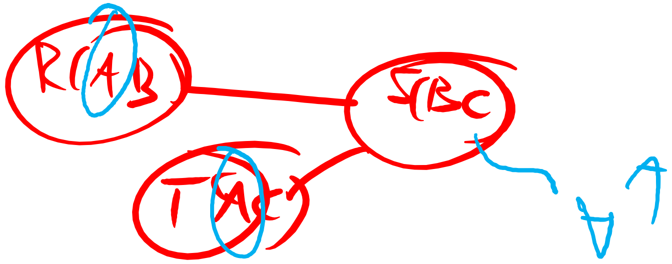
	x	v	y	u	z	w
A	1					
R	1	1				
S		1	1			
B			1	1		
T			1		1	
C					1	
D					1	1

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

α (alpha)-acyclic Conjunctive Queries have Join Trees

- Definition: A conjunctive query Q is α-acyclic if it has a join tree.
- A join tree for $Q(\mathbf{x})$:– $R_1(\mathbf{z}_1)$, $R_2(\mathbf{z}_2)$,..., $R_m(\mathbf{z}_m)$ 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 y occurring in Q, the set of the nodes of T that contain y forms a (connected) subgraph of T.
    - Also called running intersection property (see also junction tree algorithm and tree decompositions)
    - 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 $R_k(\mathbf{z}_k)$

R(A,B), S(B,C), T(A,C)



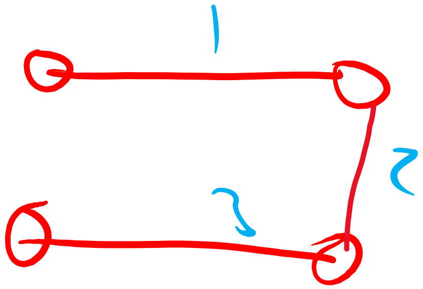
- The GYO reduction* helps us find a join tree from the hypergraph iff it is α-acyclic

Notice that the definition of join tree does not include the root. We do need to choose roots later, and different choices of roots lead to different sequences of semijoin reductions

* 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

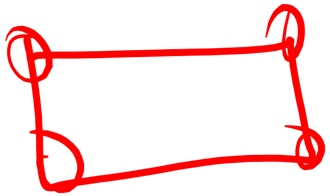
Which query is acyclic?

- Path of length 3? (4 nodes, 3 edges). Return end points.



?

- Cycle of length 4? Boolean.

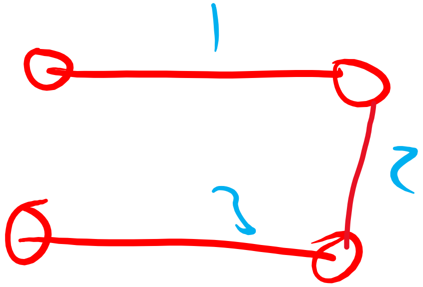


?

Which query is acyclic?

- Path of length 3? (4 nodes, 3 edges). Return end points.

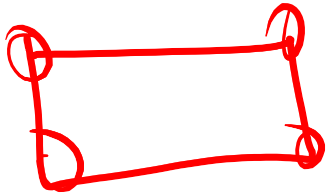
$P4(x_1,x_4) \text{ :- } E(x_1,x_2), E(x_2,x_3), E(x_3,x_4)$



?

- Cycle of length 4? Boolean.

$C4() \text{ :- } E(x_1,x_2), E(x_2,x_3), E(x_3,x_4), E(x_4,x_1)$

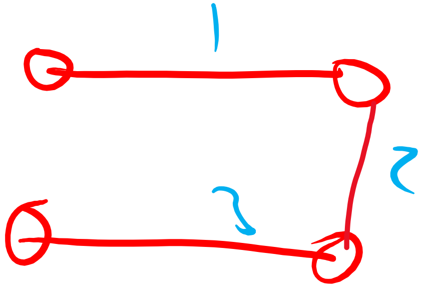


?

Which query is acyclic?

- Path of length 3? (4 nodes, 3 edges). Return end points.

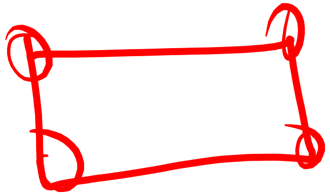
$P4(x_1,x_4)$:- $E(x_1,x_2)$, $E(x_2,x_3)$, $E(x_3,x_4)$



acyclic
(join tree is path)

- Cycle of length 4? Boolean.

$C4()$:- $E(x_1,x_2)$, $E(x_2,x_3)$, $E(x_3,x_4)$, $E(x_4,x_1)$

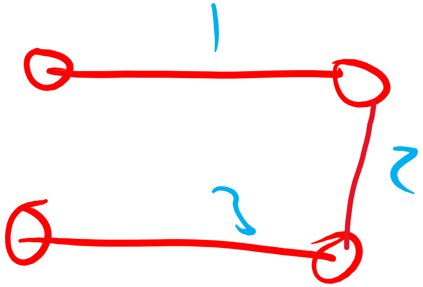


?

Which query is acyclic?

- Path of length 3? (4 nodes, 3 edges). Return end points.

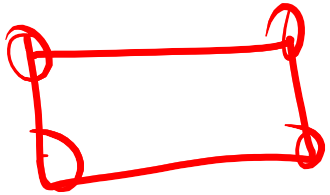
$P4(x_1,x_4) \text{ :- } E(x_1,x_2), E(x_2,x_3), E(x_3,x_4)$



acyclic
(join tree is path)

- Cycle of length 4? Boolean.

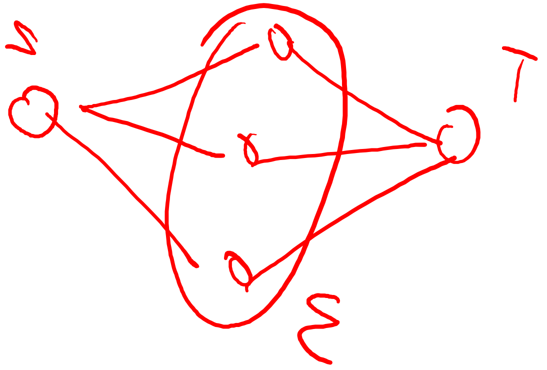
$C4() \text{ :- } E(x_1,x_2), E(x_2,x_3), E(x_3,x_4), E(x_4,x_1)$



cyclic

Acyclic Conjunctive Queries

P(x,z) :- E(x,y), E(y,z)



- Is the following query acyclic?

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

Dual Hypergraph (relations as nodes)

?

Hypergraph (variables as nodes)

?

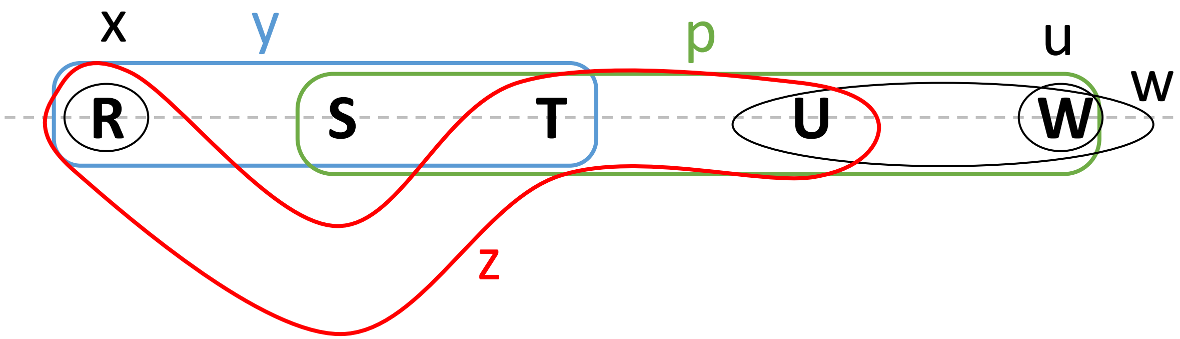
Acyclic Conjunctive Queries

- Is the following query acyclic?

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

Dual Hypergraph (relations as nodes)

Hypergraph (variables as nodes)



?

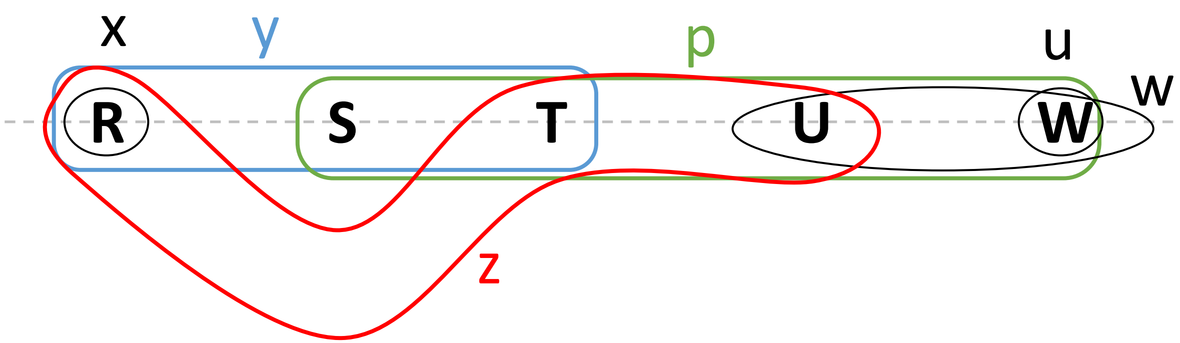
No linear order in dual hypergraph ☹
(Thus resilience is NPC!)

Acyclic Conjunctive Queries

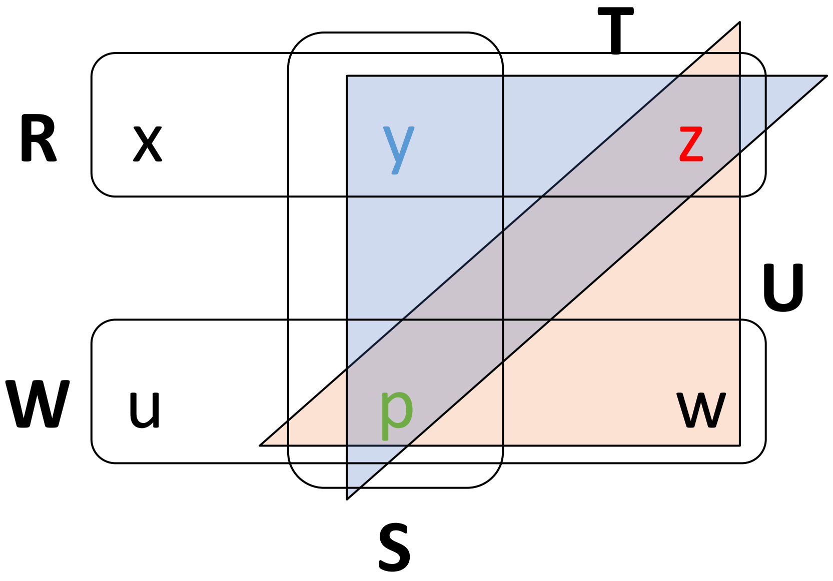
- Is the following query acyclic?

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

Dual Hypergraph (relations as nodes)



Hypergraph (variables as nodes)



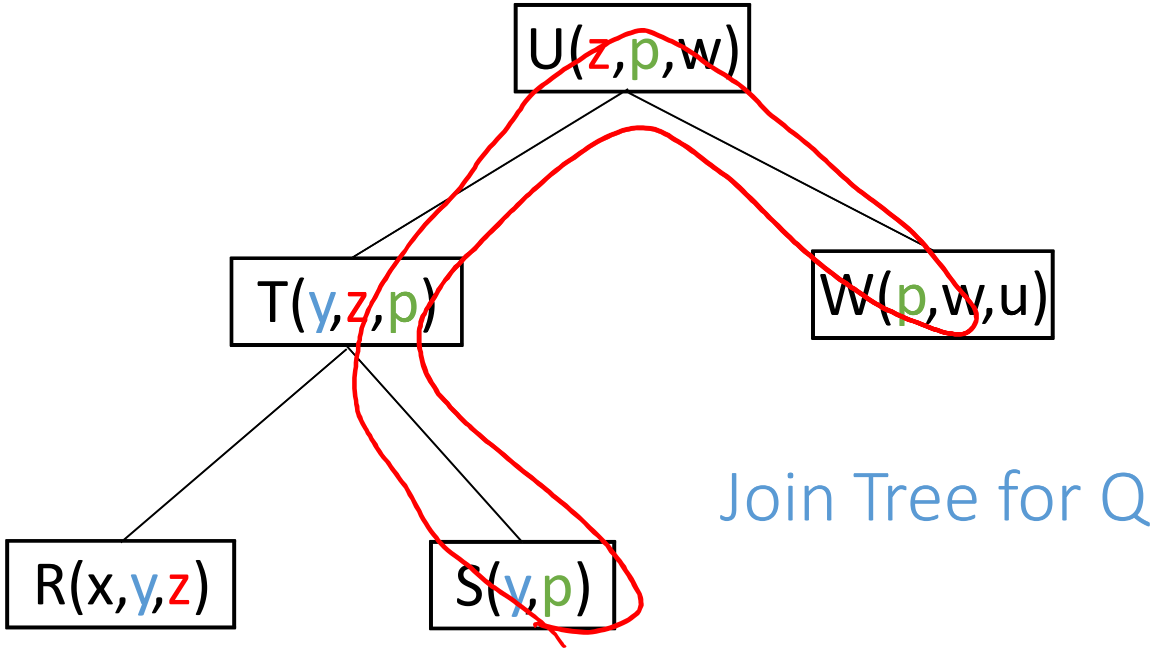
No linear order in dual hypergraph ☹
(Thus resilience is NPC!)

So does the query have a join tree?

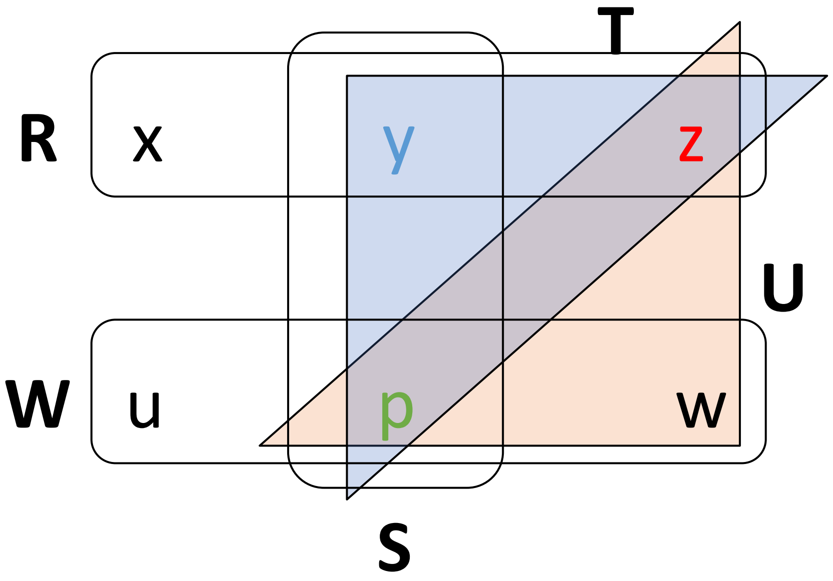
Acyclic Conjunctive Queries

- Is the following query acyclic?

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



Hypergraph (variables as nodes)



Yes the query has a join tree

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

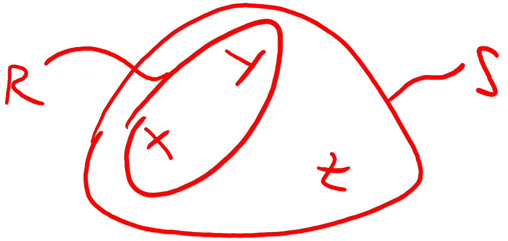
GYO reduction (Graham-Yu-Ozsoyoglu) on the hypergraph

corresponds to a projection

- An ear is a hyperedge H whose variables (=nodes) form two groups:
  1. isolated variables that appear exclusively in H, and
  2. join variables (i.e. they occur in at least one other edge). For join variables there must exist a hyperedge called a witness that contains all of them

corresponds to a join

- GYO algorithm (Ear removal)
  - Remove ears greedily from the hypergraph



- Important: any sequence of reductions that removes all hyperedges implies a join tree:
  - Just draw an edge between an ear and any witness (notice that if an ear has only isolated nodes, any remaining hyperedge is a witness)

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

Proof GYO reduction = acyclic query

- Proof (GYO ⇒ acyclic): if GYO leads to an empty hypergraph, then the resulting graph forms a valid join tree
  - notice that by construction, for any variable, those edges containing the variable will form an induced subtree

- Proof (acyclic ⇒ GYO): if there is a valid join tree, then removing leaf nodes in any order corresponds to a sequence of GYO reductions:
  - All non-exclusive variables from a leaf must be shared with the parent. Thus the parent forms a witness that consumes the leaf (notice that by construction, this also works if the leaf node shares no variables)

GYO reduction: Example 1

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

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

Join tree

?

Query hypergraph

?

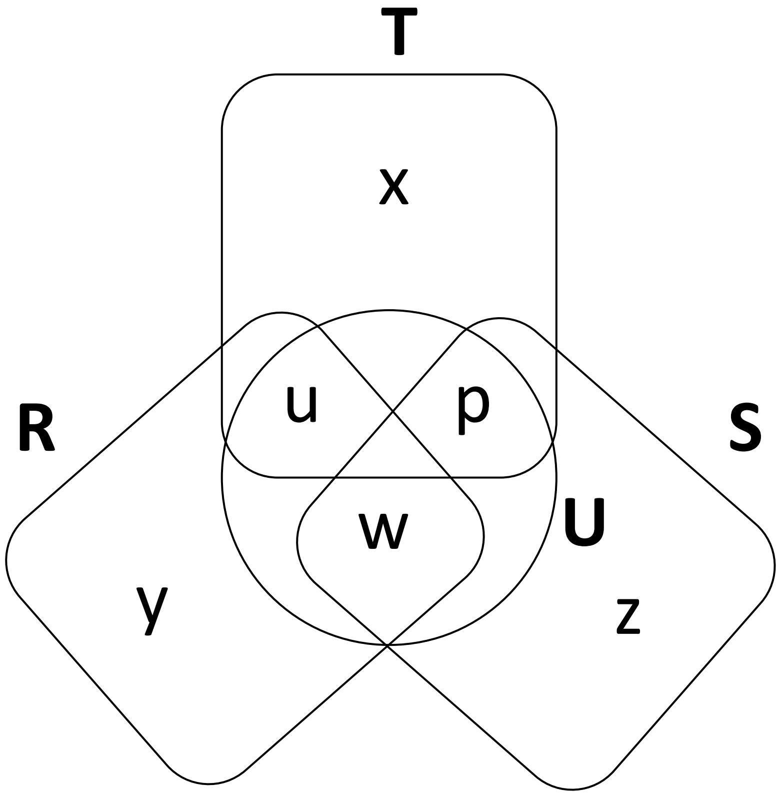
GYO reduction: Example 1

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

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

Join tree

?



GYO reduction: Example 1

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

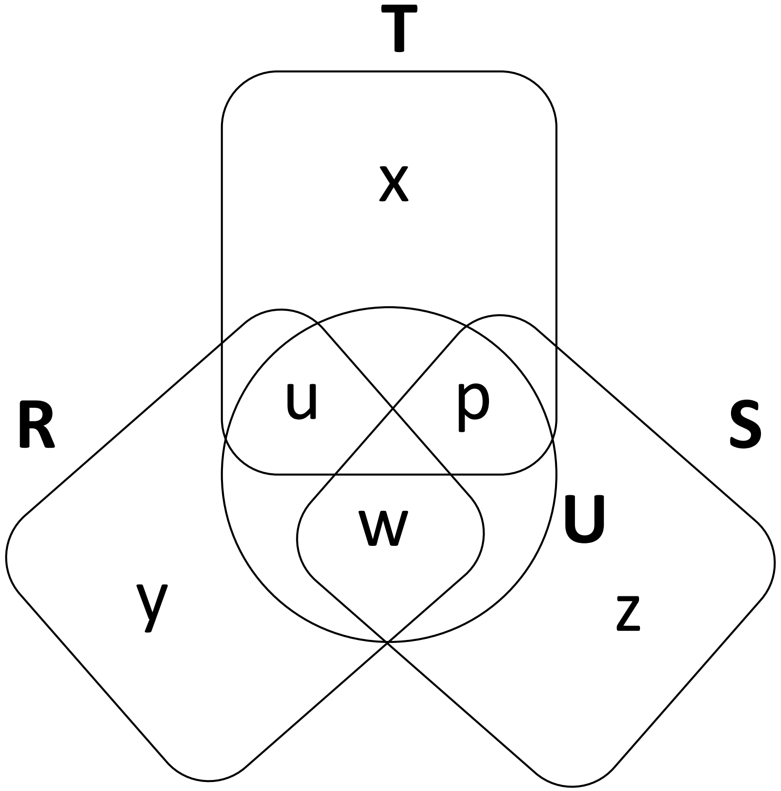
GYO REDUCTION (ear removal)

- remove isolated nodes
- remove consumed or empty edges

Join tree ?

U(u,p,w)

R(y,u,w) S(z,p,w) T(x,u,p)

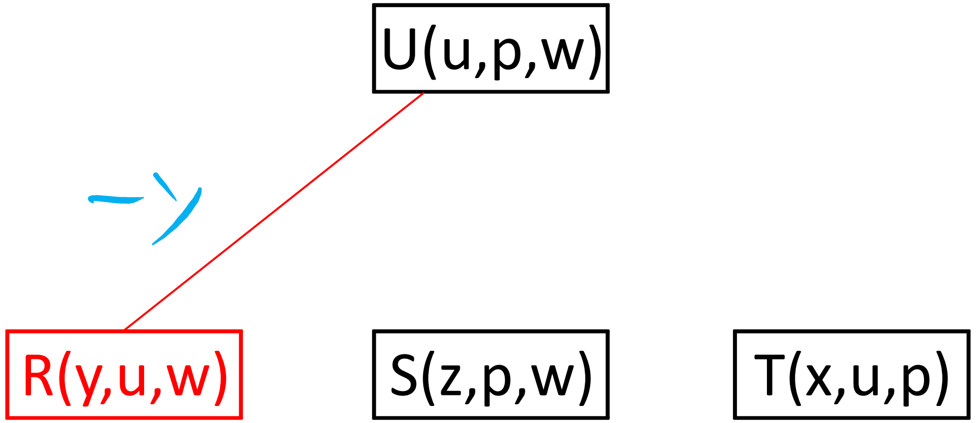


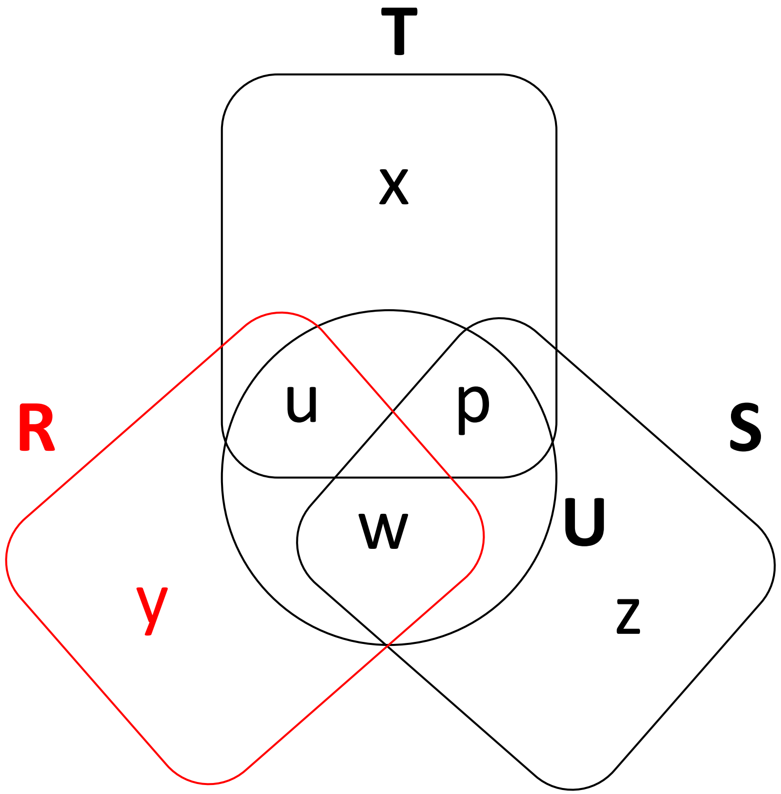
GYO reduction: Example 1

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

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

Join tree

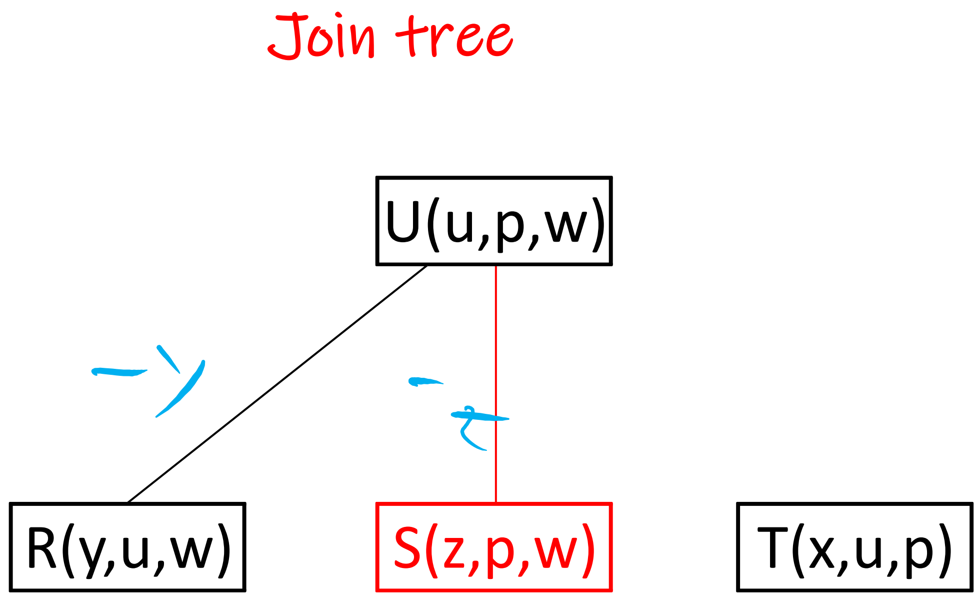

U(u,p,w)
R(y,u,w)
S(z,p,w)
T(x,u,p)

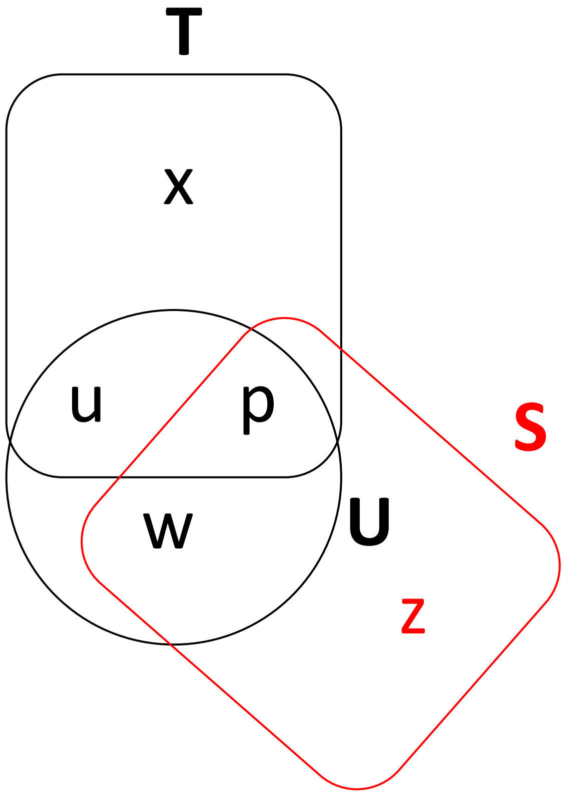


GYO reduction: Example 1

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

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





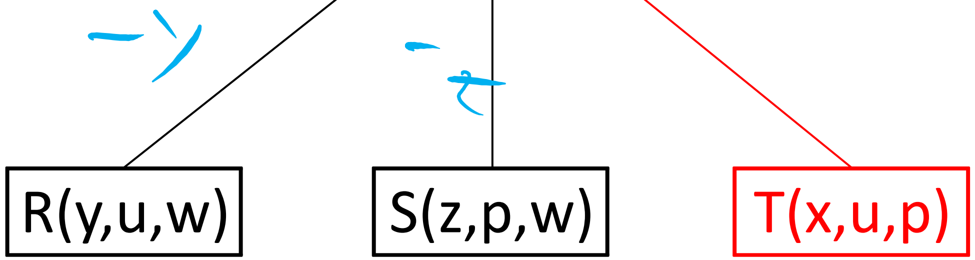
GYO reduction: Example 1

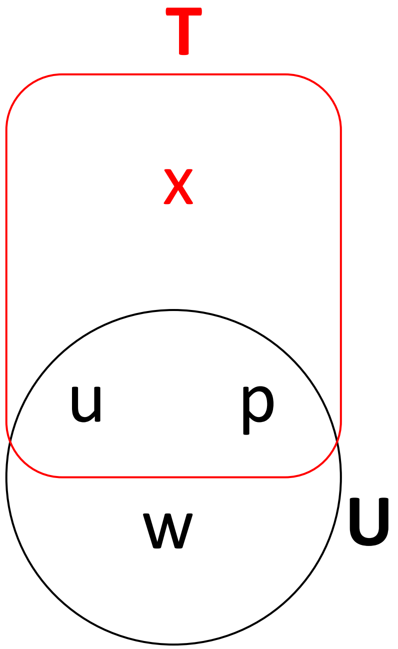
Q :- R(y,u,w), S(z,p,w), T(x,u,p), U(u,p,w).

GYO REDUCTION (ear removal)

- remove isolated nodes
- remove consumed or empty edges

Join tree





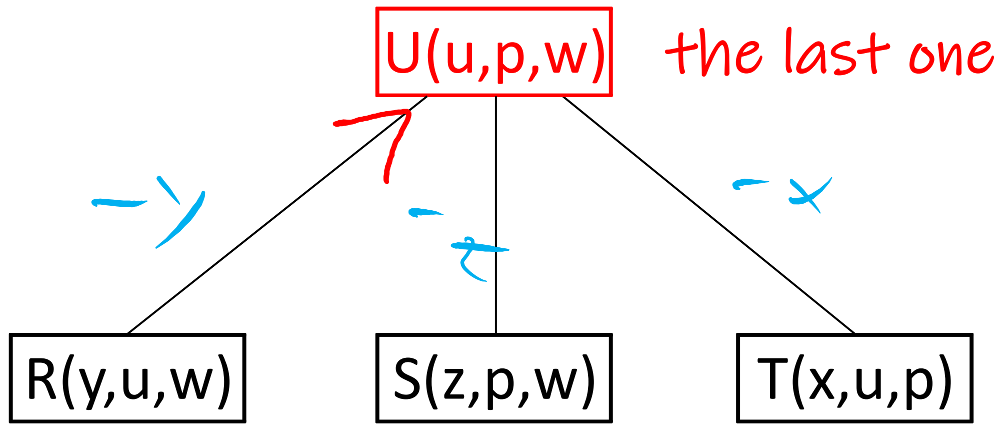
GYO reduction: Example 1

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

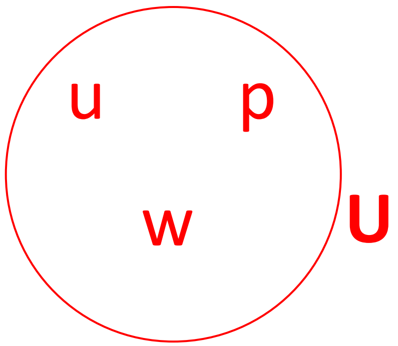
GYO REDUCTION (ear removal)

- remove isolated nodes
- remove consumed or empty edges

Join tree



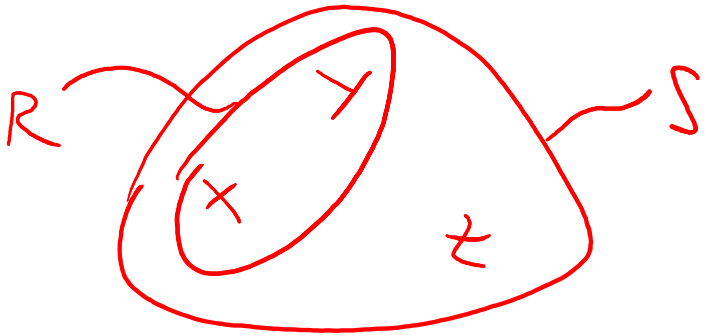
the last one is the root



GYO reduction: Example 2

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



Join tree

?

Query hypergraph

?

GYO reduction: Example 2

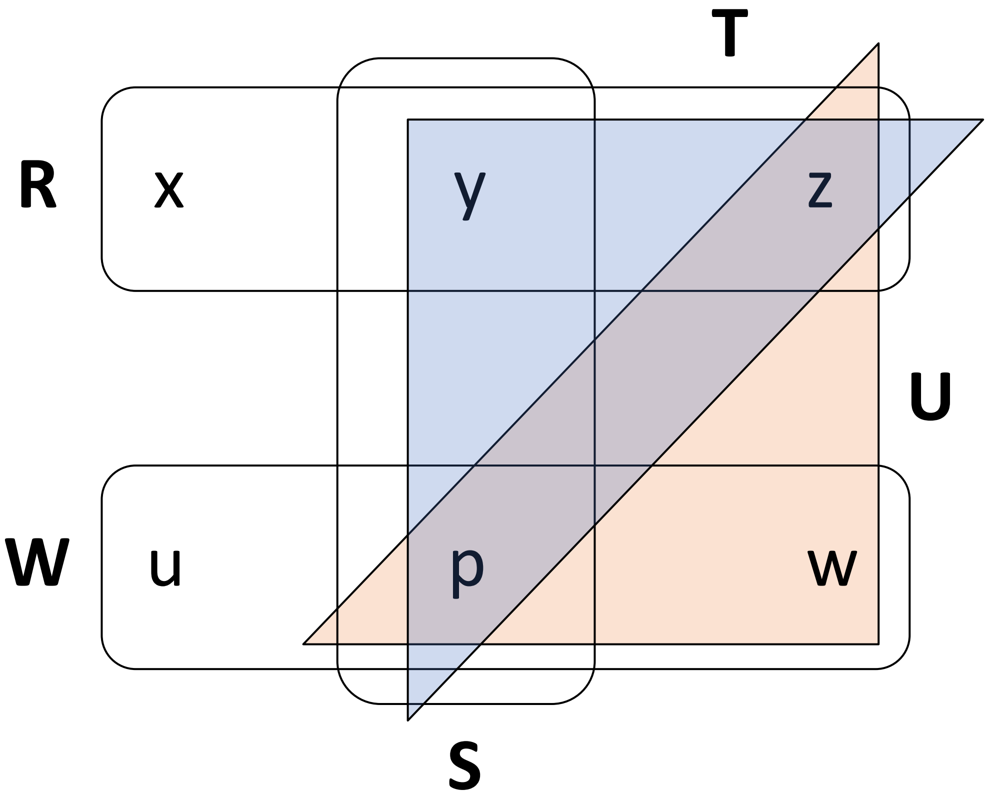
Q :- R(x,y,z), S(y,p), T(y,z,p), U(z,u,p), W(u,p,w).

GYO REDUCTION (ear removal)

- remove isolated nodes
- remove consumed or empty edges

Join tree

?



GYO reduction: Example 2

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

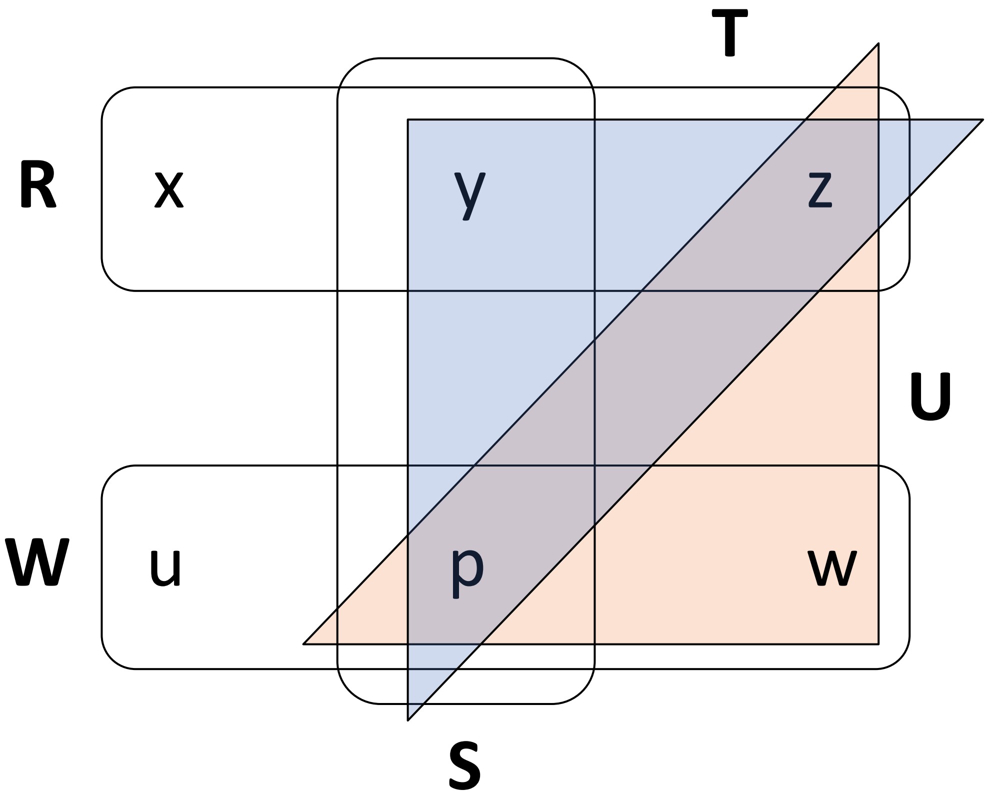
GYO REDUCTION (ear removal)

- 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: Example 2

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

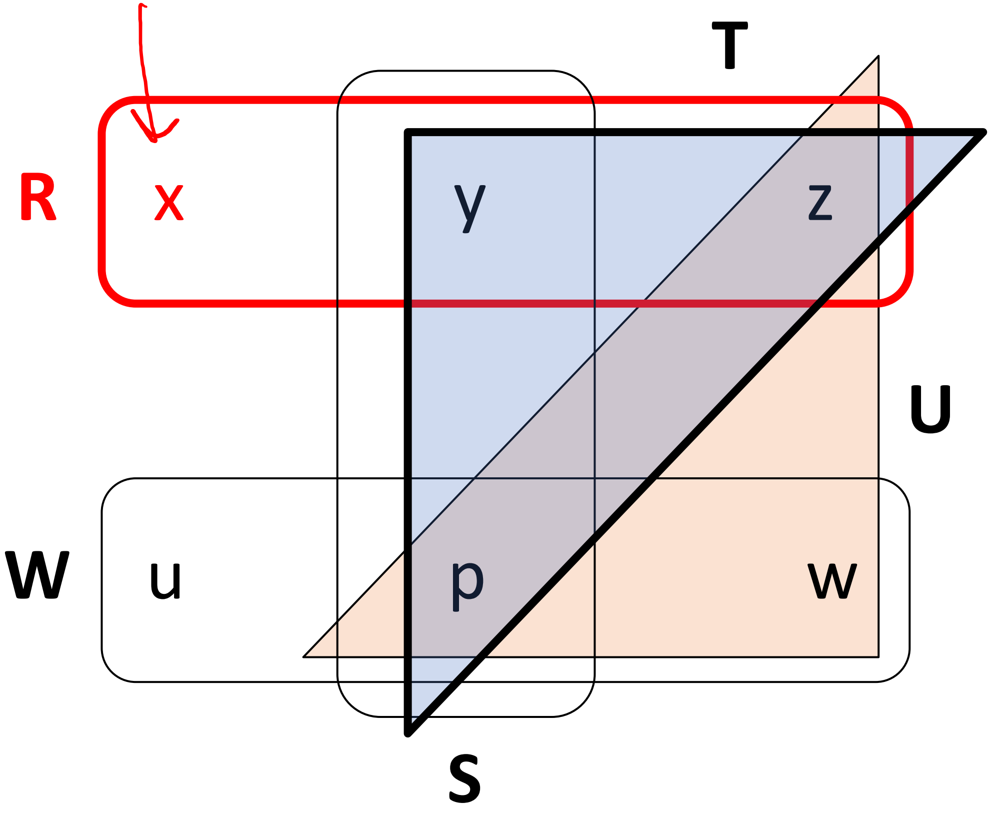
GYO REDUCTION (ear removal)

- 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)

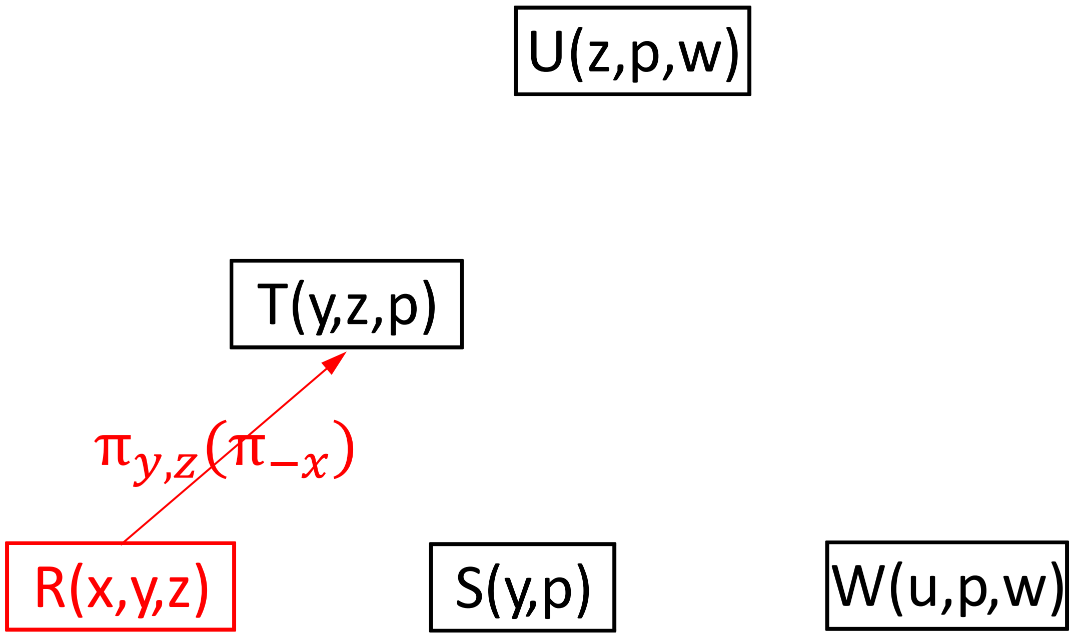


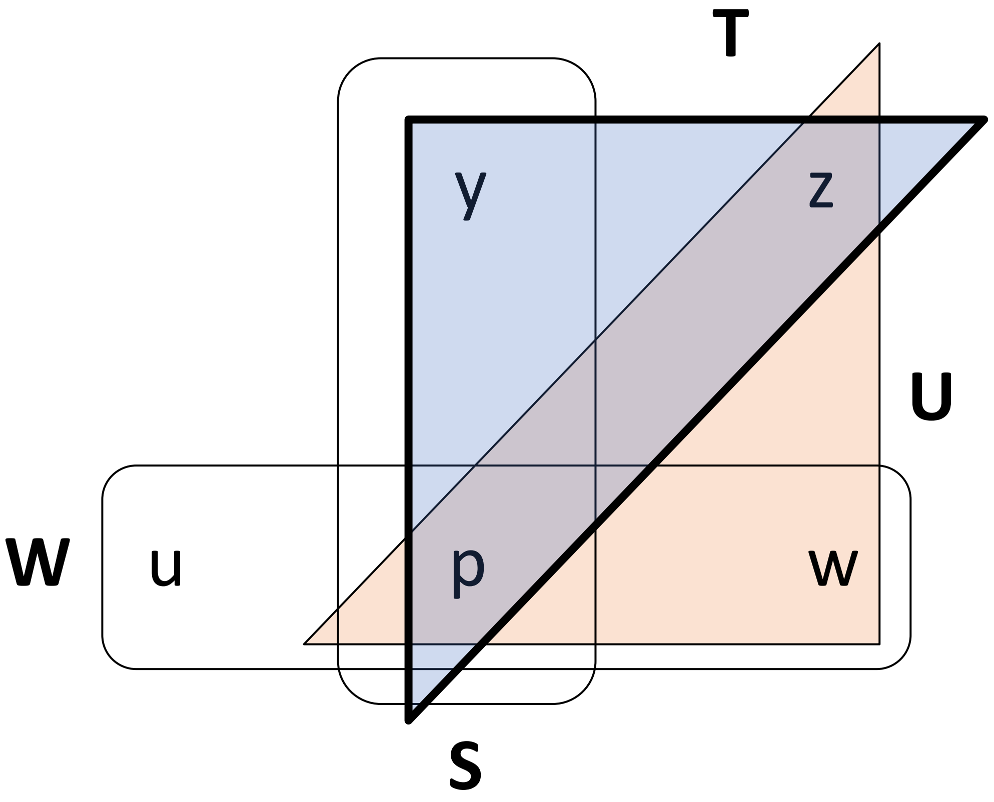
GYO reduction: Example 2

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



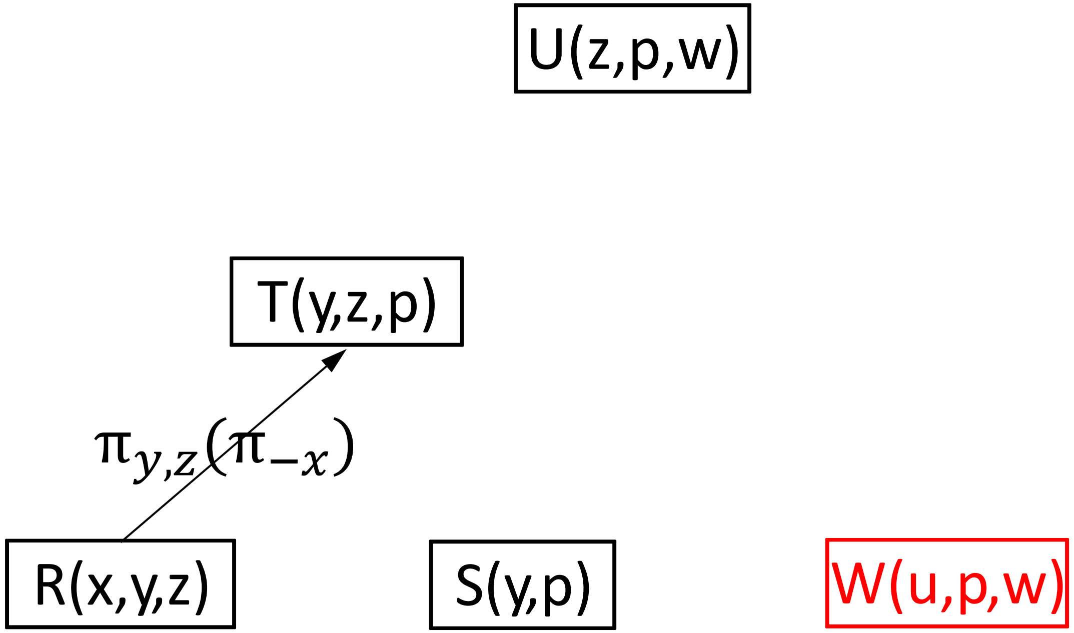


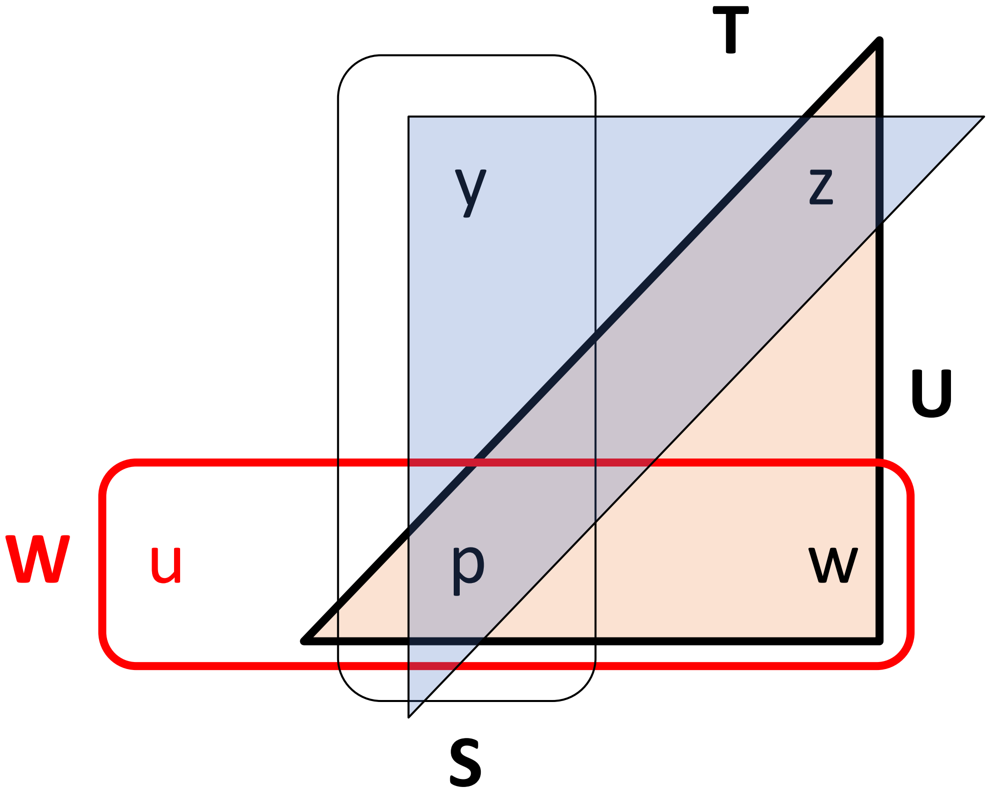
GYO reduction: Example 2

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



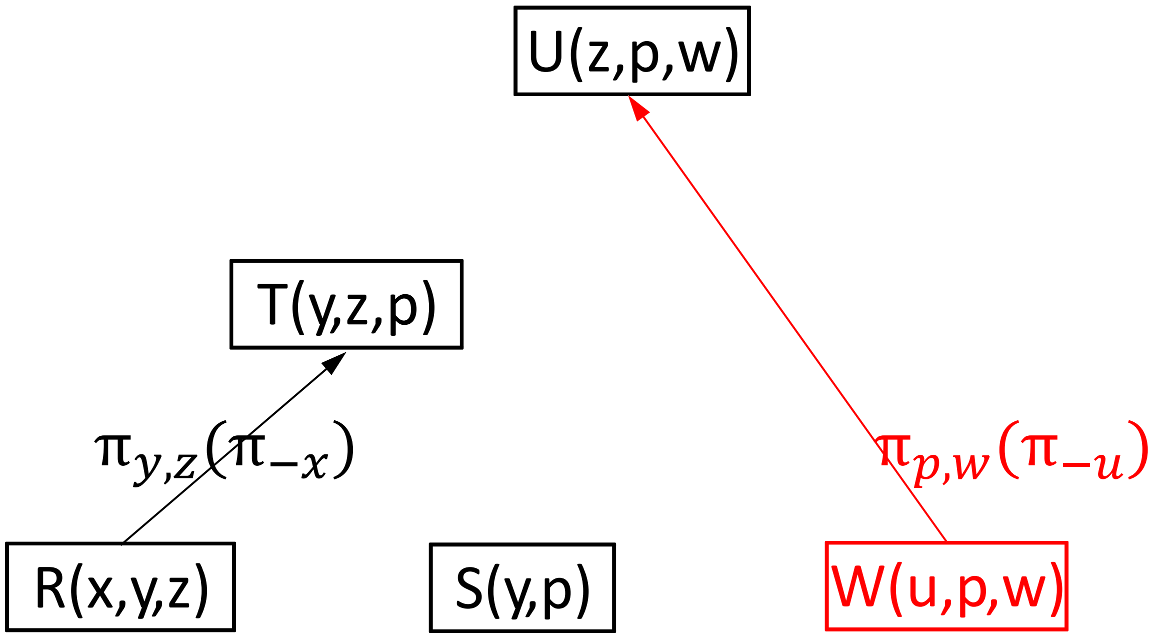


GYO reduction: Example 2

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)

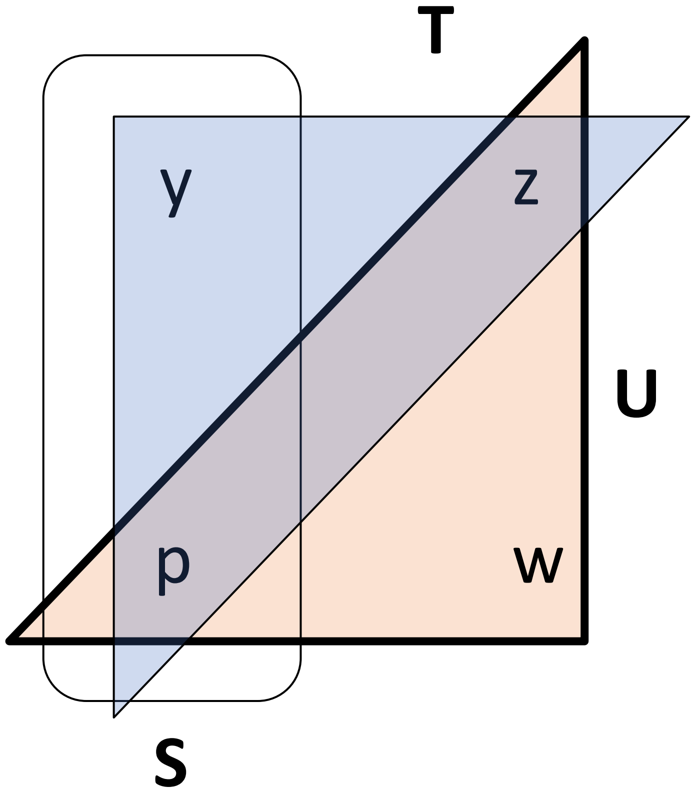
$\pi_{y,z}(\pi_{-x})$

$\pi_{p,w}(\pi_{-u})$

R(x,y,z)

S(y,p)

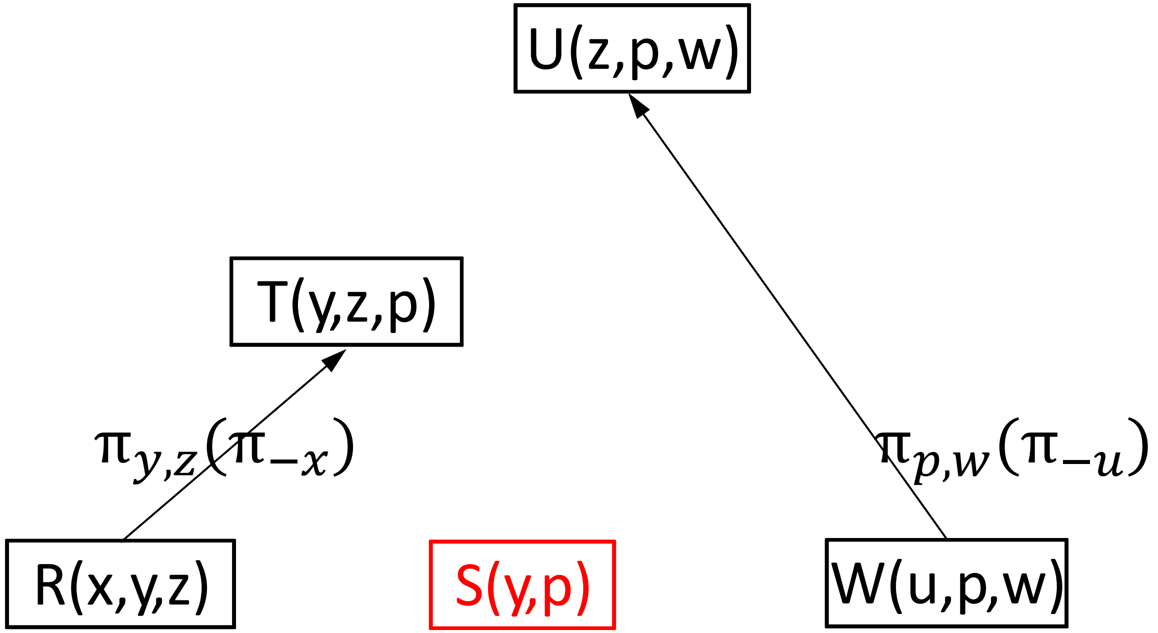
W(u,p,w)

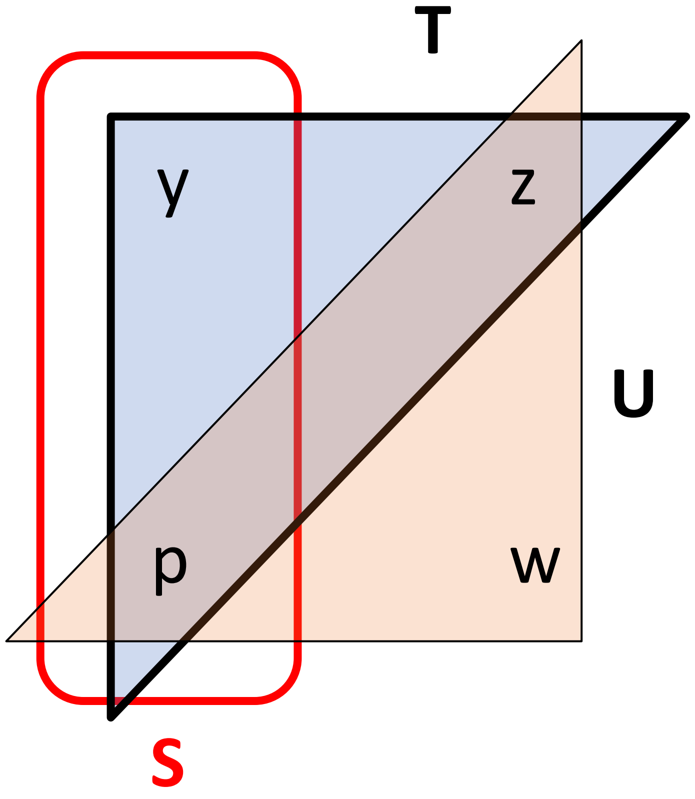


GYO reduction: Example 2

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



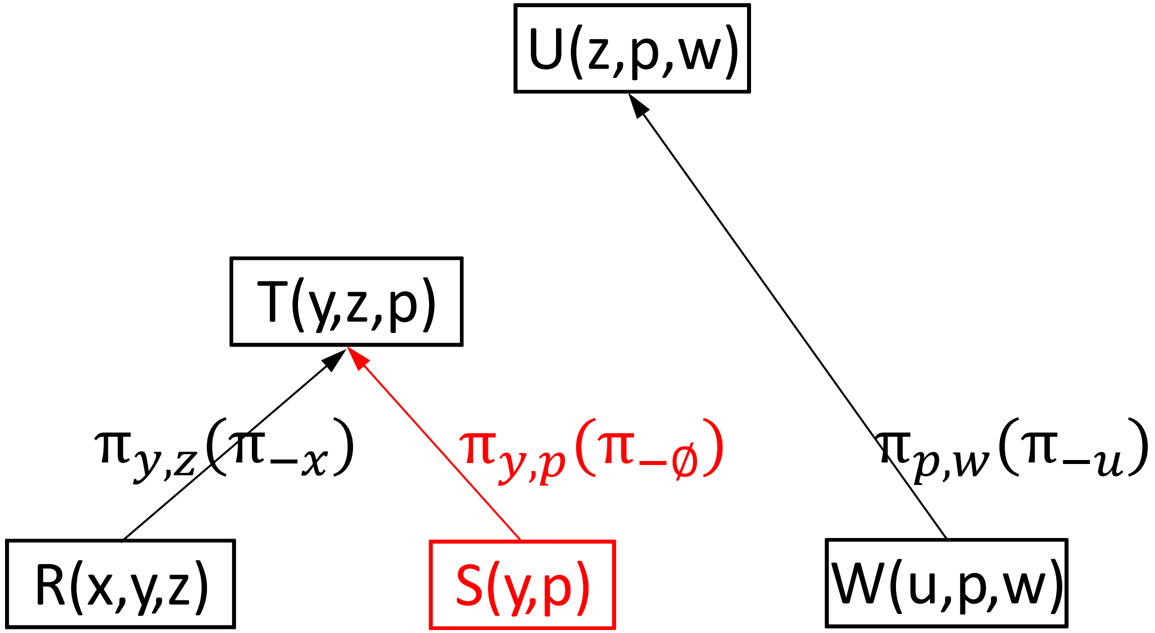


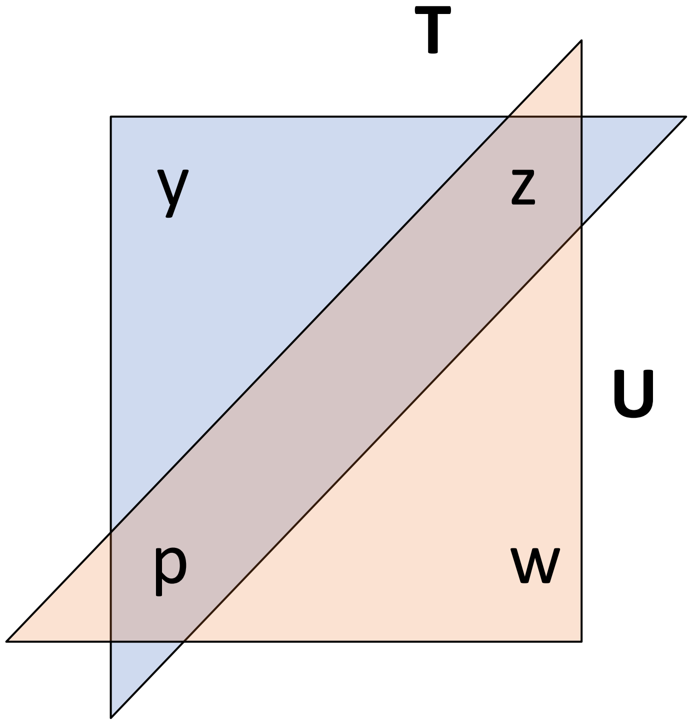
GYO reduction: Example 2

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



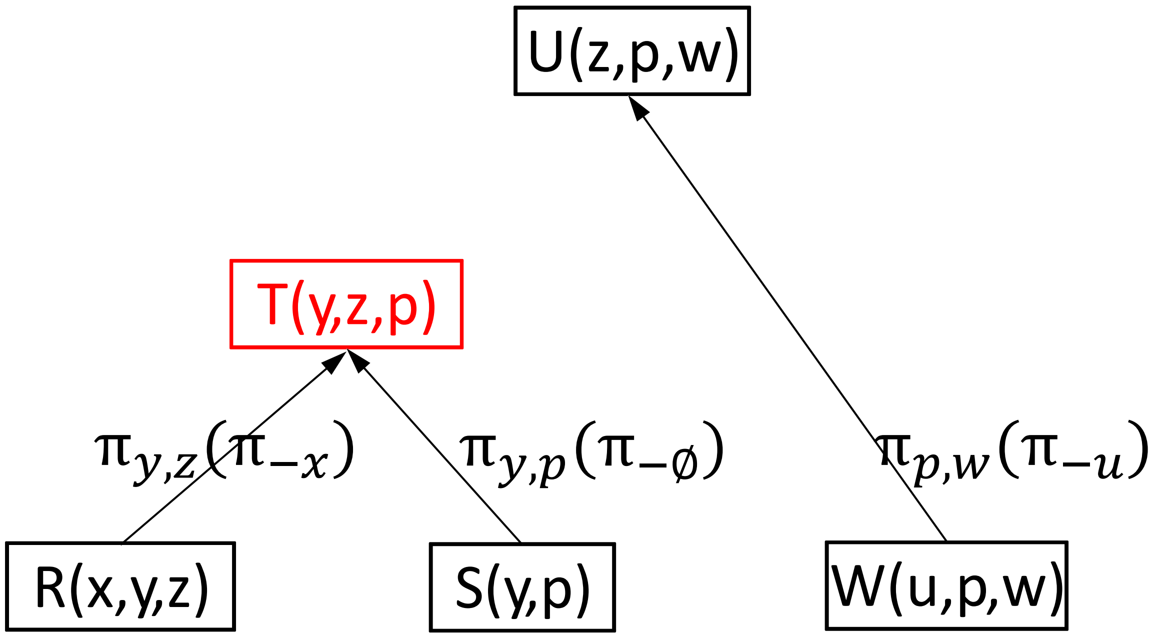


GYO reduction: Example 2

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)

$\pi_{y,z}(\pi_{-x})$

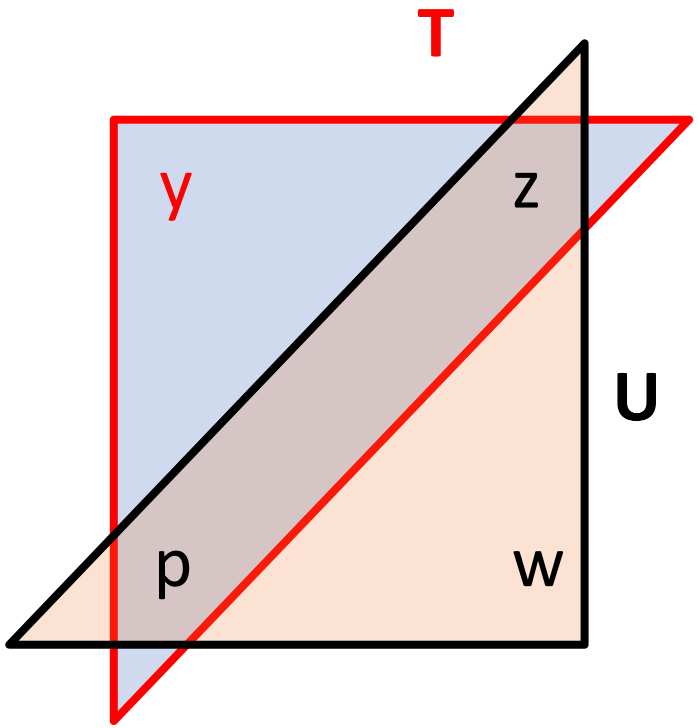
$\pi_{y,p}(\pi_{-\emptyset})$

$\pi_{p,w}(\pi_{-u})$

R(x,y,z)

S(y,p)

W(u,p,w)

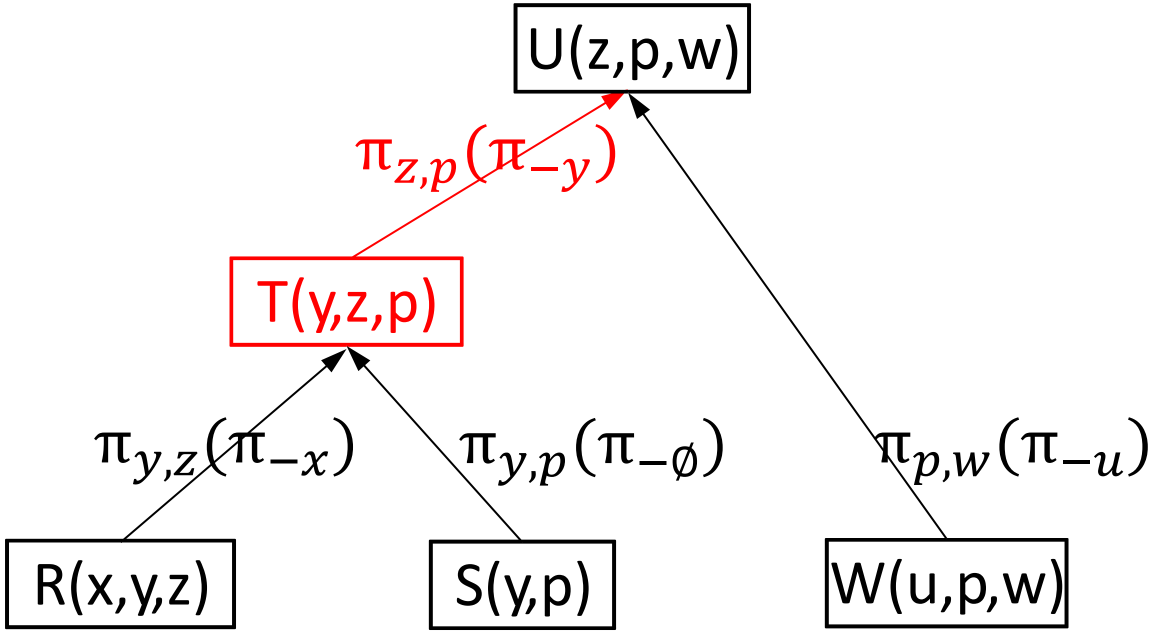


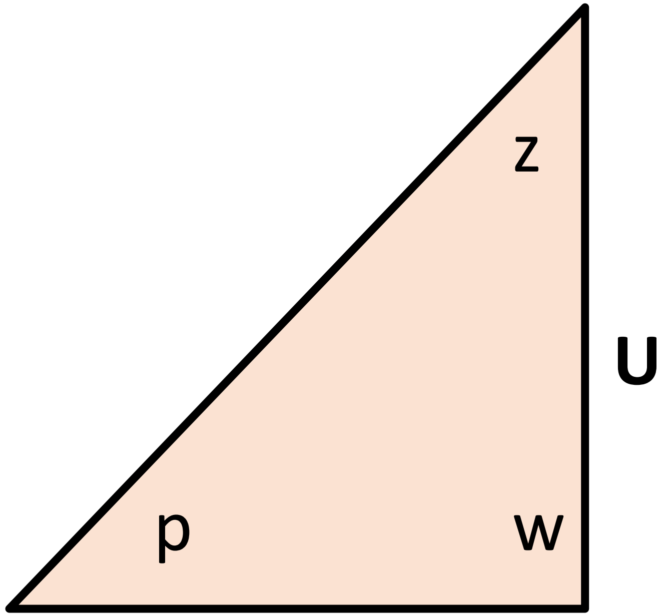
GYO reduction: Example 2

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



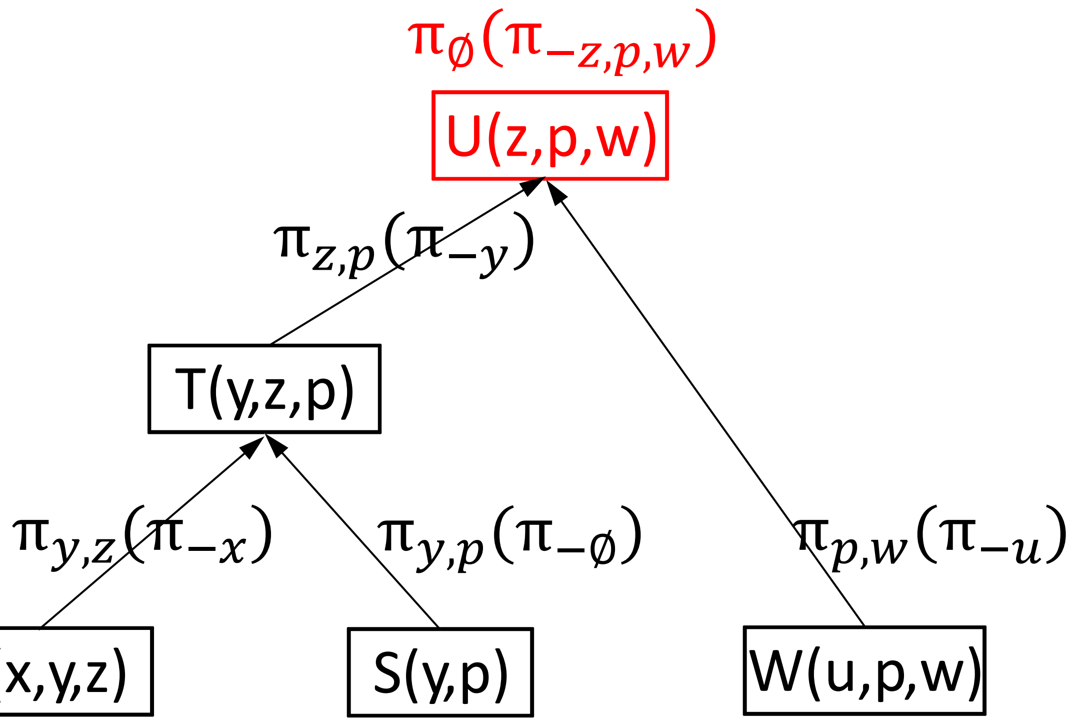


GYO reduction: Example 2

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



$\pi_{\emptyset}(\pi_{-z,p,w})$

U(z,p,w)

$\pi_{z,p}(\pi_{-y})$

T(y,z,p)

$\pi_{y,z}(\pi_{-x})$

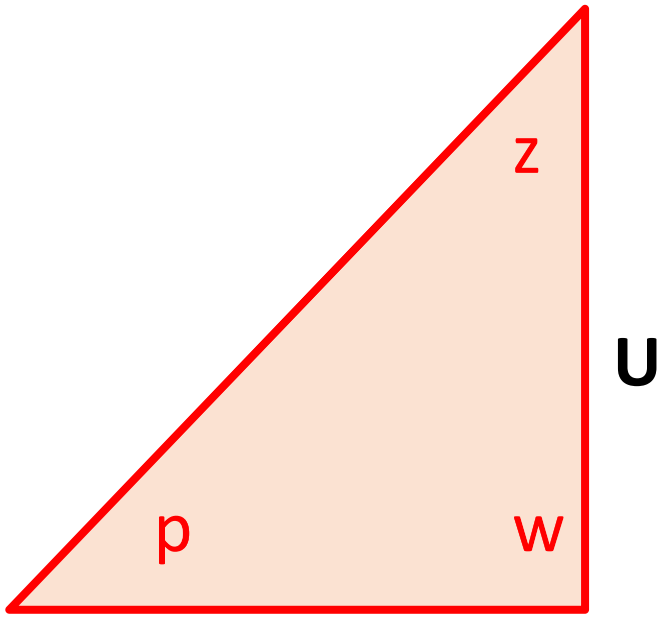
$\pi_{y,p}(\pi_{-\emptyset})$

$\pi_{p,w}(\pi_{-u})$

R(x,y,z)

S(y,p)

W(u,p,w)

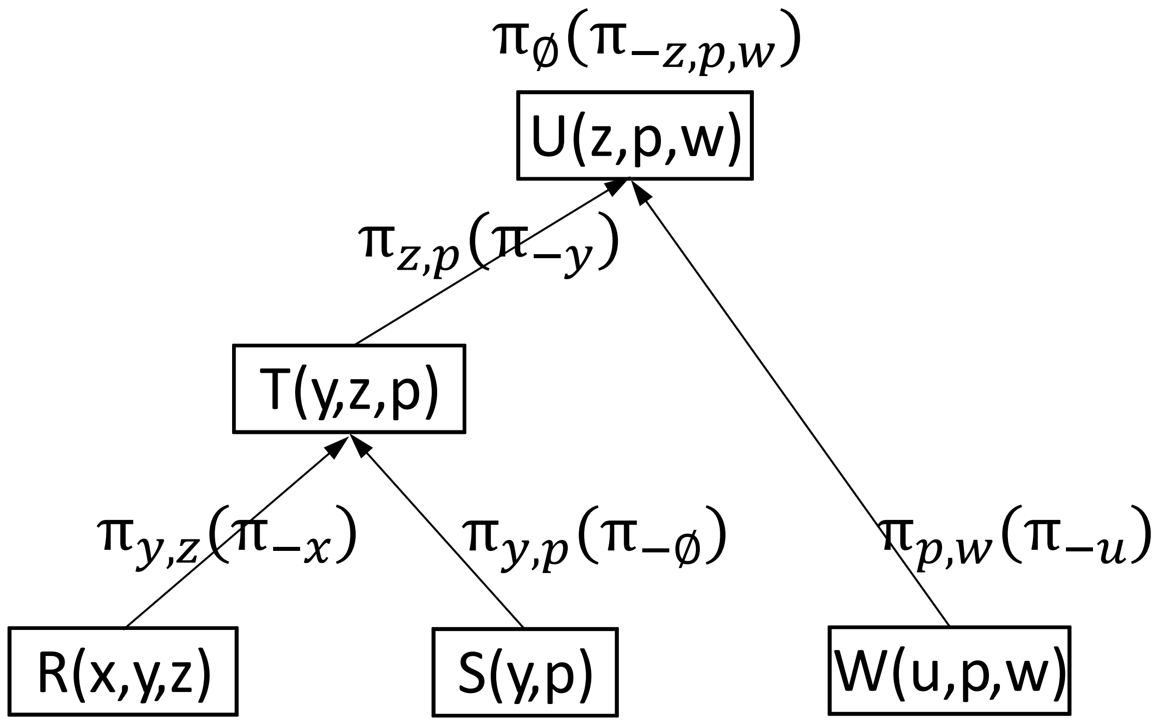


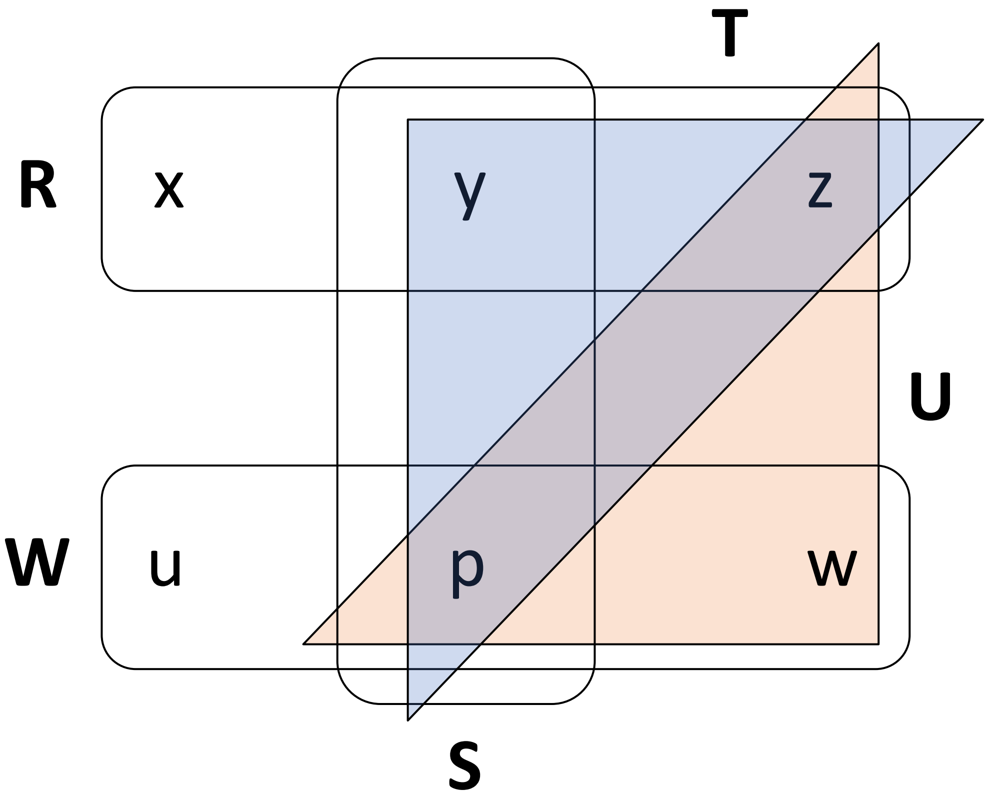
GYO reduction: Example 2

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



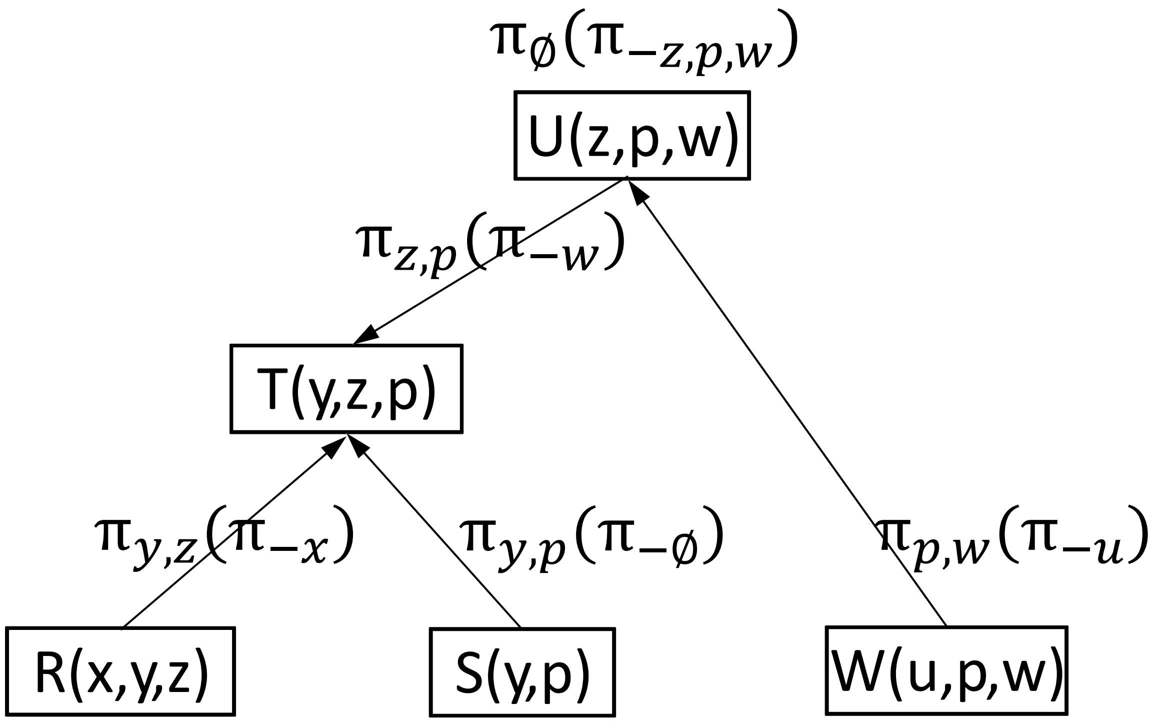


GYO reduction: Example 2

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



$\pi_{\emptyset}(\pi_{-z,p,w})$

U(z,p,w)

$\pi_{z,p}(\pi_{-w})$

T(y,z,p)

$\pi_{y,z}(\pi_{-x})$

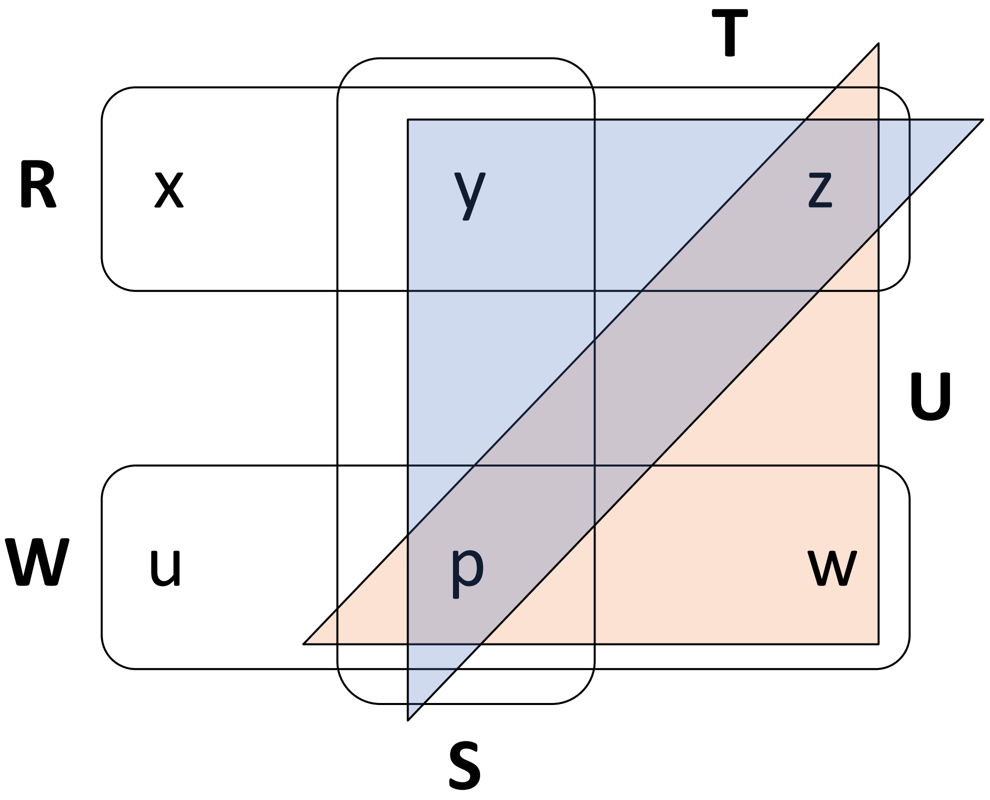
$\pi_{y,p}(\pi_{-\emptyset})$

$\pi_{p,w}(\pi_{-u})$

R(x,y,z)

S(y,p)

W(u,p,w)

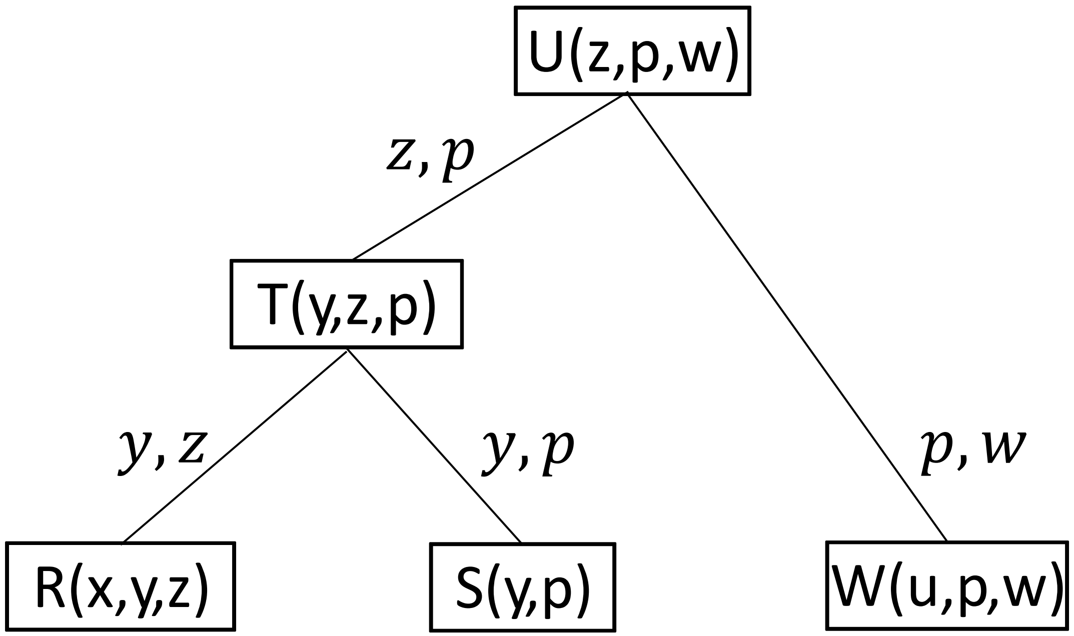


GYO reduction: Example 2

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 ☺



U(z,p,w)

z, p

p, w

T(y,z,p)

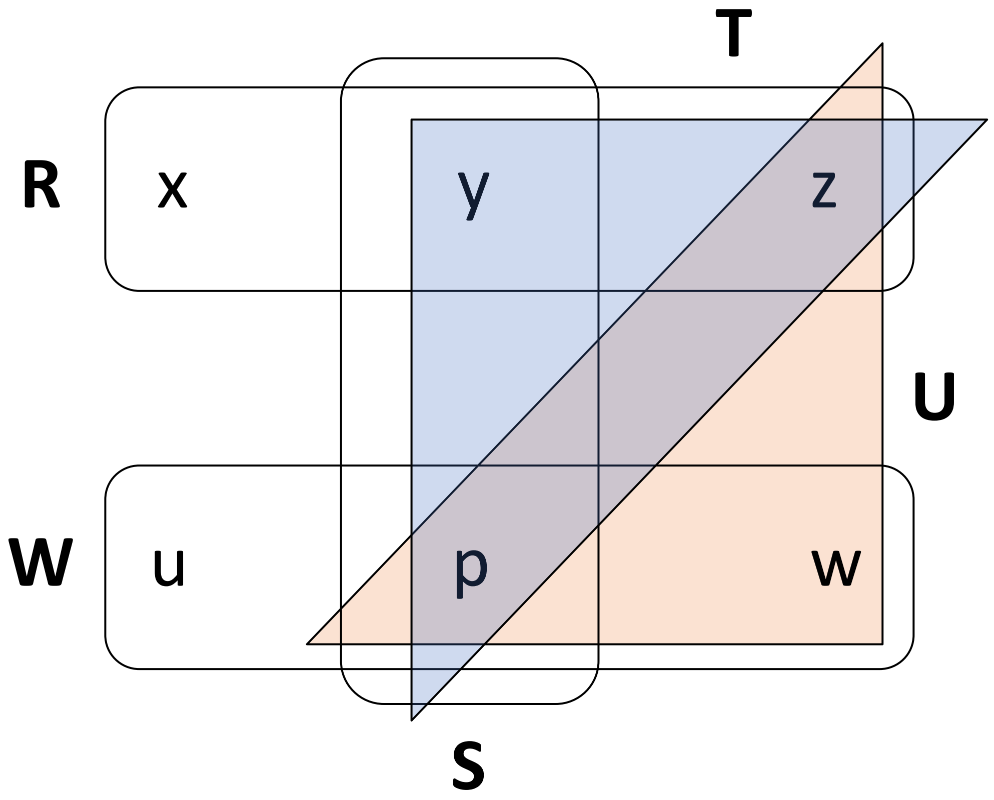
y, z

y, p

R(x,y,z)

S(y,p)

W(u,p,w)



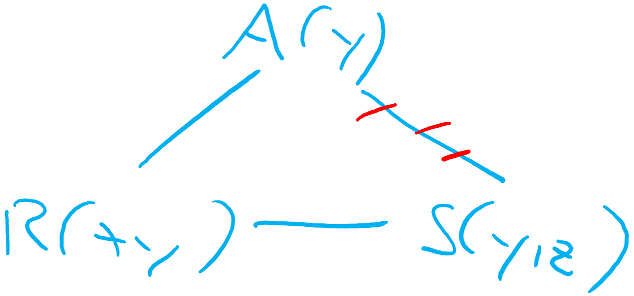
GYO reduction: Example 3: Are join trees unique?

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

?

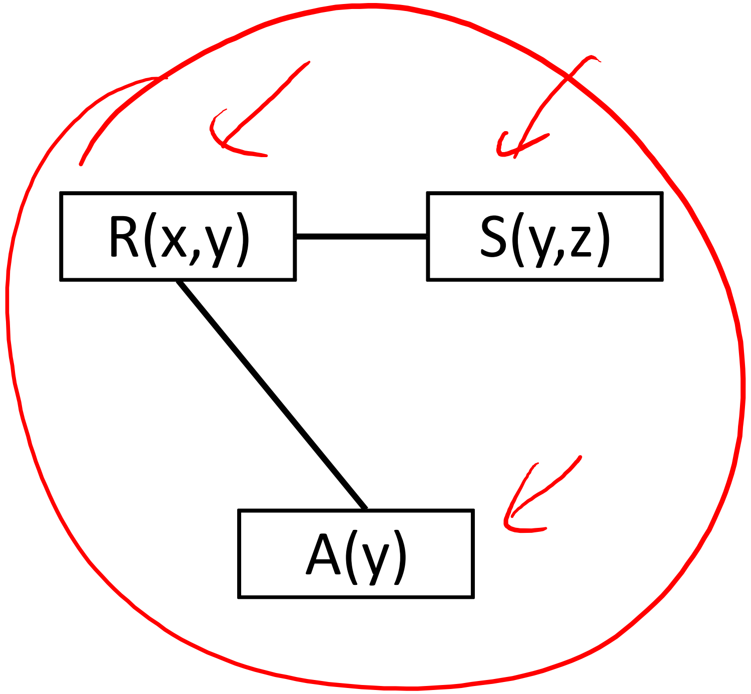
GYO reduction: Example 3: Are join trees unique?

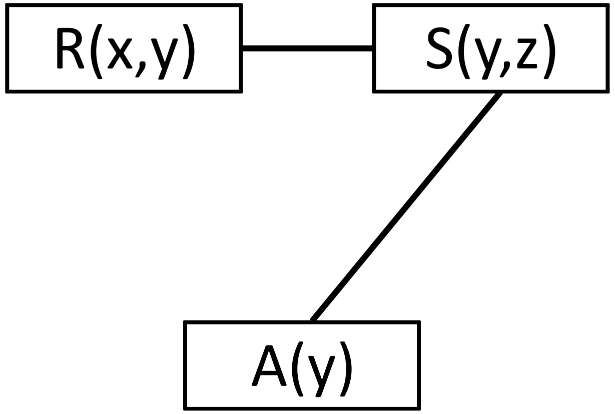
Q :- R(x,y), S(y,z), A(y).

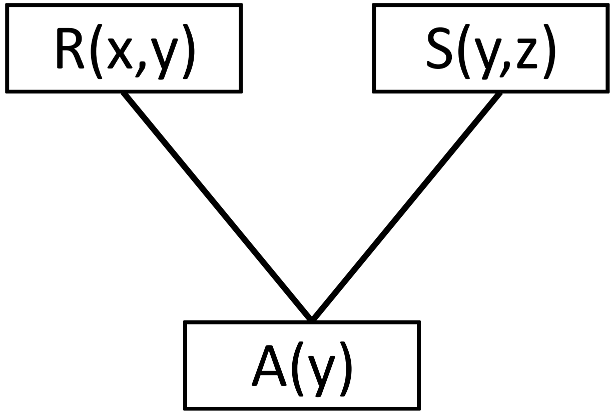


In addition, we have 3 choices as roots for each of the 3 different join trees. Leads to 9 different rooted join trees!

3 CHOICES FOR ROOT







Updated 3/24/2023

Topic 3: Efficient query evaluation
Unit 1: Acyclic query evaluation
Lecture 20

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp23)

https://northeastern-datalab.github.io/cs7240/sp23/

3/24/2023

Pre-class conversations

- Last class summary (+ many more details)
- Suggested new slide batching:
  - slide decks grouped by topics (& dates), not merely lectures
- Project: we can iterate, just approach me (P3: TUE, 3/28)
- Scribes: we are past halftime of the class (0-6 / 7)
  - please see my detailed comments, do also approach me after class with comments / questions / pointers. The goal is for the scribes and comments to be starters

- Today:
  - from semi-join to Yannakakis

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

Semijoin Reducer

$$Q(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,w)$$

DEFINITION: A full semijoin reducer (semijoin program) is:

a sequence of semijoins on the join tree

$R'(x,y) = R(y,z) \ltimes \ldots$

$S'(y,z) = S(y,z) \ltimes \ldots$

…

s.t. there no more "dangling tuples" in the reduced relations

Then you can rewrite the query over the reduced relations

$$Q(x,y,z) = R'(x,y) \bowtie S'(y,z) \bowtie T'(z,w)$$

Semijoin Reducer

$Q(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,w)$

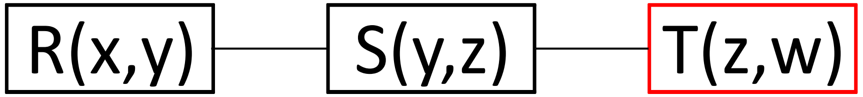
A full reducer is

?

Semijoin Reducer

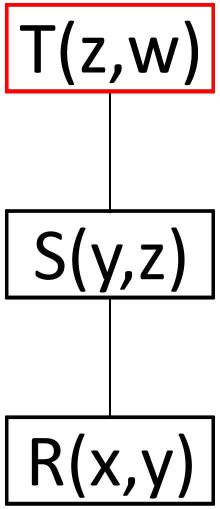
$$Q(x,y,z) = R(x, y) \bowtie S(y, z) \bowtie T(z, w)$$

A full reducer is


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

?

1. Find a join tree

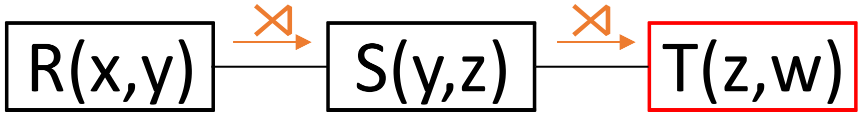

T(z,w)
S(y,z)
R(x,y)

2. pick a root

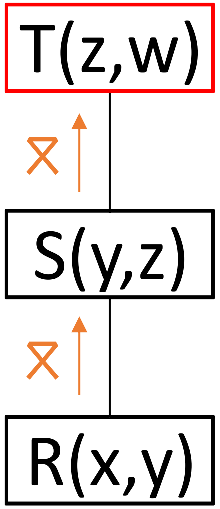
Semijoin Reducer

$$Q(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,w)$$

A full reducer is



1. Find a join tree



2. pick a root

3. collect at root (= bottom-up)

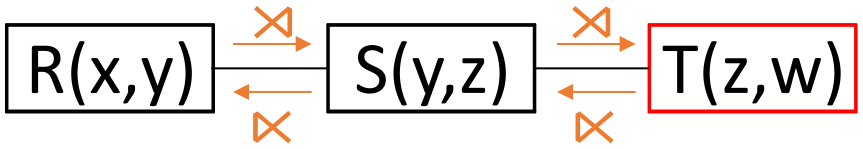
$$S_1(y,z) = S(y,z) \ltimes R(x,y)$$
$$T_1(z,w) = T(z,w) \ltimes S_1(y,z)$$

3. up

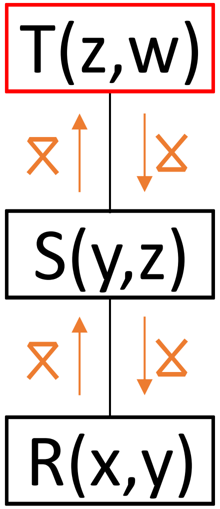
Semijoin Reducer

$$Q(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,w)$$

A full reducer is



1. Find a join tree



2. pick a root

3. up

4. down

3. collect at root (= bottom-up)

4. distribute to leaves (= top-down)

$$S_1(y,z) = S(y,z) \ltimes R(x,y)$$
$$T_1(z,w) = T(z,w) \ltimes S_1(y,z)$$
$$S_2(z,y) = S_1(y,z) \ltimes T_1(z,y)$$
$$R_1(x,y) = R(x,y) \ltimes S_2(y,z)$$

5. The rewritten query is

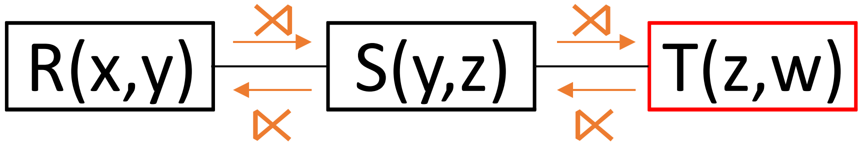
?

Semijoin Reducer

$$Q(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,w)$$

A full reducer is

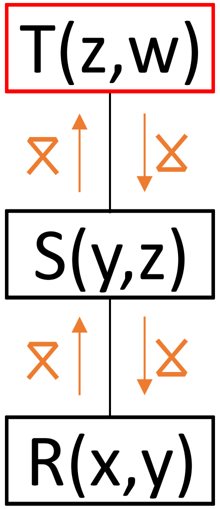
1. Find a join tree



3. collect at root (= bottom-up)

$$S_1(y,z) = S(y,z) \ltimes R(x,y)$$
$$T_1(z,w) = T(z,w) \ltimes S_1(y,z)$$
$$S_2(z,y) = S_1(y,z) \ltimes T_1(z,y)$$
$$R_1(x,y) = R(x,y) \ltimes S_2(y,z)$$

4. distribute to leaves (= top-down)



2. pick a root

3. up

4. down

5. The rewritten query is

$$Q(x,y,z) = R_1(x,y) \bowtie S_2(y,z) \bowtie T_1(z,w)$$

Notice that with 2(k-1) messages, every of the k tables has received information from every other table!

Application

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).

R(x,y)

x	y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_2
a_2	b_3
a_3	b_1
a_3	b_2
a_3	b_3

S(y,z)

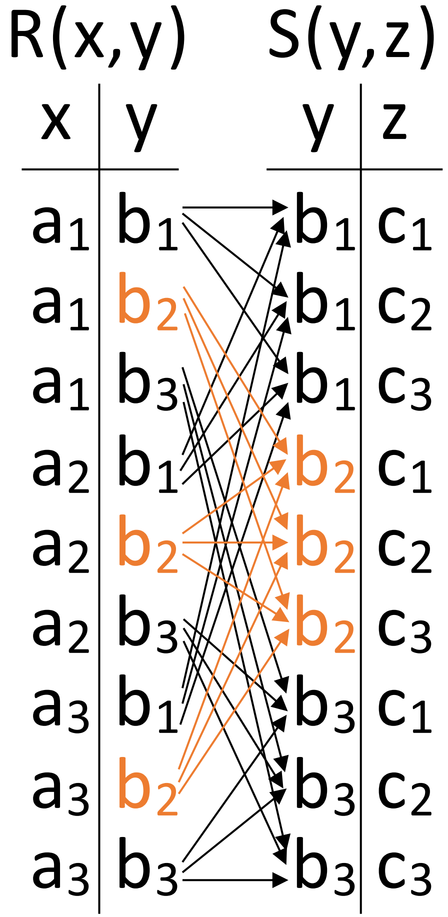
y	z
b_1	c_1
b_1	c_2
b_1	c_3
b_2	c_1
b_2	c_2
b_2	c_3
b_3	c_1
b_3	c_2
b_3	c_3

How many joins are between R and S?

?

Application

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_2
a_2	b_3
a_3	b_1
a_3	b_2
a_3	b_3

S(y,z)

y	z
b_1	c_1
b_1	c_2
b_1	c_3
b_2	c_1
b_2	c_2
b_2	c_3
b_3	c_1
b_3	c_2
b_3	c_3

How many joins are between R and S?

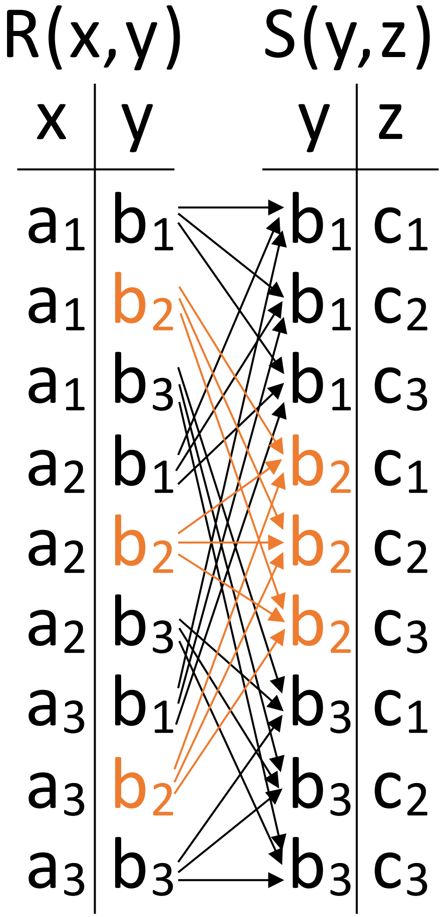
27

How can we represent the joins in a graph more compactly?

?

Application

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_2
a_2	b_3
a_3	b_1
a_3	b_2
a_3	b_3

S(y,z)

y	z
b_1	c_1
b_1	c_2
b_1	c_3
b_2	c_1
b_2	c_2
b_2	c_3
b_3	c_1
b_3	c_2
b_3	c_3

How many joins are between R and S?

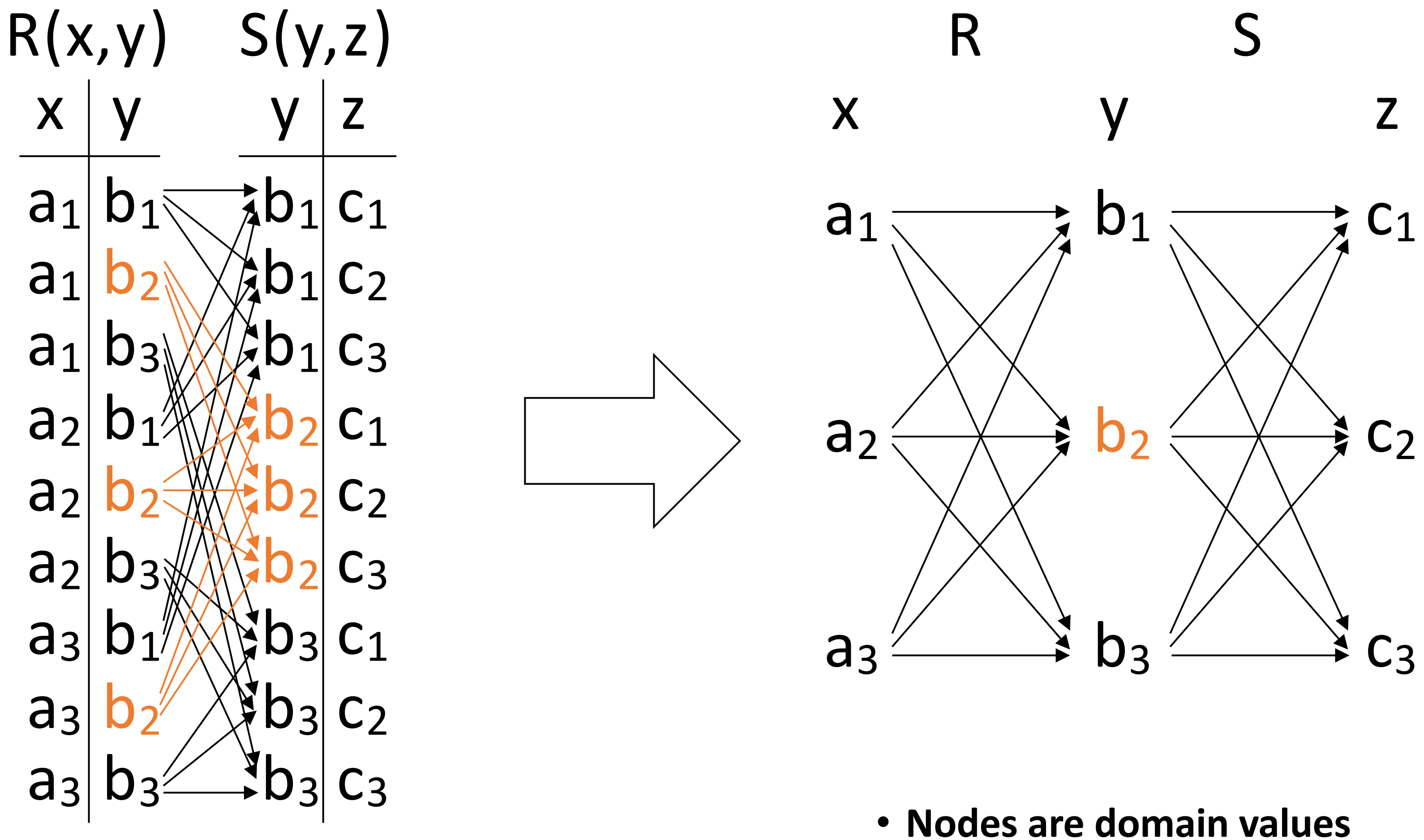
27

How can we represent the joins in a graph more compactly?

By using "the dual" (nodes as joins = domain values) and counting paths!

Application

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_2
a_2	b_3
a_3	b_1
a_3	b_2
a_3	b_3

S(y,z)

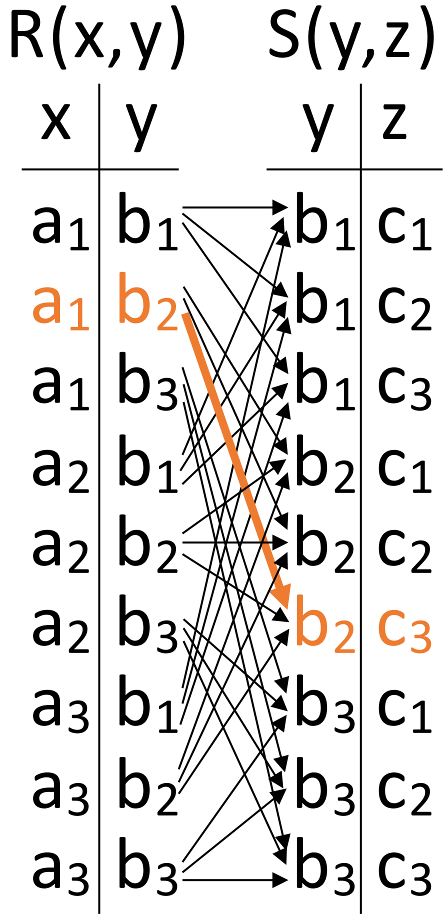
y	z
b_1	c_1
b_1	c_2
b_1	c_3
b_2	c_1
b_2	c_2
b_2	c_3
b_3	c_1
b_3	c_2
b_3	c_3

- **Nodes are the tuples**
- Edges are the joins

- **Nodes are domain values**
- Edges are the tuples
- Equi-joins b/w tuples as shared domain

Application

$Q(x,y,z,u,v,w) \text{ :- } R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).$



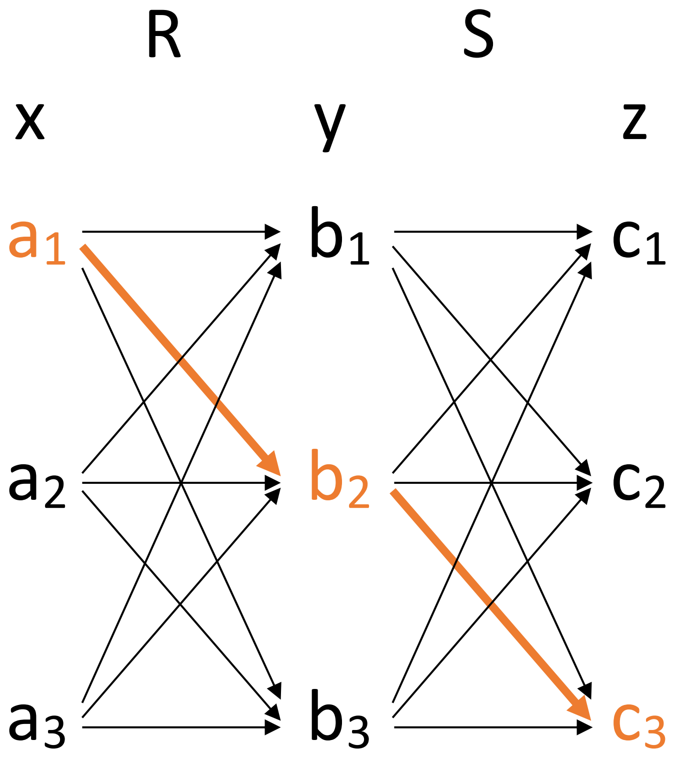
R(x,y)

x	y
a_1	b_1
a_1	b_2
a_1	b_3
a_2	b_1
a_2	b_2
a_2	b_3
a_3	b_1
a_3	b_2
a_3	b_3

S(y,z)

y	z
b_1	c_1
b_1	c_2
b_1	c_3
b_2	c_1
b_2	c_2
b_2	c_3
b_3	c_1
b_3	c_2
b_3	c_3

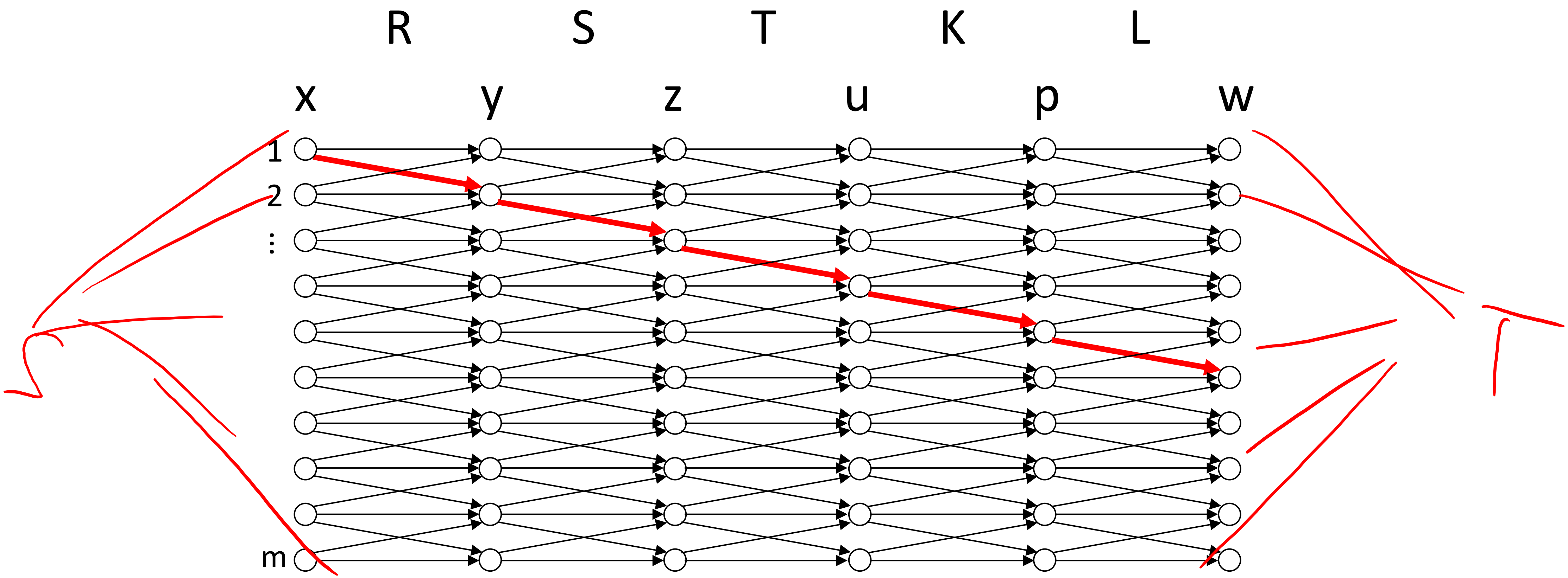
- **Nodes are the tuples**
- Edges are the joins



- **Nodes are domain values**
- Edges are the tuples
- Equi-joins b/w tuples as shared domain

Application

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).



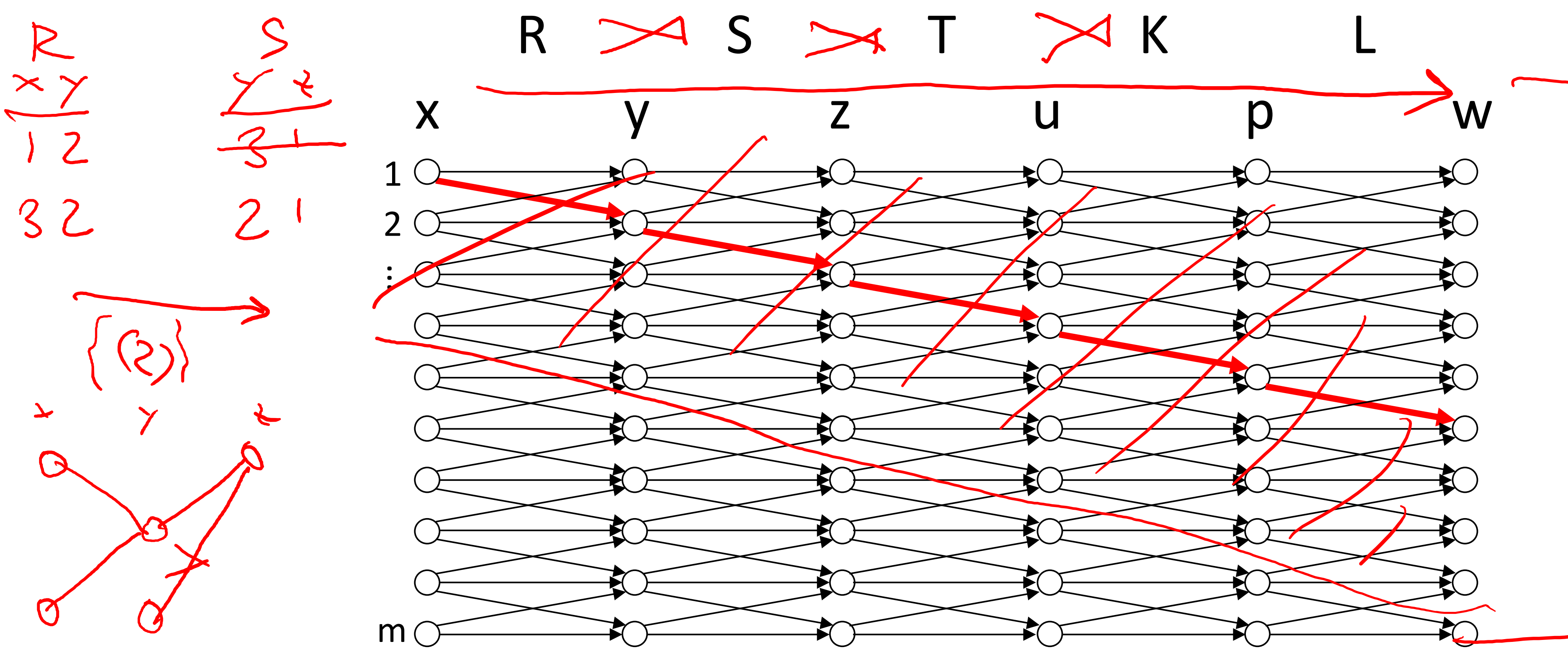
- Intermediate relations of size up to m^5 (m is here domain size!)
- But final answer can be as small as 0 (or 1...)

Application

$\Pi_y = \Pi_x$

Q(x,y,z,u,v,w) :- R(x,y),S(y,z),T(z,u),K(u,v),L(v,w).

Semi-join reduction allows pre-processing in O(n) where n is size of DB (= number of edges)



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

FWD

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

BACK

- Intermediate relations of size up to m^5 (m is here domain size!)
- But final answer can be as small as 0 (or 1...)

What is so special about acyclic queries?

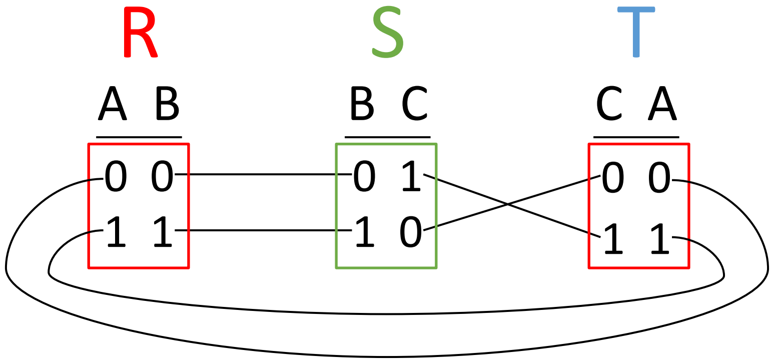
- "Consistency" = no dangling tuples (recall semi-join)
  - locally join consistent (every tuple participates in pairwise joins)

    $$\pi_{Ri}(R_i \bowtie R_j) = R_i$$

  - globally join consistent (every tuple participates in the full join)

    $$\pi_{Ri}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k) = R_i$$

- Both imply each other only for acyclic queries
  - For all queries: global ⇒ local
  - Acyclic queries: local ⇒ global; but not for cyclic queries!



R

A	B
0	0
1	1

S

B	C
0	1
1	0

T

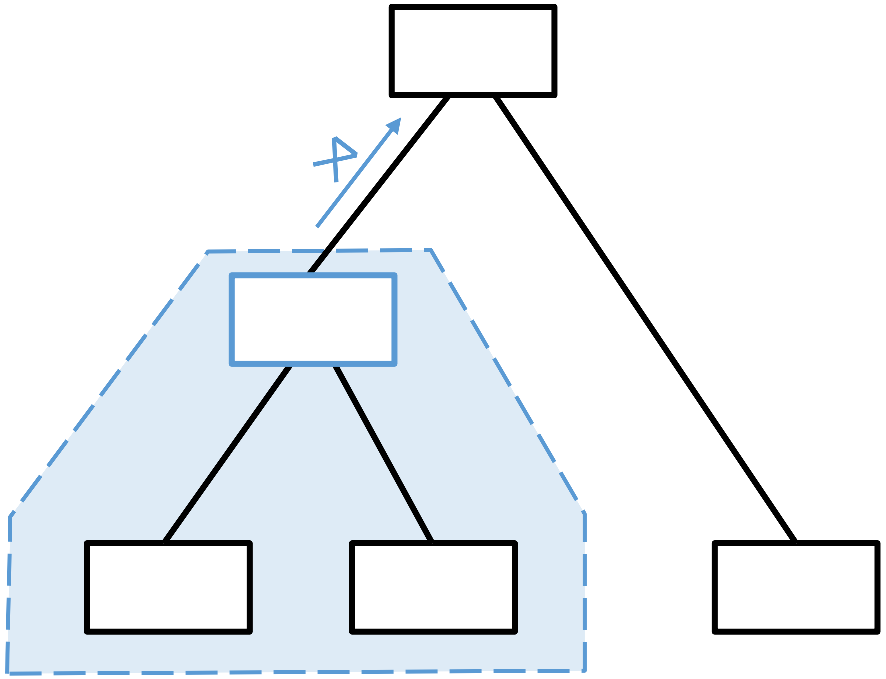
C	A
0	0
1	1

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

- 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 $R \bowtie S \bowtie T$ = Ø, 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

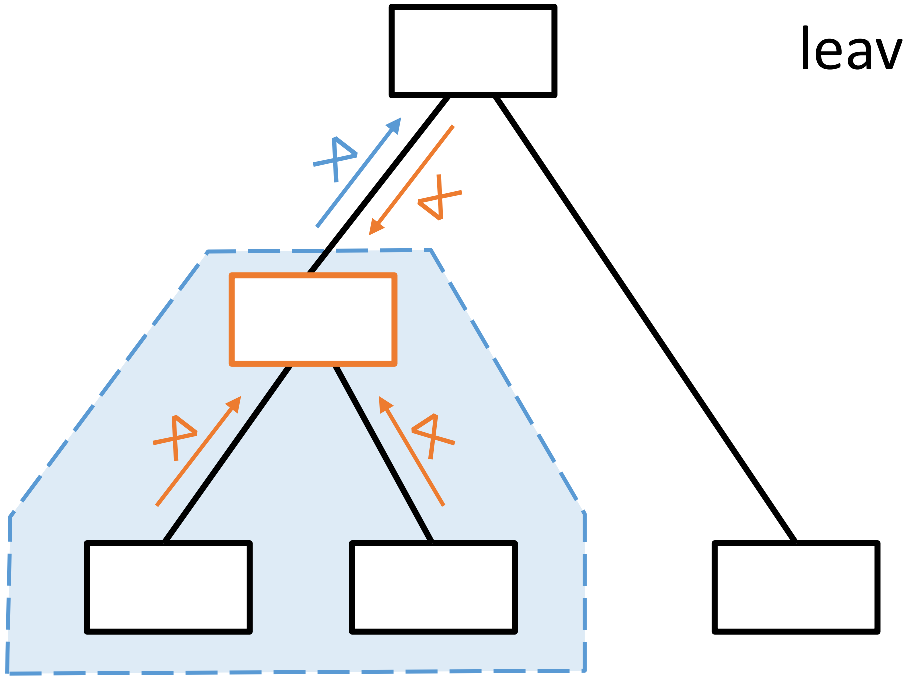
Semijoin Reduction as Message Passing on Trees



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!

Semijoin Reduction as Message Passing on Trees



Every relation (node) will be reduced d times where d is the degree of the node. Thus leaves (also the root) are reduced only once)

Each edge transmits 2 messages: one up and one down.

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

Join tree

?

Query hypergraph

?

Semi-join reducers fail with the triangle

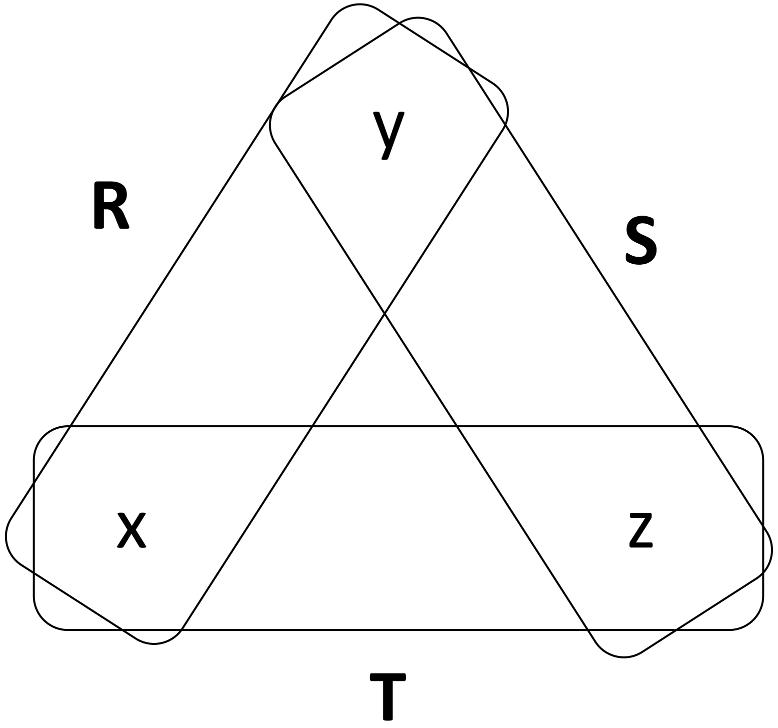
Q(x,y,z) :- R(x,y), S(y,z), T(x,z).

GYO reduction (ear removal)

- remove isolated nodes
- remove consumed or empty edges

Join tree

?

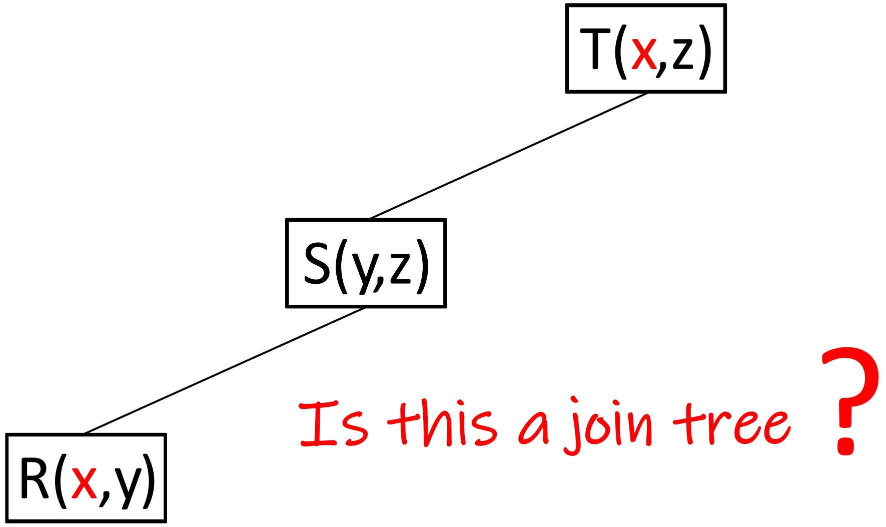


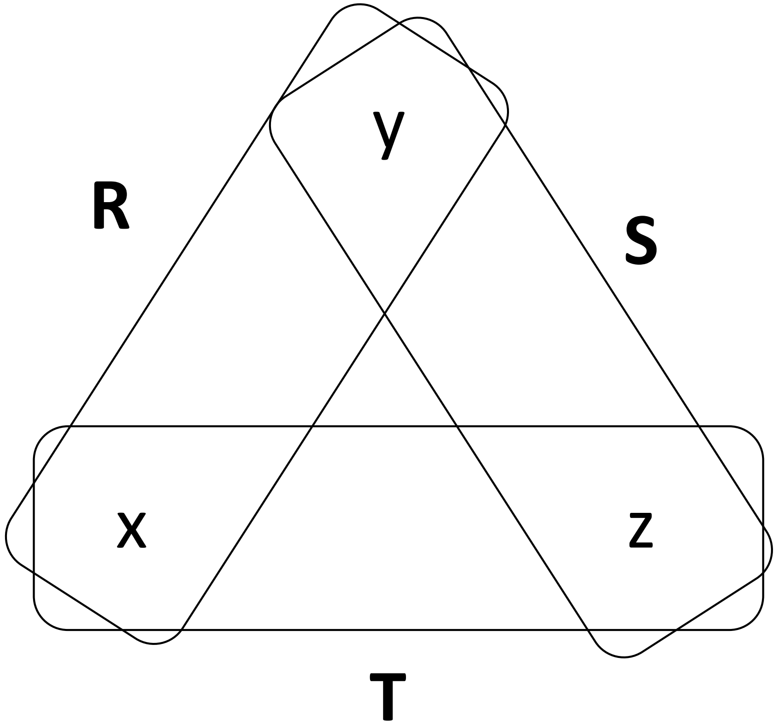
Semi-join reducers fail with the triangle

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

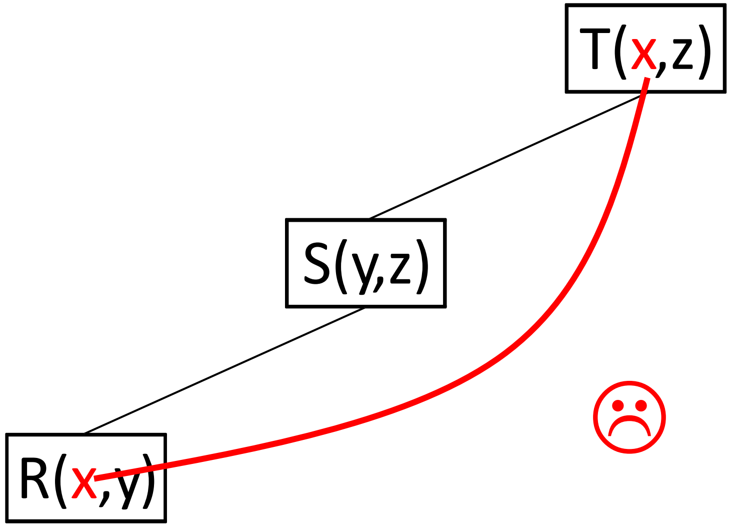


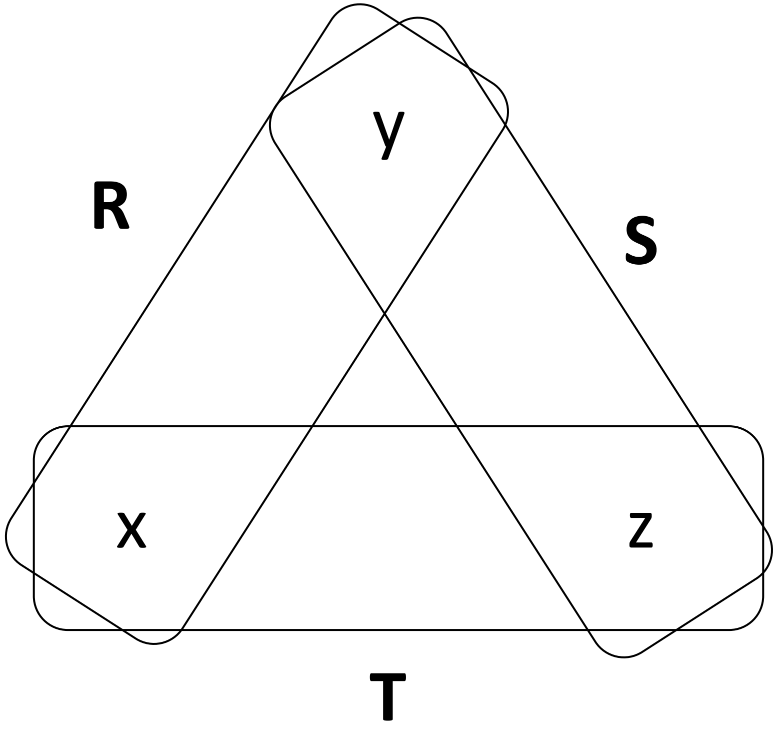


Semi-join reducers fail with the triangle

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...

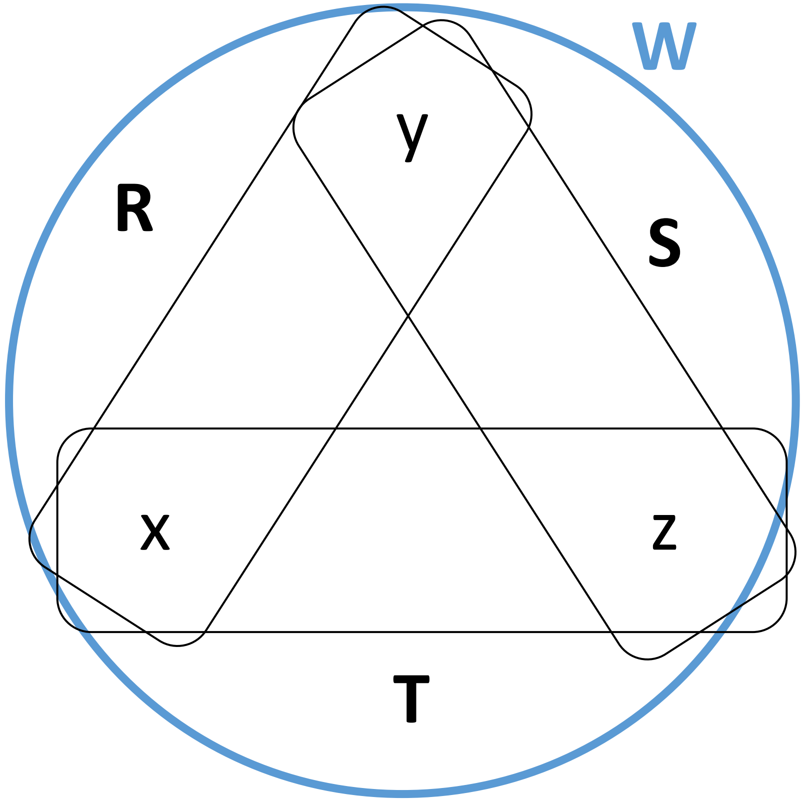
Semi-join reducers work with the "beta-triangle"

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

Join tree

?

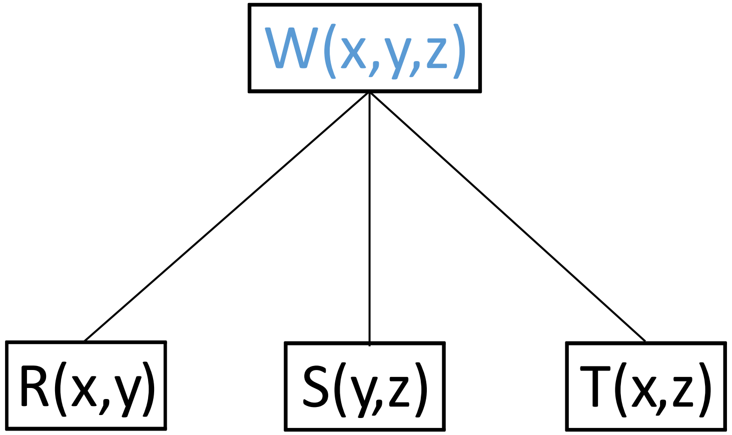


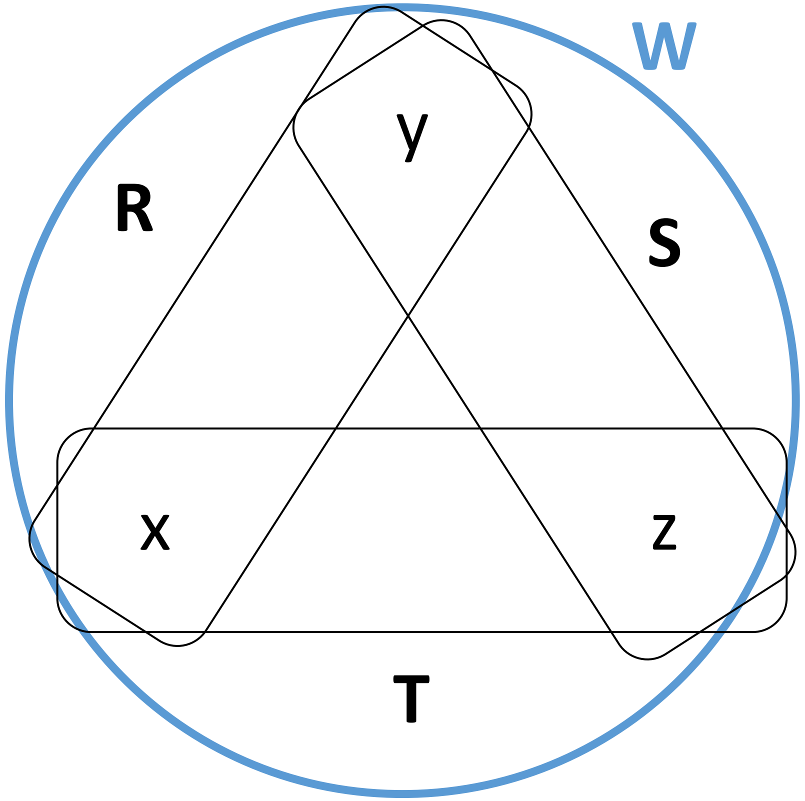
Semi-join reducers work with the "beta-triangle"

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

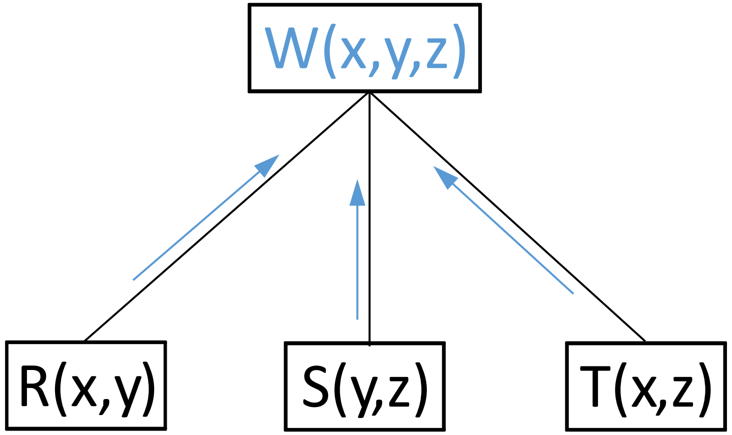




Next: Write 1) a full reducer and then 2) the new join expression in RA ?

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(x,y,z)

R(x,y)

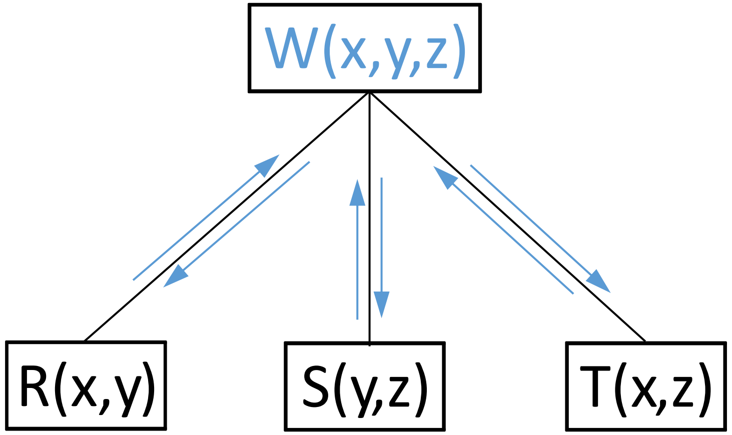
S(y,z)

T(x,z)

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

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).



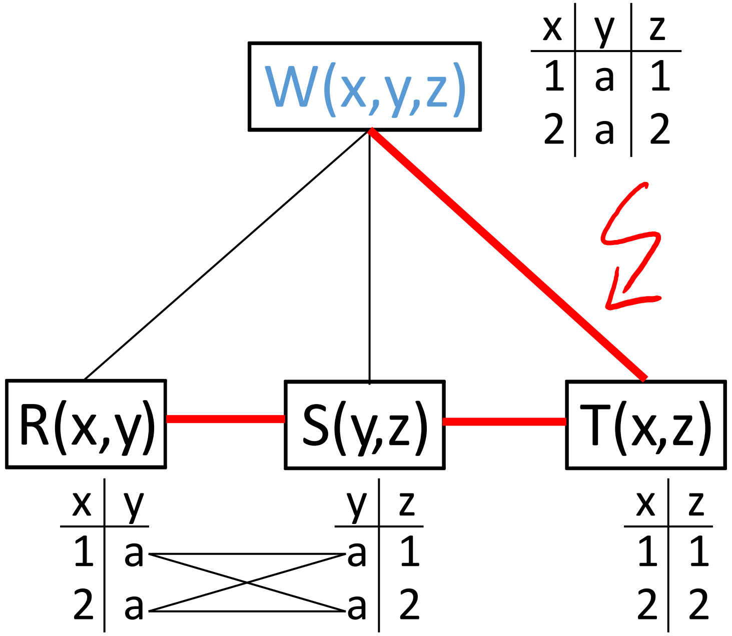
$$W_1(x,y,z) = W(x,y,z) \ltimes R(x,y)$$
$$W_2(x,y,z) = W_1(x,y,z) \ltimes S(y,z)$$
$$W_3(x,y,z) = W_2(x,y,z) \ltimes T(x,z)$$
$$R_1(x,y) = R(x,y) \ltimes W_3(x,y,z)$$
$$S_1(y,z) = S(y,z) \ltimes W_3(x,y,z)$$
$$T_1(x,z) = T(x,z) \ltimes W_3(x,y,z)$$

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

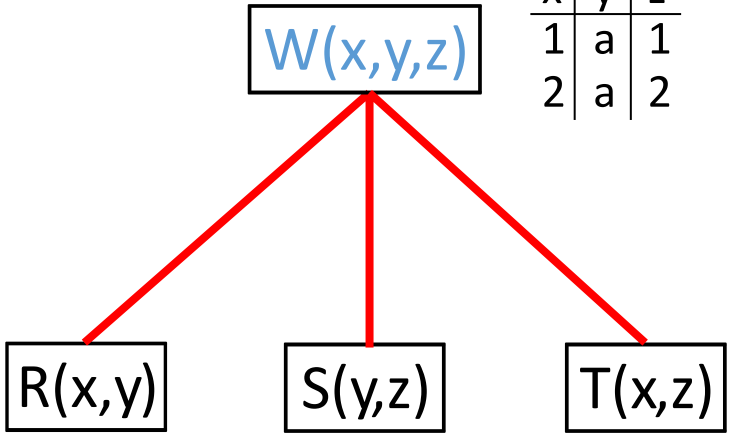
$$W_1(x,y,z) = W(x,y,z) \ltimes R(x,y)$$
$$W_2(x,y,z) = W_1(x,y,z) \ltimes S(y,z)$$
$$W_3(x,y,z) = W_2(x,y,z) \ltimes T(x,z)$$
$$R_1(x,y) = R(x,y) \ltimes W_3(x,y,z)$$
$$S_1(y,z) = S(y,z) \ltimes W_3(x,y,z)$$
$$T_1(x,z) = T(x,z) \ltimes W_3(x,y,z)$$

$$Q(x,y,z) = \Big(\underbrace{\big(R_1(x,y) \bowtie S_1(y,z)\big)}_{\begin{array}{ccc} 1 & a & 1 \\ 2 & a & 2 \\ 1 & a & 1 \\ 2 & a & 2 \end{array}} \bowtie T_1(x,z)\Big) \bowtie W_3(x,y,z)$$

☹ #=4

… 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

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

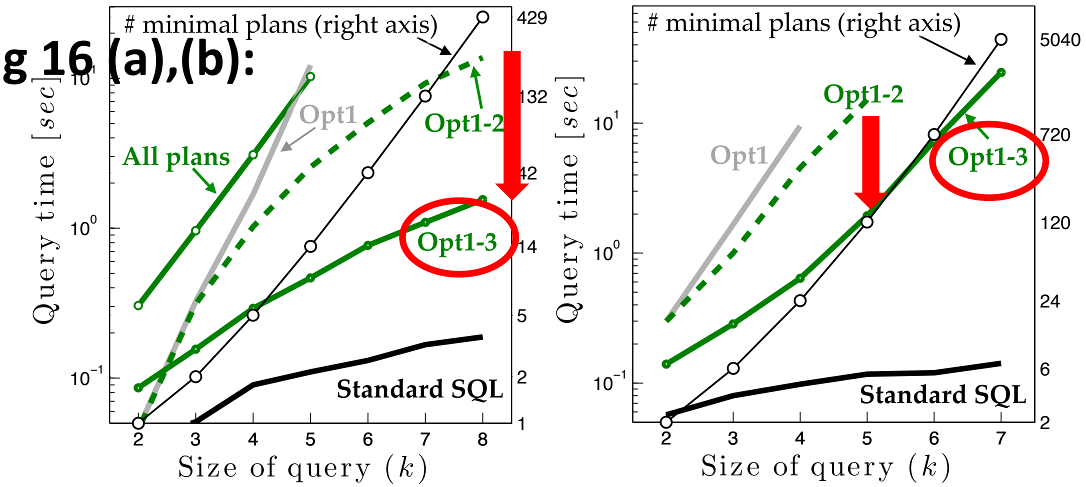
$$\xcancel{Q(x,y,z) = \Big(\big(R_1(x,y) \bowtie S_1(y,z)\big) \bowtie T_1(x,z)\Big) \bowtie W_3(x,y,z)}$$

We still need to follow the join tree! ☺

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

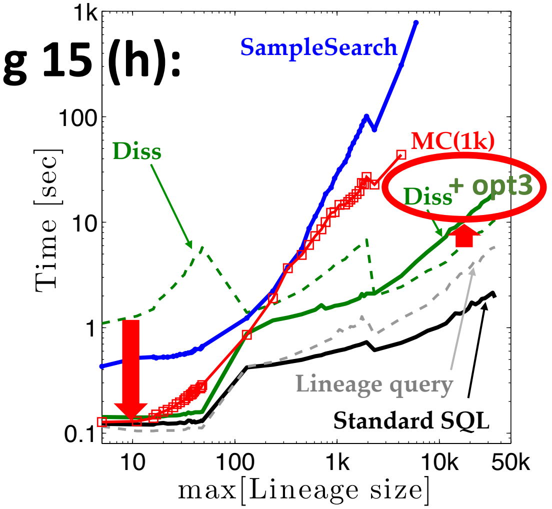
Semi-join reductions can be extremely powerful

Fig 16 (a),(b):



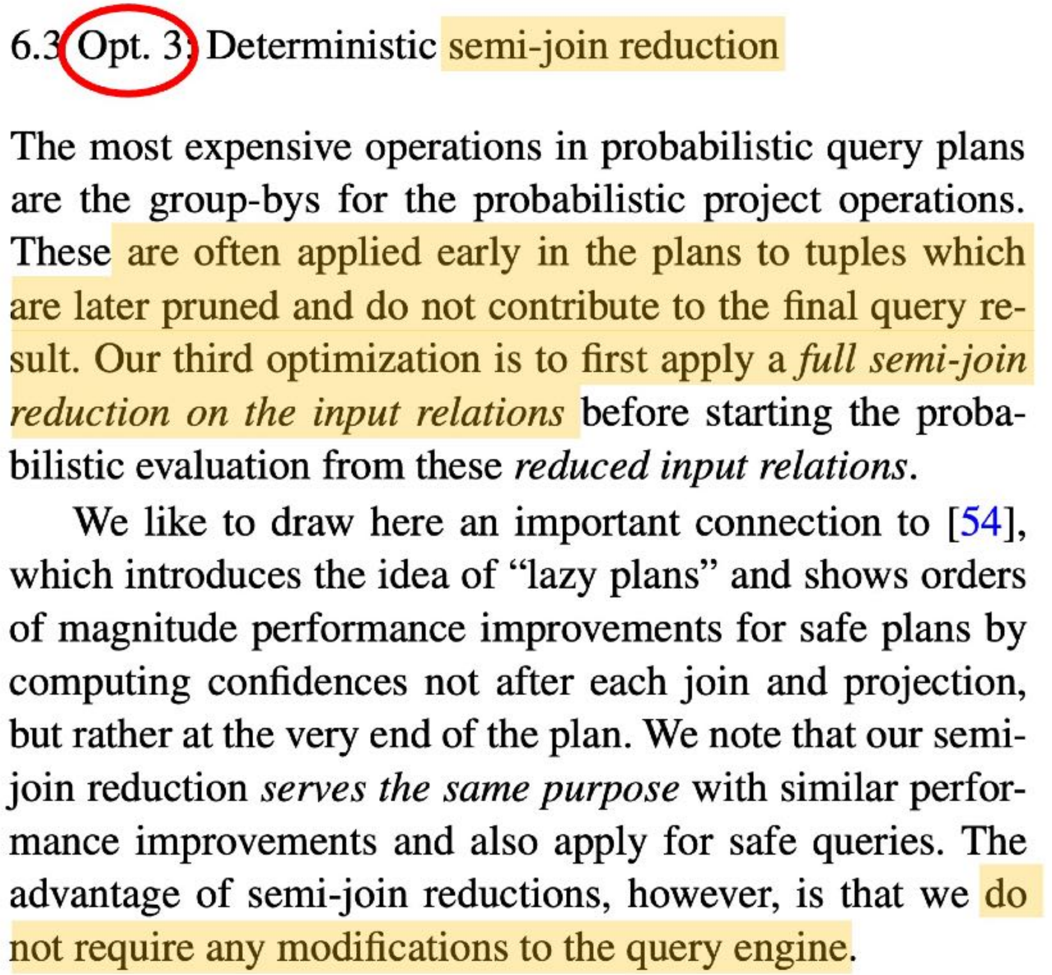
(a) *k*-chain queries (b) *k*-star queries

Fig 15 (h):



(h) Combining (a)-(c)

Semi-join reductions can be extremely powerful in different contexts (yet is speculative, i.e. 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

Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

Key idea of [Yannakakis'81]: Reduction by Semi-joins

- Key insight for acyclic queries:
  - If there are no dangling tuples, then the result can never shrink with an additional join
  - Thus, for a "reduced database", every additional join can only increase the size of the intermediate query results

Yannakakis, "Algorithms for acyclic database schemes", VLDB 1981. https://dl.acm.org/doi/10.5555/1286831.1286840

Yannakakis Algorithm [Yannakakis'81]

- Given: acyclic full conjunctive query Q (full = no projections)
- Compute Q on any database in time O(|Input|+|Output|) by using the join tree
- Step 1: semi-join reduction (two sweeps)
  - Pick any root node R in the join tree of Q
  - Step 1a: Do a semi-join reduction from the leaves to R (bottom-up)
  - Step 1b: Do a semi-join reduction from R to the leaves (top-down)
- 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

[Yannakakis'81] Acyclic Conjunctive Queries

- Theorem [Yannakakis'81]:
  - The Acyclic Conjunctive Query Evaluation Problem is tractable.
  - There is an algorithm for query evaluation with following properties:
    - If Q is a Boolean acyclic conjunctive query,
      then the algorithm runs in time: O(|Q| · |D|)
    - If Q is a full (i.e. no projections) acyclic conjunctive query
      then the algorithm runs in time: O(|Q| · |D| + |Q(D)|)
  - In terms of data complexity, this means O(|Input| + |Output|)
    - i.e., it runs in input/output linear time, which is the “right” notion of tractability in this case (you can't do better, in general)

Yannakakis, "Algorithms for acyclic database schemes", VLDB 1981. https://dl.acm.org/doi/10.5555/1286831.1286840

Yannakakis' Algorithm for Boolean acyclic CQs

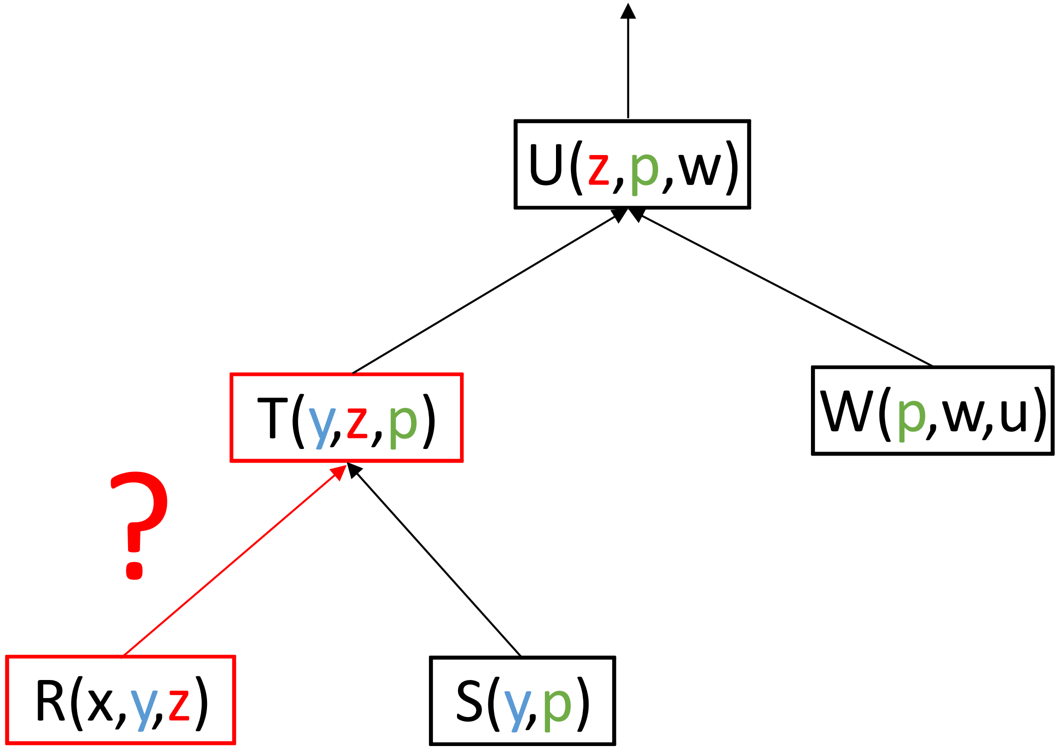
Basically the standard Dynamic Programming Algorithm

- Input: Boolean acyclic conjunctive query Q, database D
- Construct a join tree T of Q
- Populate the nodes of T with the matching relations of D.
- Traverse the tree T bottom up (also called "collection phase"):
  - For each node compute the semi-joins of the (current) relation in the node with the (current) relations in the children of the node
- Examine the resulting relation R at the root of T
  - If R is non-empty, then output Q(D) = 1 (D satisfies Q).
  - If R is empty, then output Q(D) = 0 (D does not satisfy Q).

Acyclic Conjunctive Queries

- Where are the semi-join reductions in following query:

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



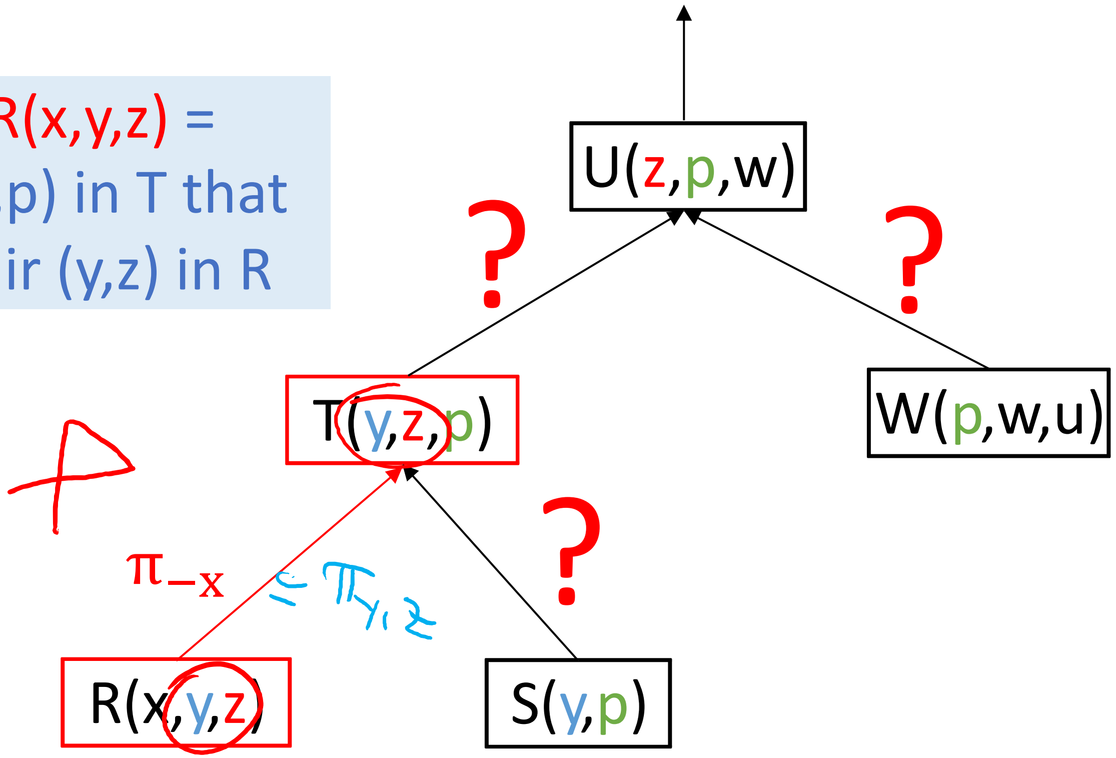
Rooted Join Tree for Q

Acyclic Conjunctive Queries

- Where are the semi-join reductions in following query:

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

T(y,z,p) ⋉ R(x,y,z) =
all triples (y,z,p) in T that
"match" a pair (y,z) in R



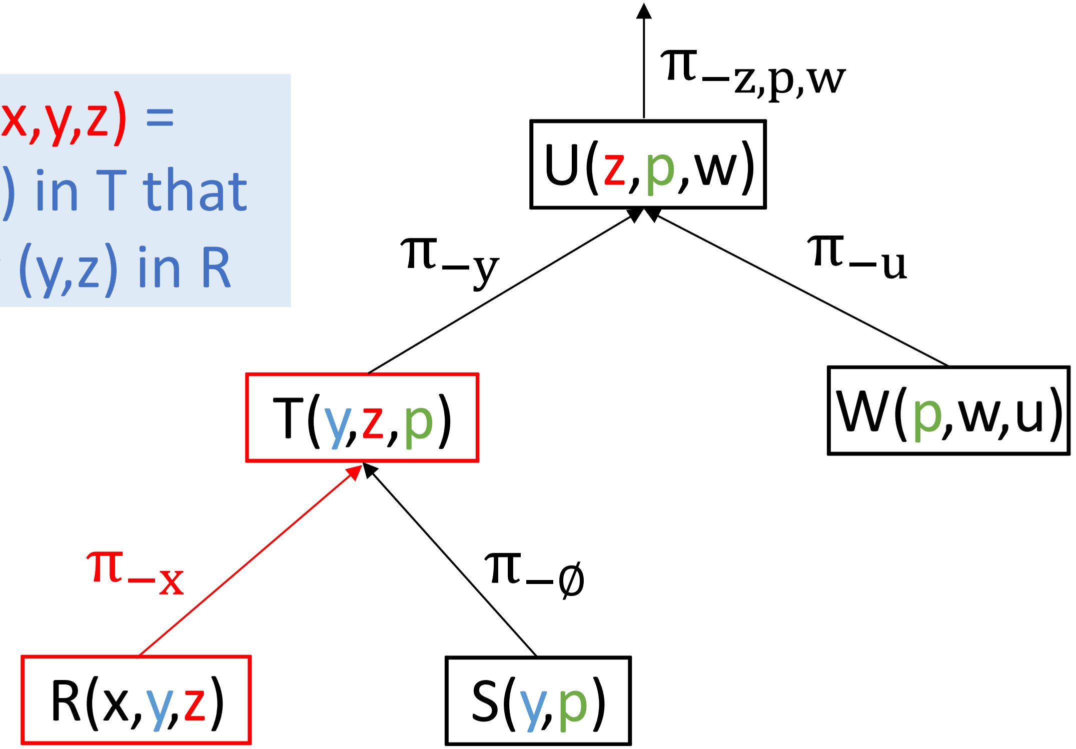
Rooted Join Tree for Q

Acyclic Conjunctive Queries

- Where are the semi-join reductions in following query:

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

T(y,z,p) ⋉ R(x,y,z) =
all triples (y,z,p) in T that "match" a pair (y,z) in R



Rooted Join Tree for Q

More on Yannakakis' Algorithm

- The join tree makes it possible to avoid exponential explosion (in size |Q|) of the size of the intermediate computations.
- The algorithm can be extended to non-Boolean conjunctive queries using one additional top-down traversal of the join tree.
  - Bottom up (or "collect")
  - Top down (or "distribute")
- There are efficient algorithms for detecting acyclicity & computing a join tree.
  - [Tarjan, Yannakakis'84] Linear-time algorithm for detecting acyclicity and computing a join tree.
  - [Gottlob, Leone, Scarcello'01 (FOCS'98)] Detecting acyclicity is LOGCFL-complete and thus highly parallelizable)

Tarjan, Yannakakis. "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs", SIAM J. Comput. 1984. https://doi.org/10.1137/0213035 / Gottlob, Leone, Scarcello. "The Complexity of Acyclic Conjunctive Queries", JACM 2001. https://doi.org/10.1145/382780.382783

A detailed example for the Yannakakis algorithm

Yannakakis Algorithm

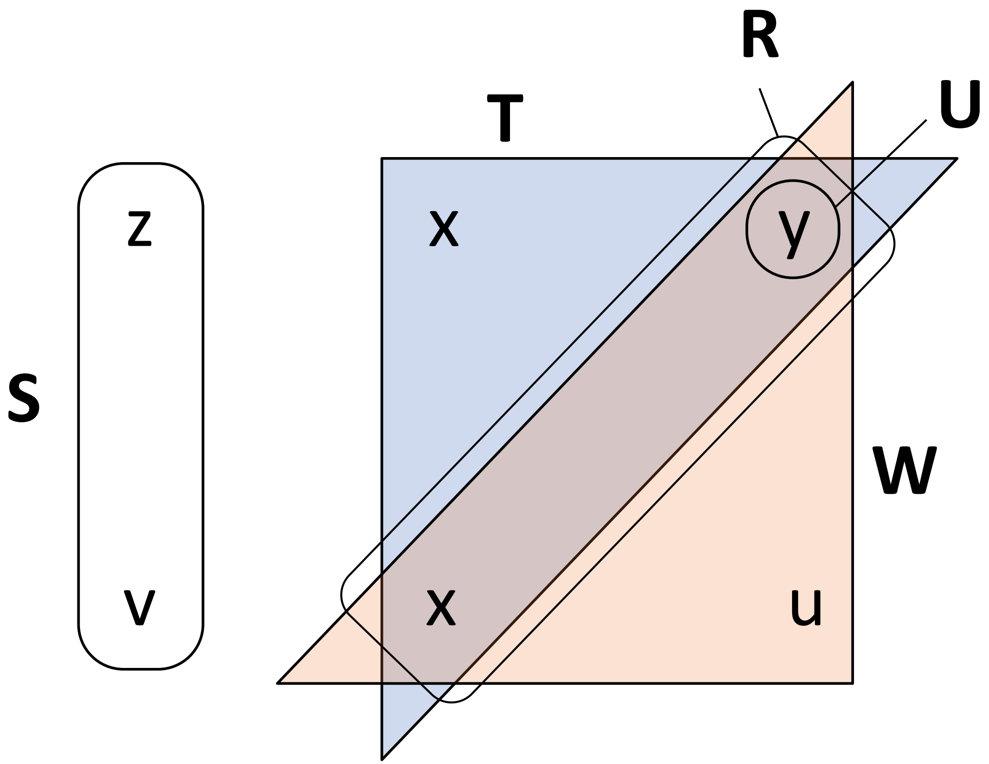
- Given: acyclic full conjunctive query Q (full = no projections)
- Compute Q on any database in time O($\overset{\text{Input}}{n}+\overset{\text{Output}}{r}$) by using the join tree
- Step 1: semi-join reduction (two sweeps)
  - Pick any root node R in the join tree of Q
  - Sweep bottom-up: Do a semi-join 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

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

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

one of several possible ones

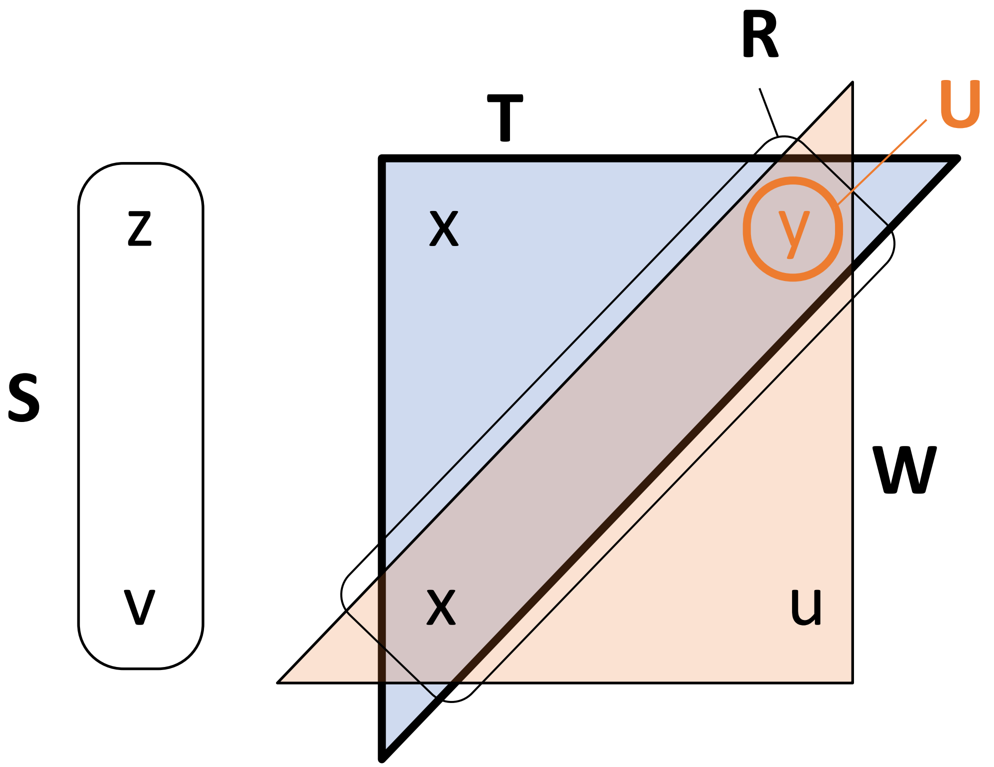
Rooted Join tree

?

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

R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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

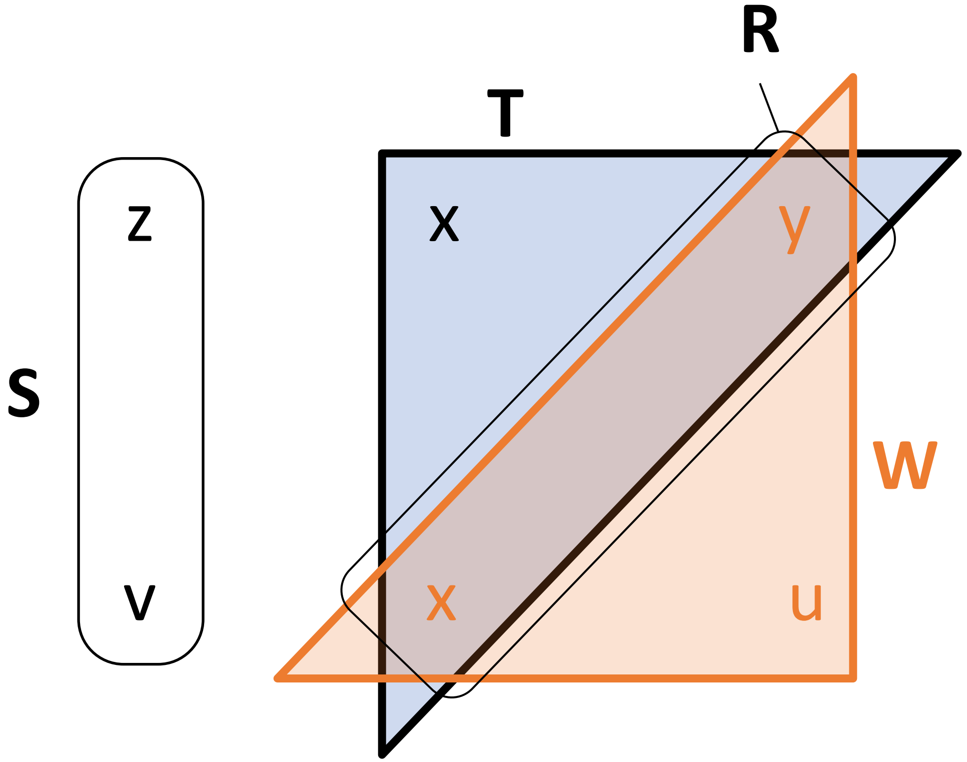
W(u,x,y)

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

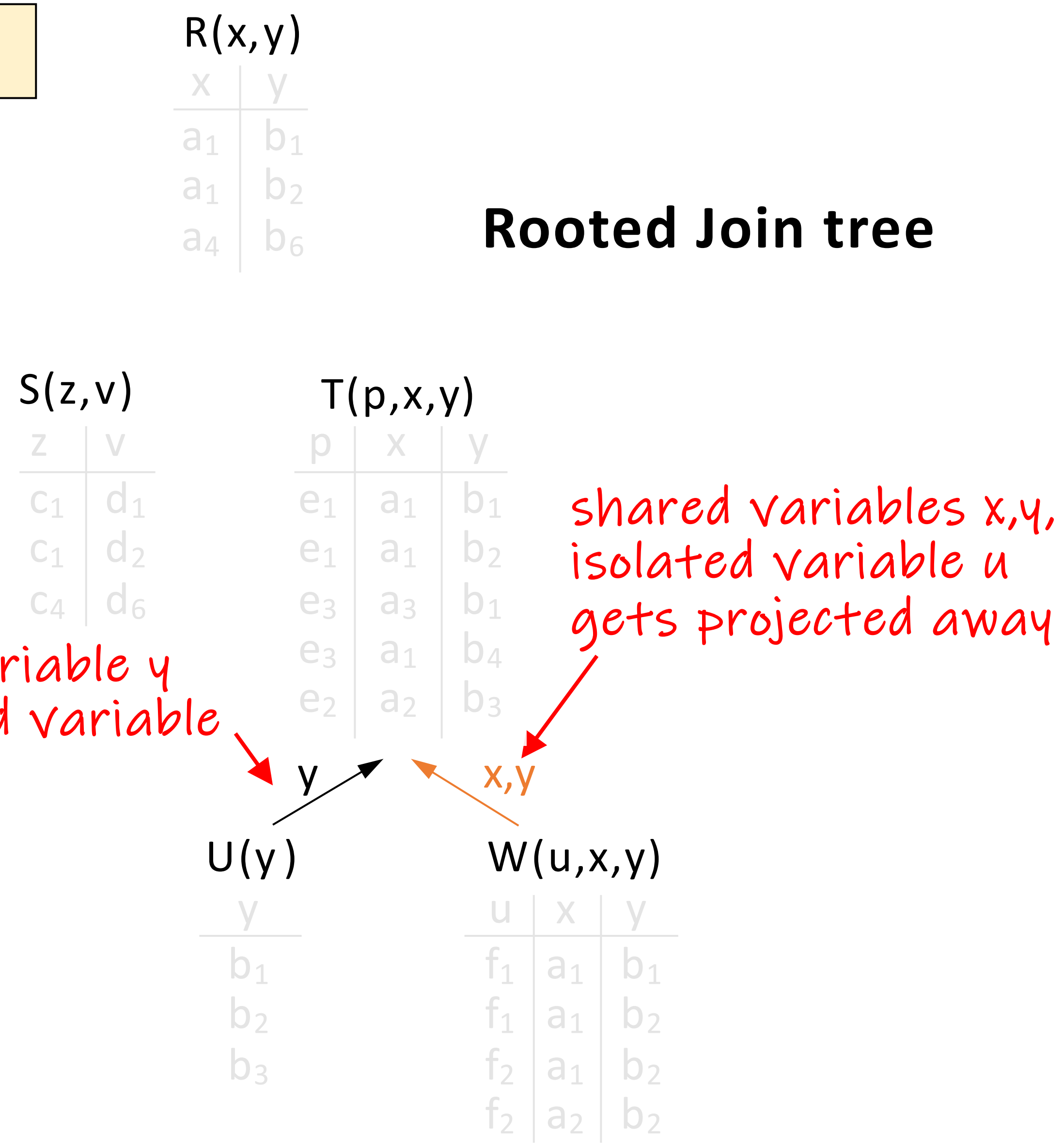
Yannakakis example: first use GYO to get the join tree

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

Hypergraph



Rooted Join tree



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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

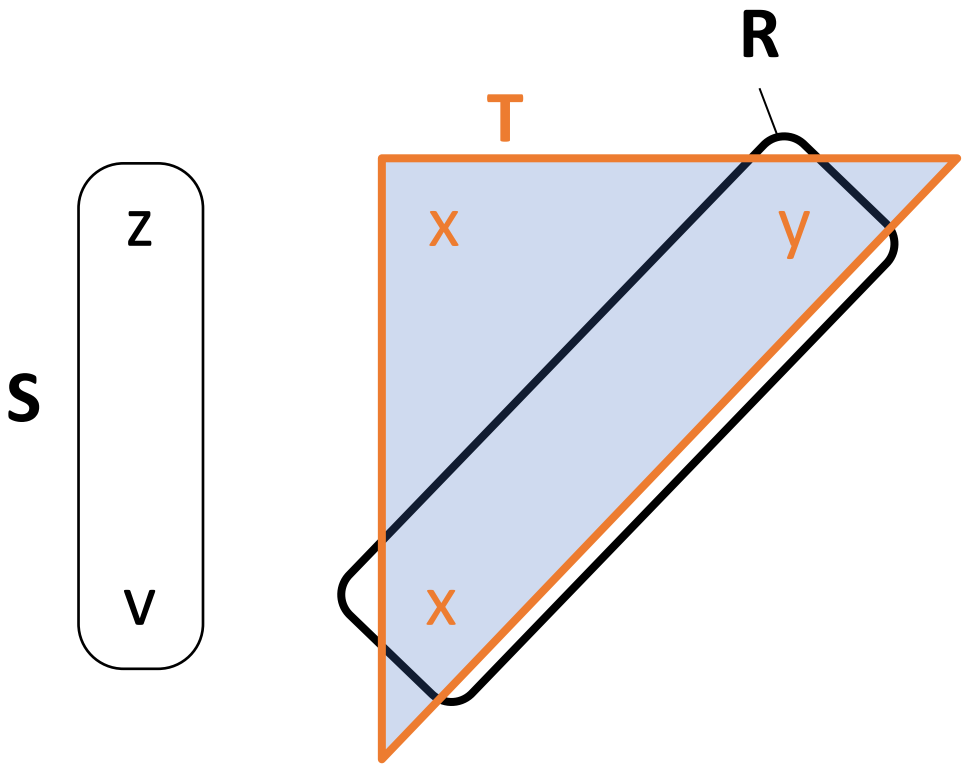
shared variable y
no isolated variable

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

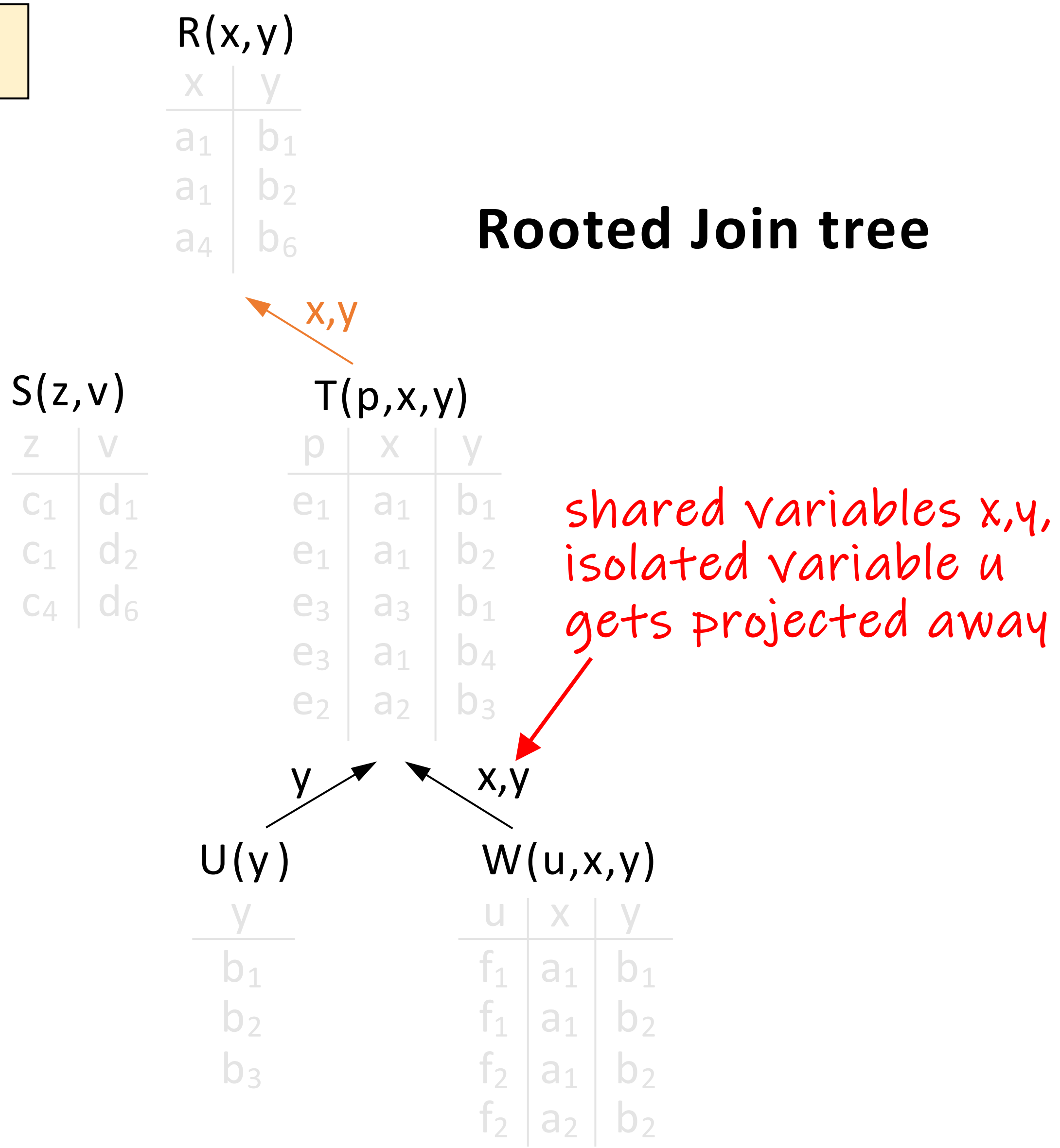
Yannakakis example: first use GYO to get the join tree

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

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)

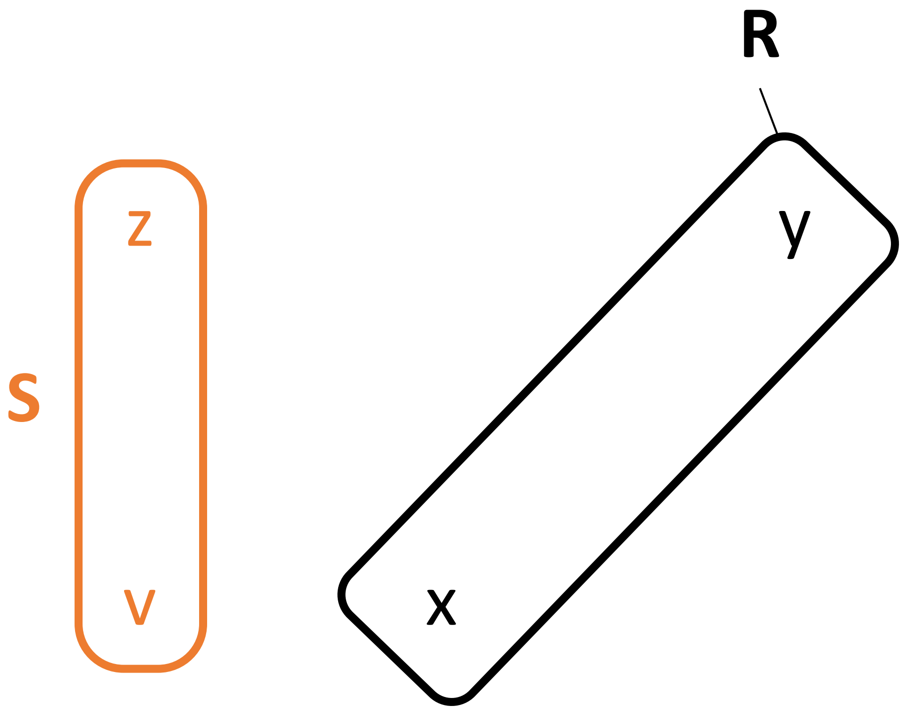
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

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

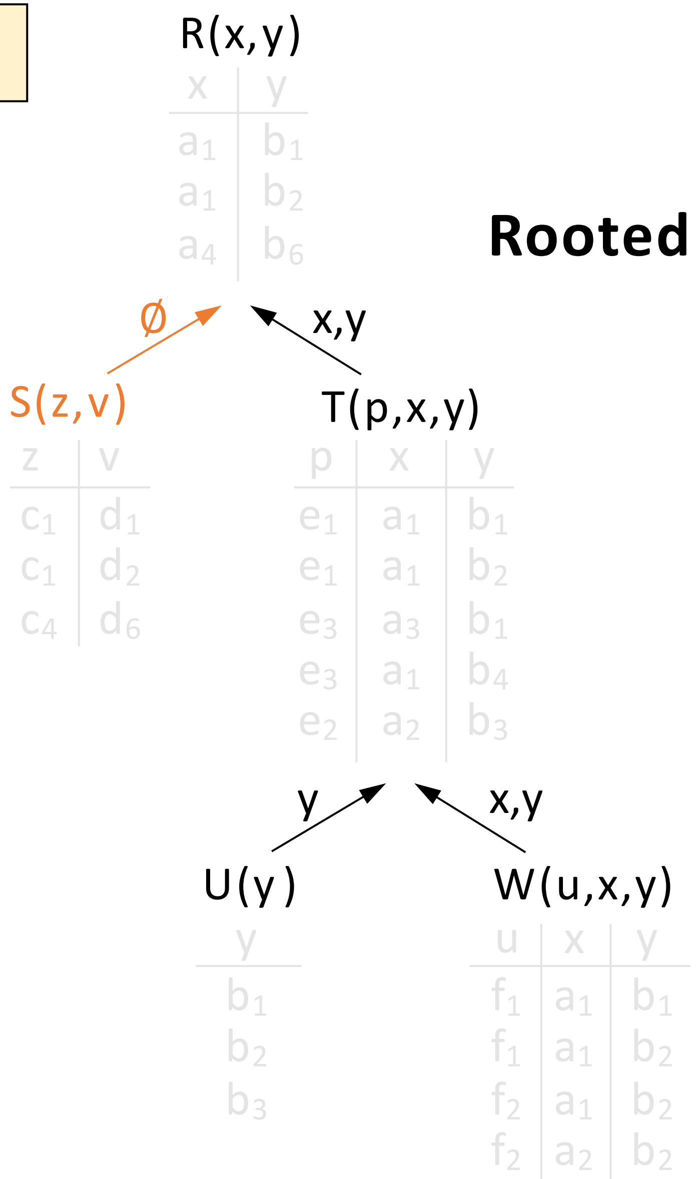
Yannakakis example: first use GYO to get the join tree

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

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

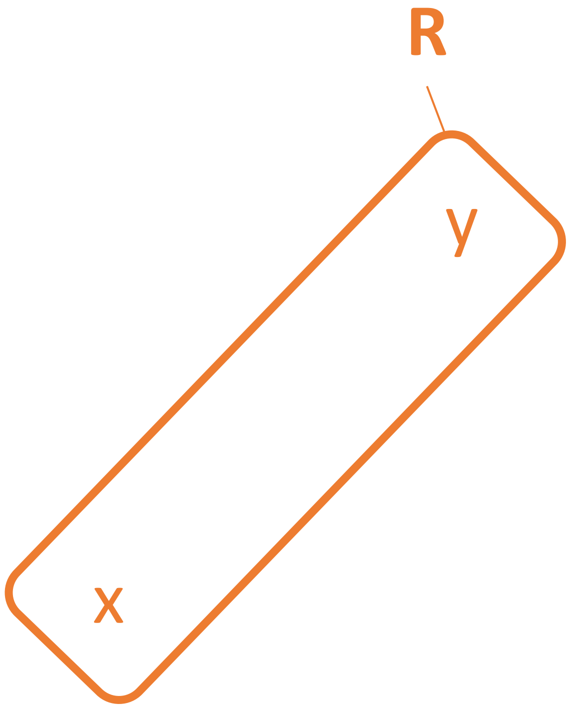
W(u,x,y)

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

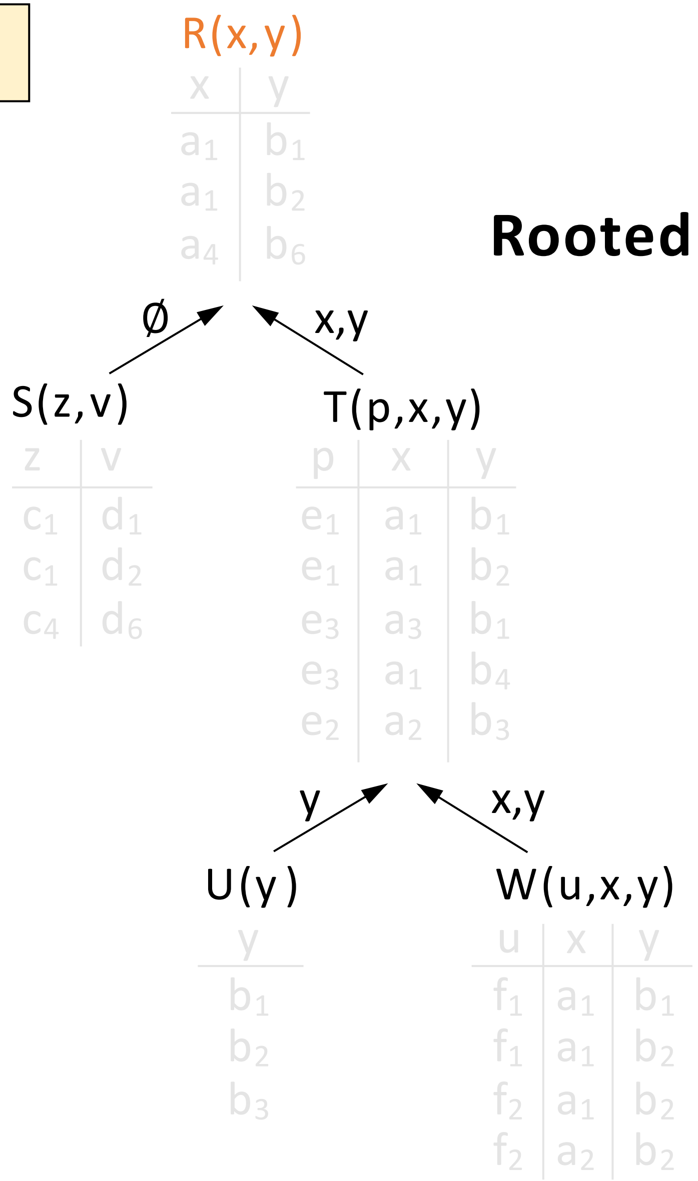
Yannakakis example: first use GYO to get the join tree

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

Hypergraph



Rooted Join tree



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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)

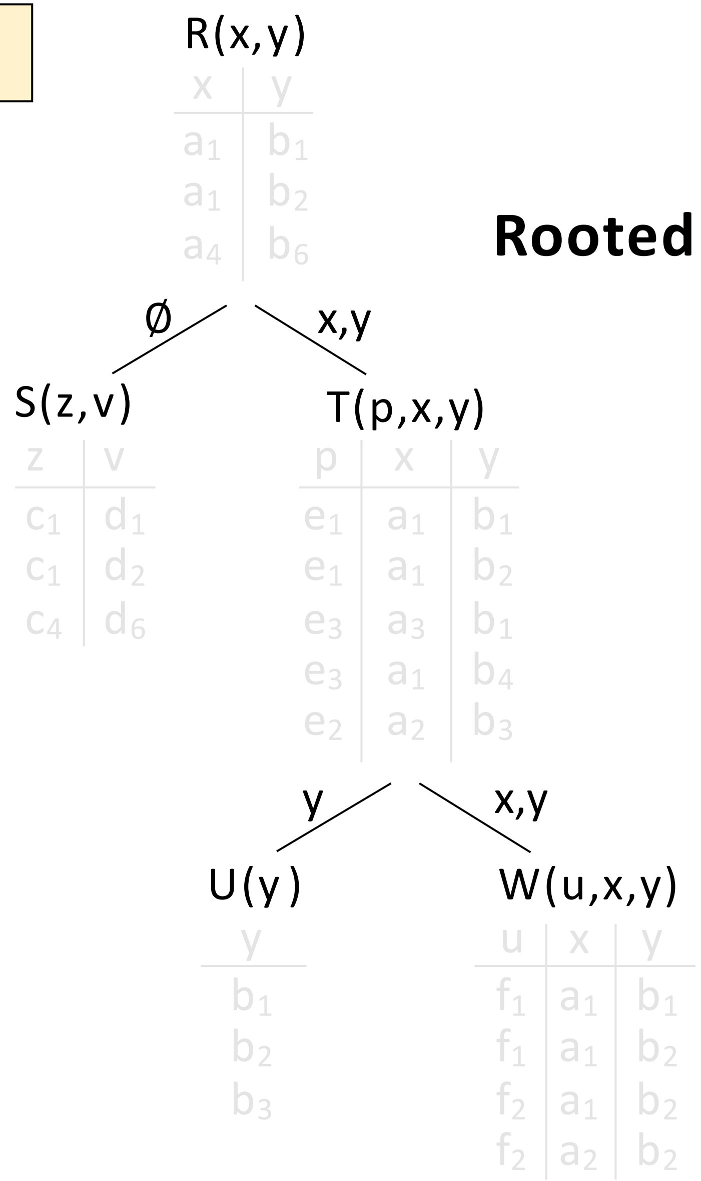
y
b_1
b_2
b_3

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).



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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

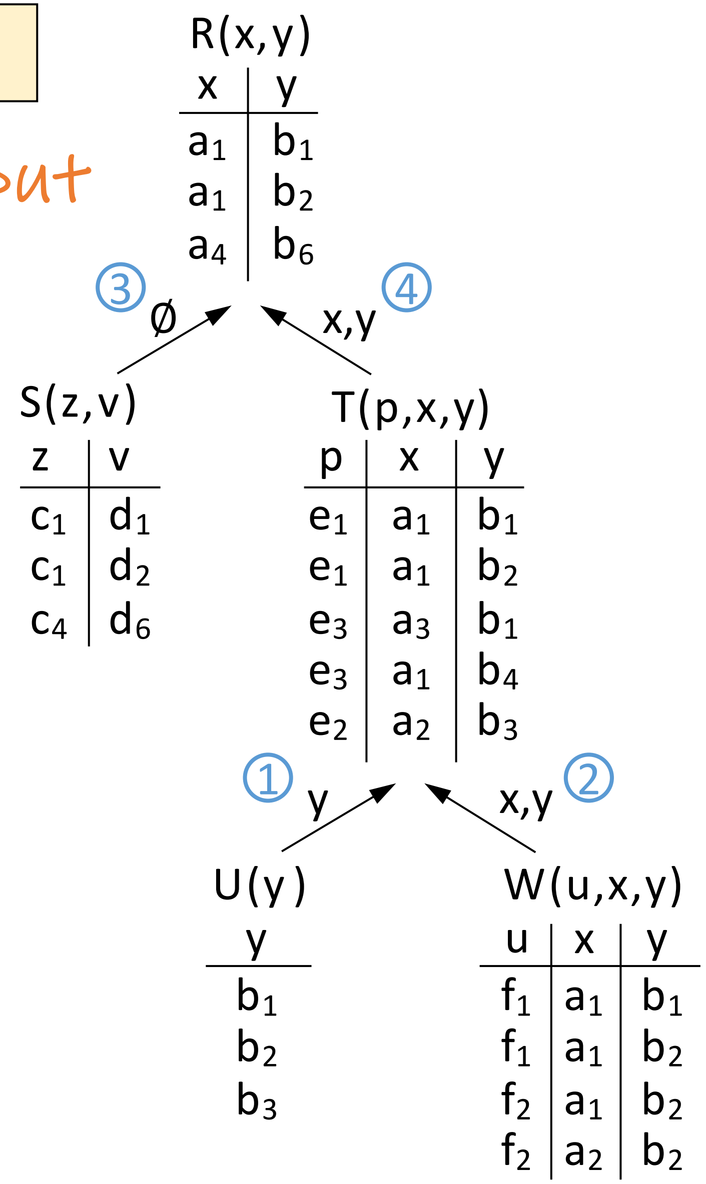
Rooted Join tree

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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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)

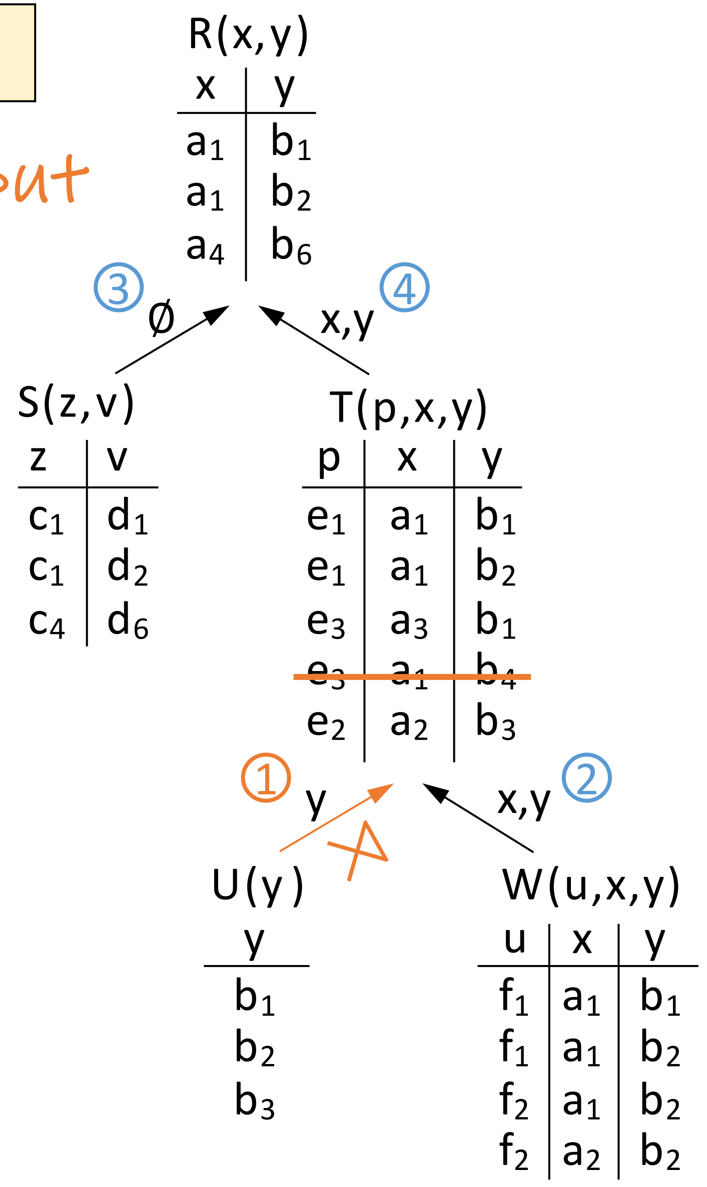
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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)

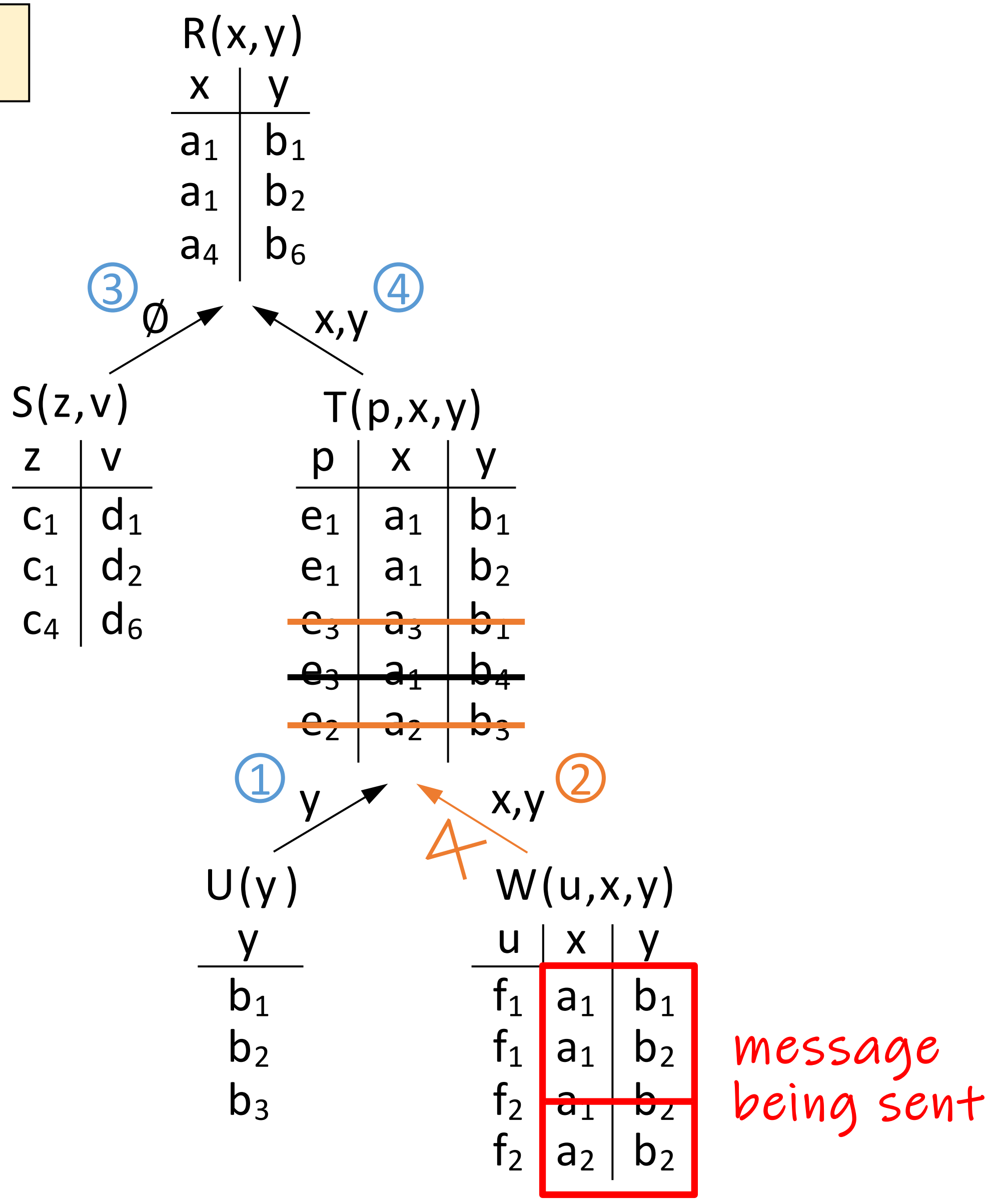
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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

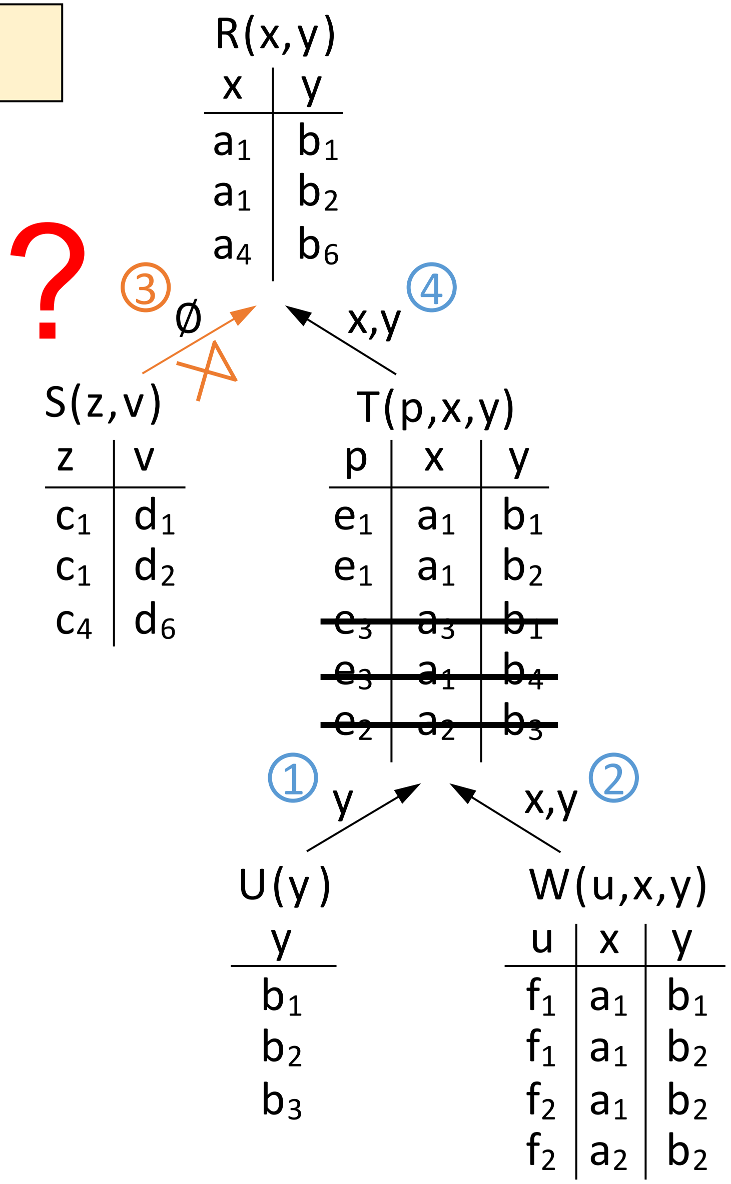
message being sent

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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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)

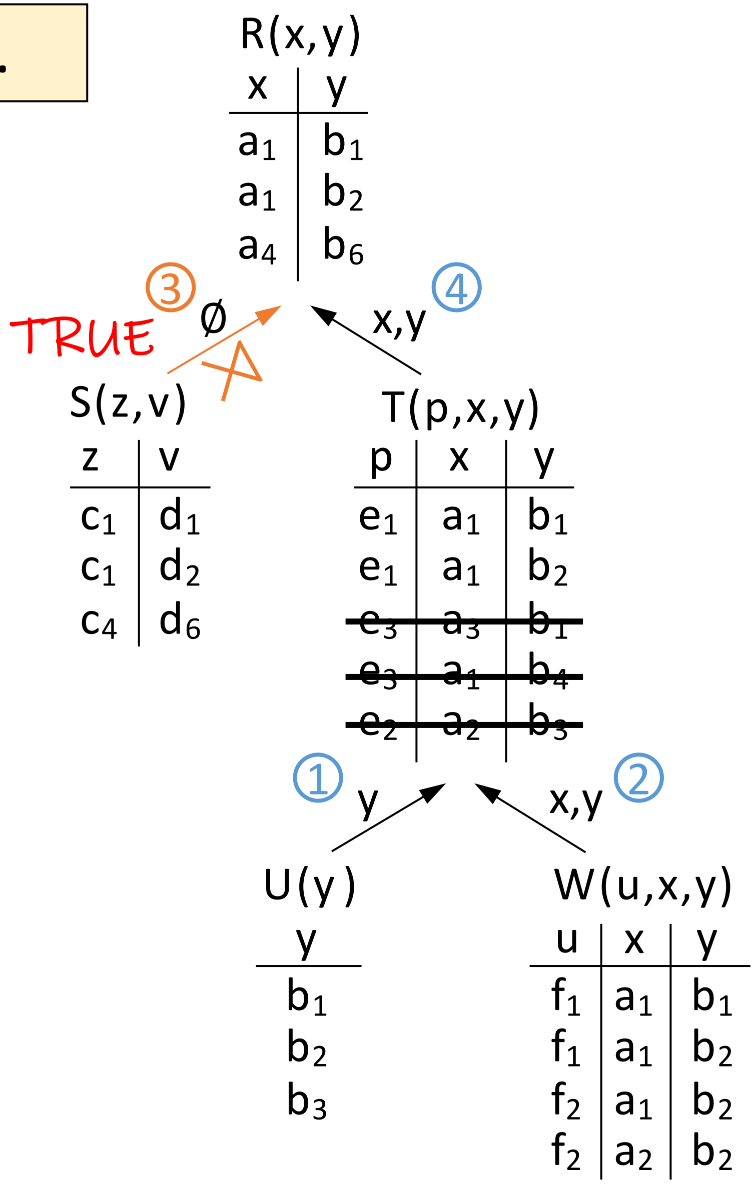
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
a_4	b_6

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)

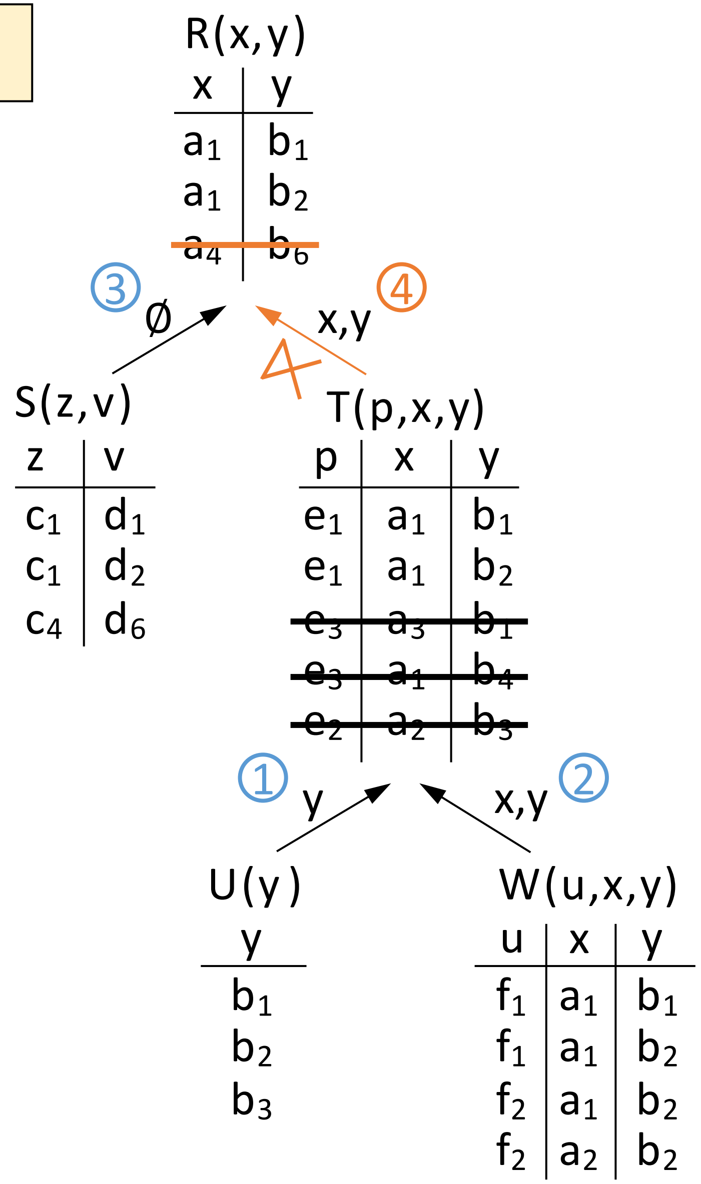
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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 Algorithm example: 1st pass over the data

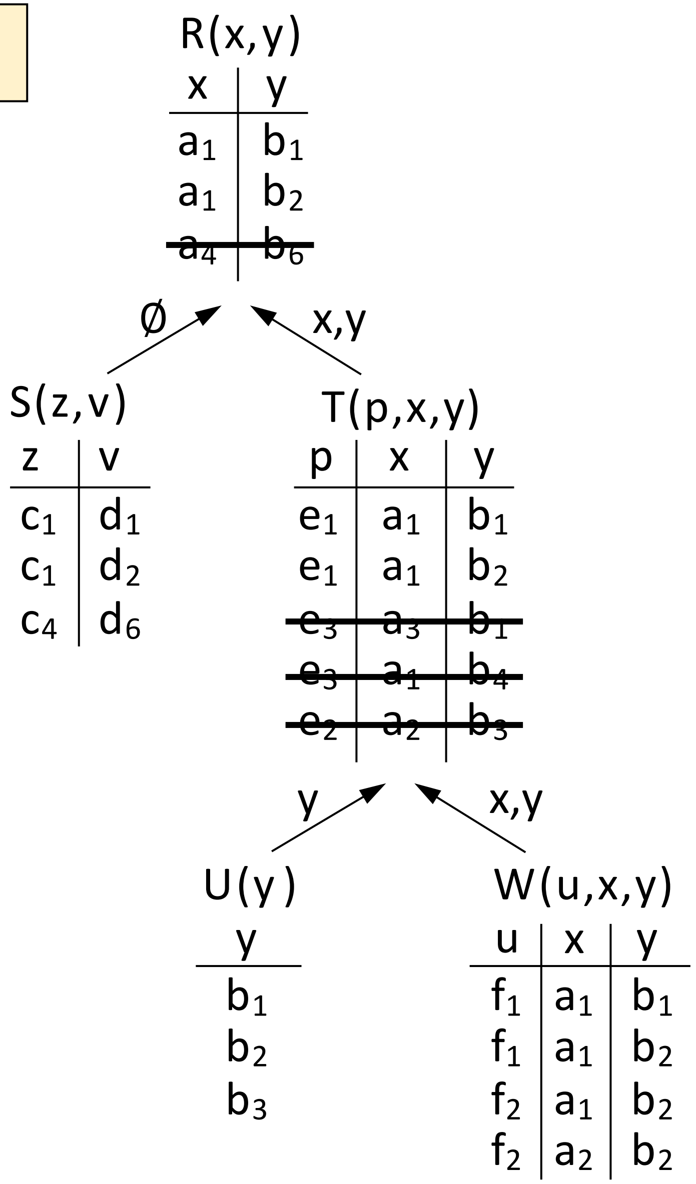
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

1. Semi-join phase $\rtimes$ (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.



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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)

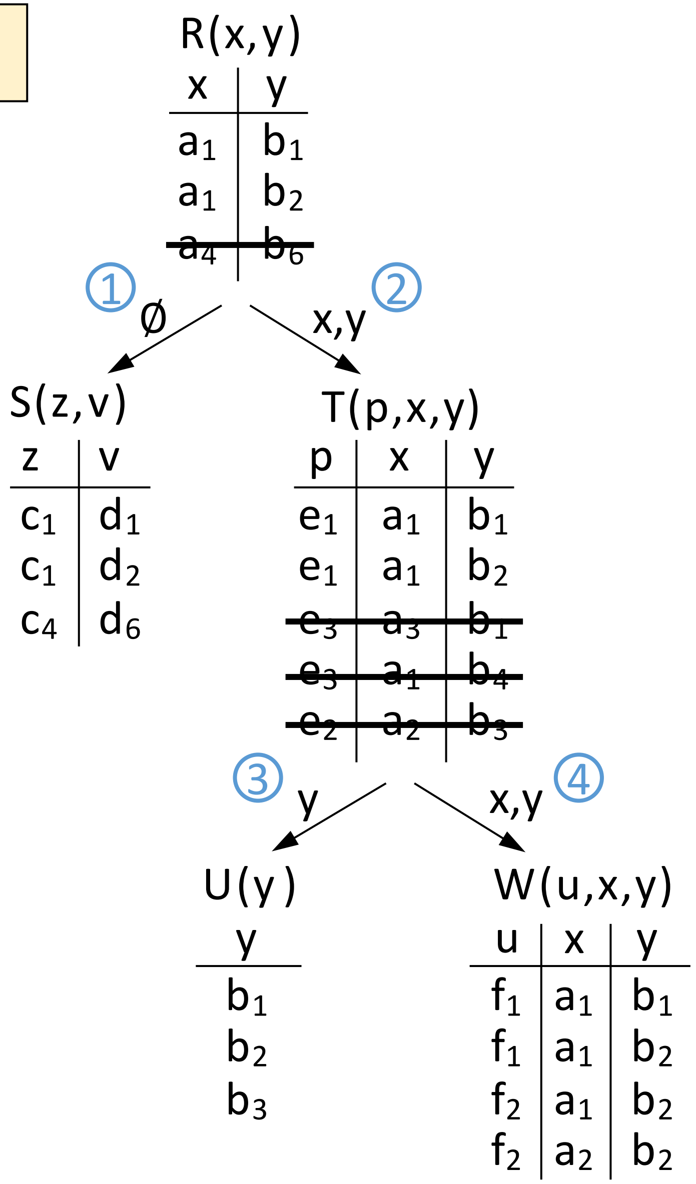
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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 $\rtimes$ (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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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)

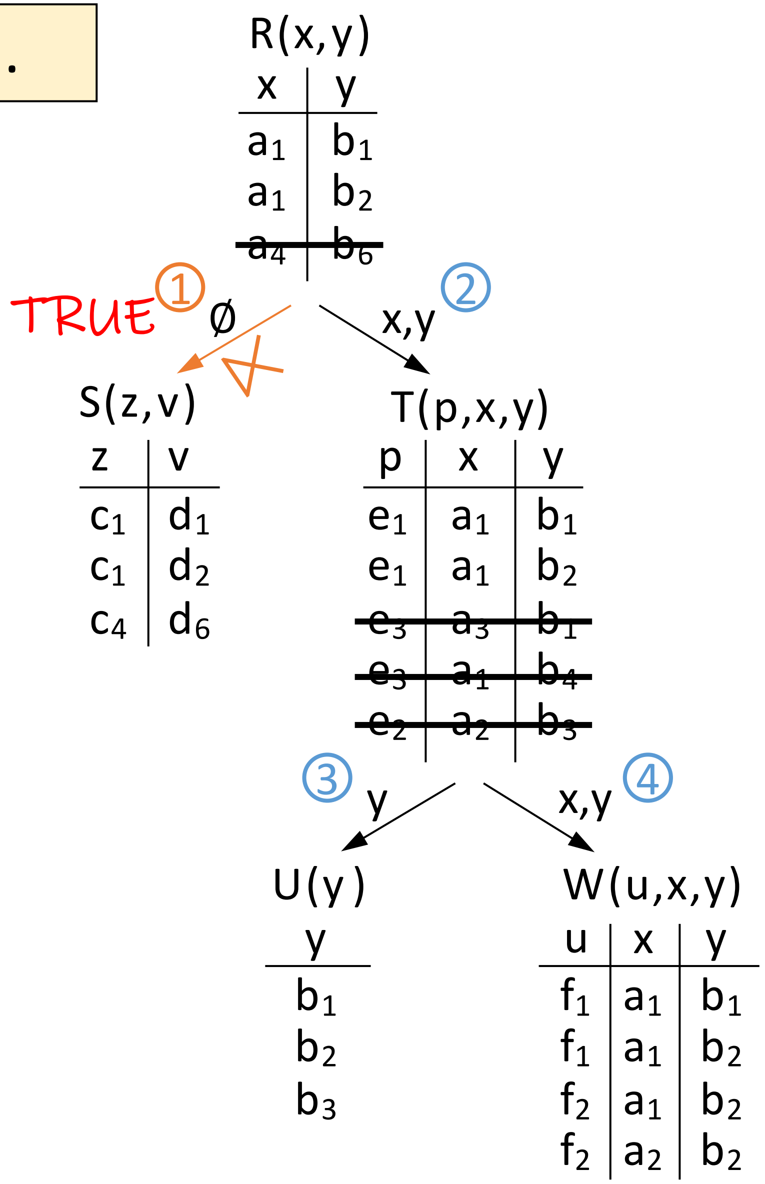
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis Algorithm example: 2^{nd} 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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)

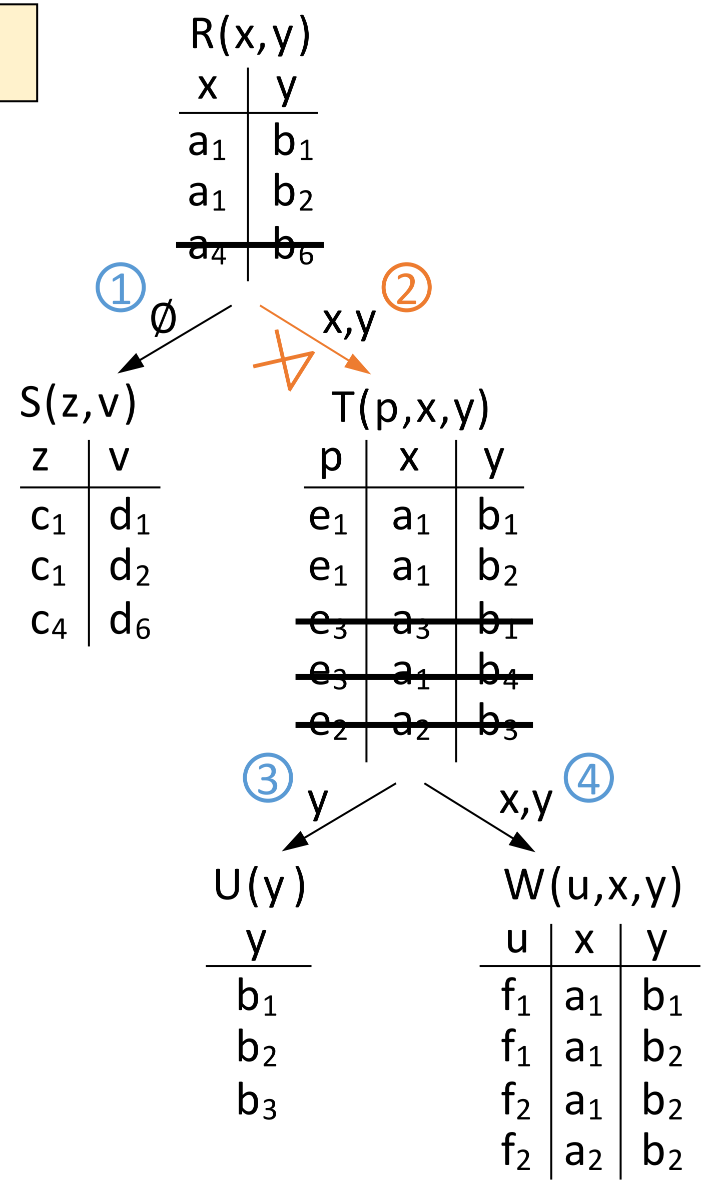
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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)

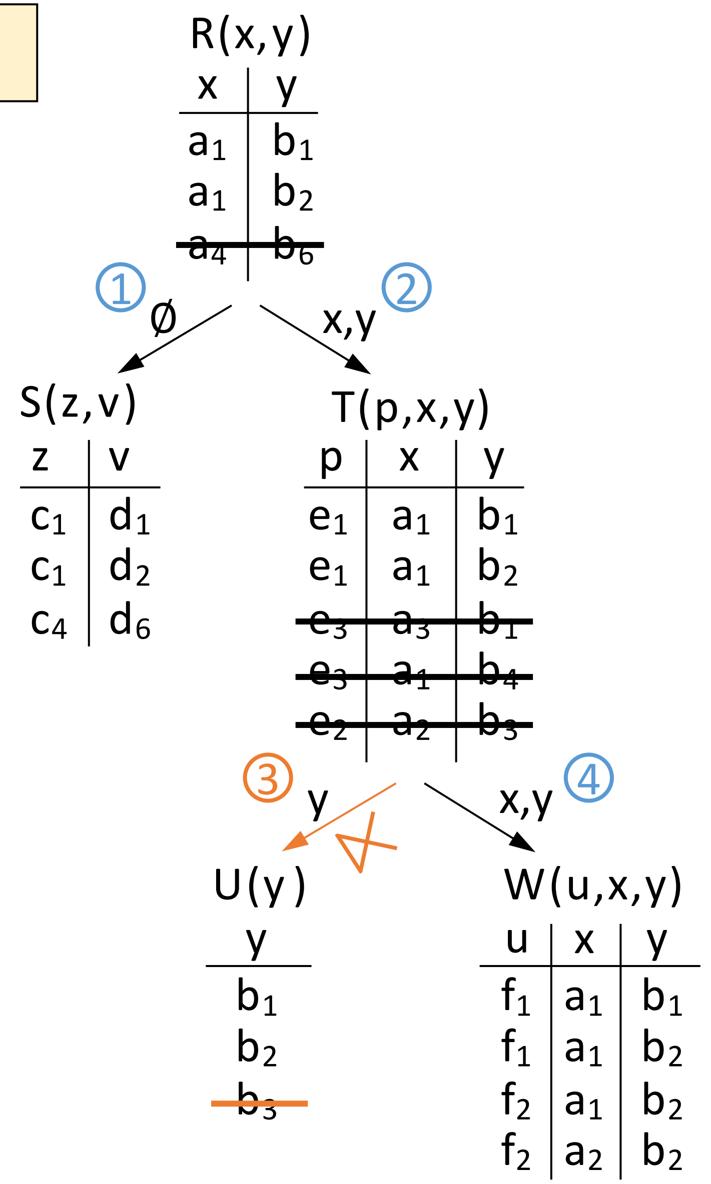
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis Algorithm example: 2^{nd} 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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)

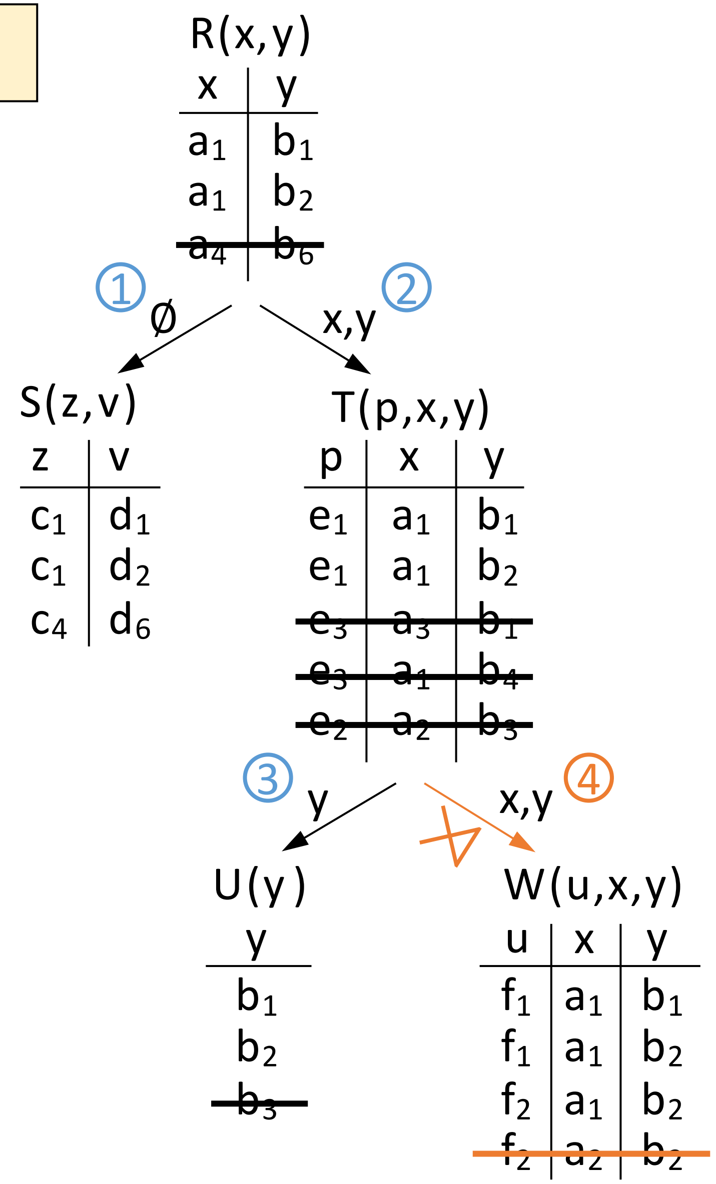
u	x	y
f_1	a_1	b_1
f_1	a_1	b_2
f_2	a_1	b_2
f_2	a_2	b_2

Yannakakis 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**



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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 Algorithm example: 2nd pass over the data

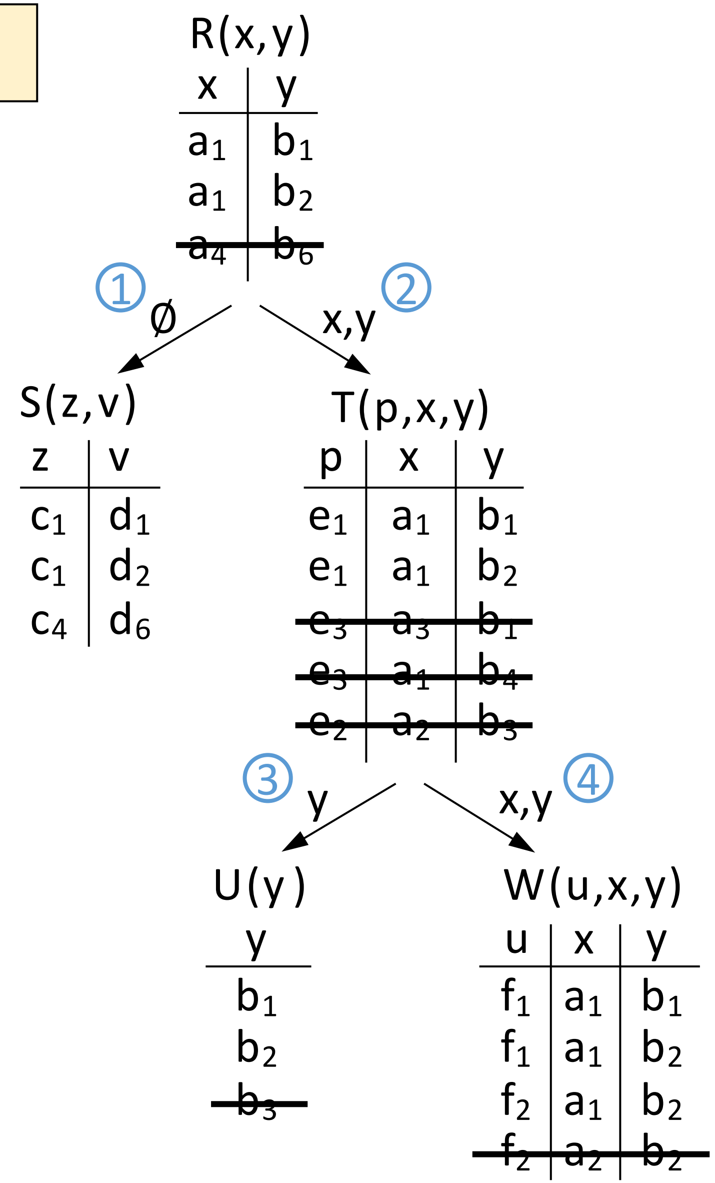
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

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.



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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 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).

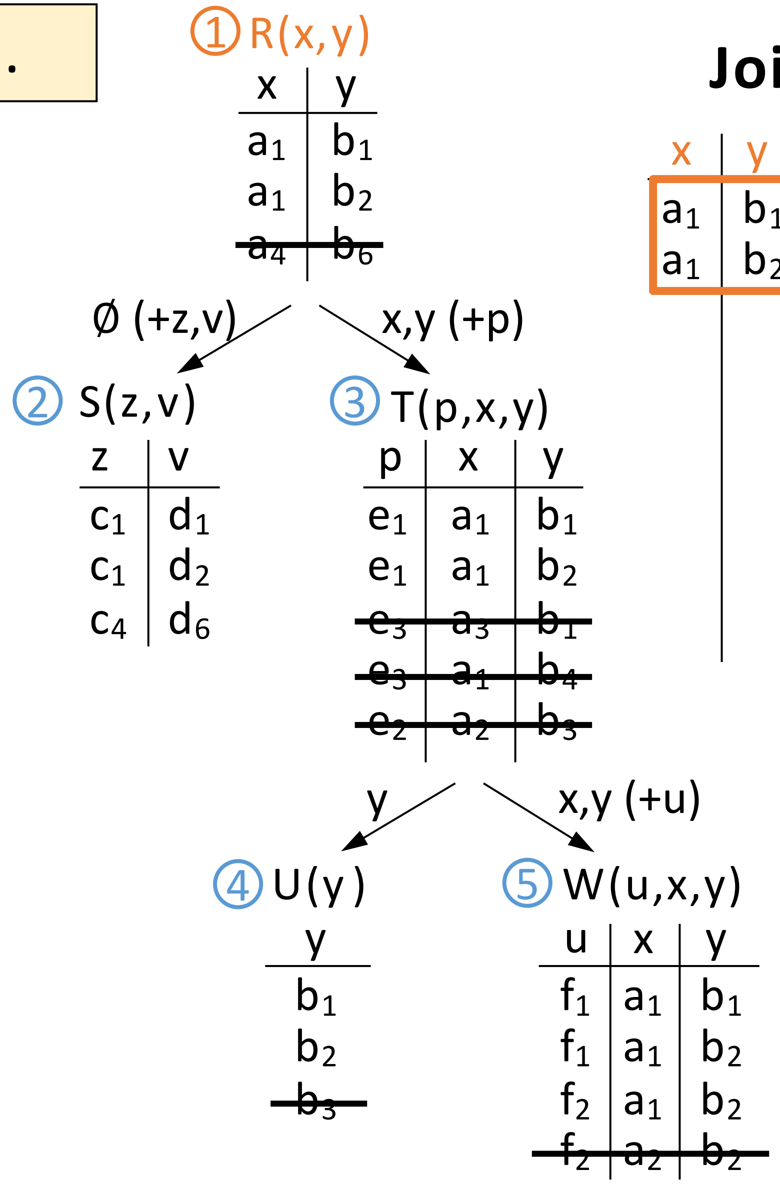
1. Semi-join phase ⋊ (remove dangling tuples) in *O*(n)

- 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!



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

x	y	z	v	p	u
a_1	b_1				
a_1	b_2				

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).

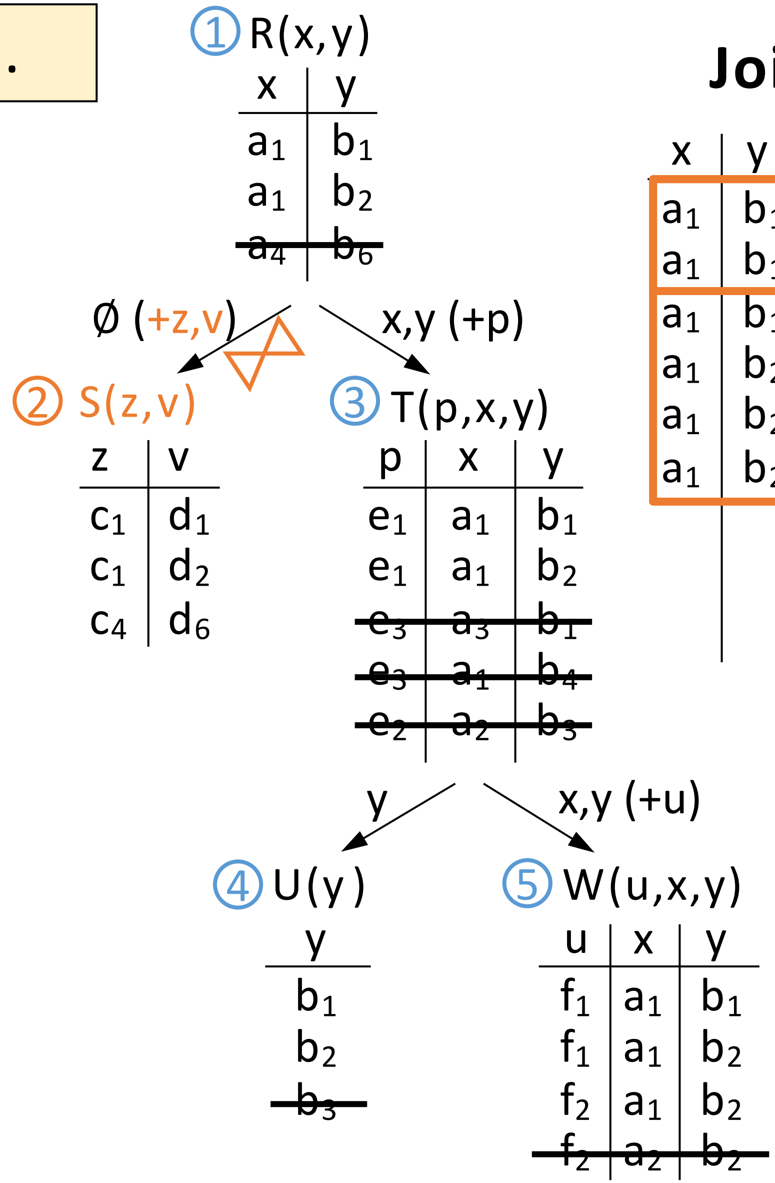
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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!



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

x	y	z	v	p	u
a_1	b_1	c_1	d_1		
a_1	b_1	c_1	d_2		
a_1	b_1	c_4	d_6		
a_1	b_2	c_1	d_1		
a_1	b_2	c_1	d_2		
a_1	b_2	c_4	d_6		

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).

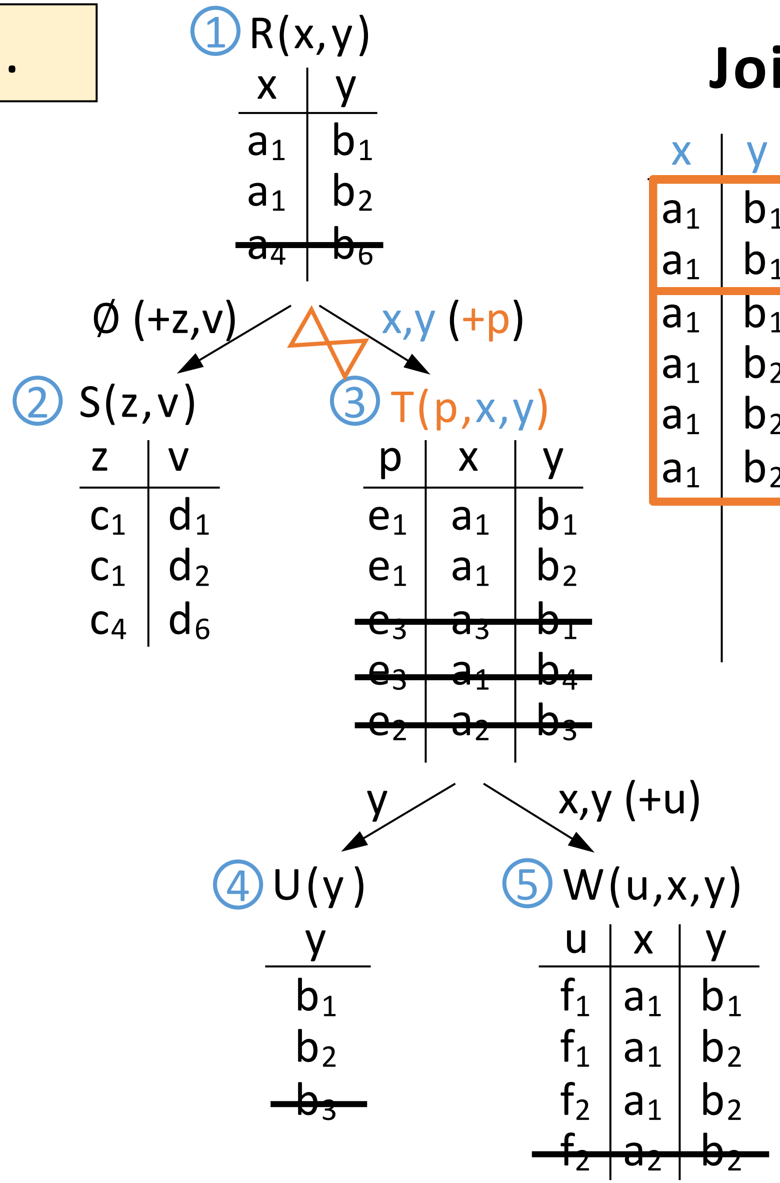
1. Semi-join phase ⋊ (remove dangling tuples) in O(n)

- 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!



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

x	y	z	v	p	u
a_1	b_1	c_1	d_1	e_1	
a_1	b_1	c_1	d_2	e_1	
a_1	b_1	c_4	d_6	e_1	
a_1	b_2	c_1	d_1	e_1	
a_1	b_2	c_1	d_2	e_1	
a_1	b_2	c_4	d_6	e_1	

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).

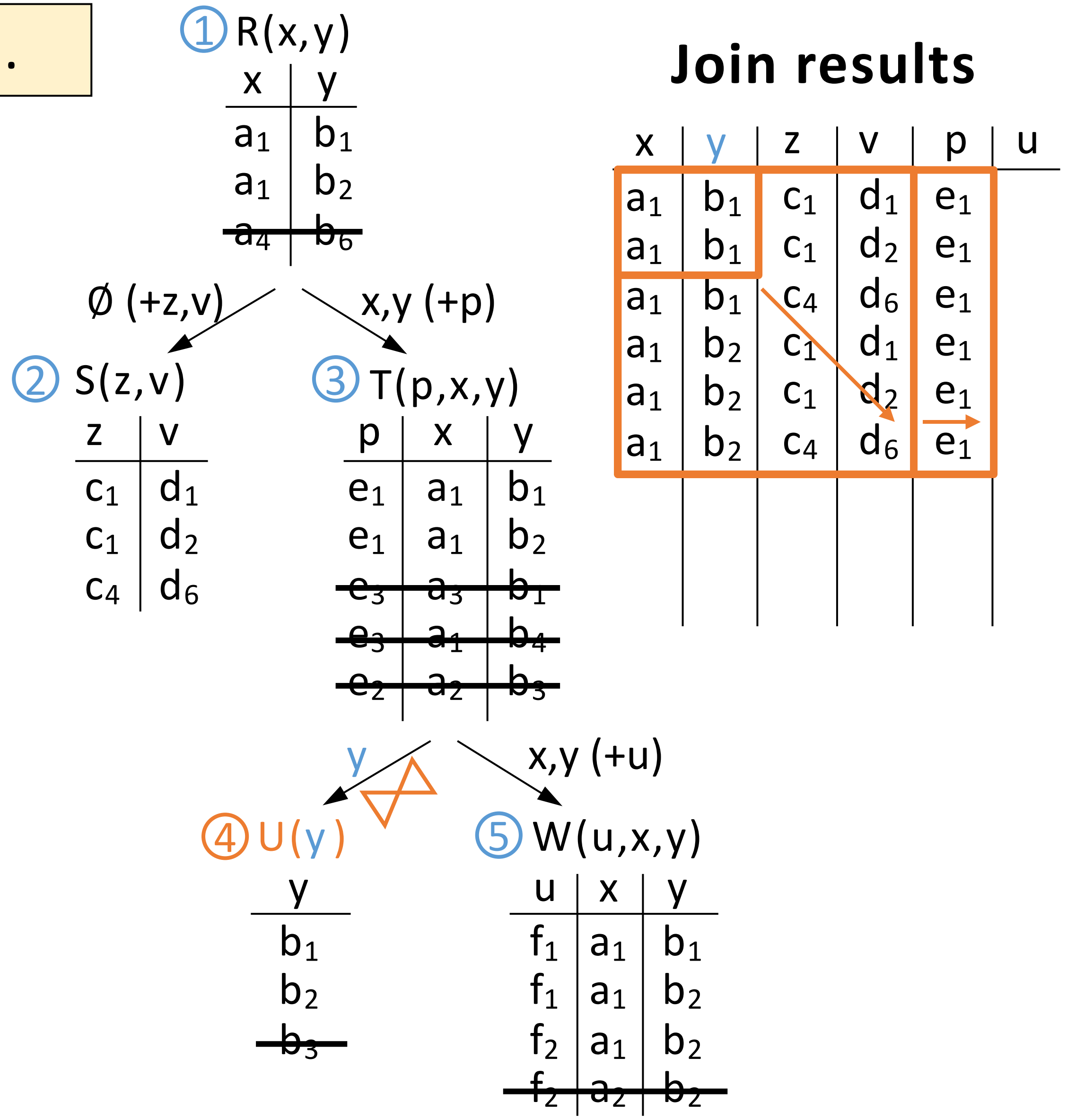
1. Semi-join phase ⋊ (remove dangling tuples) in *O*(n)

- 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!



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

Ø (+z,v)

x,y (+p)

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

y

x,y (+u)

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

x	y	z	v	p	u
a_1	b_1	c_1	d_1	e_1	
a_1	b_1	c_1	d_2	e_1	
a_1	b_1	c_4	d_6	e_1	
a_1	b_2	c_1	d_1	e_1	
a_1	b_2	c_1	d_2	e_1	
a_1	b_2	c_4	d_6	e_1	

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).

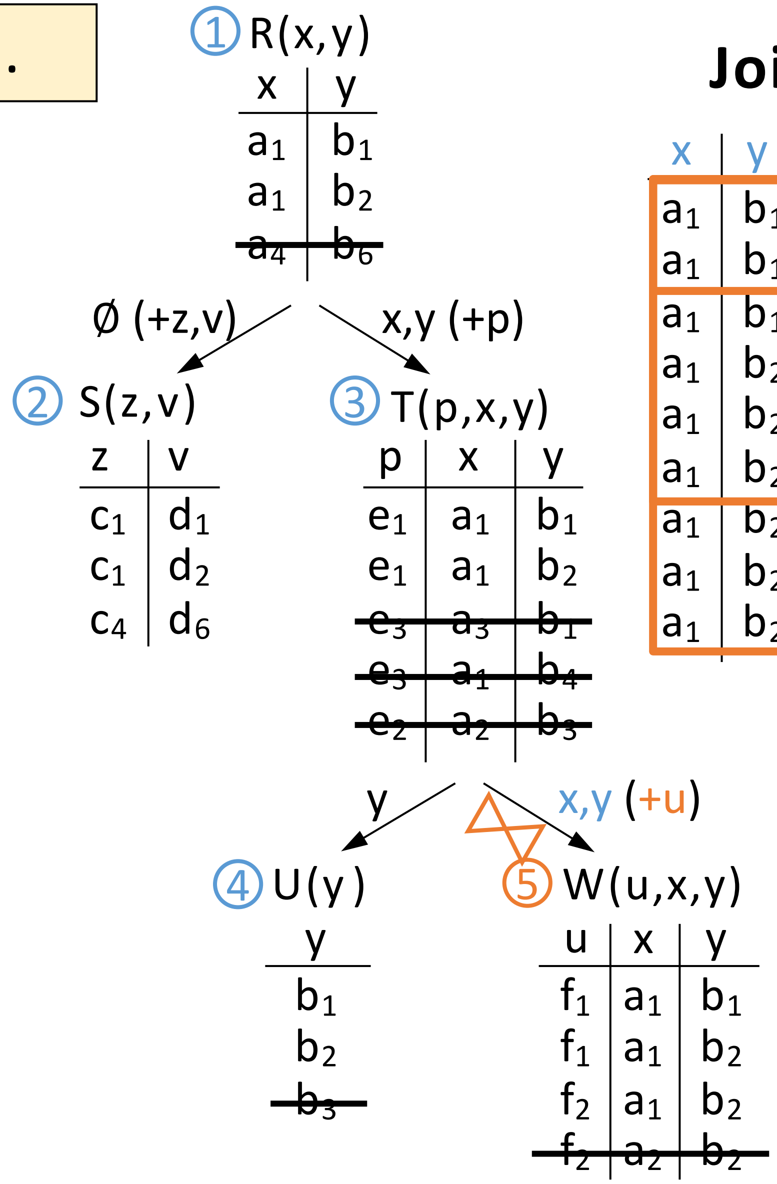
1. Semi-join phase ⋊ (remove dangling tuples) in *O*(n)

- 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!



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

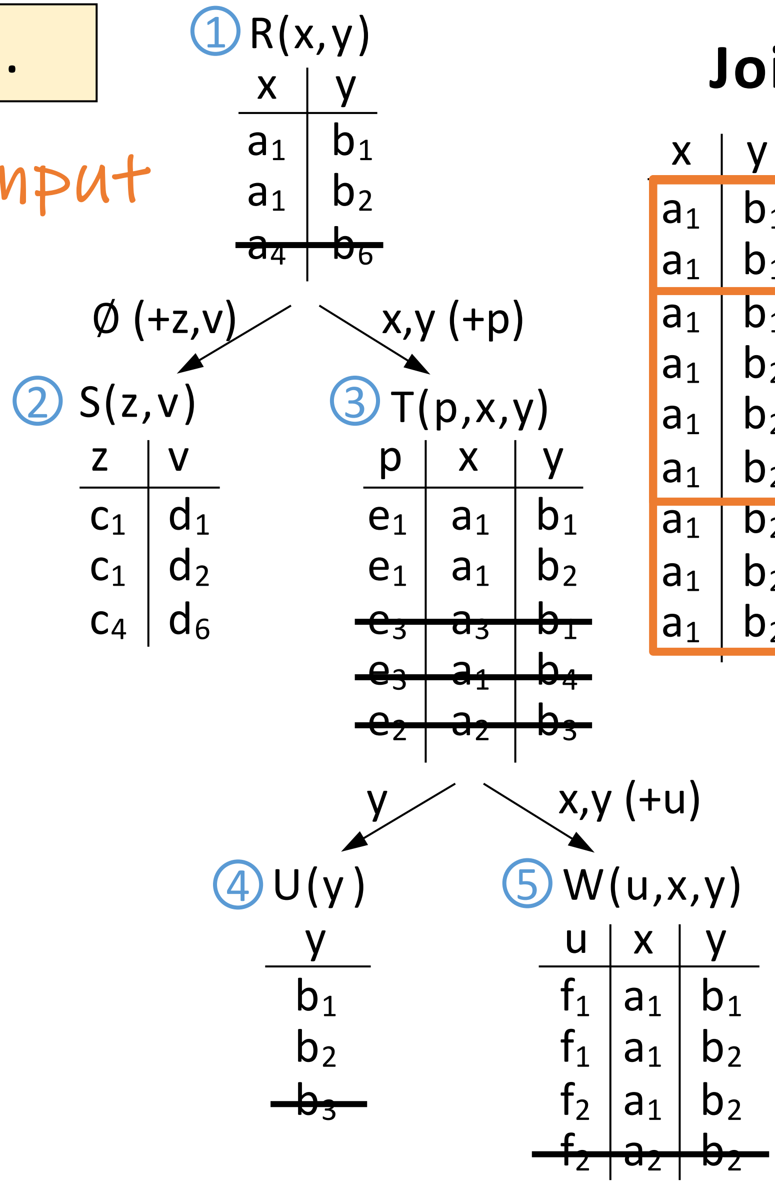
1. Semi-join phase ⋊ (remove dangling tuples) in O(n) ← Input

- 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)≠∅ is in O(n)
Computing Q(D) is in O(n+r)



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

Deciding ACQs Efficiently [Yannakakis'81]

- Non-Serial Dynamic Programming (NSDP) algorithm over a join tree T = (V, E) of a query Q, given database instance D
- Decide Boolean variant Q(D) = ∅ as follows:
  - Pick a root and assign to each $R_j \in V$ the corresponding relation R_j^D of D
  - In a bottom-up reverse topological order of T: compute semijoins of R_j^D
  - If the thus reduced relation at root is empty, then Q(D) = ∅, else Q(D) ≠ ∅.
- Theorem:
  - For ACQs Q: Deciding Q(D) = ∅ is feasible in INPUT linear time.
  - Computing Q(D) can be done in OUTPUT polynomial time.
  - For full queries (no projections) in OUTPUT linear time.

Recap topics 2 & 3

- T2:
  - The notions and complexity of query equivalence and containment
  - The Homomorphism Theorem
  - Minimization of conjunctive queries
- T3 so far: Acyclic conjunctive queries
  - Semi-join reductions, notions of hypergraph acyclicity
  - The Yannakakis algorithm
- T3 yet to be seen:
  - What do we do with cycles?
  - What about ranked retrieval?

Pointers to original work

- The original GYO algorithm was developed concurrently by Graham and Yu-Ozsoyoglu:
  - [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
- Yannakakis. *Algorithms for acyclic database schemes*. VLDB 1981 https://dl.acm.org/doi/10.5555/1286831.1286840
- Bernstein, Chiu. *Using semi-joins to solve relational queries*. JACM 1981. https://doi.org/10.1145/322234.322238
- Bernstein, Goodman. *Power of natural semi-joins*. SIAM J. 1981. https://doi.org/10.1137/0210059
- Beeri, Fagin, Maier, Yannakakis. *On the desirability of acyclic database schemes*. JACM 1983. https://doi.org/10.1145/2402.322389

Updated 3/28/2023

Topic 3: Efficient query evaluation
Unit 1: Acyclic query evaluation
Lecture 21

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp23)

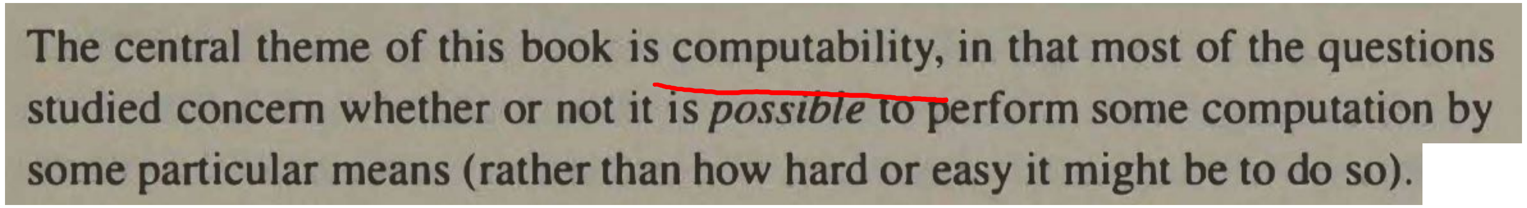
https://northeastern-datalab.github.io/cs7240/sp23/

3/28/2023

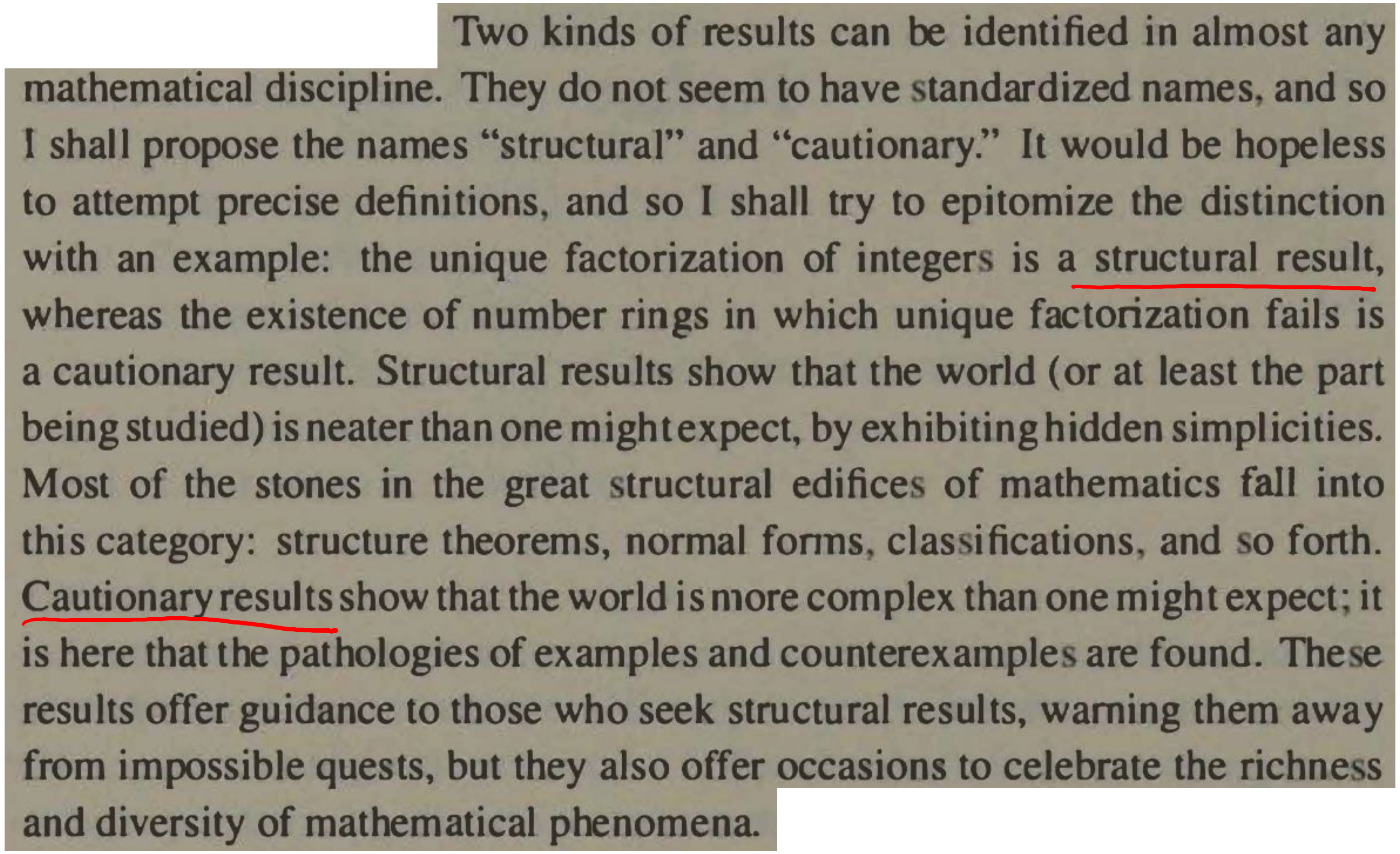
Pre-class conversations

- Last class summary
- Suggested new slide batching:
  - slide decks grouped by topics (& dates), not merely lectures
- Project: (P3: today ~~TUE~~ FR, 3/28)
- Today:
  - Better than Yannakakis
  - Reducing cycles to trees (tree decompositions)
  - Reducing cycles in CQs to trees based (hypertree decompositions)
- Next time:
  - Linear Programming Duality

Preface to "Theories of computability" by Pippenger (1997)



The central theme of this book is computability, in that most of the questions studied concern whether or not it is *possible* to perform some computation by some particular means (rather than how hard or easy it might be to do so).



Two kinds of results can be identified in almost any mathematical discipline. They do not seem to have standardized names, and so I shall propose the names "structural" and "cautionary." It would be hopeless to attempt precise definitions, and so I shall try to epitomize the distinction with an example: the unique factorization of integers is a structural result, whereas the existence of number rings in which unique factorization fails is a cautionary result. Structural results show that the world (or at least the part being studied) is neater than one might expect, by exhibiting hidden simplicities. Most of the stones in the great structural edifices of mathematics fall into this category: structure theorems, normal forms, classifications, and so forth. Cautionary results show that the world is more complex than one might expect; it is here that the pathologies of examples and counterexamples are found. These results offer guidance to those who seek structural results, warning them away from impossible quests, but they also offer occasions to celebrate the richness and diversity of mathematical phenomena.

Source: Nicholas Pippenger. Theories of computability. 1997. https://dl.acm.org/doi/book/10.5555/264484

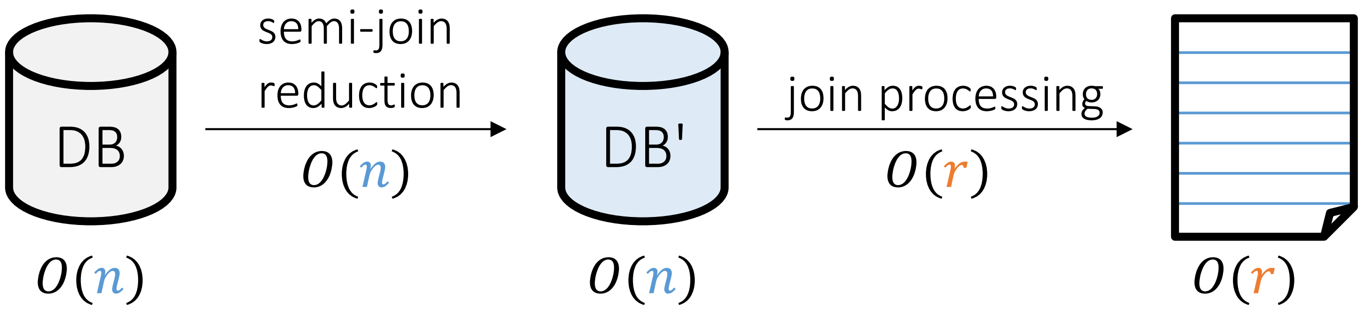
Outline: T3-1: Acyclic conjunctive queries

- T3-1: Acyclic conjunctive queries
  - The semijoin operator
  - alpha-acyclic hypergraphs, join trees
  - GYO reduction
  - Full semi-join reductions
  - Yannakakis algorithm
  - Enumeration algorithms
- T3-2: Cyclic conjunctive queries

Several parts are an extended version of a tutorial from ICDE'22:
https://www.youtube.com/watch?v=toi7ysuyRkw
https://northeastern-datalab.github.io/

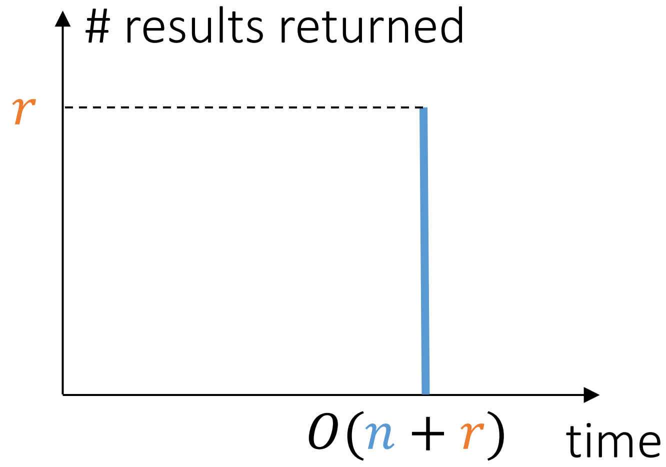
The enumeration framework

Standard Yannakakis framework for acyclic join processing

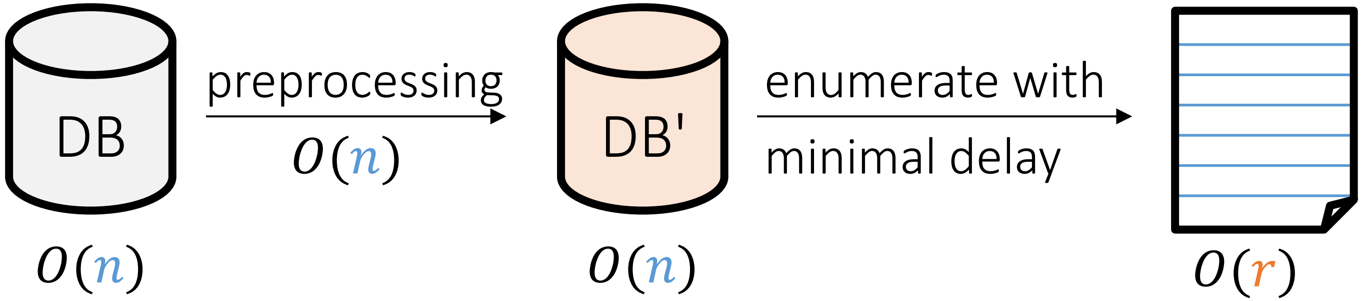


DB $\xrightarrow[O(n)]{\text{semi-join reduction}}$ DB' $\xrightarrow[O(r)]{\text{join processing}}$

$O(n)$ $O(n)$ $O(r)$



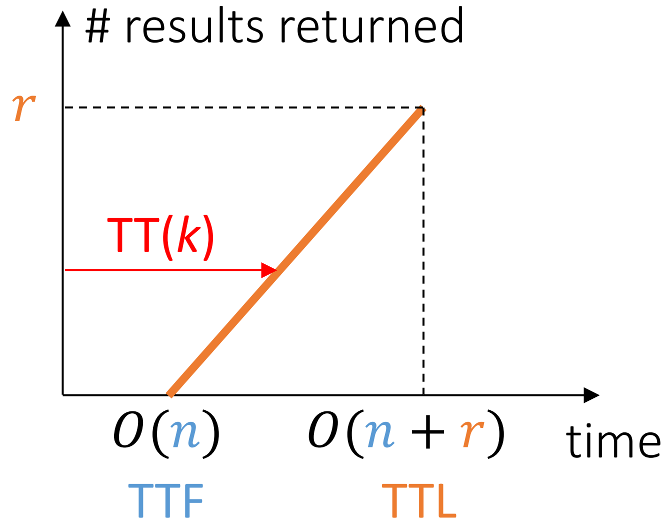
Enumeration framework for acyclic queries



DB $\xrightarrow[O(n)]{\text{preprocessing}}$ DB' $\xrightarrow[\text{minimal delay}]{\text{enumerate with}}$

$O(n)$ $O(n)$ $O(r)$

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

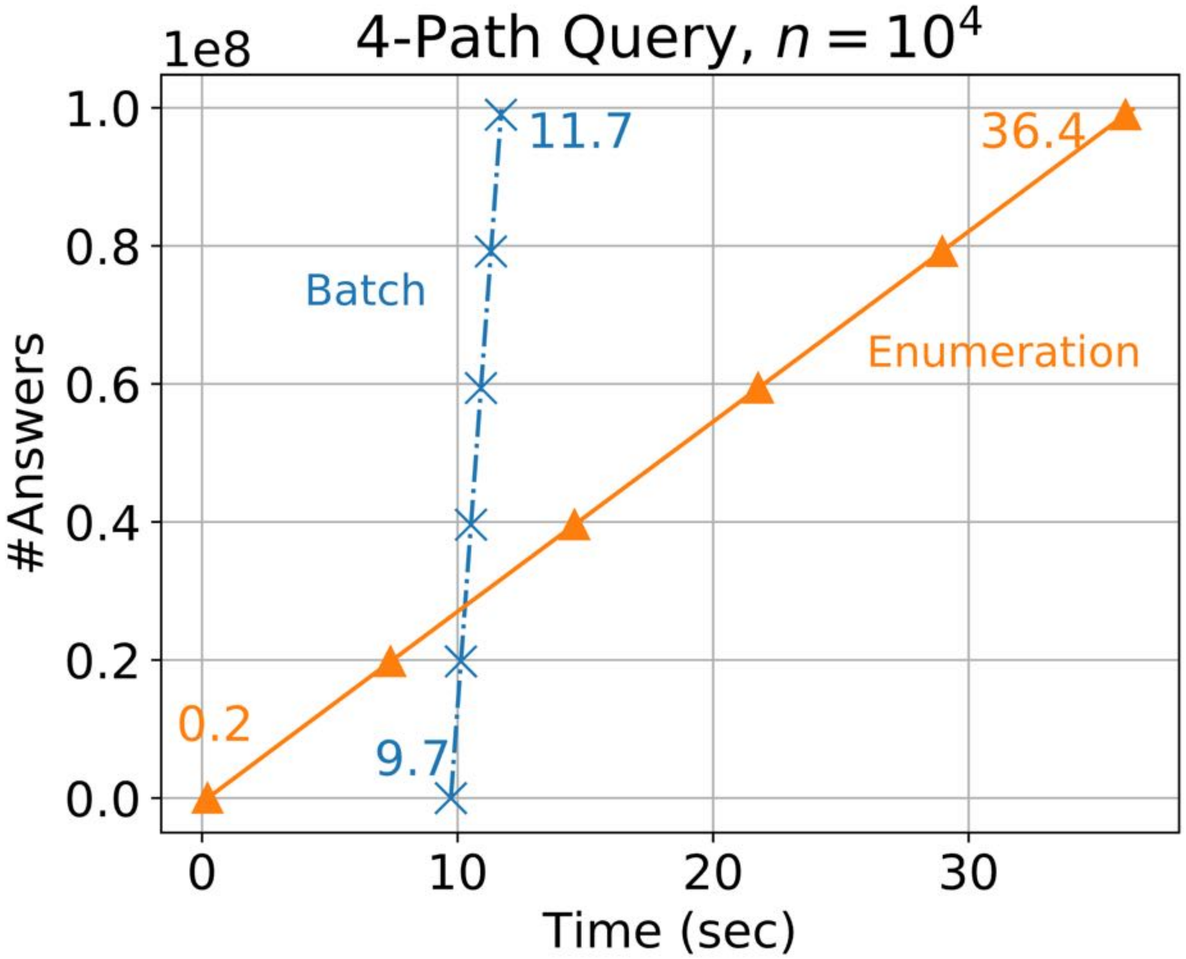


$r = O(n^{\rho^*})$ worst-case result size (AGM bound). ρ^* = fractional edge cover

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

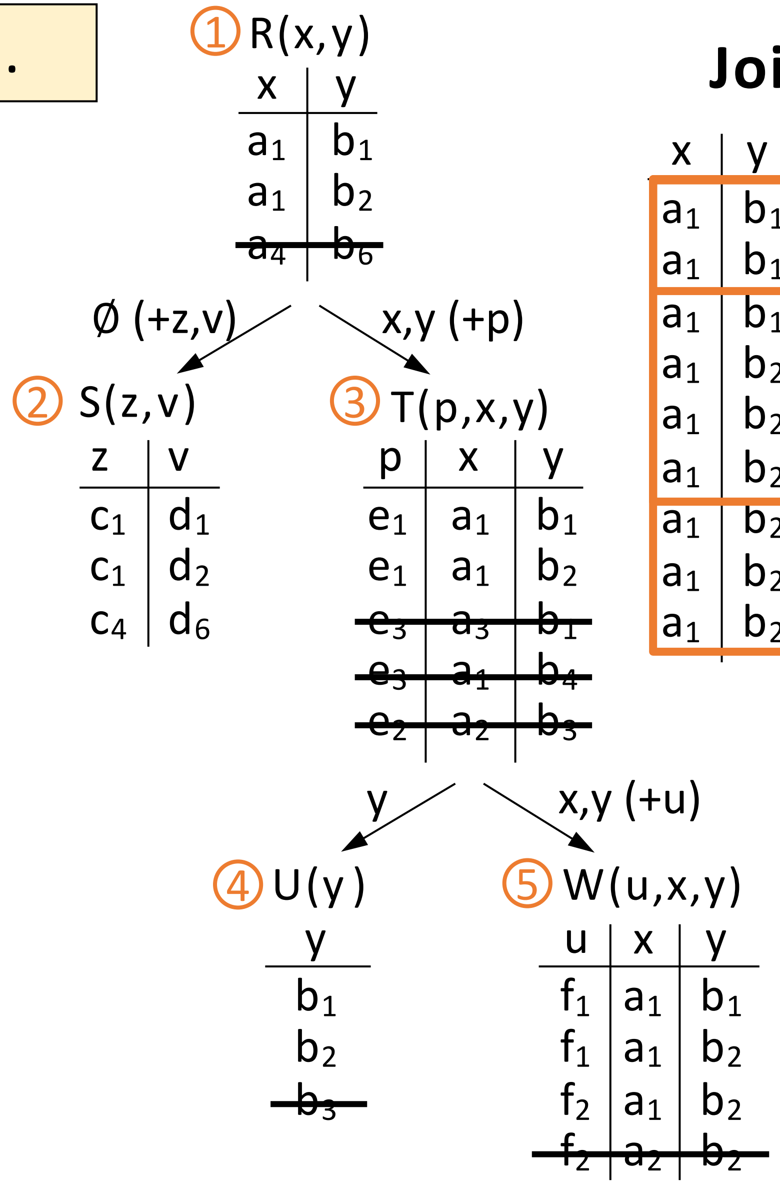
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

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 ☺~~



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

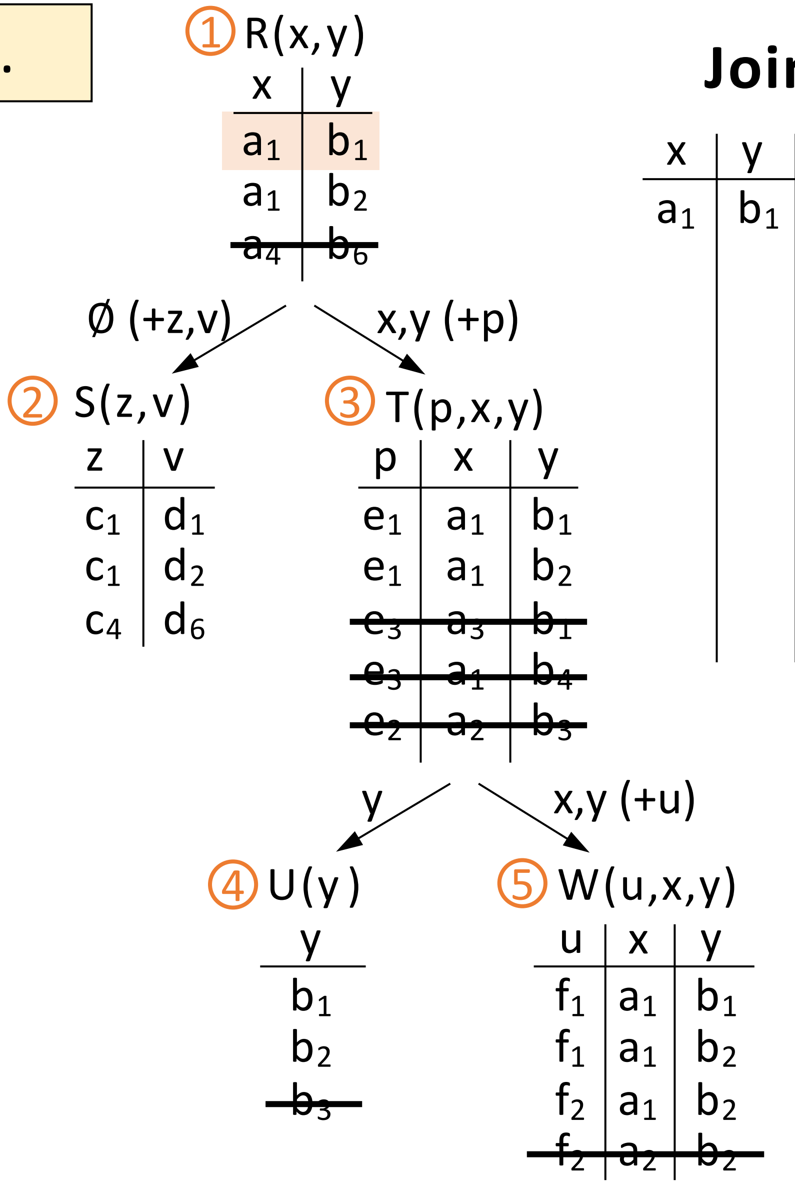
Q(x,y,z,v,p,u) :- R(x,y), S(z,v), T(p,x,y), U(y), W(u,x,y).

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}+Ø+{u}



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

x	y	z	v	p	u
a_1	b_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).

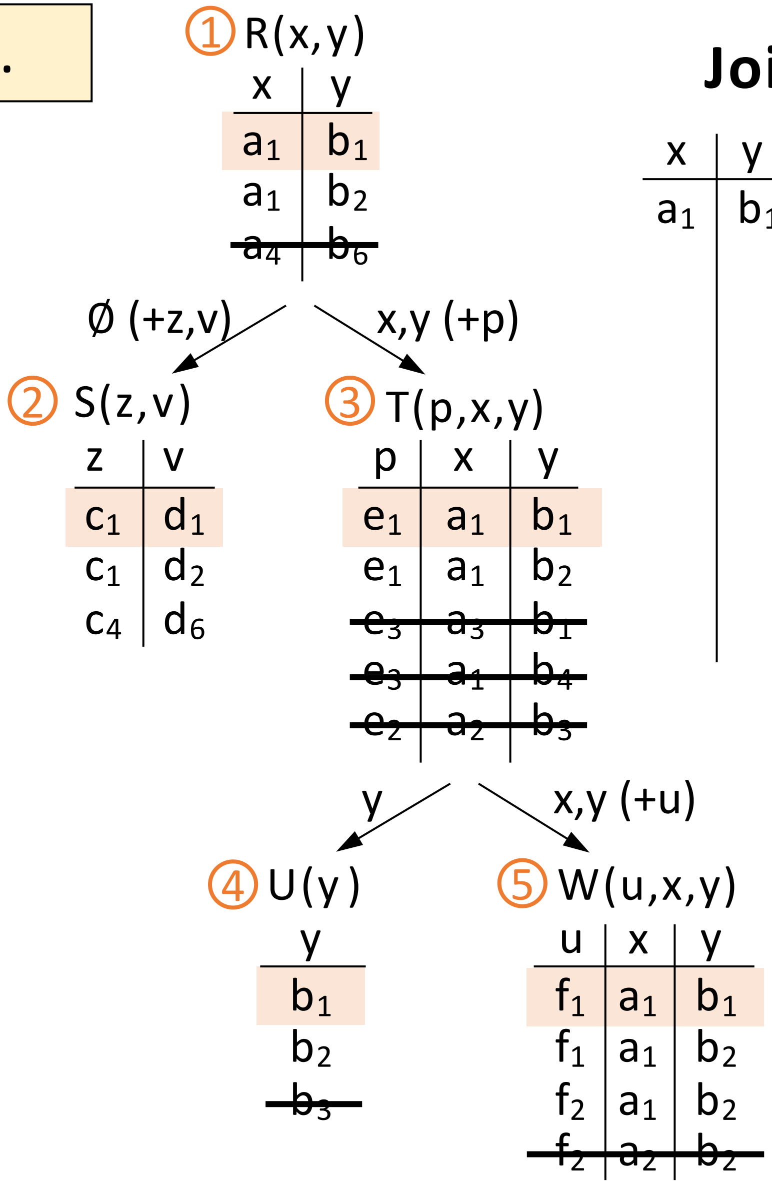
1. Semi-join phase ⋊ (remove dangling tuples) in *O*(n)

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

Join results

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

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).

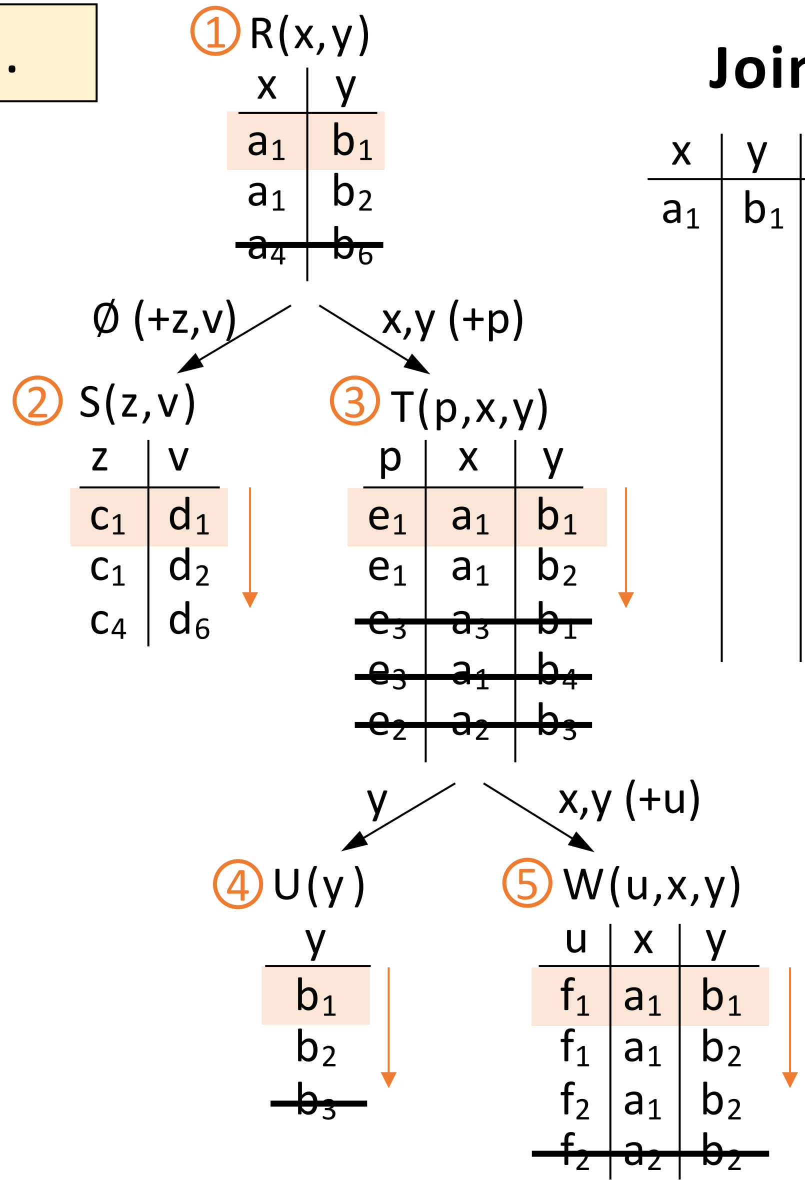
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

Ø (+z,v)

x,y (+p)

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

y

x,y (+u)

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

next tuple consistent with (x,y)

Join results

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

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).

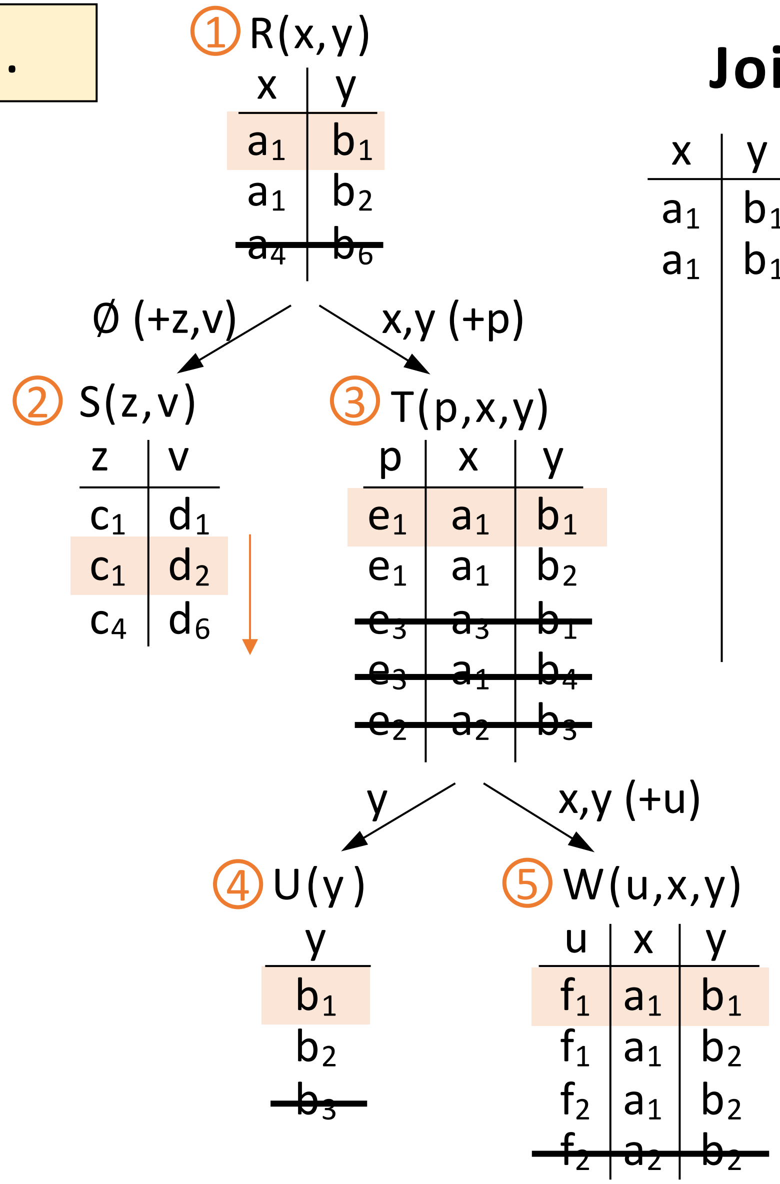
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

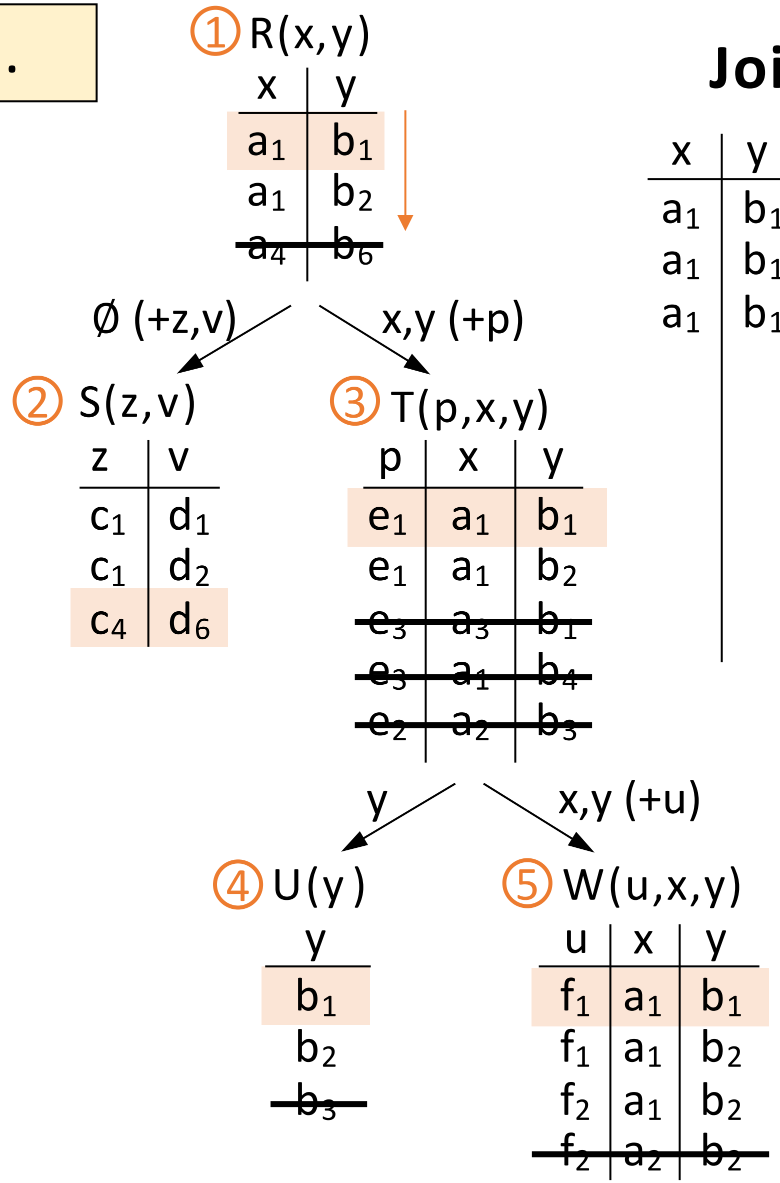
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

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~~

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

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).

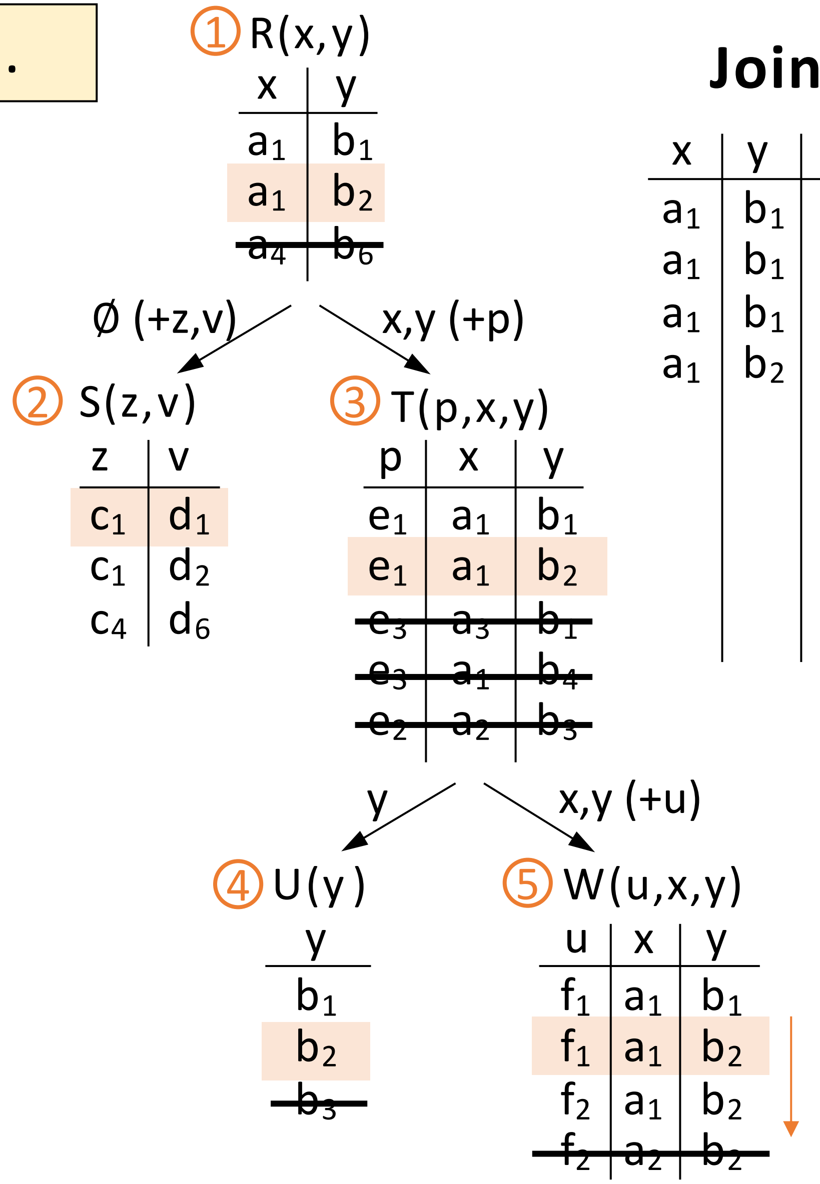
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

Ø (+z,v)

x,y (+p)

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

y

x,y (+u)

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

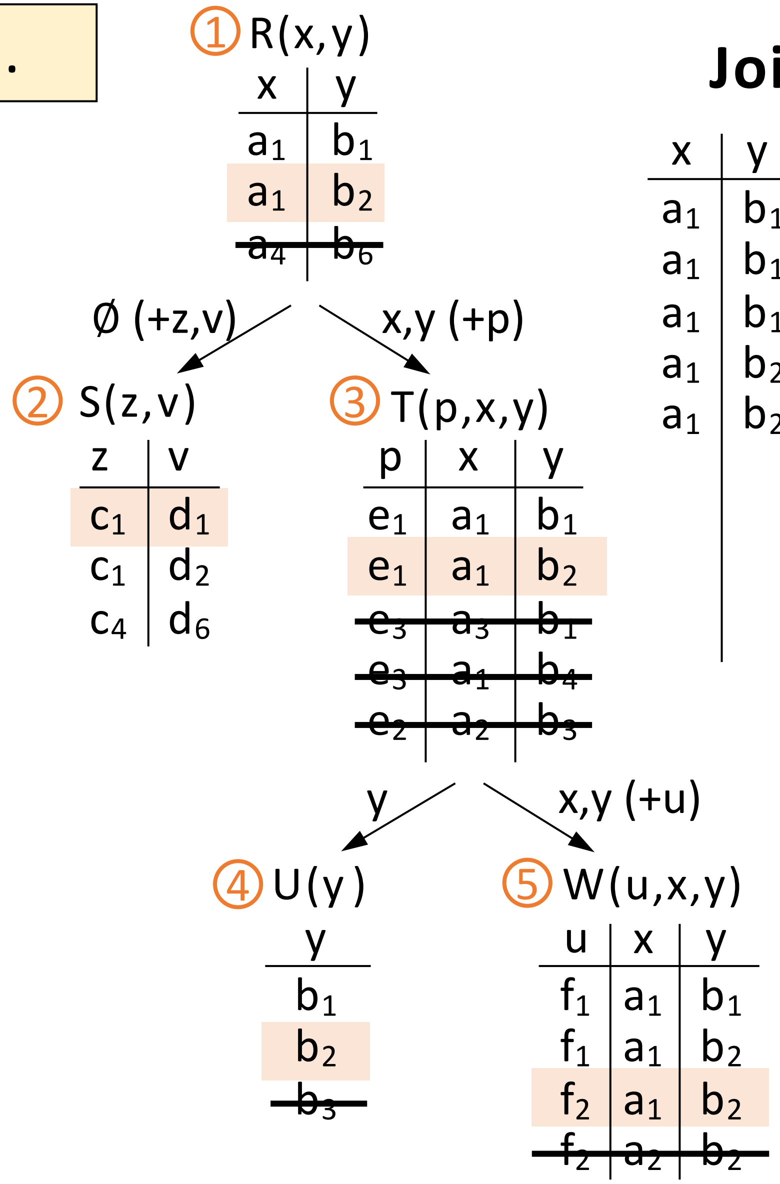
1. Semi-join phase ⋊ (remove dangling tuples) in O(n)

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

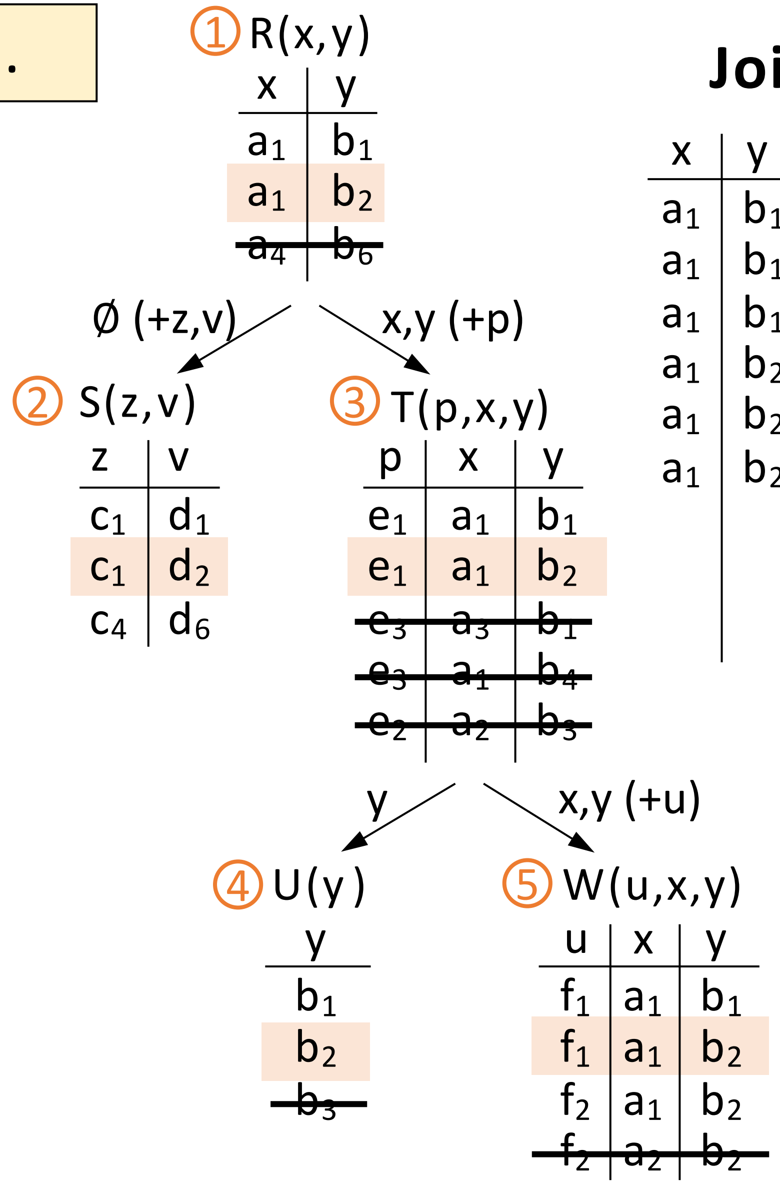
1. Semi-join phase ⋊ (remove dangling tuples) in O(n)

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

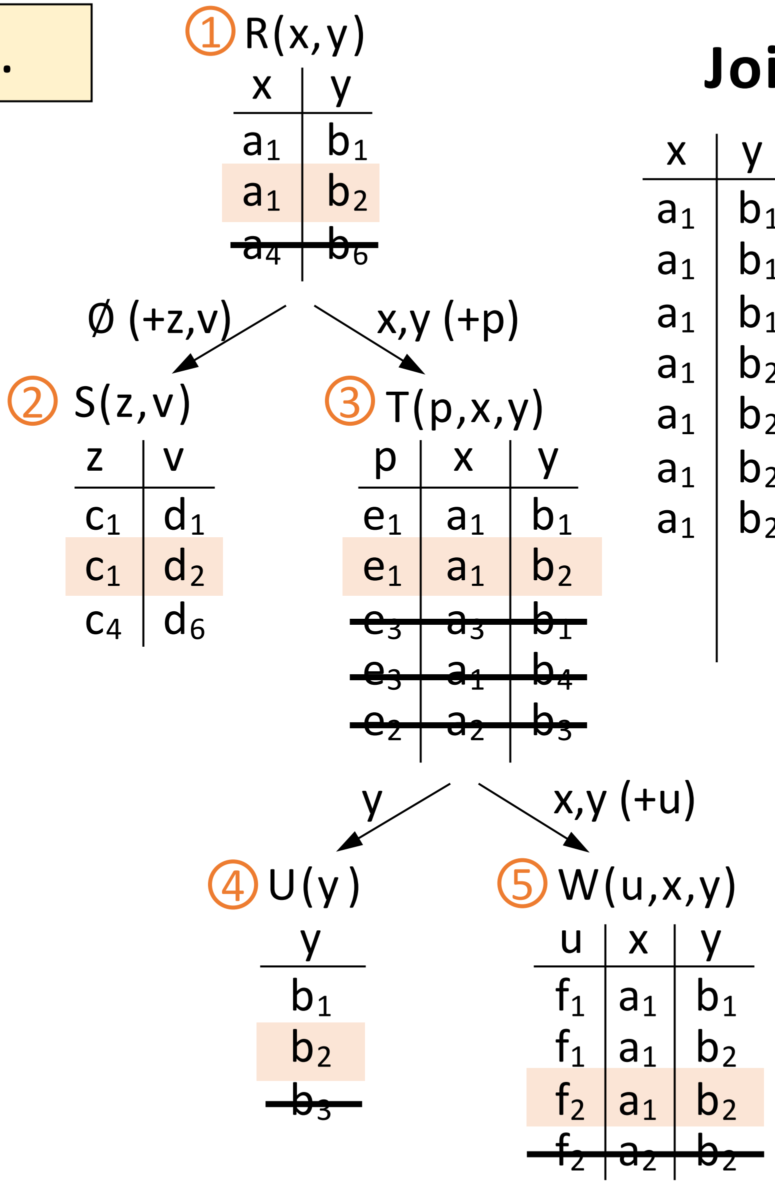
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

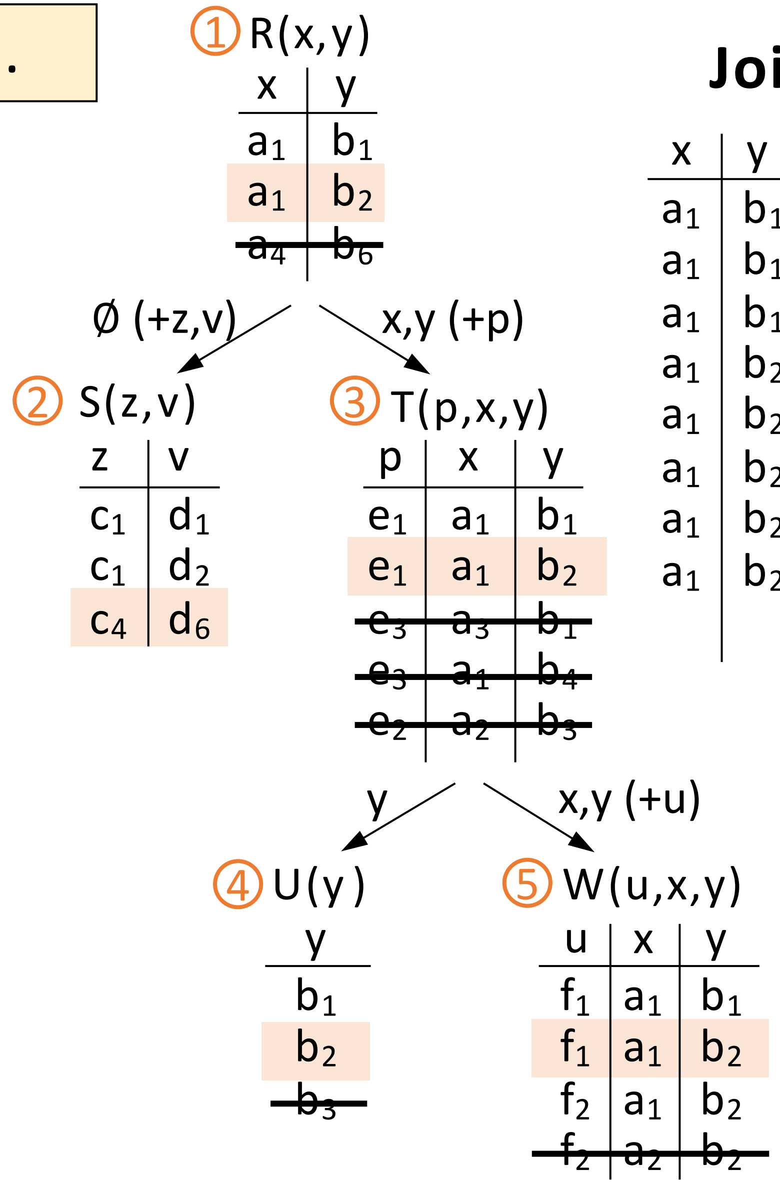
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

Ø (+z,v)

x,y (+p)

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

y

x,y (+u)

④ U(y)

y
b_1
b_2
~~b_3~~

⑤ 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~~

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

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).

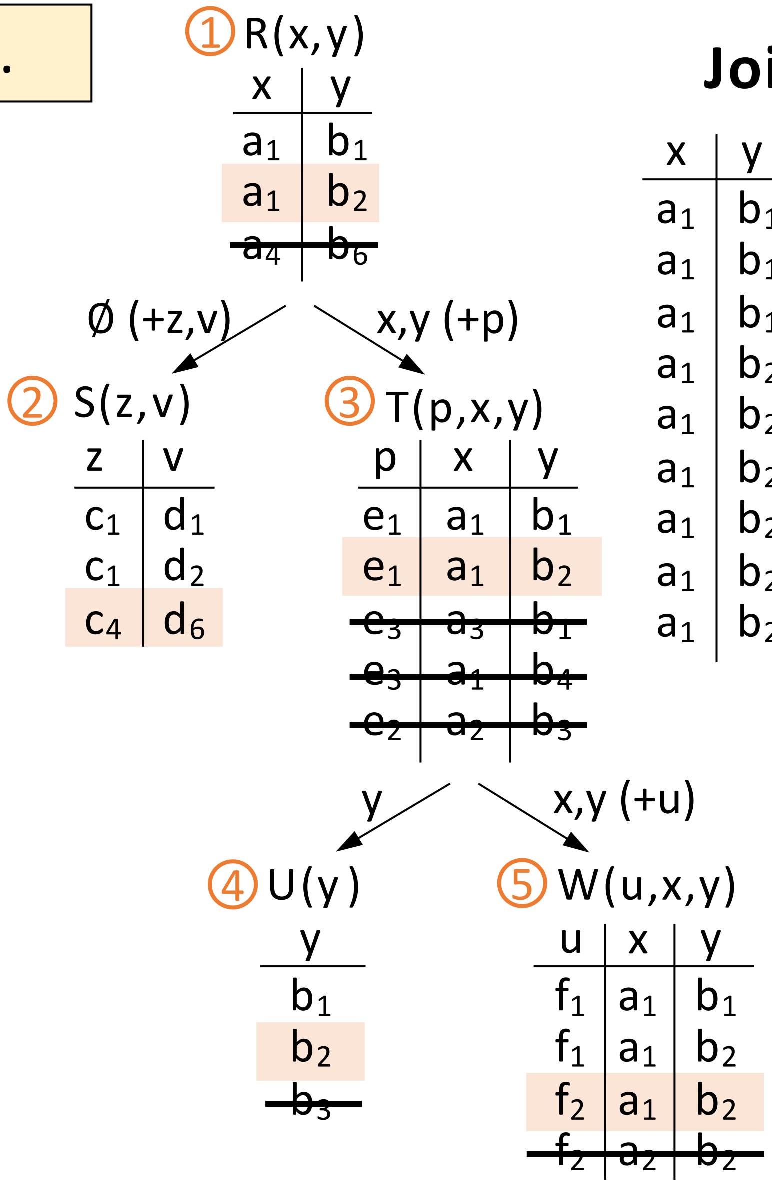
1. Semi-join phase ⋊ (remove dangling tuples) in $O(n)$

- 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}+Ø+{u}

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



① R(x,y)

x	y
a_1	b_1
a_1	b_2
~~a_4~~	~~b_6~~

② S(z,v)

z	v
c_1	d_1
c_1	d_2
c_4	d_6

③ 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~~

④ U(y)

y
b_1
b_2
~~b_3~~

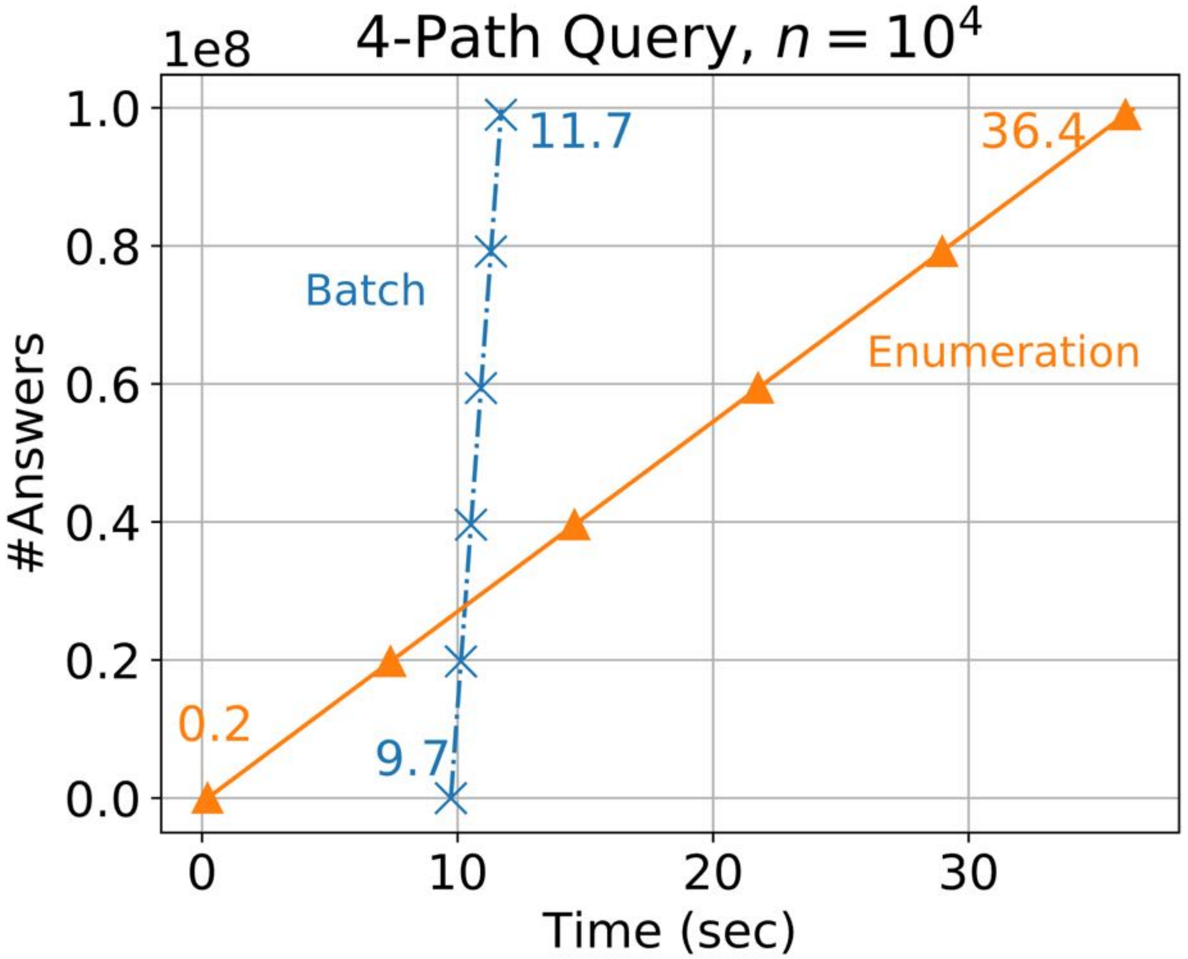
⑤ W(u,x,y)

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~~

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

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

How to deal with projections in enumeration ("free-connex")

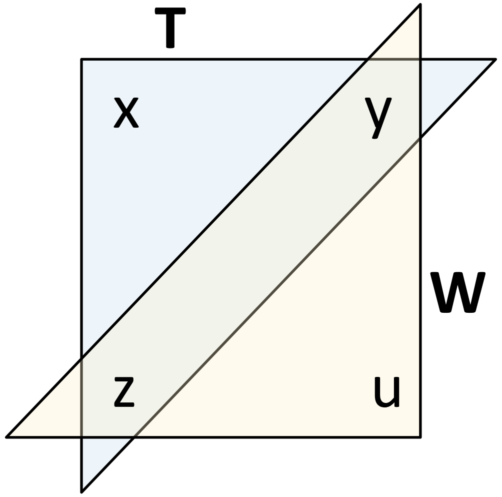
For acyclic queries, what changes for projections?

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

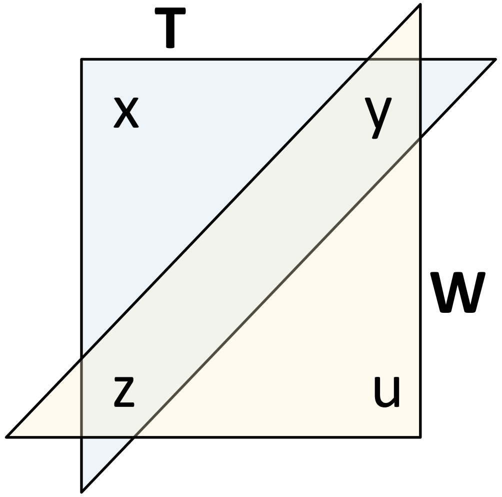
- Yannakakis works in O(n+n·r) for arbitrary projections
  - Then Enumeration works with linear delay and TTL is also O(n+n·r)
- Yannakakis works in O(n+r) for queries that are "free-connex"
  - Then Enumeration works with constant delay and TTL is also O(n+r)

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

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



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



Idea of "free-connex" from: Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18

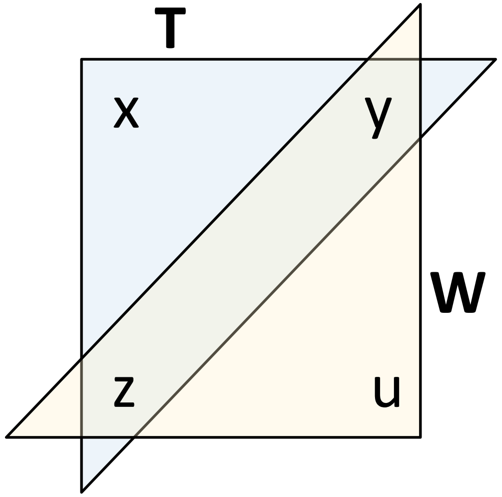
For acyclic queries, what changes for projections?

usually written as $O(n \cdot r)$ assuming output size $r \geq 1$

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

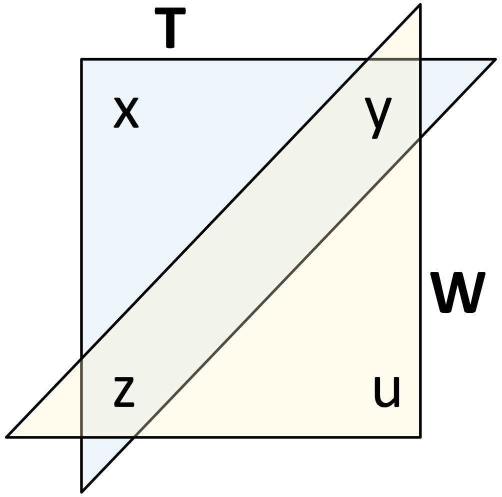
Free-connex: the query remains acyclic after adding an edge for the set of projected variables

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



?

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



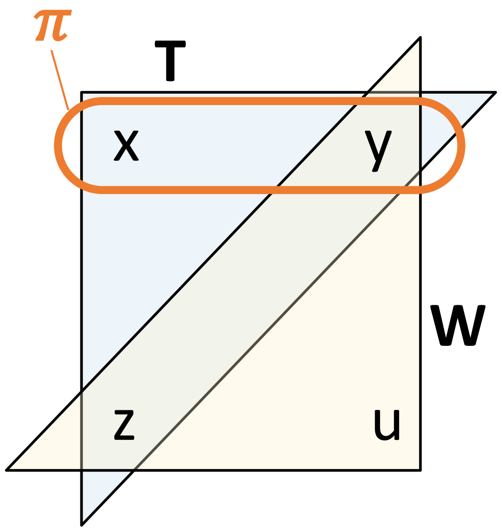
Idea of "free-connex" from: Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18

For acyclic queries, what changes for projections?

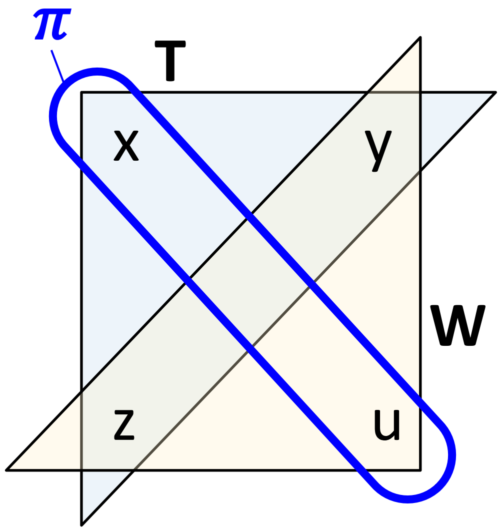
usually written as O(n·r) assuming output size r ≥1

- 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)

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



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



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

Which one is now free-connex ?

Idea of "free-connex" from: Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18

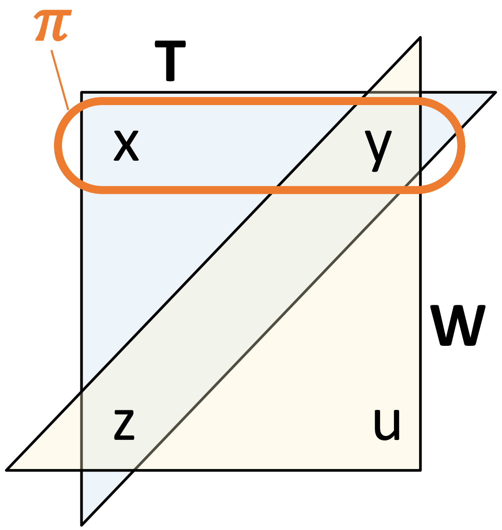
For acyclic queries, what changes for projections?

usually written as $O(n\cdot r)$ assuming output size $r \geq 1$

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

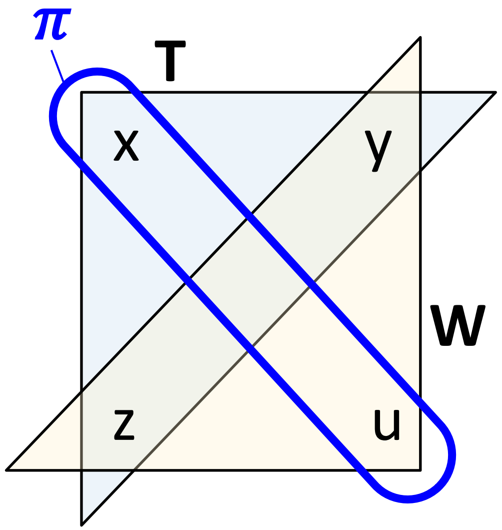
Free-connex: the query remains acyclic after adding an edge for the set of projected variables

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



free-connex ☺

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



not free-connex ☹

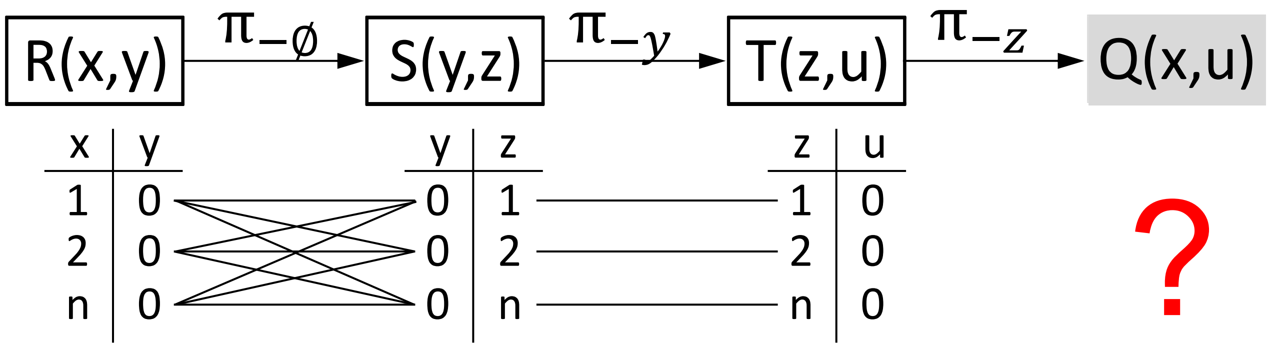
Idea of "free-connex" from: Bagan, Durand, Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL 2007. https://doi.org/10.1007/978-3-540-74915-8_18

Wolfgang Gatterbauer. Principles of scalable data management: https://northeastern-datalab.github.io/cs7240/

For acyclic queries, what changes for projections?

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

Join phase:

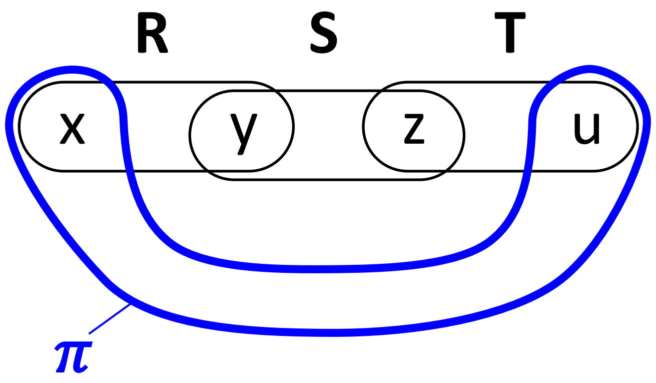


x	y
1	0
2	0
n	0

y	z
0	1
0	2
0	n

z	u
1	0
2	0
n	0

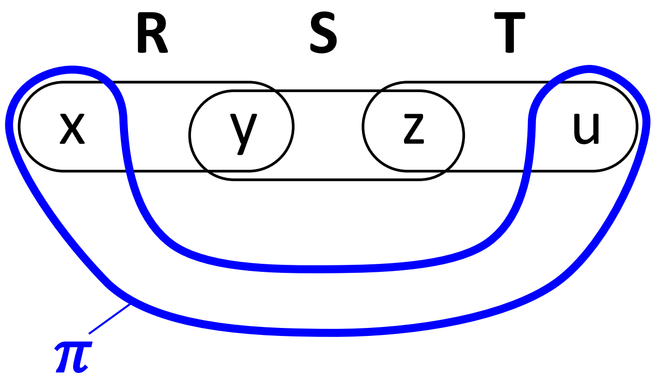
?



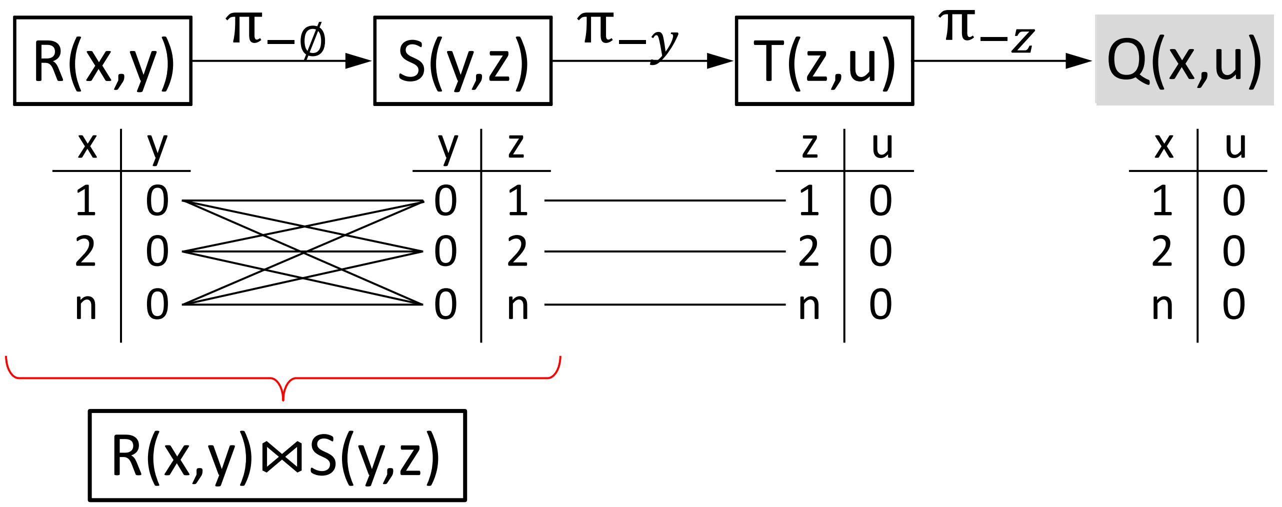
not free-connex ☹

For acyclic queries, what changes for projections?

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



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

O(n)

not free-connex ☹

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 rediscovered 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·r) for arbitrary projections
4. Enumeration slightly modifies Yannakakis to start returning results earlier (depth-first instead of breadth-first)