Updated 2/25/2020

T3: Efficient query evaluation
L14: Acyclic query evaluation

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp20)

https://northeastern-datalab.github.io/cs7240/sp20/

Version 2/21/2020

Outline: Acyclic conjunctive queries

- Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- Cyclic conjunctive queries

Islands of Tractability of CQ Evaluation

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

  Constraint Satisfaction Problem

  ≡ [Feder,Vardi'93]

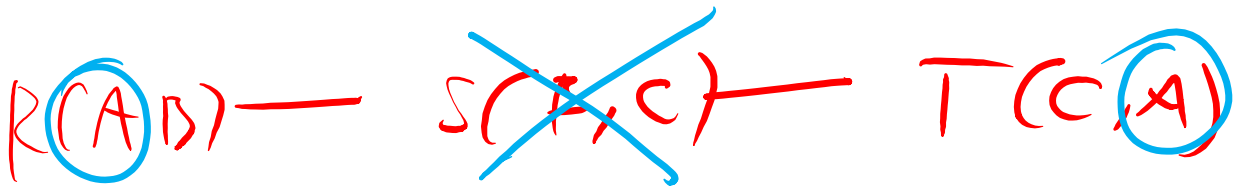
  Homomorphism Problem

  ≡ [Chandra,Merlin'77)

  Conjunctive Query Evaluation

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 [Yannakakis'81]
- A query is acyclic if its relations can be placed in a tree such that
  - 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|)



Outline: Acyclic conjunctive queries

- Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- Cyclic conjunctive queries

Semijoin ($\ltimes$)

Actor (<u>aid</u>, name, age)
Casts (<u>aid, mid, role</u>)
Movie(<u>mid</u>, name, year)

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

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

where A_1, …, A_n are the attributes in R

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

RA: ?

Semijoin ($\ltimes$)

Actor (aid, name, age)
Casts (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.

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

where A_1, …, A_n are the attributes in R

SQL: ?

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

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

Semijoin (⋉)

Actor (aid, name, age)
Casts (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.

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

where A_1, …, A_n are the attributes in R

Intuition: remove "dangling tuples" in R

- Example:

"Find actors who play some role."

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

Equivalent only if no duplicates in R!

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

```
SELECT *
FROM Actor
WHERE aid IN
        (SELECT aid
        FROM Casts)
```

Also works

Duplicates in R are preserved!

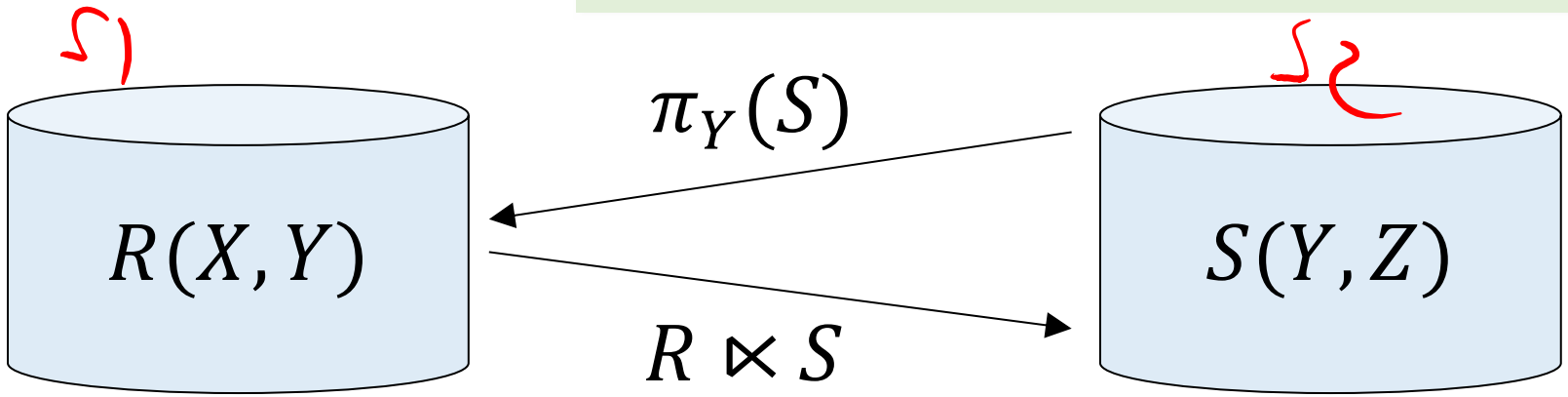
```
SELECT DISTINCT
         A.aid, name, age
FROM Actor A, Casts C
WHERE A.aid = C.aid
```

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!



$R \bowtie S = (R \ltimes S) \bowtie S$ law of semijoins

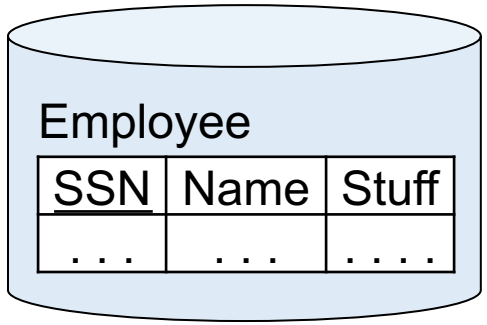
Semijoins in Distributed Databases

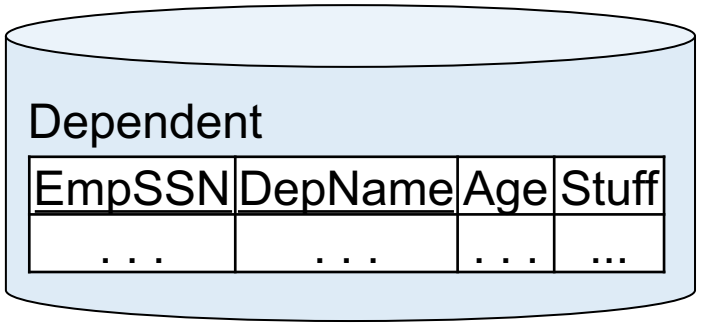
Task: compute with minimum data transfer:

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

Assumptions:

1. Very few employees have dependents.
2. Very few dependents have age > 71.
3. "Stuff" is big.





?

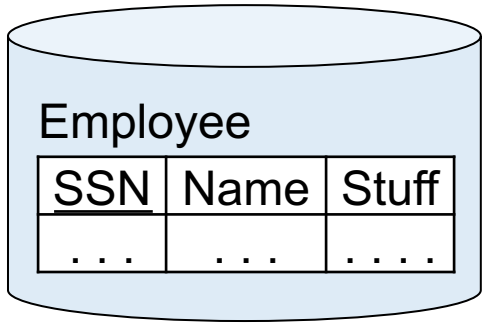
Example by Dan Suciu

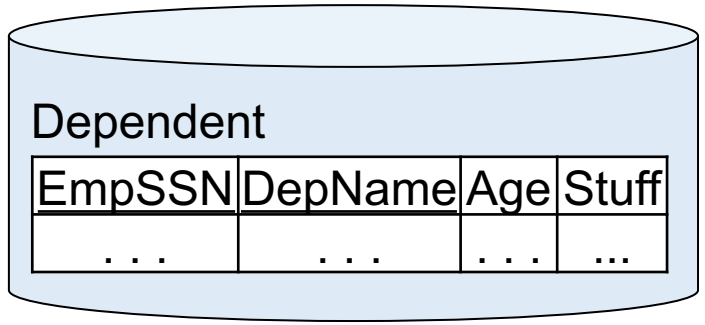
Semijoins in Distributed Databases

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. "Stuff" is big.





R

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

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

L

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

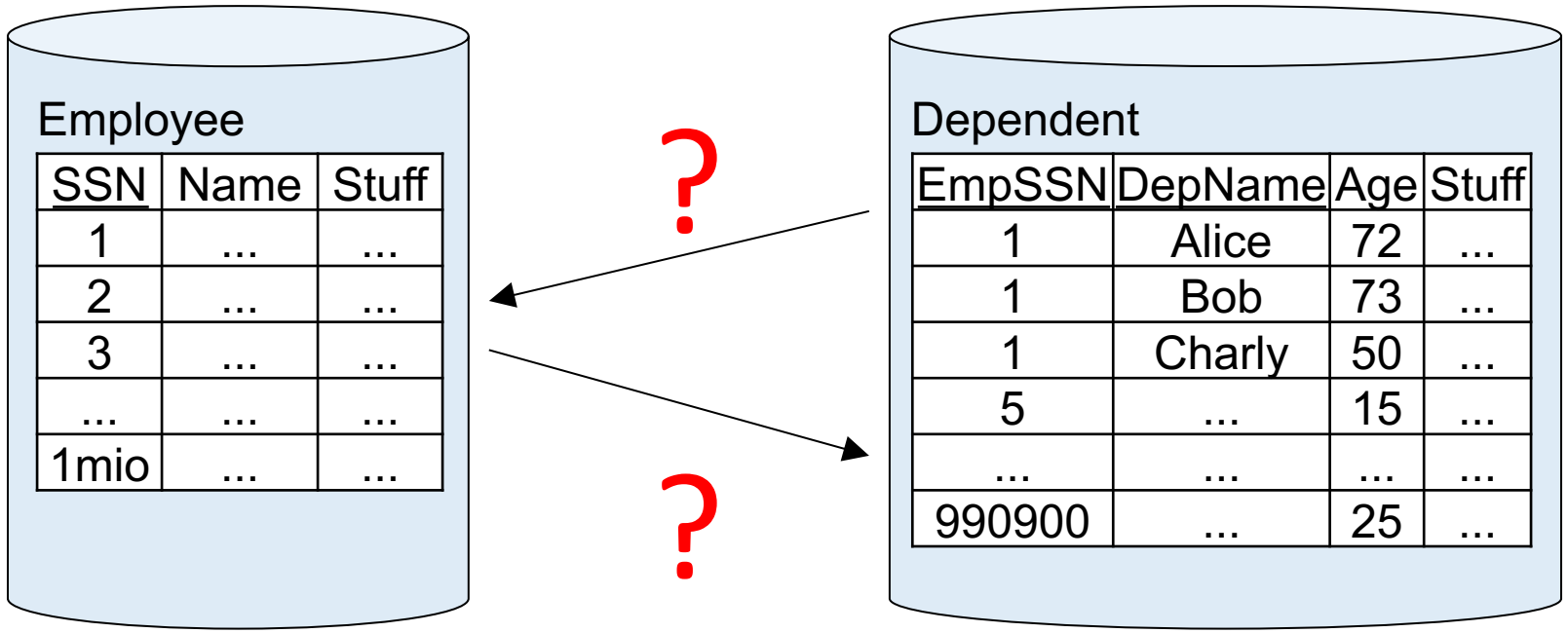
Example by Dan Suciu

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. "Stuff" is big.



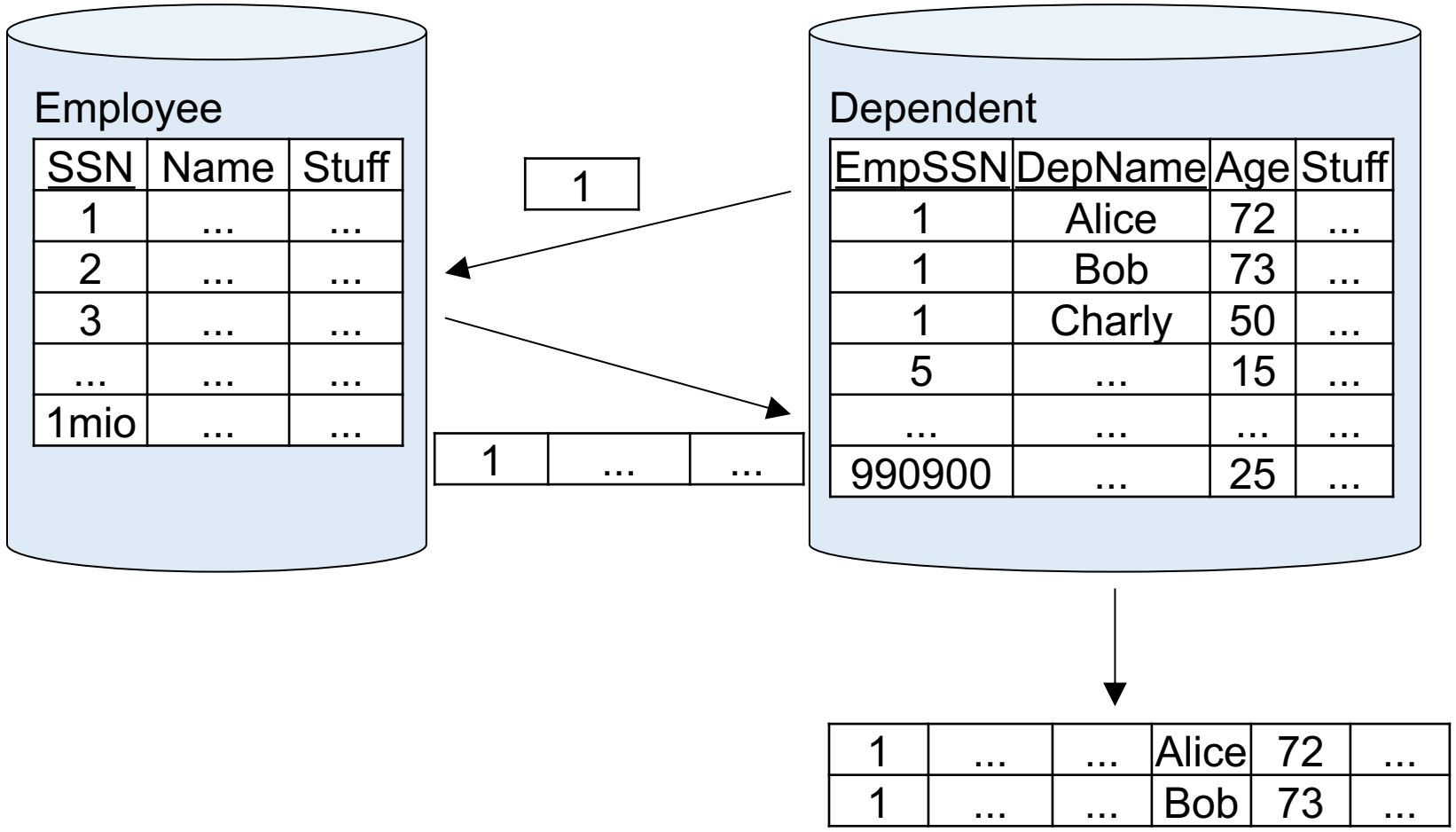
Example by Dan Suciu

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. "Stuff" is big.



Example by Dan Suciu

Summary: Law of Semijoins

- Definition: the semi-join operation is:

$$R \ltimes_C S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$

  - Formally, $R \ltimes_C S$ 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: $R(A_1,\ldots,A_n)$, $S(B_1,\ldots,B_m)$, Output: $T(A_1,\ldots,A_n)$

- The law of semijoins is:

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

  - Thus, removing dangling tuples does not change the query result

Outline: Acyclic conjunctive queries

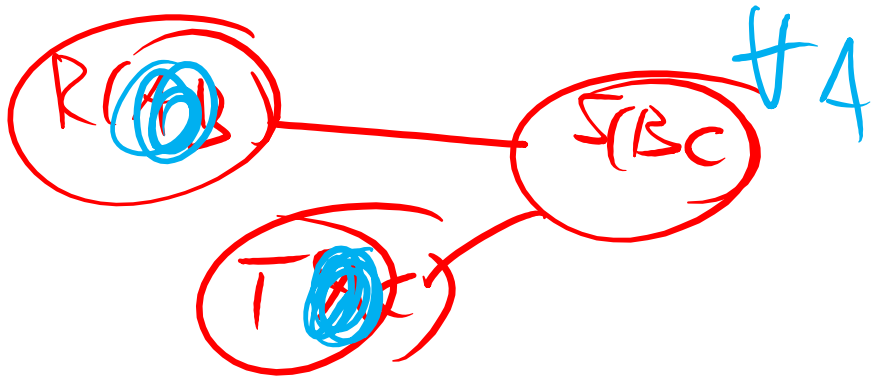
- Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- Cyclic conjunctive queries

Acyclic Conjunctive Queries

- Definition: A conjunctive query Q is acyclic if it has a join tree.
- Let Q be a conjunctive query of the form

  $Q(\mathbf{x}) : \exists \mathbf{y}\, [R_1(\mathbf{z}_1) \wedge R_2(\mathbf{z}_2) \wedge \ldots \wedge R_m(\mathbf{z}_m)]$

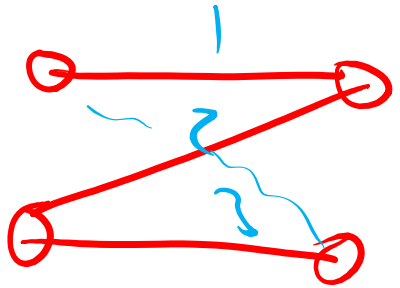
  $Q(\mathbf{x}) :- R_1(\mathbf{z}_1), R_2(\mathbf{z}_2), \ldots, R_m(\mathbf{z}_m)$



- A join tree for Q is a tree T =(V,E) such that:
  - V: The nodes of T are the atoms $R_i(\mathbf{z}_i)$, $1 \le i \le m$, of Q.
  - E: For every variable w occurring in Q, the set of the nodes of T that contain w forms a (connected) subtree of T;
    - $\{R_j(\mathbf{z}_j) \in V \mid z \text{ occurs in } R_j(\mathbf{z}_j)\}$ induces a connected subtree in T
    - 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 $R_k(\mathbf{z}_k)$
    - also called: running intersection property

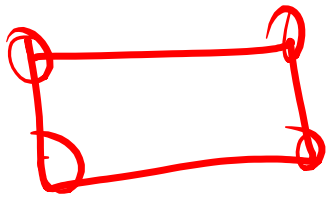
Acyclic Conjunctive Queries

- Path of length 4 ?



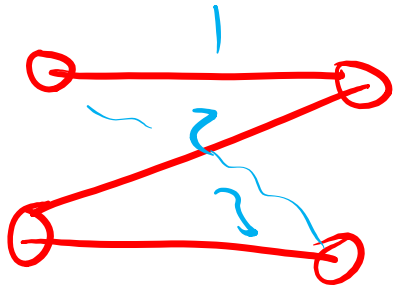
?

- Cycle of length 4 ?



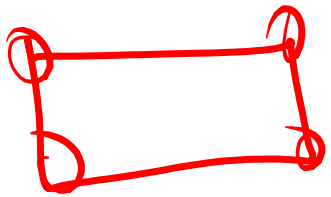
?

Acyclic Conjunctive Queries



- Path of length 4 ?
  - $P4(x_1,x_4) : \exists x_2\ x_3\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4)]$

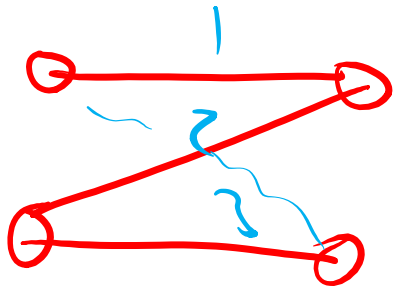
?



- Cycle of length 4 ?
  - $C4(\ ) : \exists\ x_1\ x_2\ x_3\ x_4\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4) \wedge E(x_4,x_1)]$

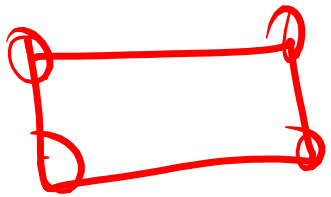
?

Acyclic Conjunctive Queries



- Path of length 4 ?
  - $P4(x_1,x_4) : \exists x_2\ x_3\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4)]$

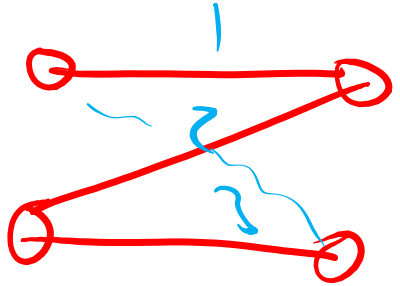
acyclic
(join tree is path)



- Cycle of length 4 ?
  - $C4(\ ) : \exists\ x_1\ x_2\ x_3\ x_4\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4) \wedge E(x_4,x_1)]$

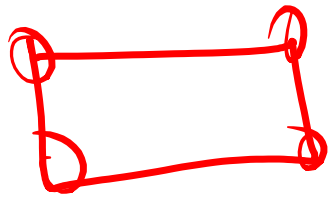
?

Acyclic Conjunctive Queries



- Path of length 4 ?
  - $P4(x_1,x_4) : \exists x_2\ x_3\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4)]$

acyclic
(join tree is path)



- Cycle of length 4 ?
  - $C4(\ ) : \exists\ x_1\ x_2\ x_3\ x_4\ [E(x_1,x_2) \wedge E(x_2,x_3) \wedge E(x_3,x_4) \wedge E(x_4,x_1)]$

cyclic

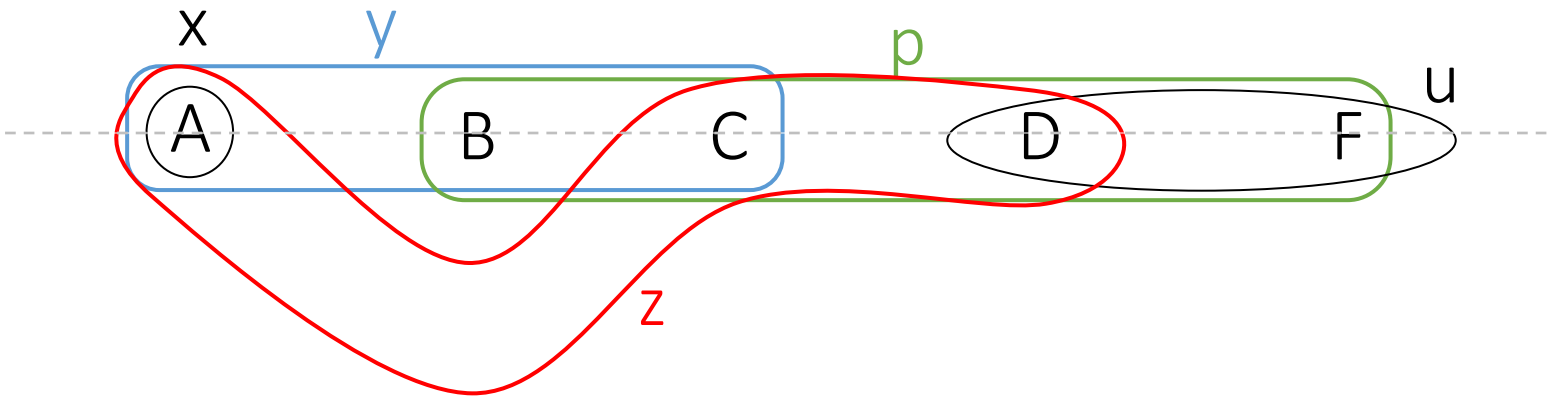
Acyclic Conjunctive Queries

- What about the following query?
  - Q( ) : ∃ x y z u p w [A(x,y,z) ∧ B(y,p) ∧ C(y,z,p) ∧ D(z,u,p) ∧ F(u,p,w)]

Acyclic Conjunctive Queries

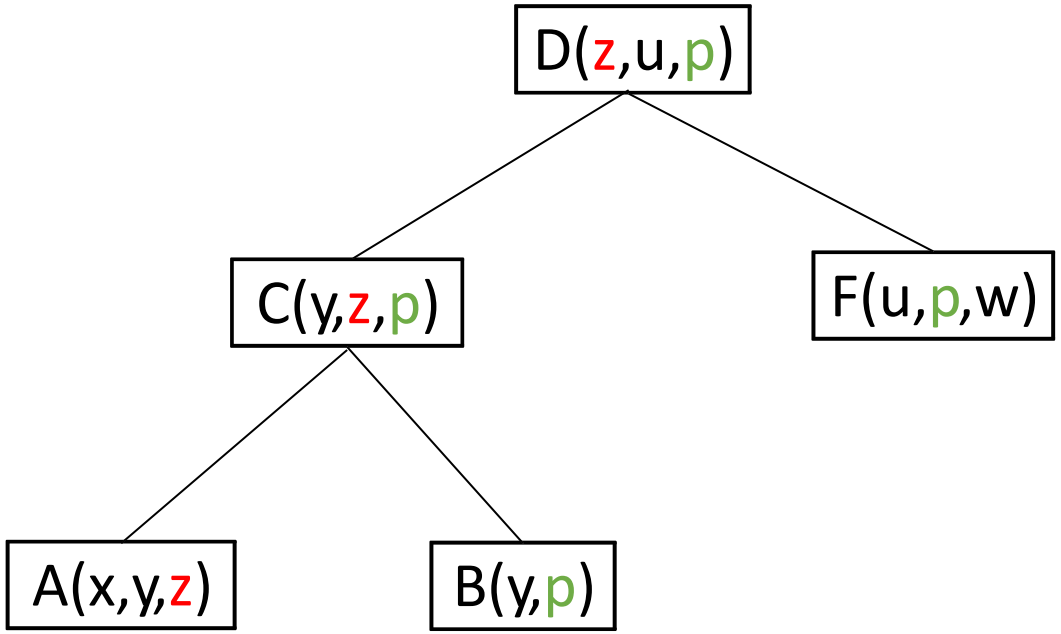
- What about the following query?
  - Q( ) : ∃ x y z u p w [A(x,y,z) ∧ B(y,p) ∧ C(y,z,p) ∧ D(z,u,p) ∧ F(u,p,w)]

Notice that it is not linear
(thus resilience is NPC)



Acyclic Conjunctive Queries

- What about the following query?
  - Q( ) : ∃ x y z u p w [A(x,y,z) ∧ B(y,p) ∧ C(y,z,p) ∧ D(z,u,p) ∧ F(u,p,w)]



Join Tree for Q

[Yannakakis'81] Acyclic Conjunctive Queries

- Theorem [Yannakakis'81]:
  - The Acyclic Conjunctive Query Evaluation Problem is tractable.
  - There is an algorithm for this problem with following properties:
    - If Q is a Boolean acyclic conjunctive query,
      then the algorithm runs in time: $O(|Q| \cdot |D|)$
    - If Q is a k-ary acyclic conjunctive query, $k \geq 1$,
      then the algorithm runs in time: $O(|Q| \cdot |D| + |Q(D)|)$,

    - i.e., it runs in input/output polynomial time,
      which is the "right" notion of tractability in this case

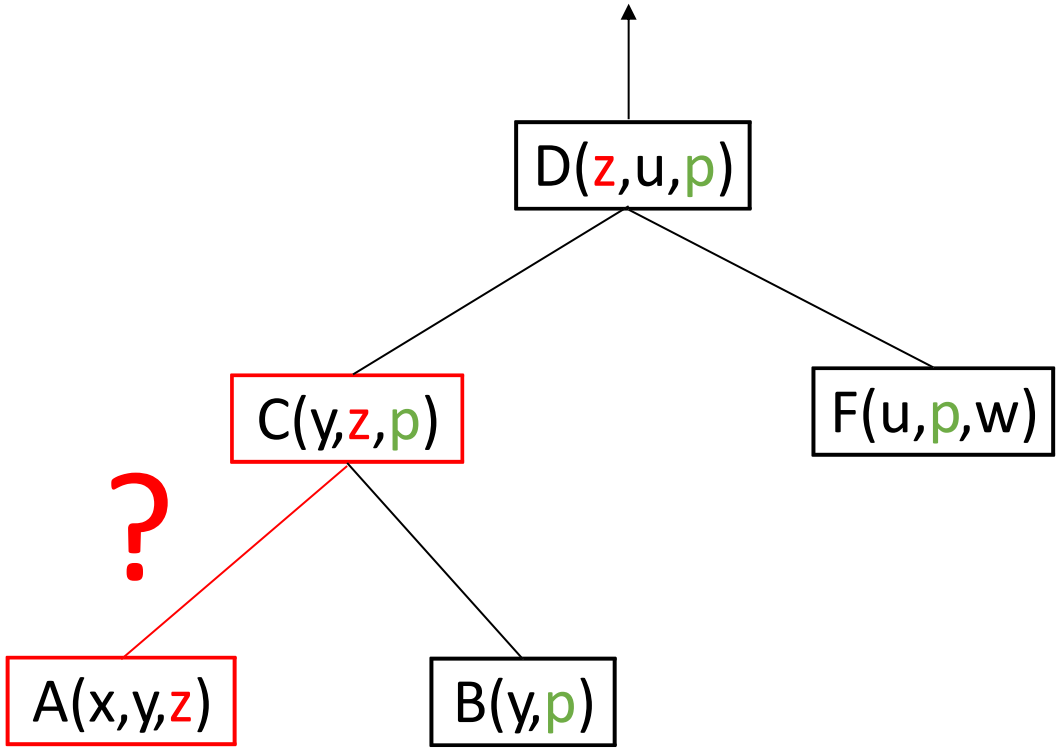
Yannakakis' Algorithm for Boolean acyclic CQs

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:
  - For each node $R_k(\mathbf{z}_k)$, compute the semi-joins of the (current) relation in the node $R_k(\mathbf{z}_k)$ with the (current) relations in the children of the node $R_k(\mathbf{z}_k)$.
- 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( ) : ∃ x y z u p w [A(x,y,z) ∧ B(y,p) ∧ C(y,z,p) ∧ D(z,u,p) ∧ F(u,p,w)]

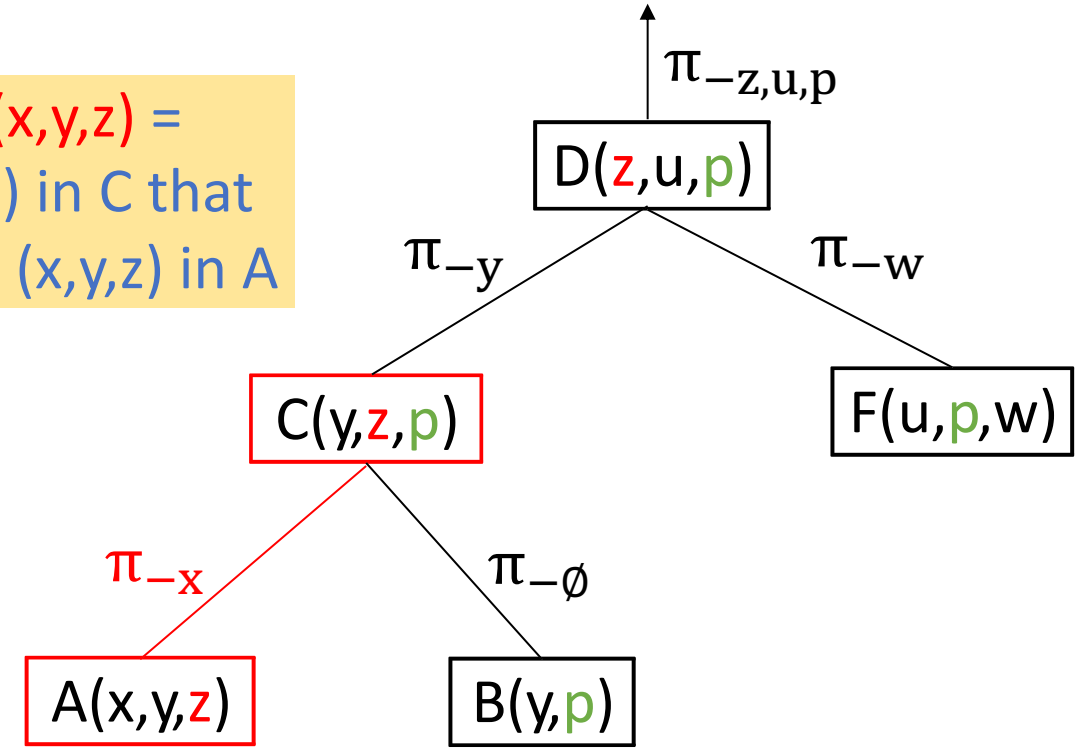


Join Tree for Q

Acyclic Conjunctive Queries

- Where are the semi-join reductions in following query:
  - Q( ) : ∃ x y z u p w [A(x,y,z) ∧ B(y,p) ∧ C(y,z,p) ∧ D(z,u,p) ∧ F(u,p,w)]

C(y,z,p) ⋉ A(x,y,z) =
all triples (y,z,p) in C that
"match" a triple (x,y,z) in A



Join Tree for Q

More on Yannakakis' Algorithm

- The join tree makes it possible to avoid exponential explosion in intermediate computations.
- The algorithm can be extended to non-Boolean conjunctive queries using two more traversals of the join tree.
- There are efficient algorithms for detecting acyclicity and computing a join tree.
  - 1984: Tarjan and Yannakakis: Linear-time algorithm for detecting acyclicity and computing a join tree.
  - 1998: Gottlob, Leone, Scarcello: Detecting acyclicity is LOGCFL-complete

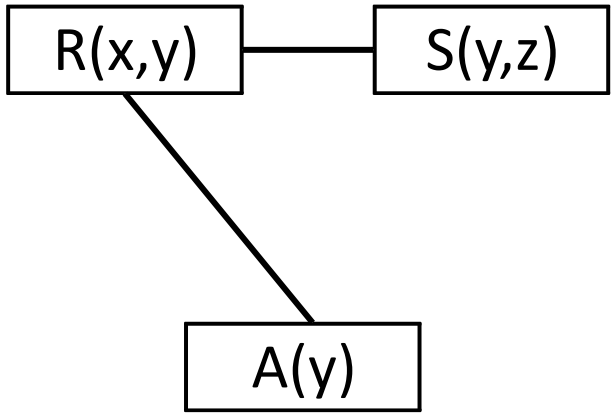
Are join trees unique?

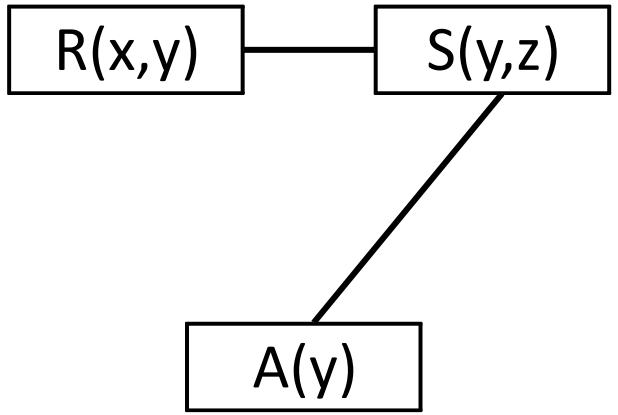
R(x,y),S(y,z), A(y)

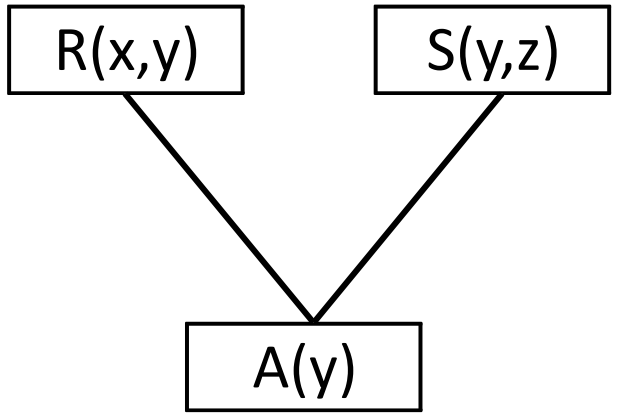
?

Are join trees unique?

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





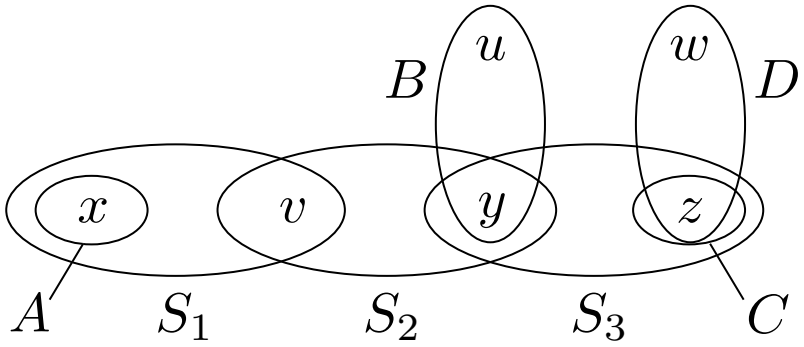


Outline: Acyclic conjunctive queries

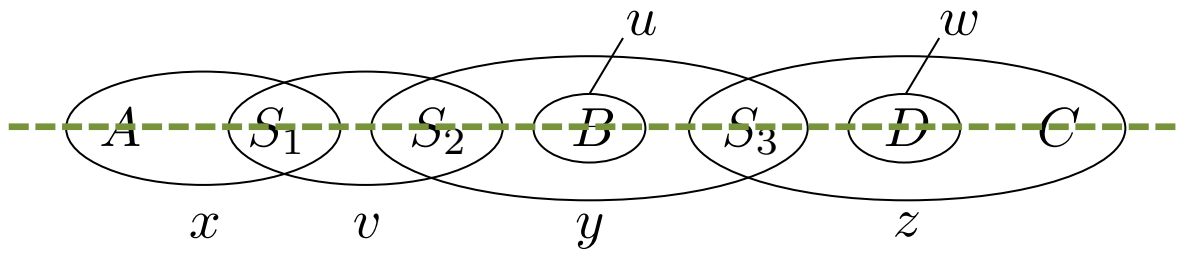
- Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- Cyclic conjunctive queries

Query Hypergraph (vs. Dual Hypergraph)

$$q :- A(x)S_1(x,v)S_2(v,y)B(y,u)S_3(y,z)D(z,w)C(z)$$



Query hypergraph



Query dual hypergraph

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - 1. remove isolated nodes (variables)
    - 2. remove consumed or empty edges (atoms)
    - (An ear is a hyperedge H such that we can divide its nodes into two groups:
      - those that appear in H and no other hyperedge, and
      - those that are contained in another hyperedge G.)
  - Reduction sequence allows to build a join tree efficiently

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

- Background:
  - Atom R(**z**) is empty if $|\mathbf{z}|=0$
  - Atom $R_1(\mathbf{z}_1)$ is contained in $R_2(\mathbf{z}_2)$ if $\mathbf{z}_1 \subseteq \mathbf{z}_2$

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

Join tree

?

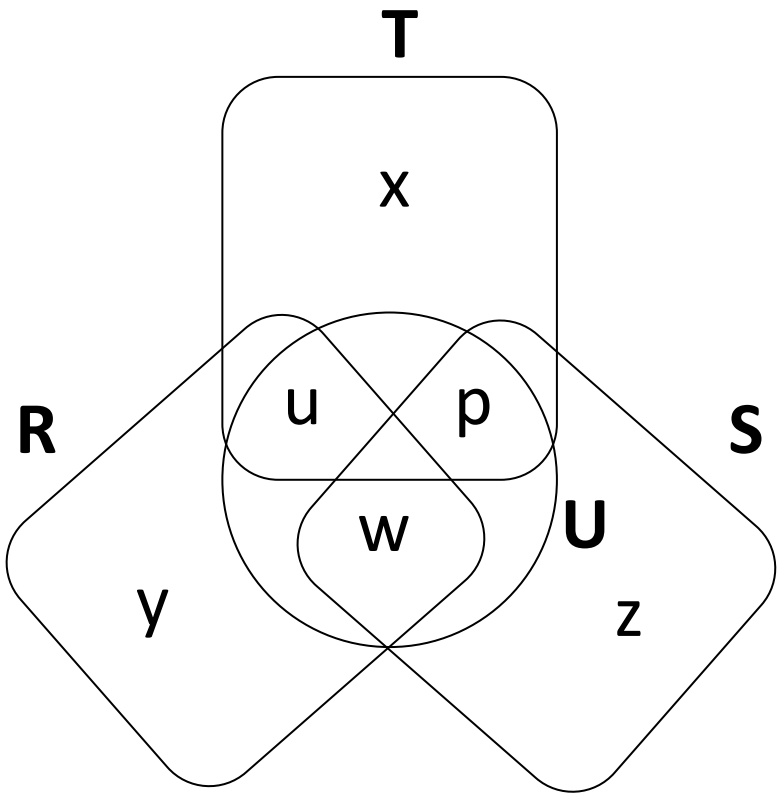
Query hypergraph

?

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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



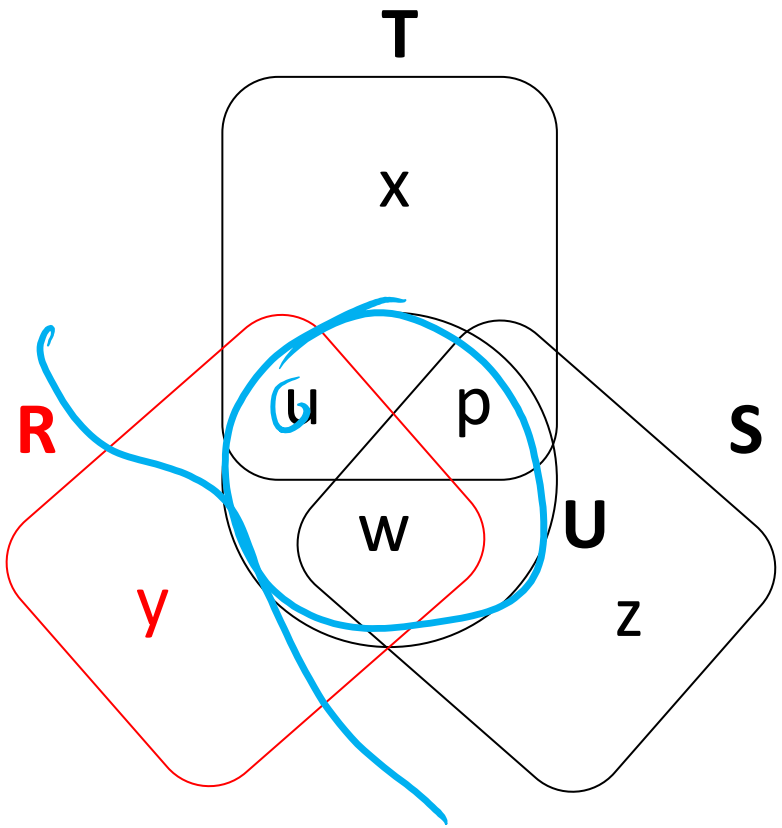
Join tree

?

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

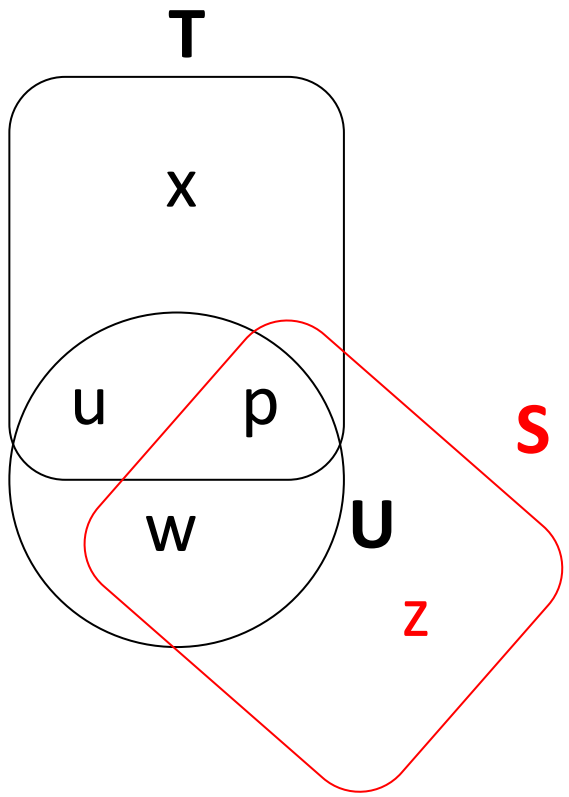


R(y,u,w)

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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



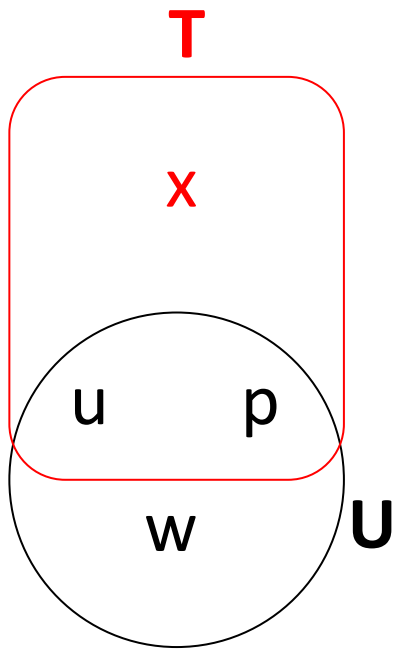
R(y,u,w)

S(z,p,w)

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

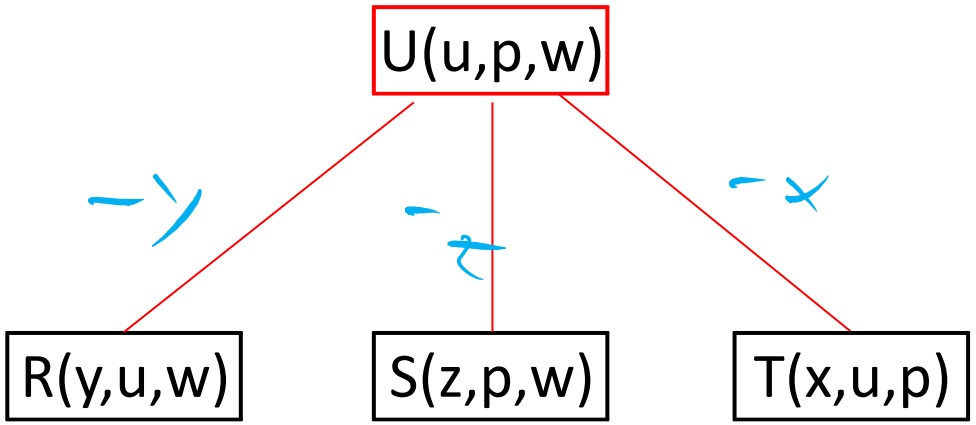


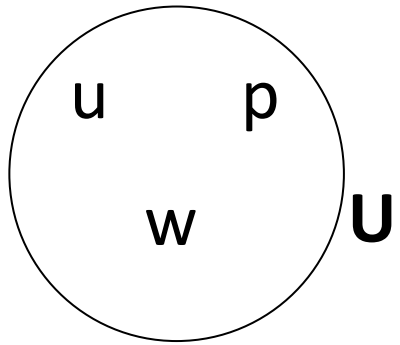
R(y,u,w) S(z,p,w) T(x,u,p)

GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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





GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

Join tree

?

Query hypergraph

?

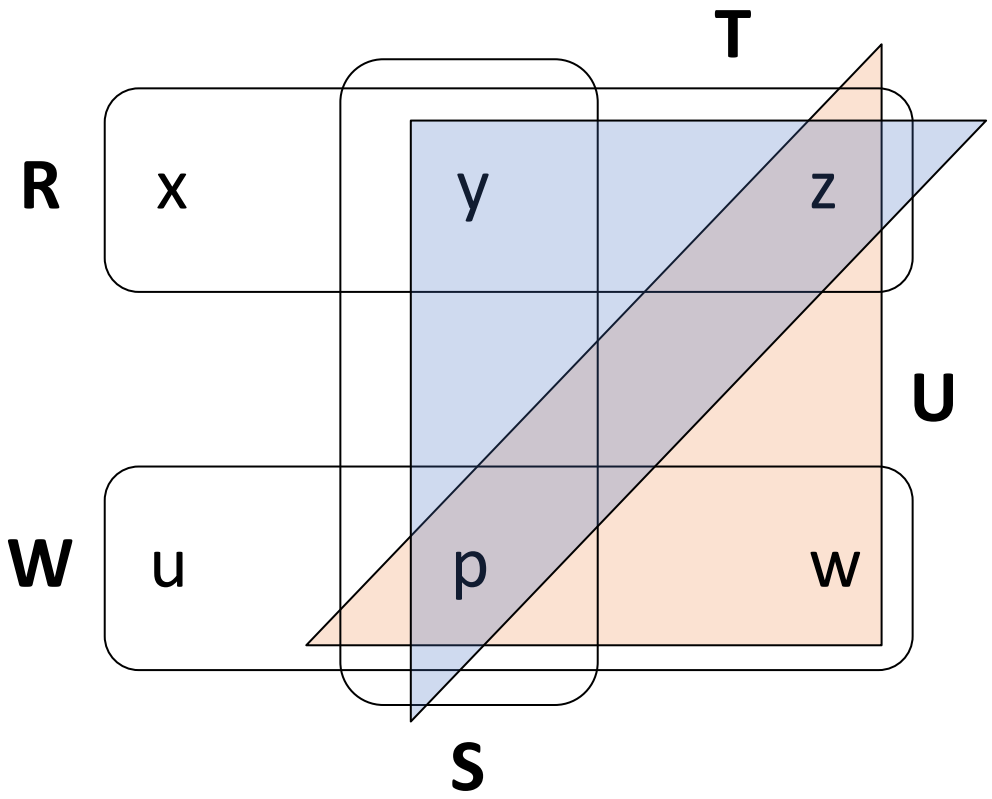
GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

Join tree

?



GYO reduction (Graham-Yu-Ozsoyoglu)

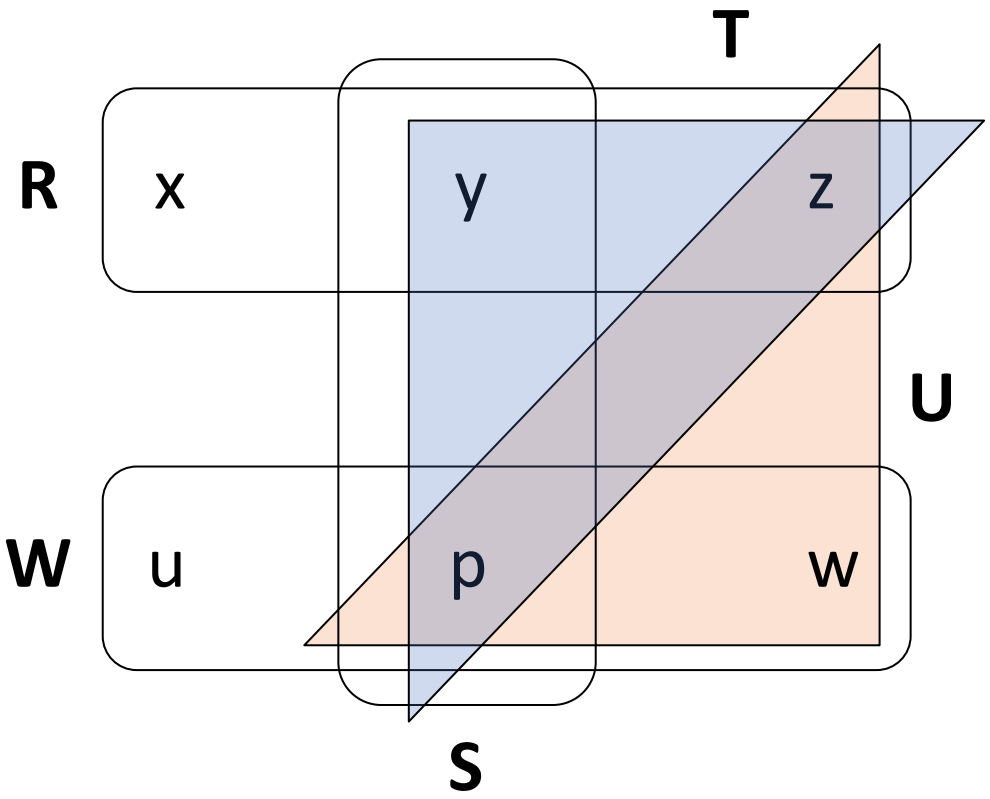
- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

U(z,p,w)

T(y,z,p)

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



GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

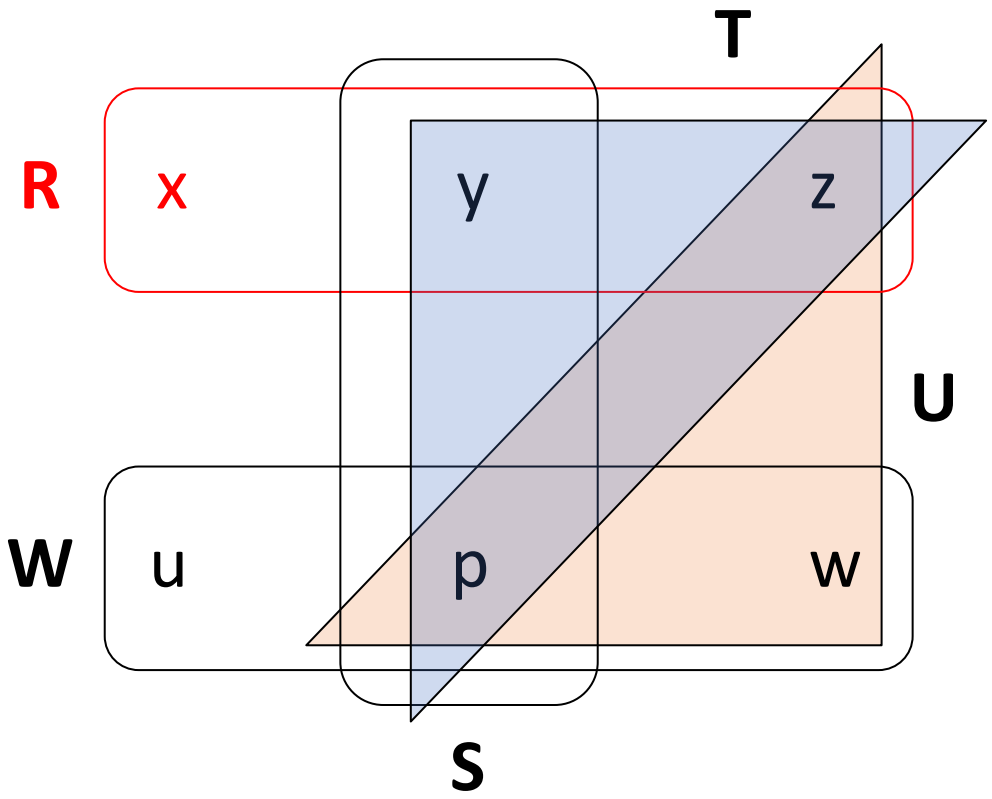
U(z,p,w)

T(y,z,p)

R(x,y,z)

S(y,p)

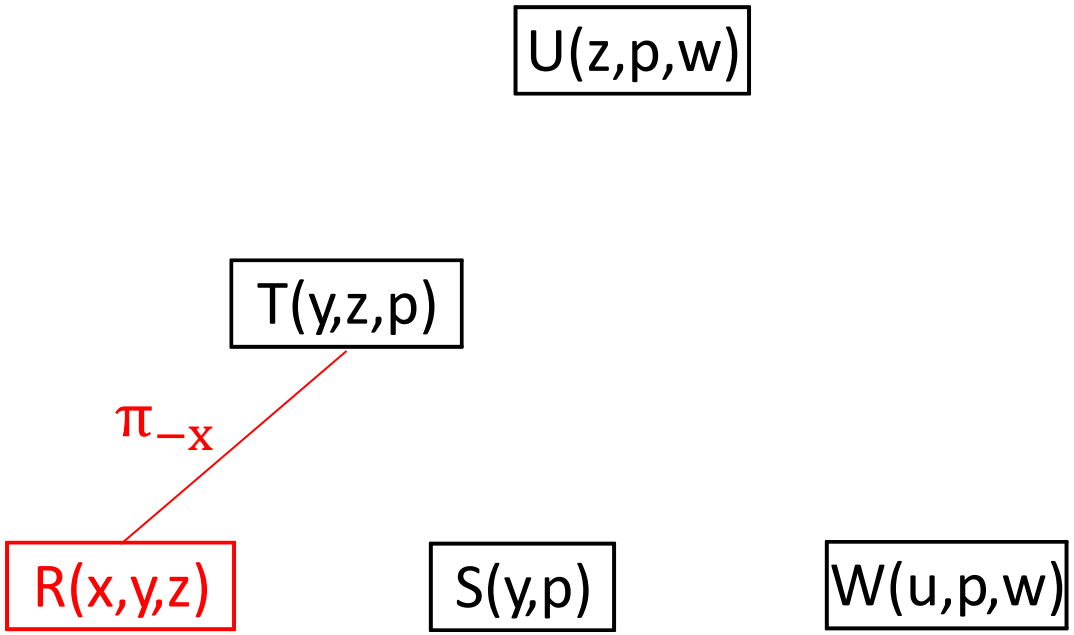
W(u,p,w)

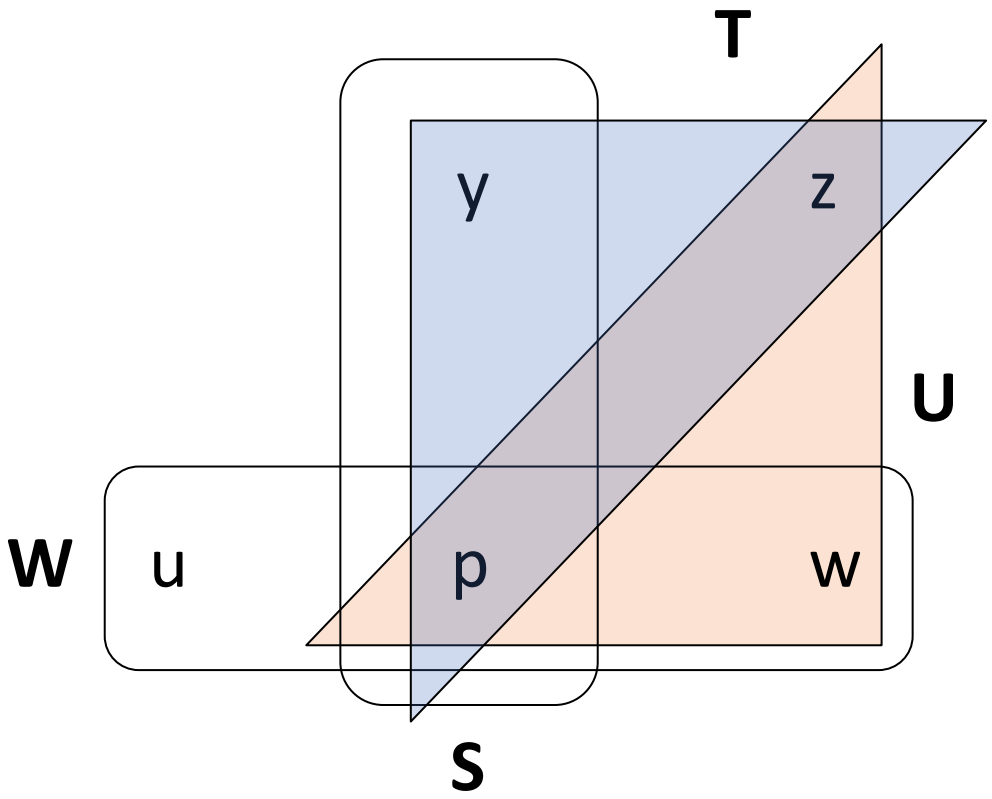


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

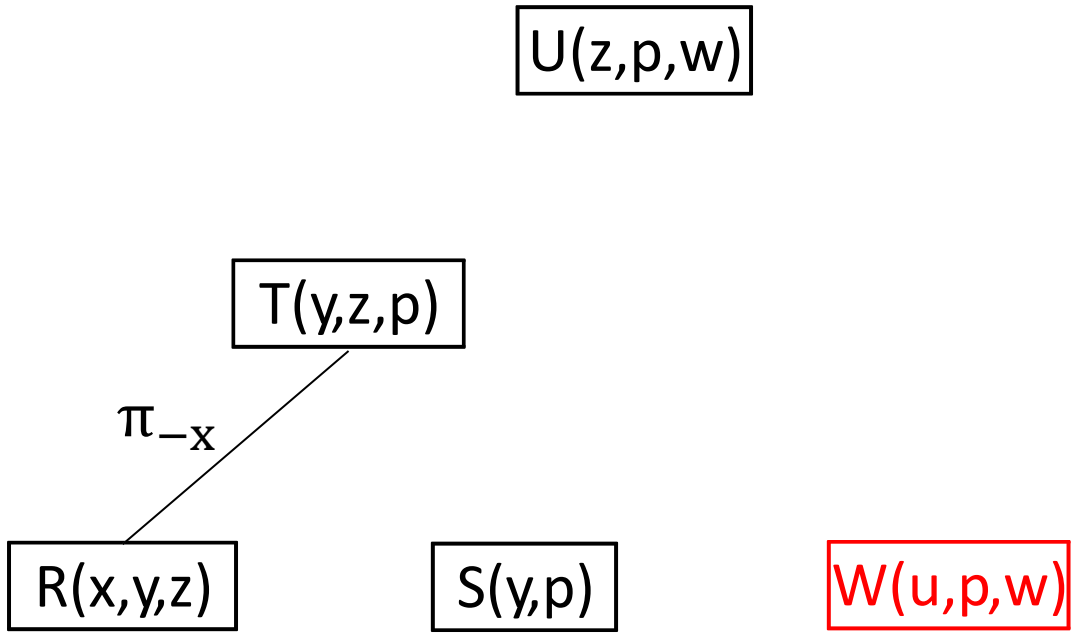


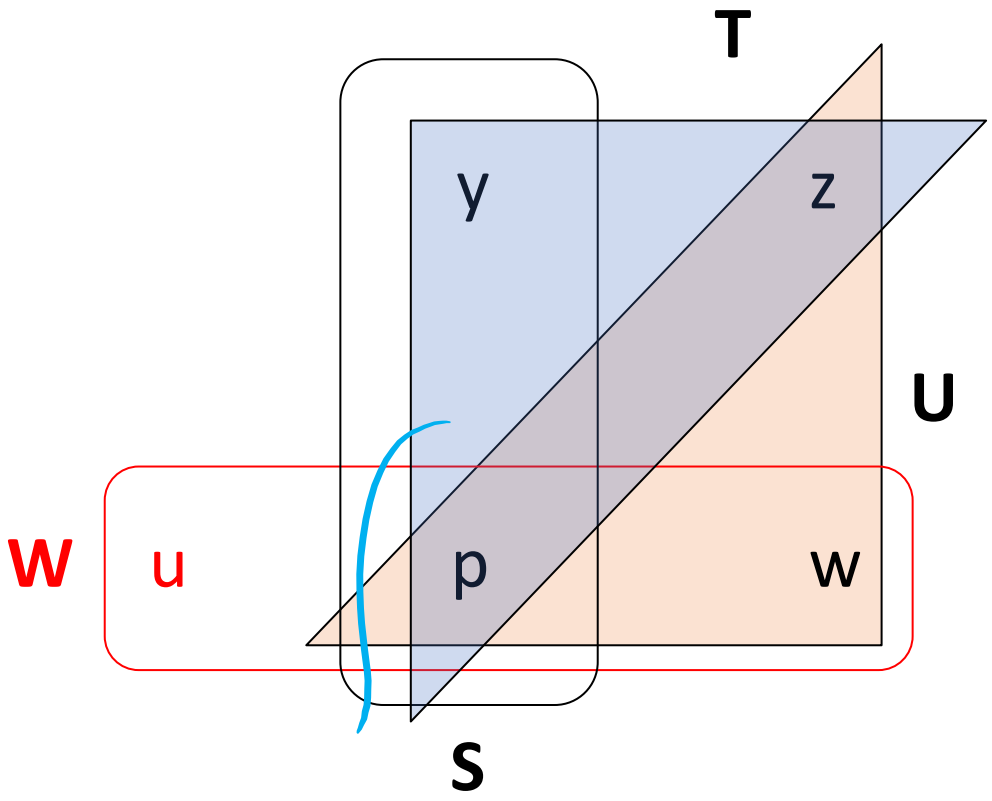


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

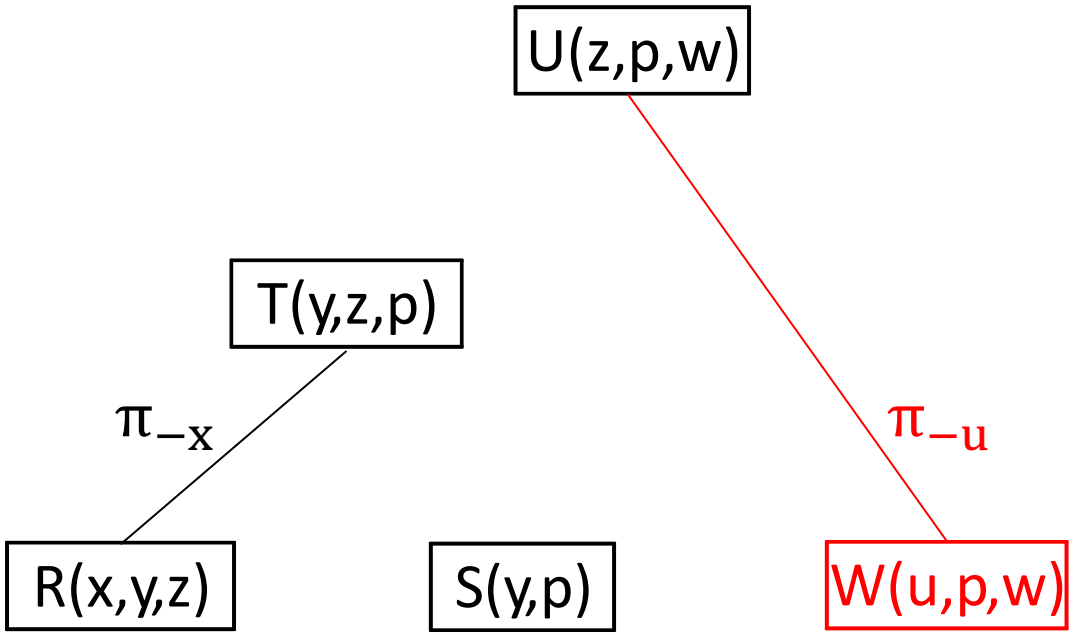


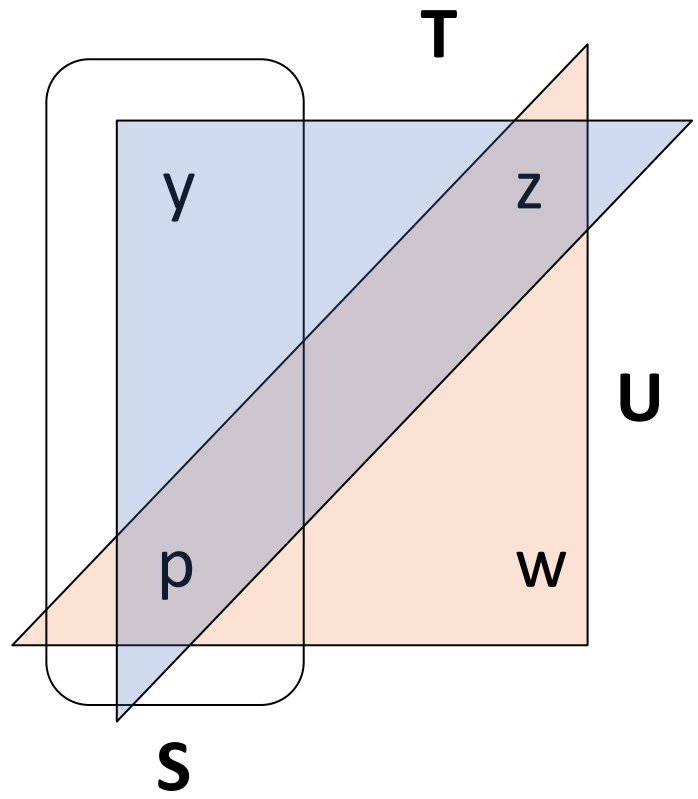


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

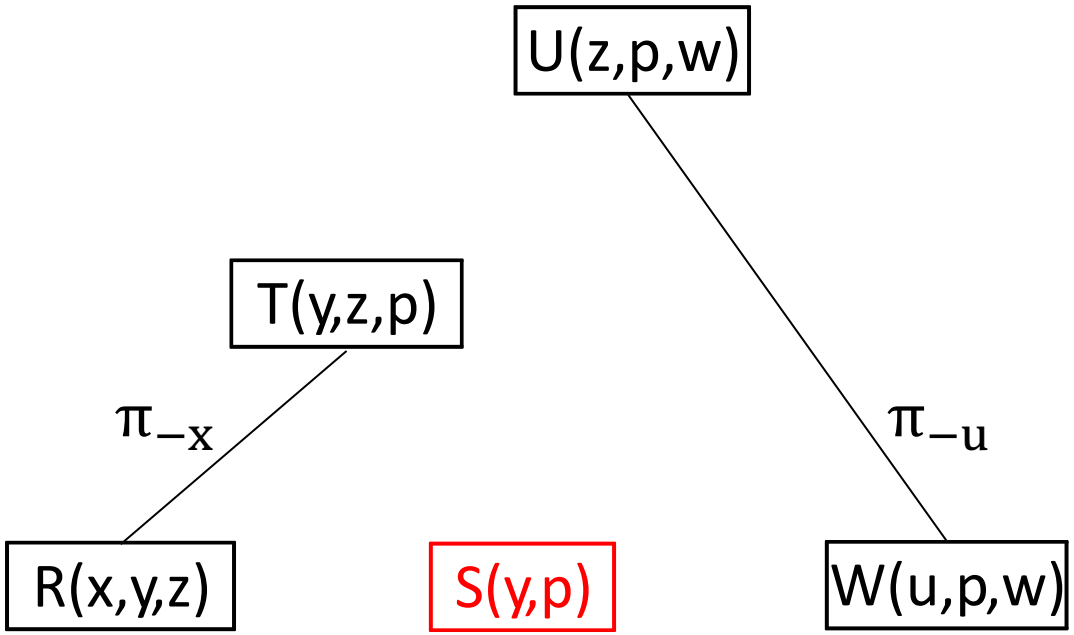


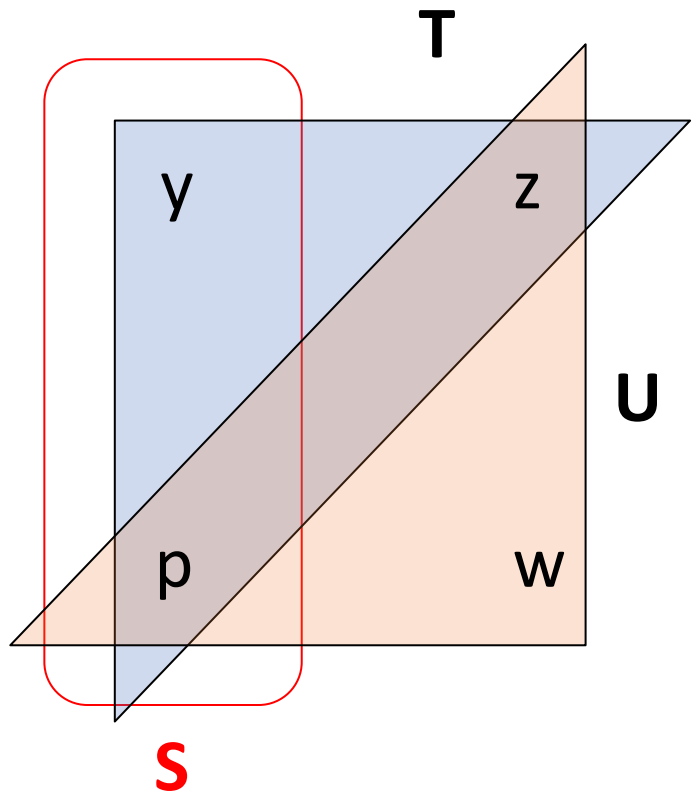


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

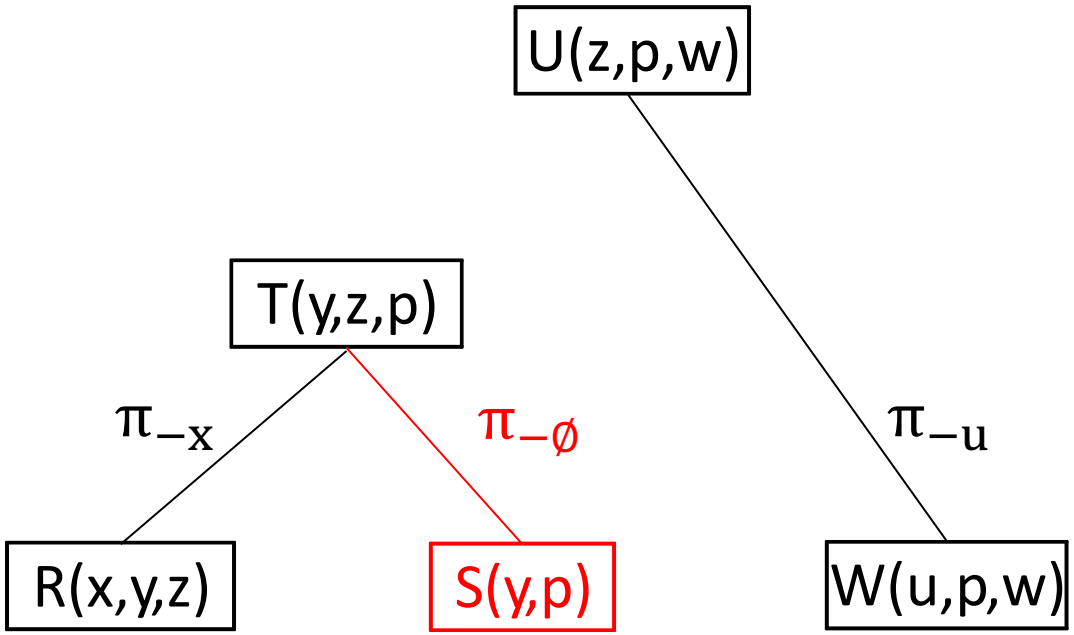


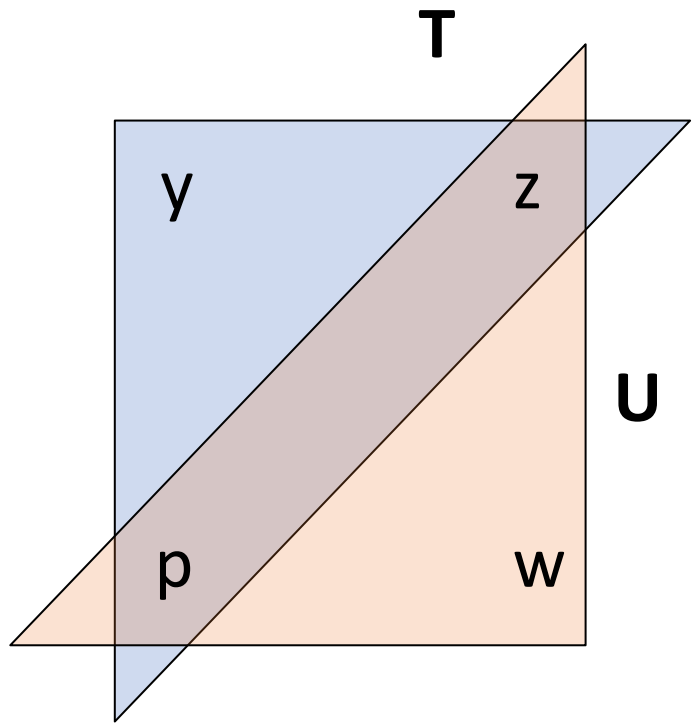


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

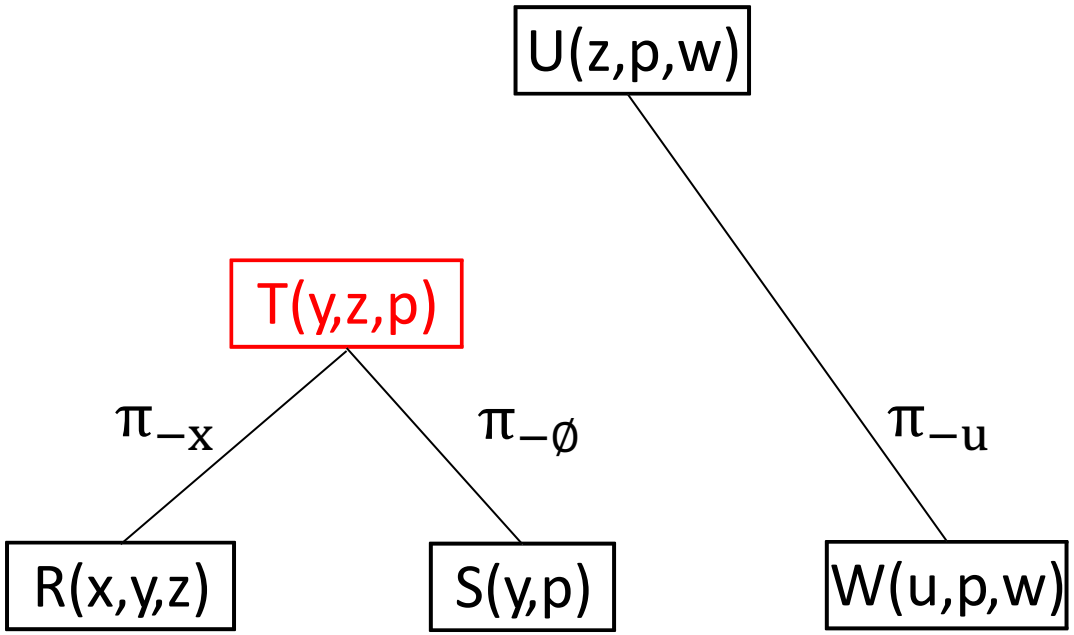


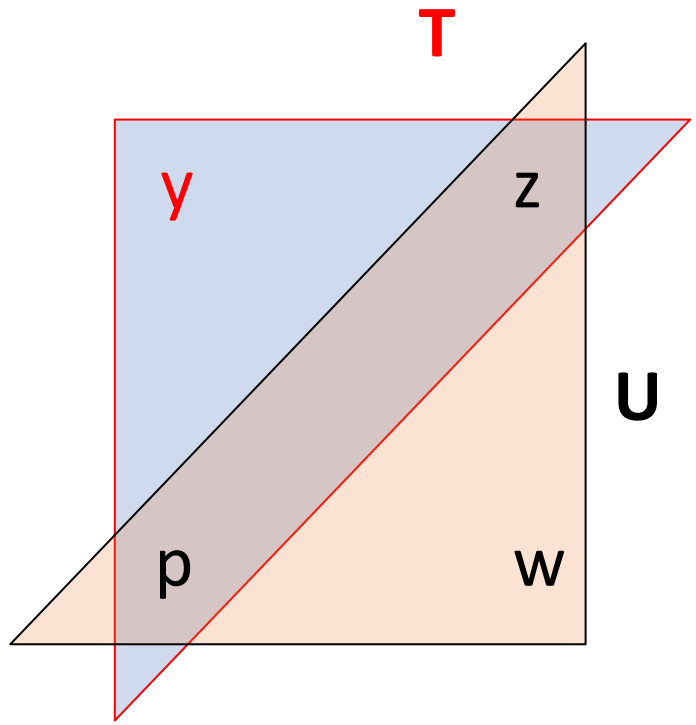


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

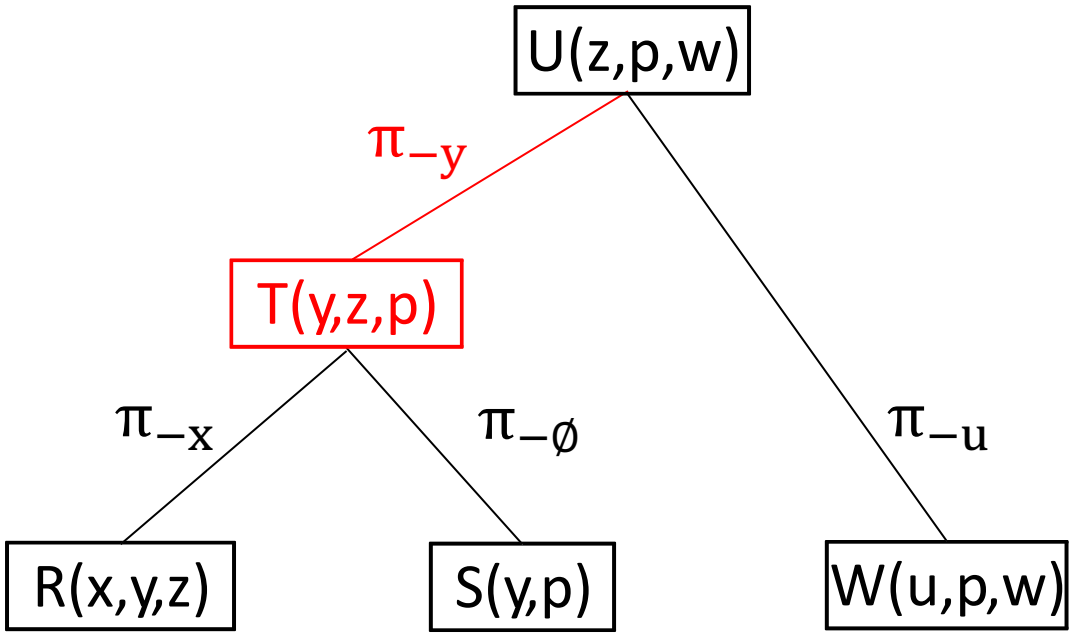


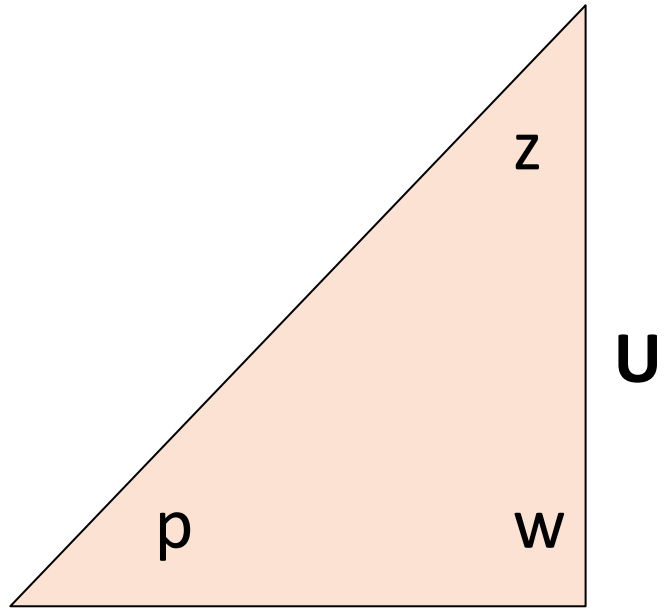


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

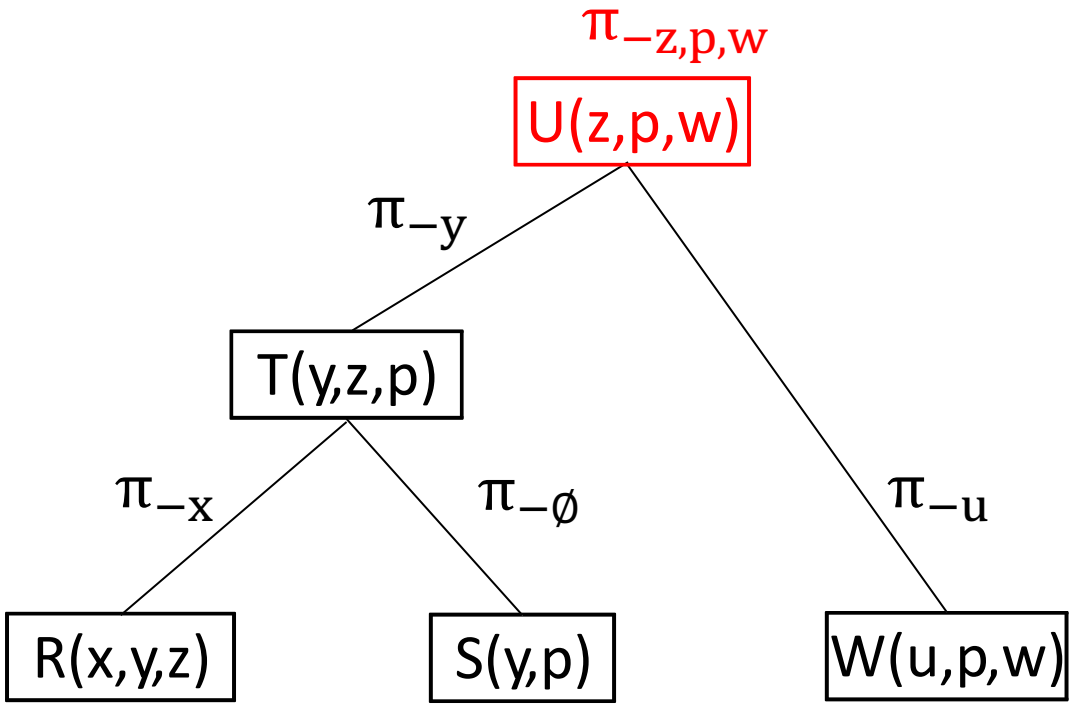




GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

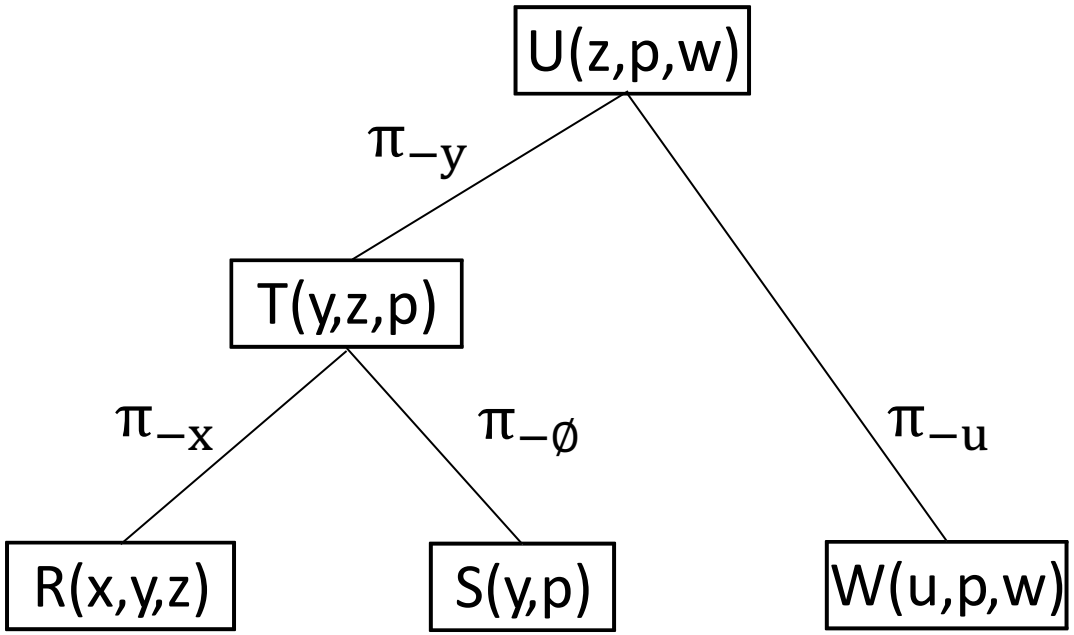
R(x,y,z), S(y,p), T(y,z,p), U(z,p,w), W(u,p,w)

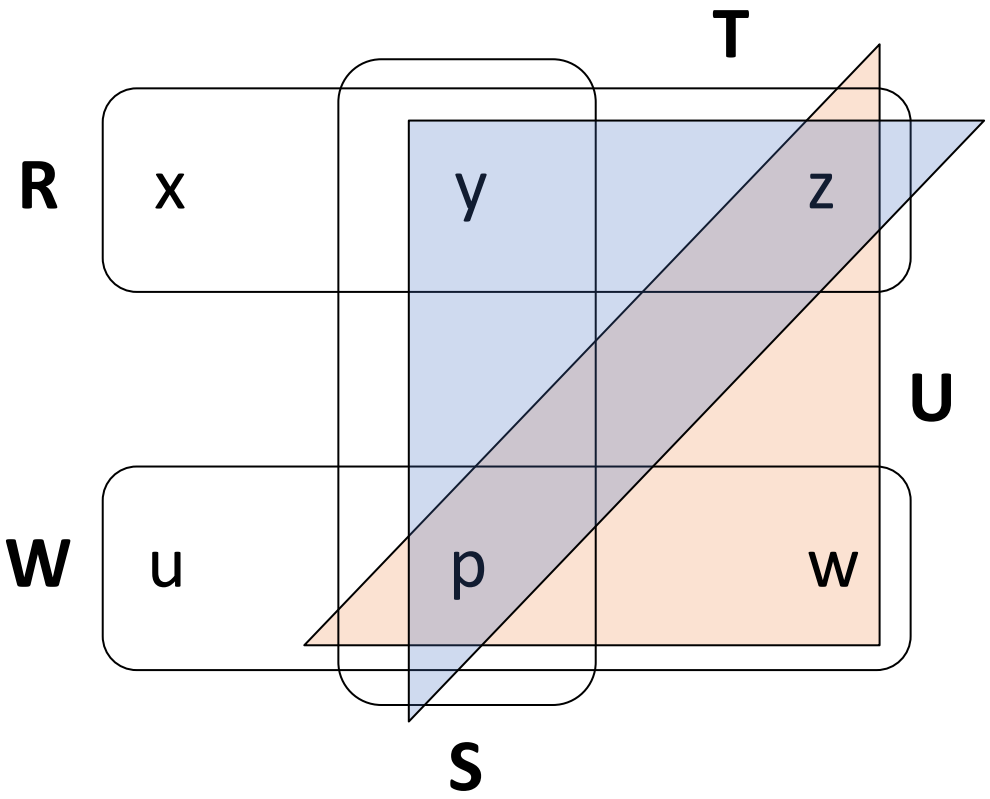


GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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





GYO reduction (Graham-Yu-Ozsoyoglu)

- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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

- What about the triangle query?

Join tree

?

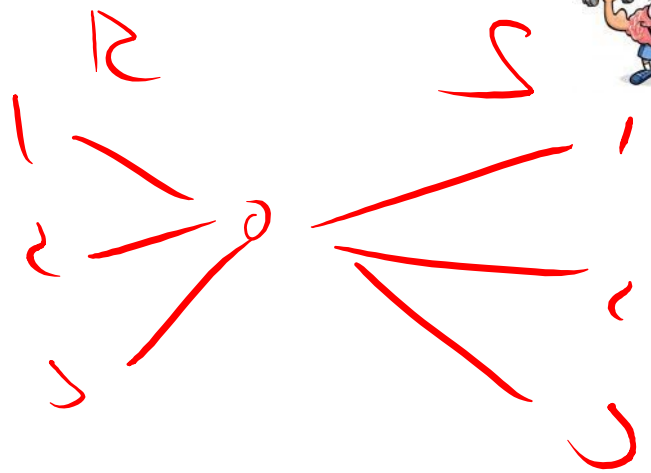
Query hypergraph

?

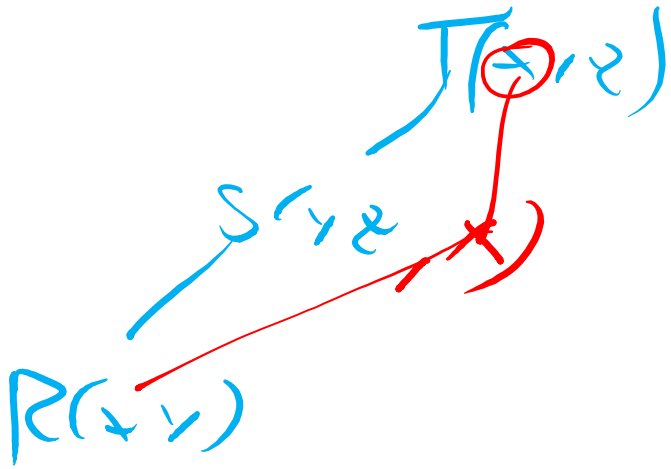
GYO reduction (Graham-Yu-Ozsoyoglu)

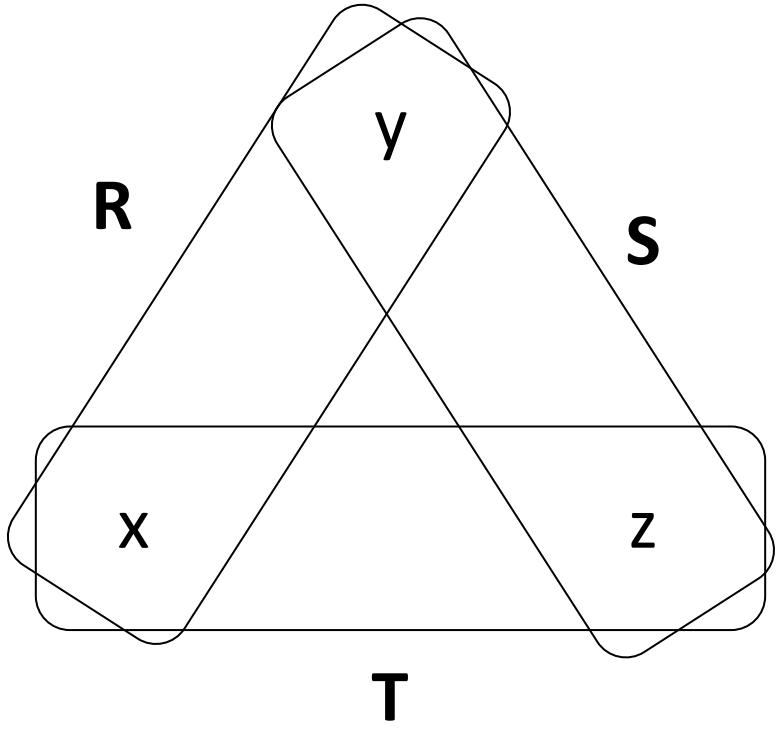
- GYO Ear removal
  - remove ears (= edges)
    - remove isolated nodes (variables)
    - remove consumed or empty edges (atoms)

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



- What about the triangle query?





Acyclic queries

- "Consistency" = no dangling tuples (recall semi-join)
  - locally consistent $\pi_{R_i}(R_i \ltimes R_j) = R_i$
  - globally consistent $\pi_{R_i}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n) = R_i$
- One implies the other
  - For all queries: global $\Rightarrow$ local
  - Acyclic queries: local $\Rightarrow$ global; but not for cyclic queries!

Example

Consider $AB = \{00, 11\}$, $BC = \{00, 11\}$, and $AC = \{01, 10\}$.

- Any two relations are join-consistent. E.g., $AB \bowtie AC = \{001, 110\}$, which projected onto AB is $\{00, 11\}$.
- But $AB \bowtie BC \bowtie AC = \emptyset$, so the relations are not globally consistent.

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

Repetition: Law of Semijoins

- Definition: the semi-join operation is:

$$R \ltimes_C S = \pi_{A_1,\ldots,A_n}(R \bowtie S)$$

  - Formally, $R \ltimes_C S$ 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: $R(A_1,\ldots,A_n)$, $S(B_1,\ldots,B_m)$, Output: $T(A_1,\ldots,A_n)$

- The law of semijoins is:

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

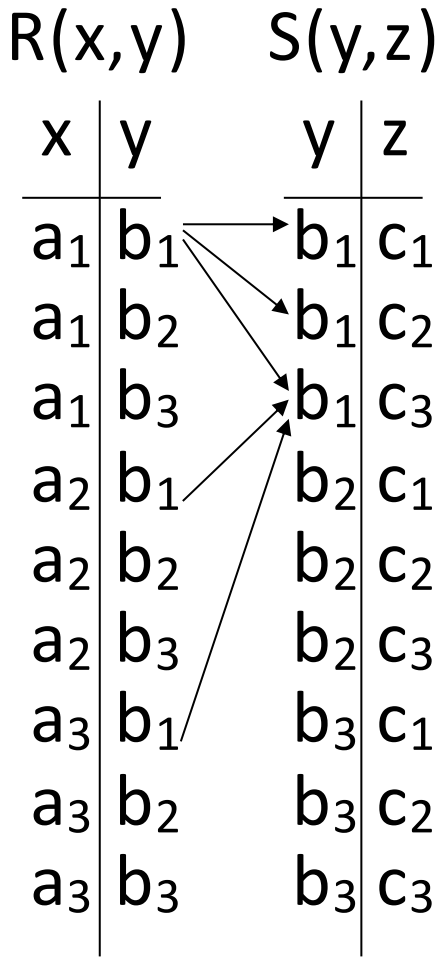
  - Thus, removing dangling tuples does not change the query result

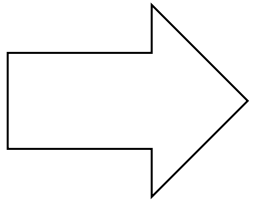
Yannakakis Algorithm

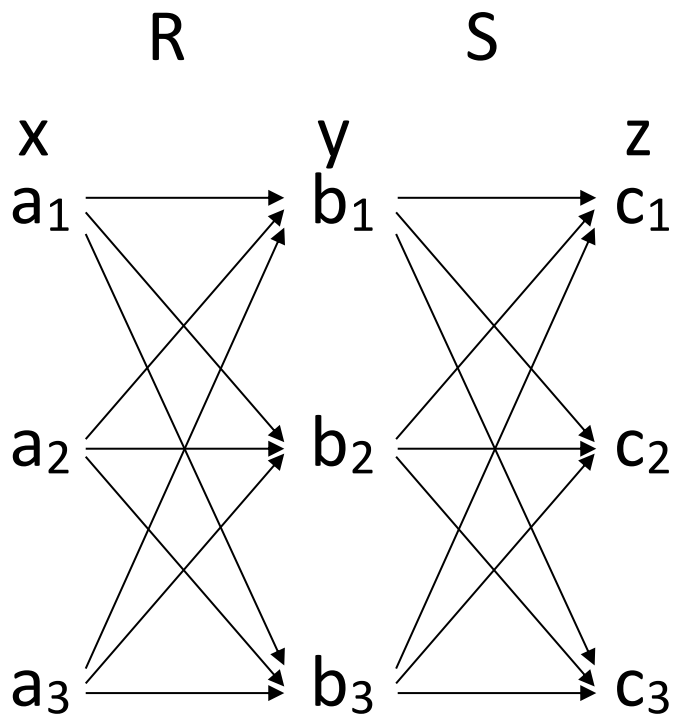
- Given: acyclic query Q
- Compute Q on any database in time O(|Input|+|Output|)
- We use the join tree!
- Step 1: semi-join reduction (two sweeps)
  - Pick any root node x in the tree decomposition of Q
  - Do a semi-join reduction sweep from the leaves to x
  - Do a semi-join reduction sweep from x to the leaves
- Step 2: use the join tree as query plan; compute the joins bottom up, with early projections
  - Notice that step 2 can be pushed into the top-down sweep

Application

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



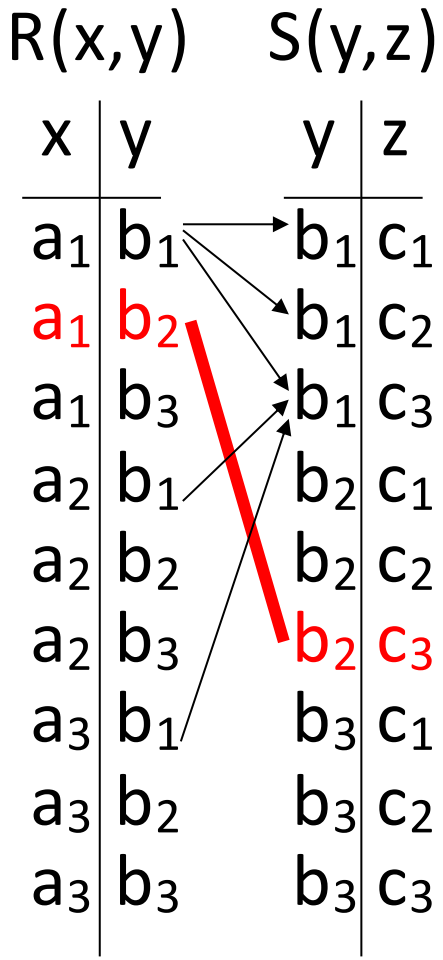


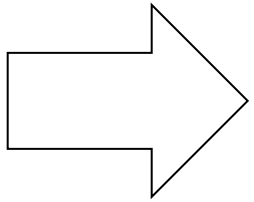


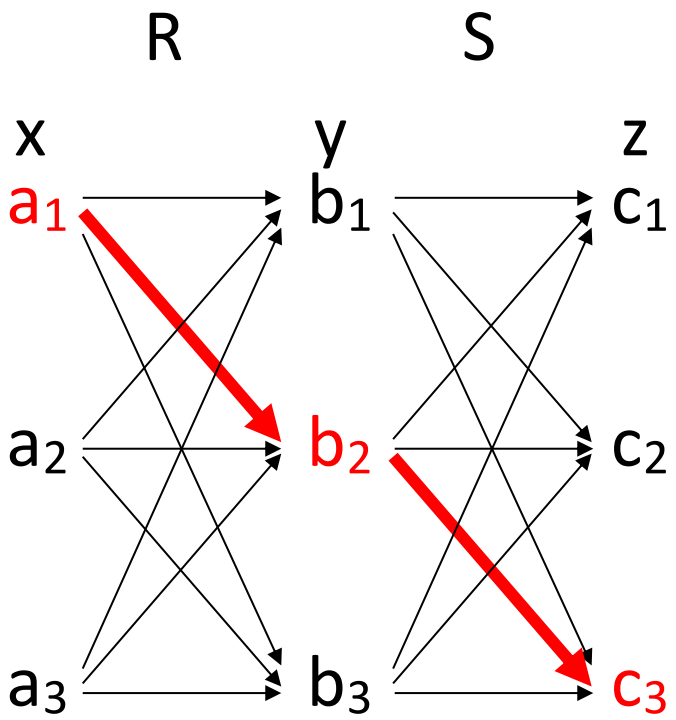
- Domain as nodes
- Tuples as edges
- Equi-joins between tuples as shared domain

Application

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



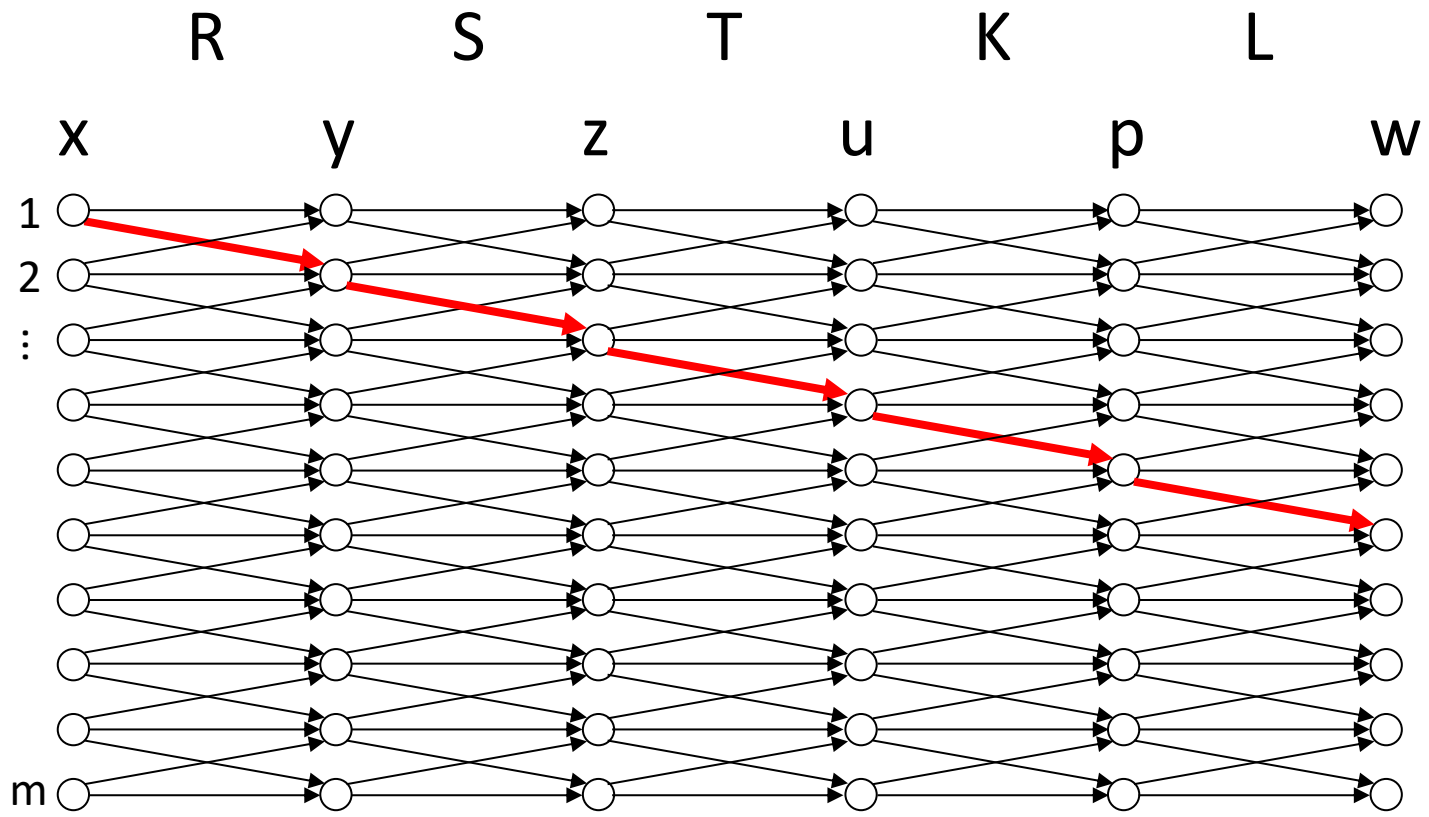




- Domain as nodes
- Tuples as edges
- Equi-joins between tuples as shared domain

Application

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

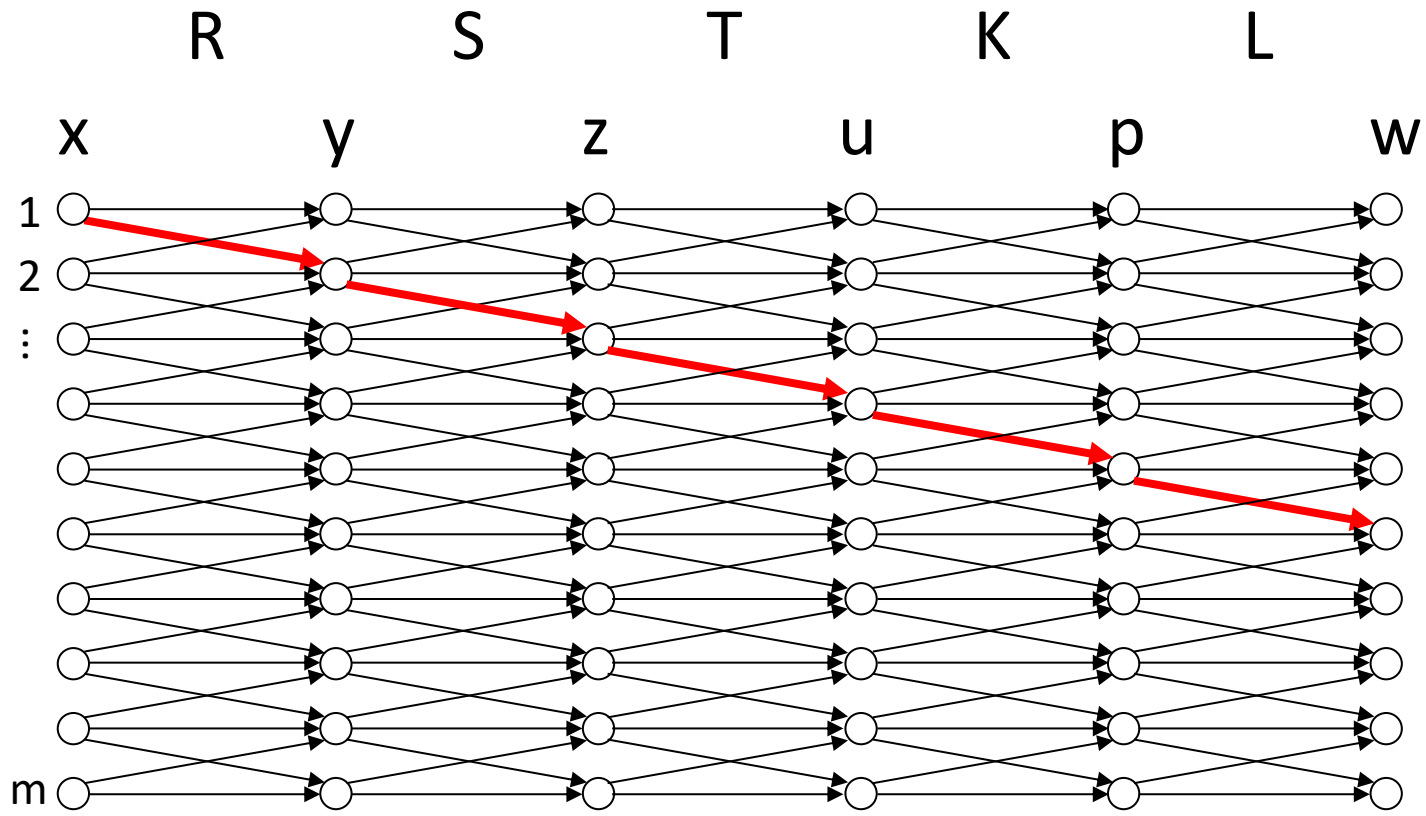


- Intermediate relations of size up to m^5 (notice m is here domain size!)
- But final answer can be as small as 0 (or 1...)

Application

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

Semi-join reduction allows pre-processing in O(n)



/* Forwards: */
S := S ⋉ R
T := T ⋉ S
K := K ⋉ T
L := L ⋉ K

/* then backwards: */
K := K ⋉ L
T := T ⋉ K
S := S ⋉ T
R := R ⋉ S

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

Outline: Acyclic conjunctive queries

- Acyclic conjunctive queries
  - The semijoin operator
  - Join trees & Yannakakis algorithm
  - Query hypergraphs & GYO reduction
  - A detailed Yannakakis example
  - Full semijoin reductions
- Cyclic conjunctive queries

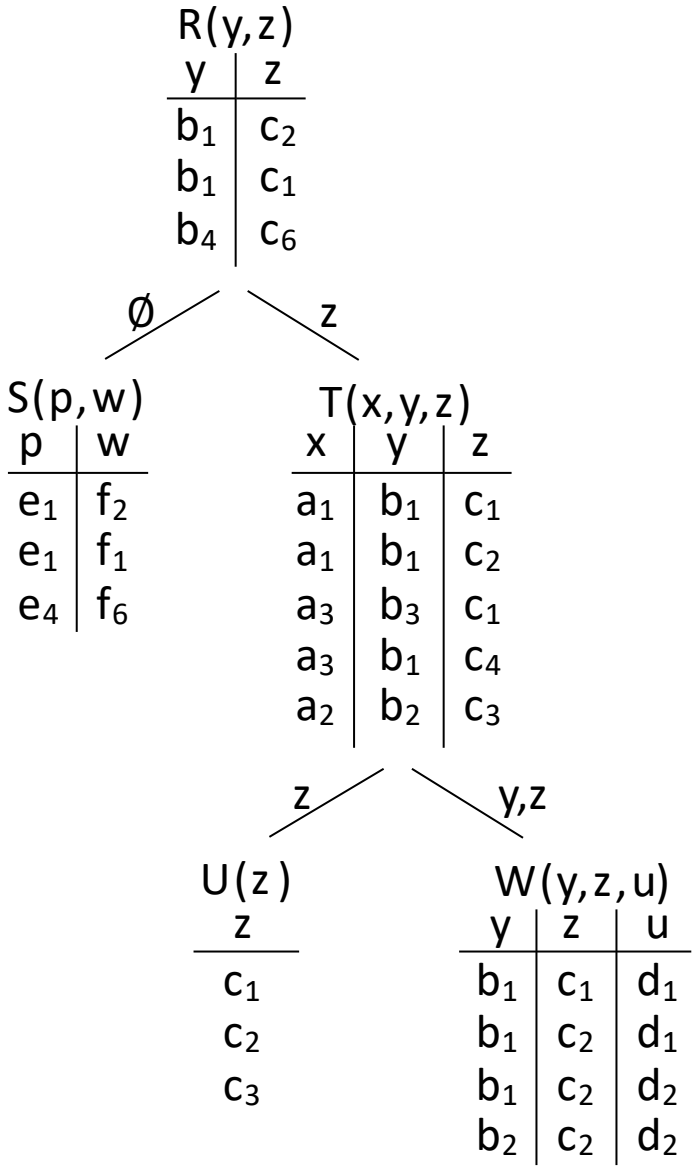
Deciding ACQs Efficiently [Yannakakis'81]

- Dynamic Programming Algorithm over the join tree T = (V , E)
- Let T = (V, E) be a join tree of a query Q. Assign to each $R_j(\mathbf{z}_j) \in V$ the corresponding relation R_j^D of D
- Given database instance D, decide Q(D) = ∅ as follows:
  - In a bottom up traversal of T: compute semijoins of R_j^D
  - If the resulting relation at root node is
    - empty, then Q(D) = ∅,
    - nonempty, then Q(D) ≠ ∅.
- Theorem:
  - For ACQs Q: Deciding Q(D) = ∅ is feasible in polynomial time.
  - Computing Q(D) can be done in output polynomial time.

Yannakakis Algorithm – Example

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

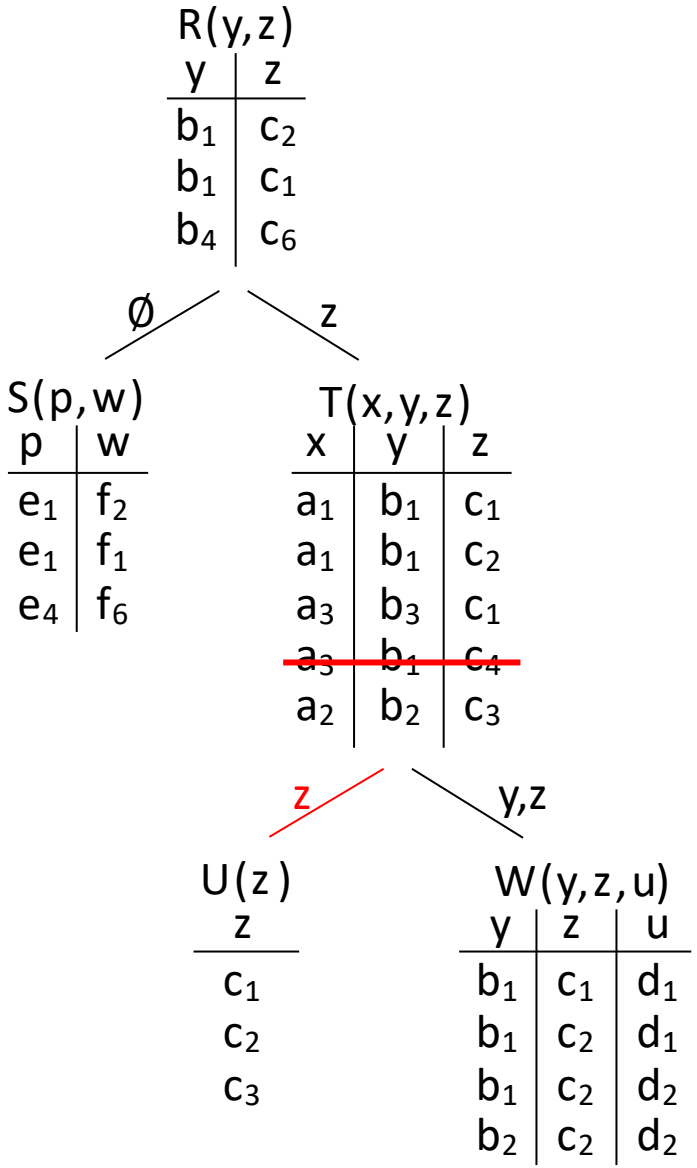
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

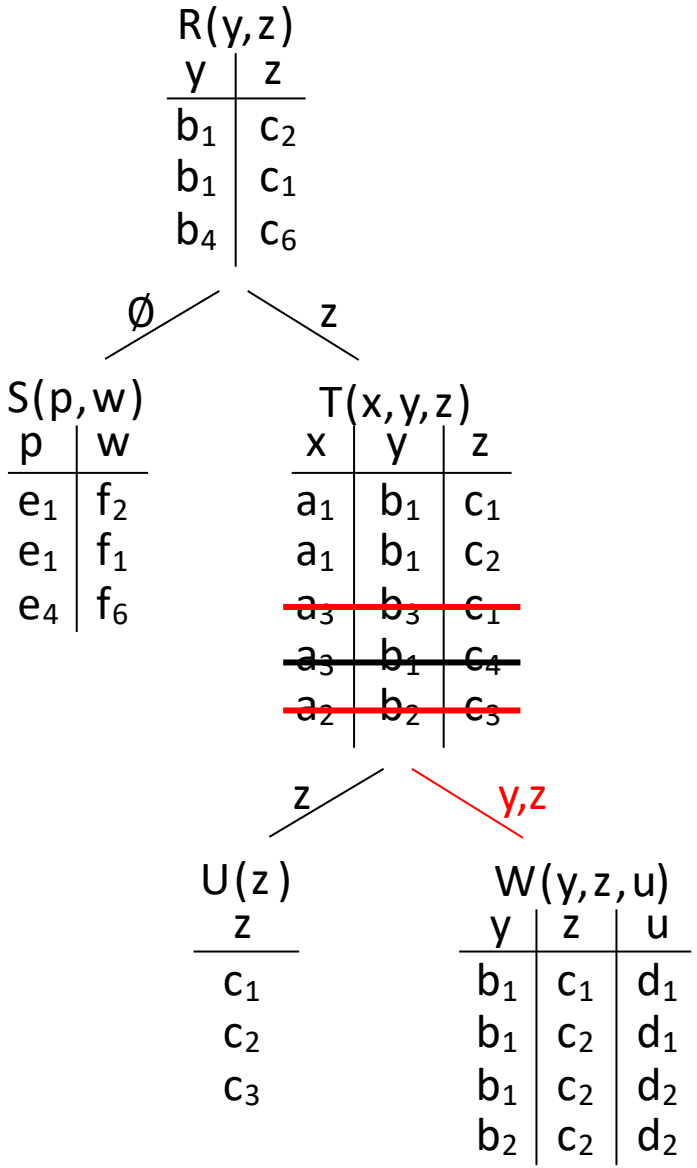
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

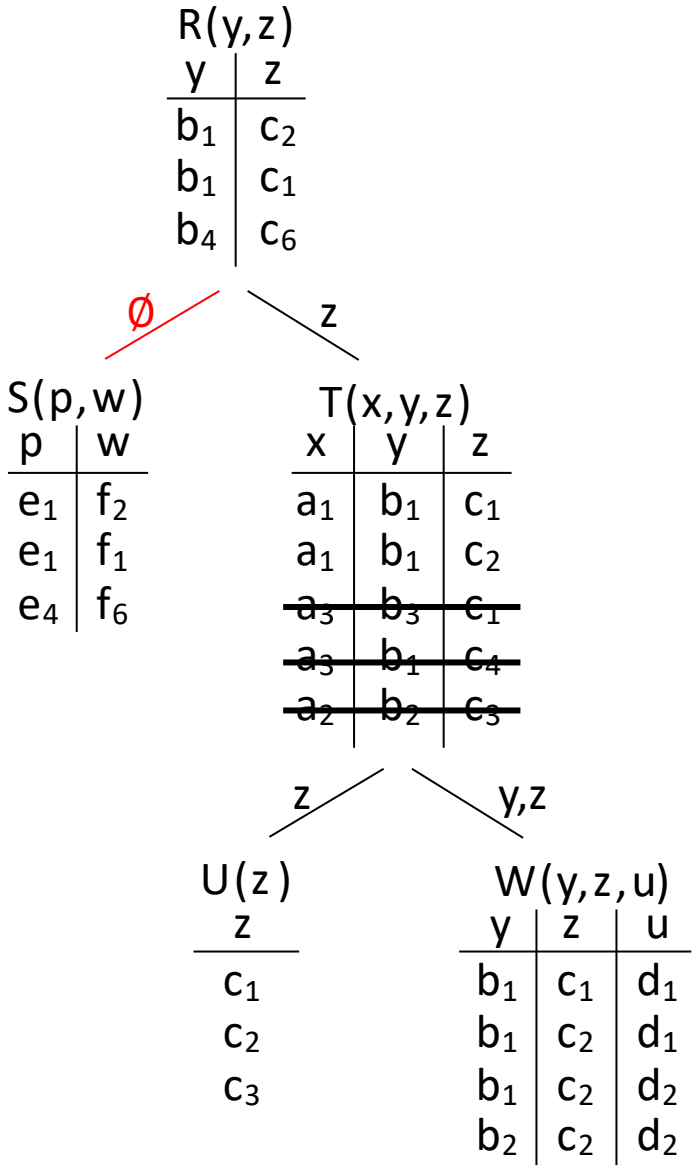
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

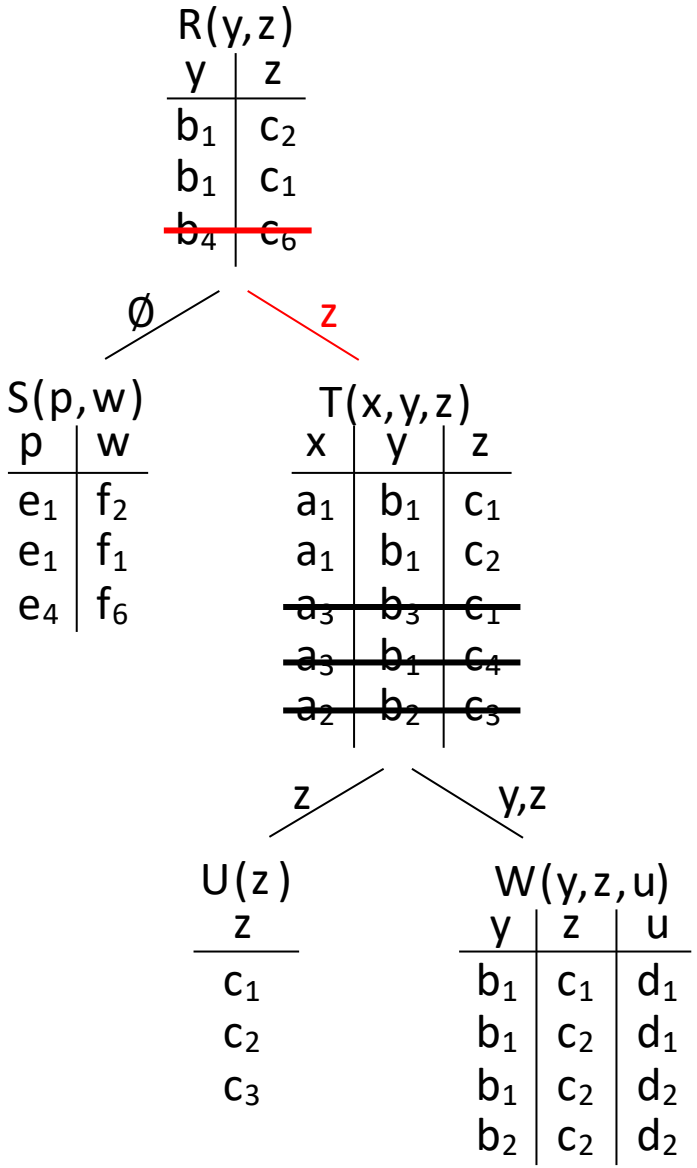
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

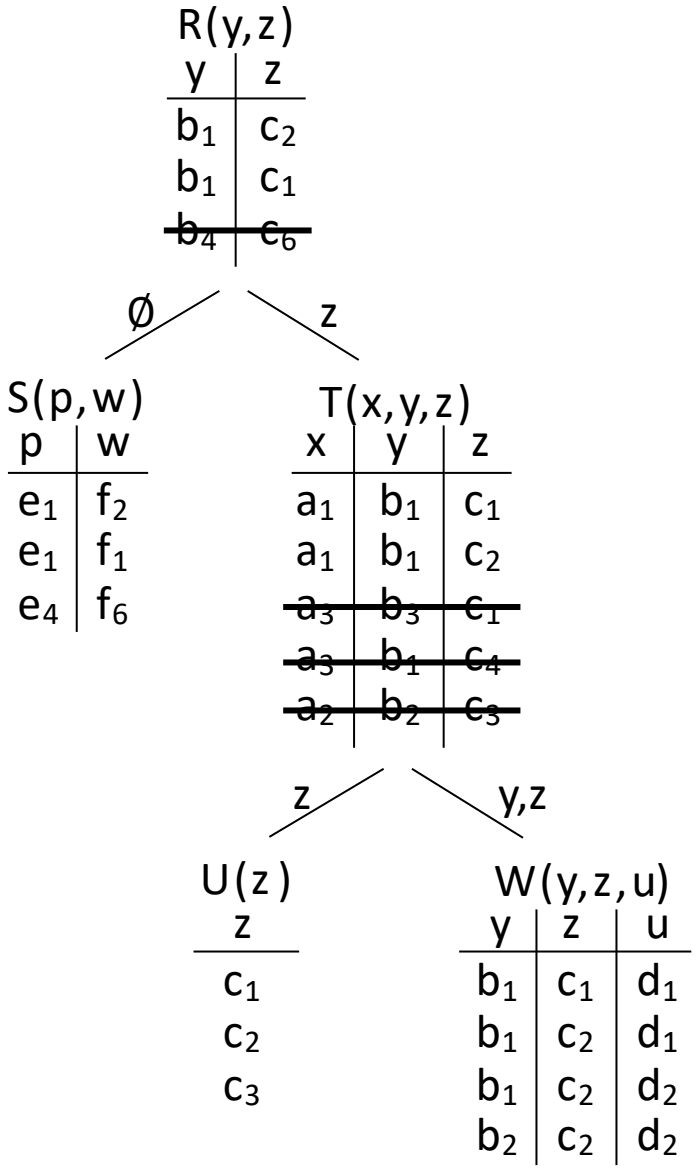
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

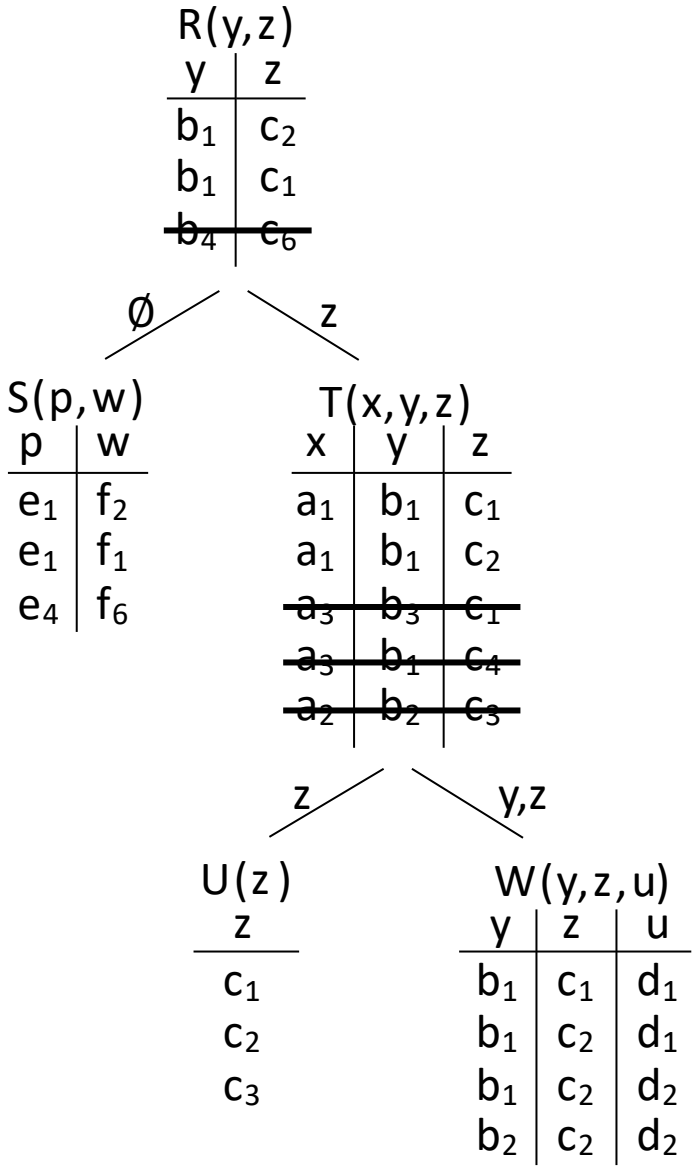
1. Bottom-up traversal (semi-joins)



Yannakakis Algorithm – Example

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

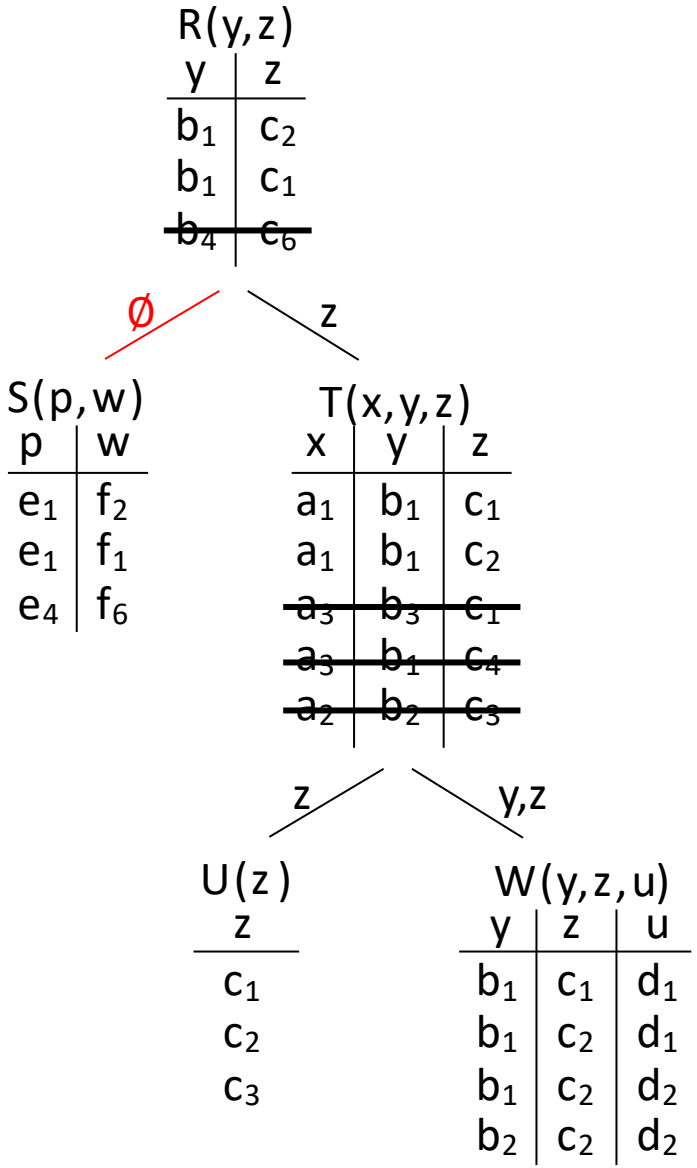
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

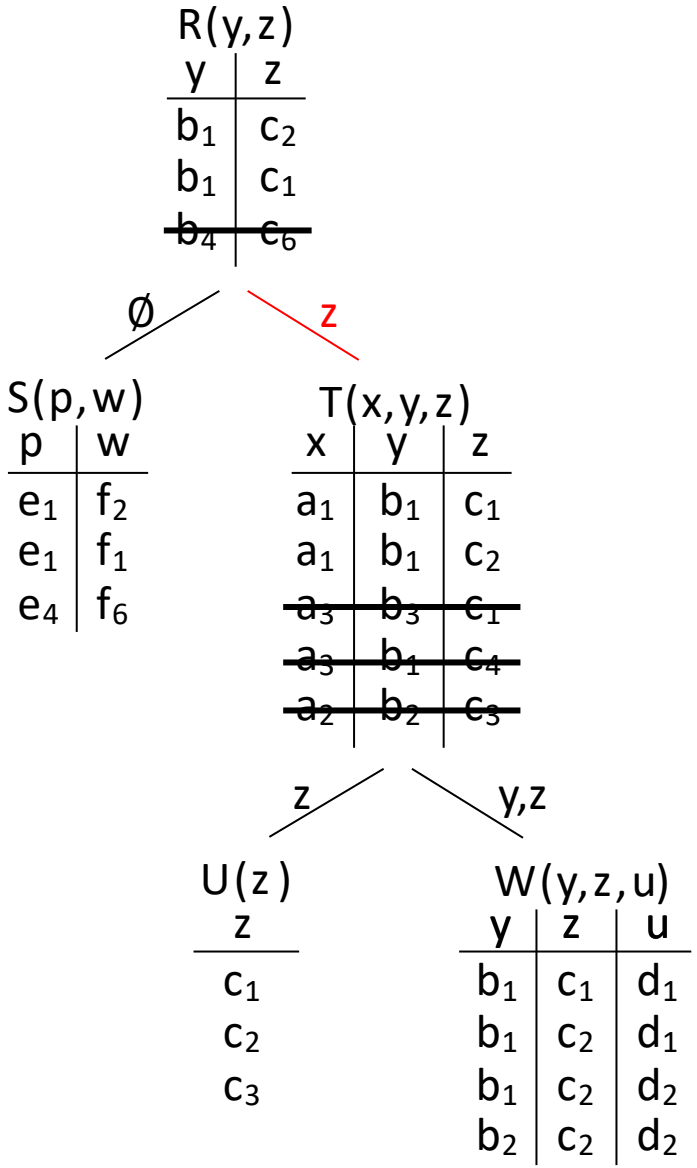
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

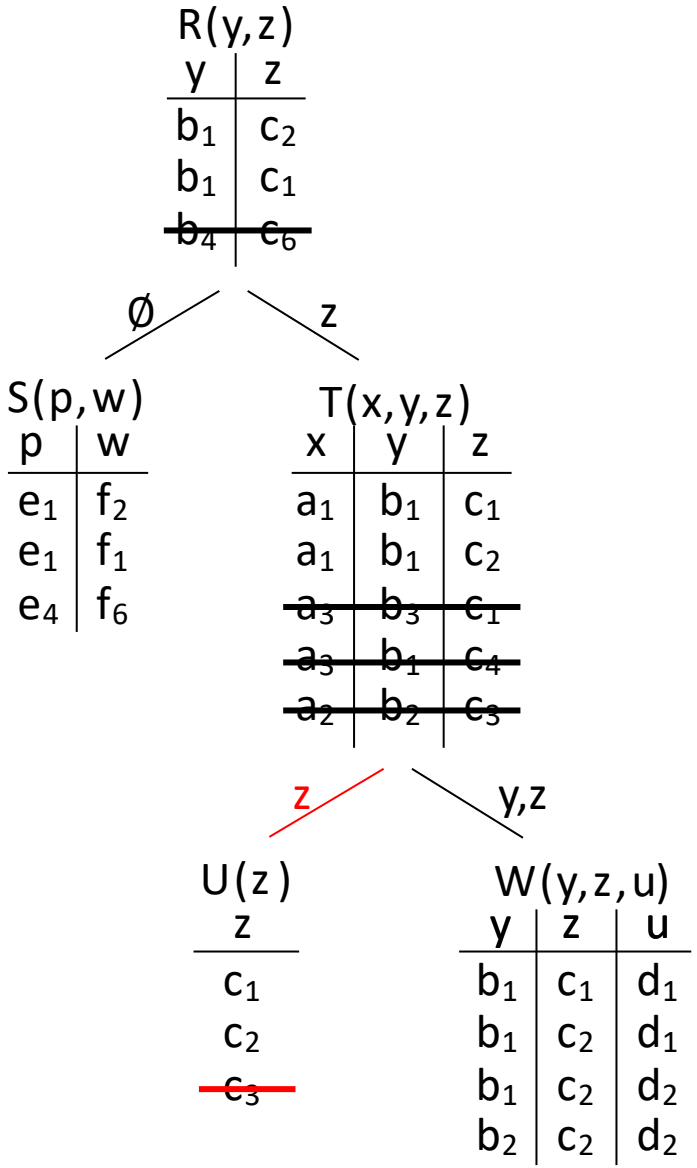
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

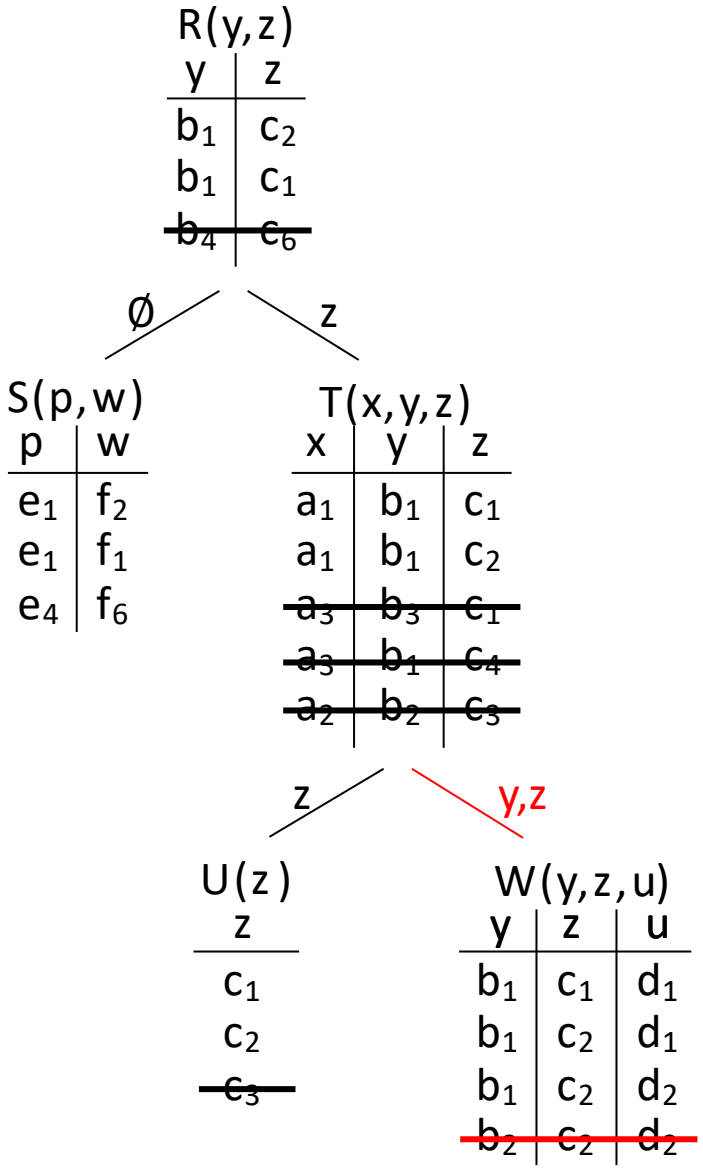
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

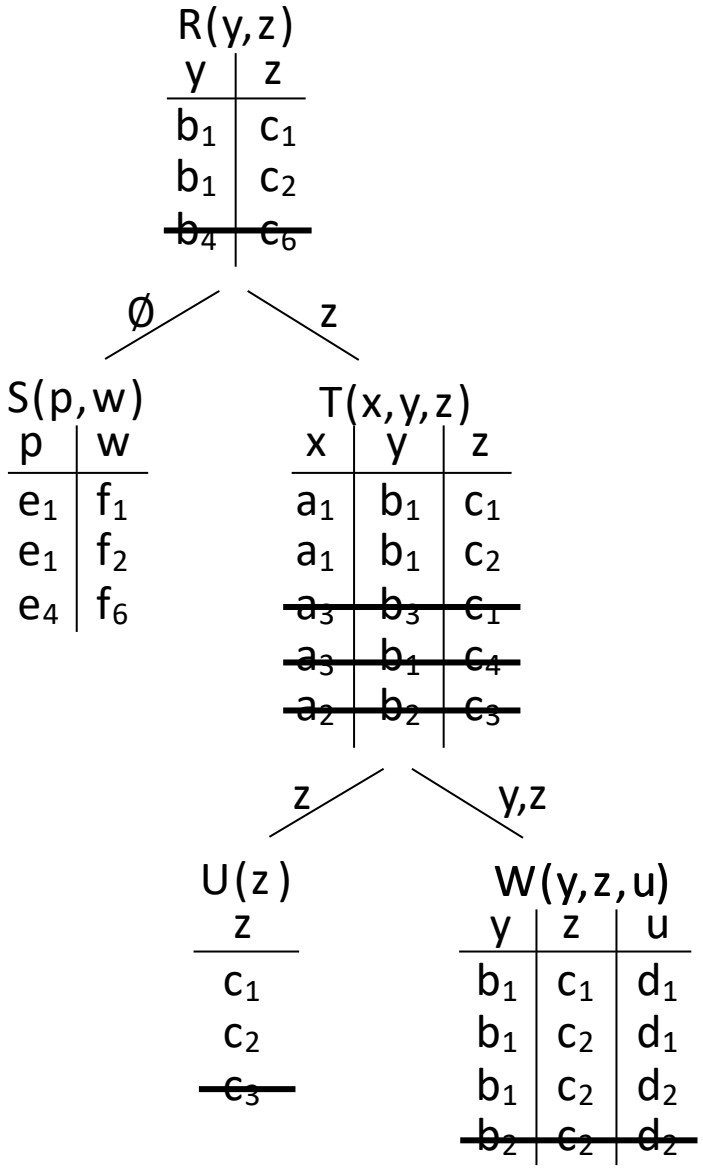
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

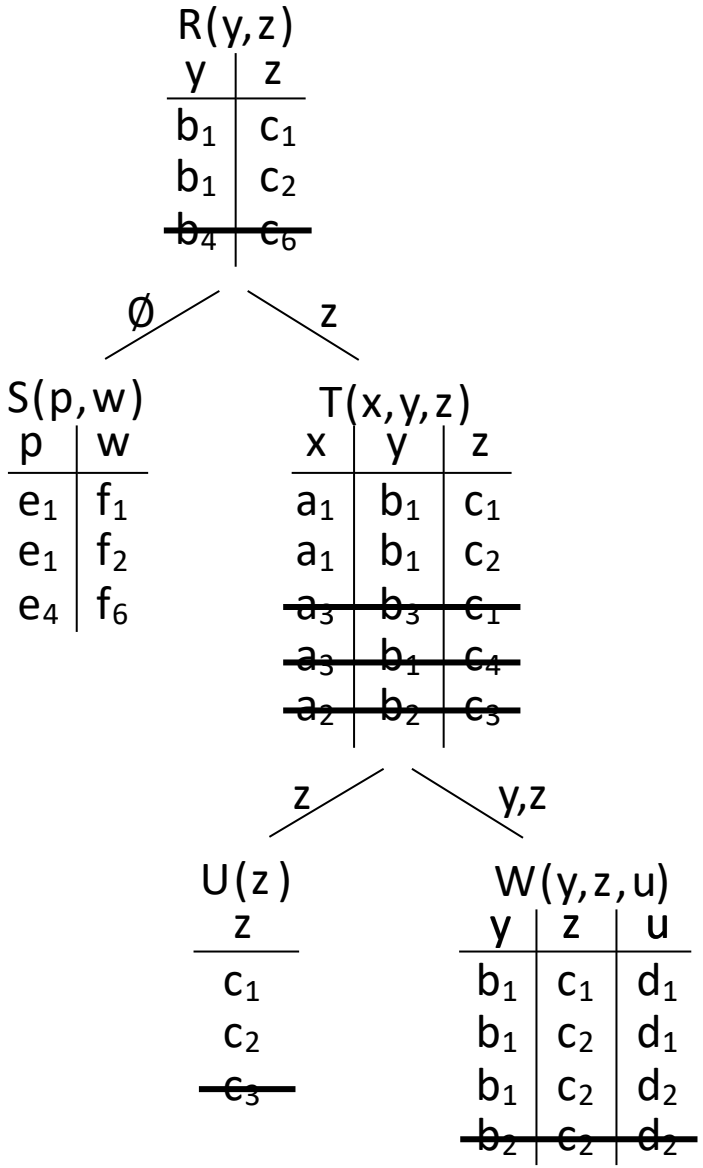
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)



Yannakakis Algorithm – Example

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

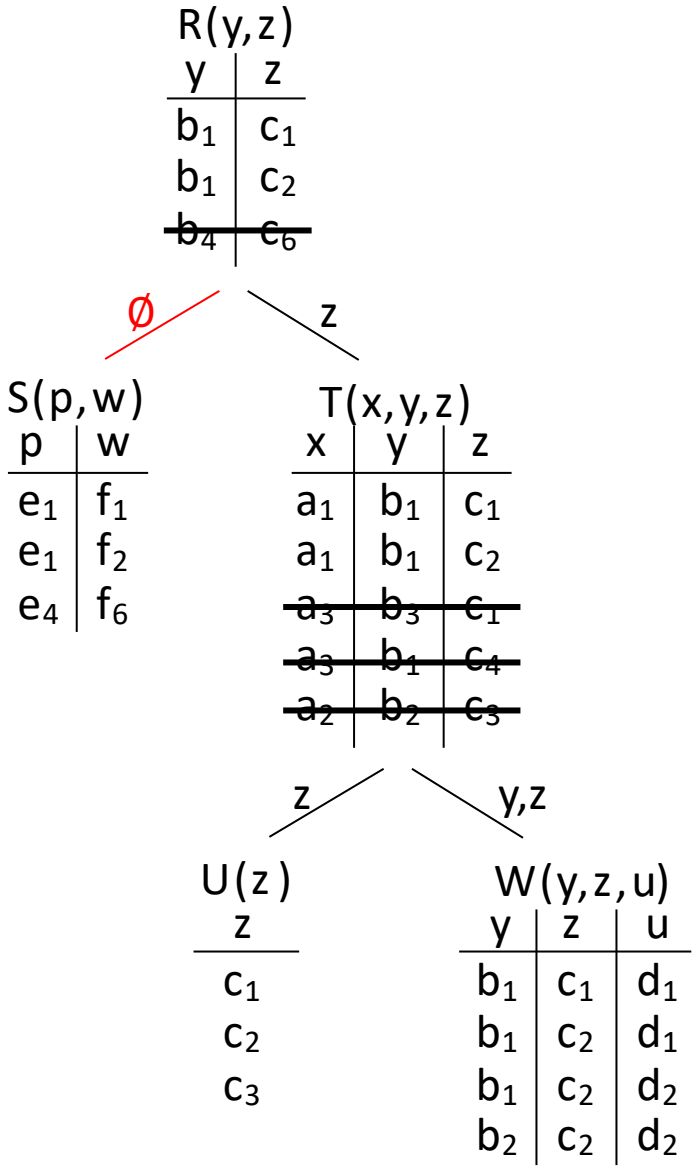
1. Bottom-up traversal (semi-joins)
2. Top-down traversal (semi-joins)
3. Compute results in 2nd bottom-up traversal
   - Or, better, just push computation of results into top-down traversal ☺



Yannakakis Algorithm – Example

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

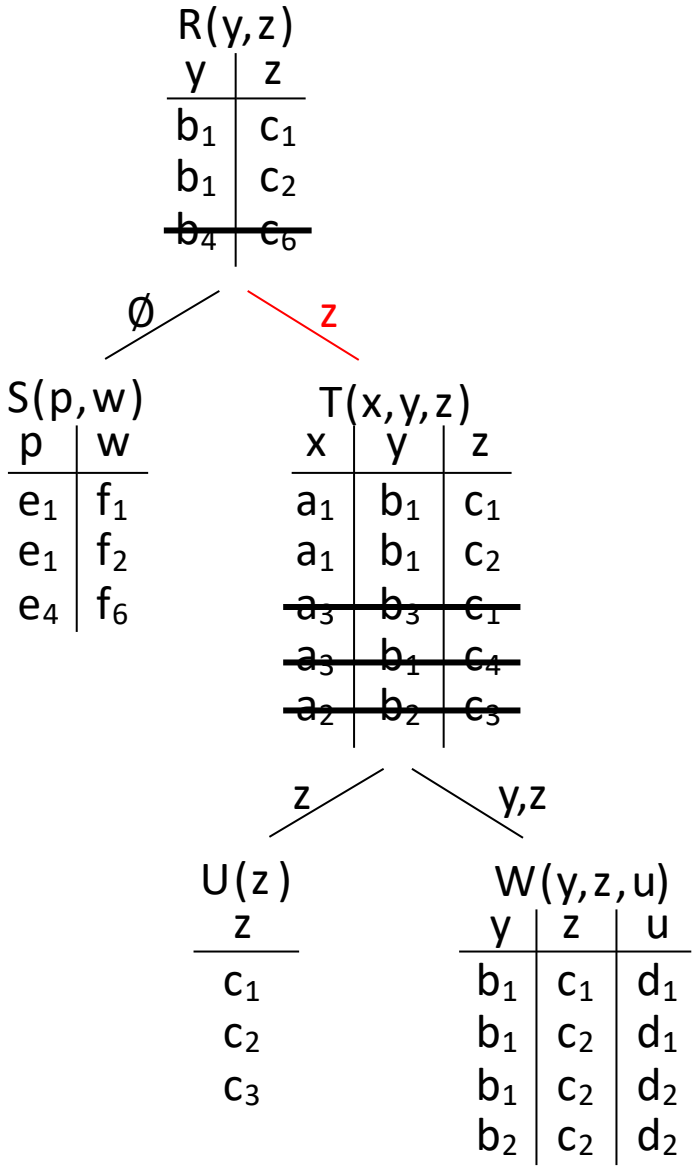
y	z	p	w
b_1	c_1	e_1	f_1
b_1	c_1	e_1	f_2
b_1	c_1	e_4	f_6
b_1	c_2	e_1	f_1
b_1	c_2	e_1	f_2
b_1	c_2	e_4	f_6



Yannakakis Algorithm – Example

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

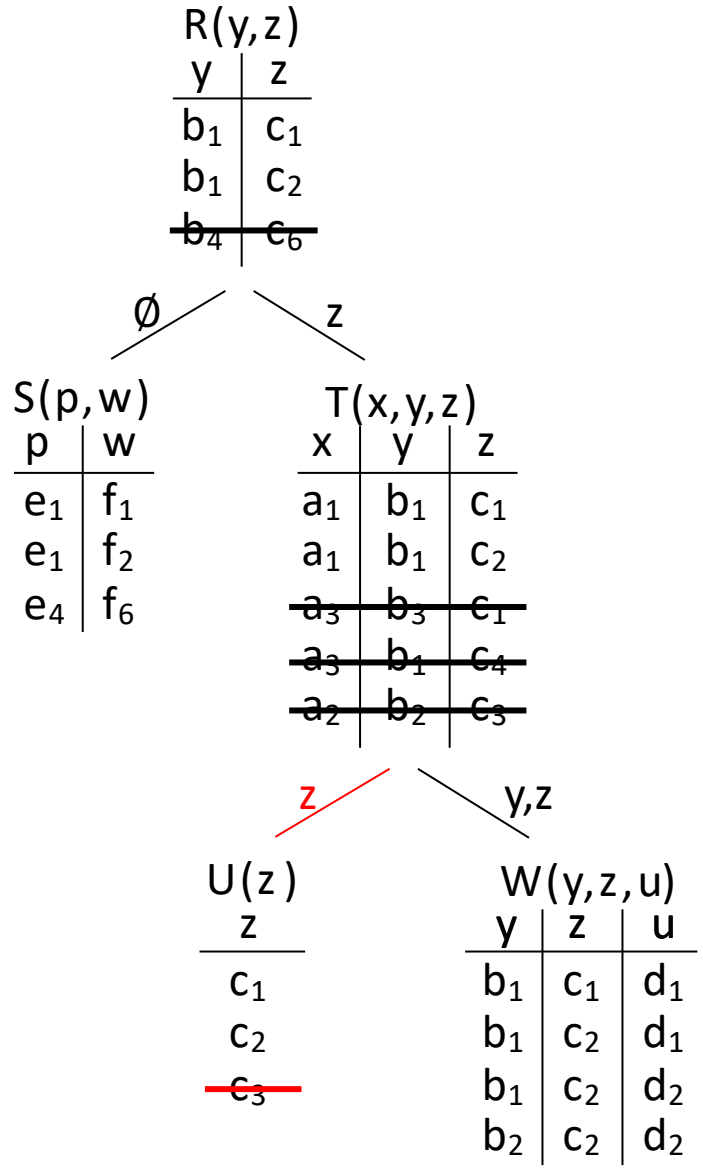
y	z	p	w	x
b_1	c_1	e_1	f_1	a_1
b_1	c_1	e_1	f_2	a_1
b_1	c_1	e_4	f_6	a_1
b_1	c_2	e_1	f_1	a_1
b_1	c_2	e_1	f_2	a_1
b_1	c_2	e_4	f_6	a_1



Yannakakis Algorithm – Example

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

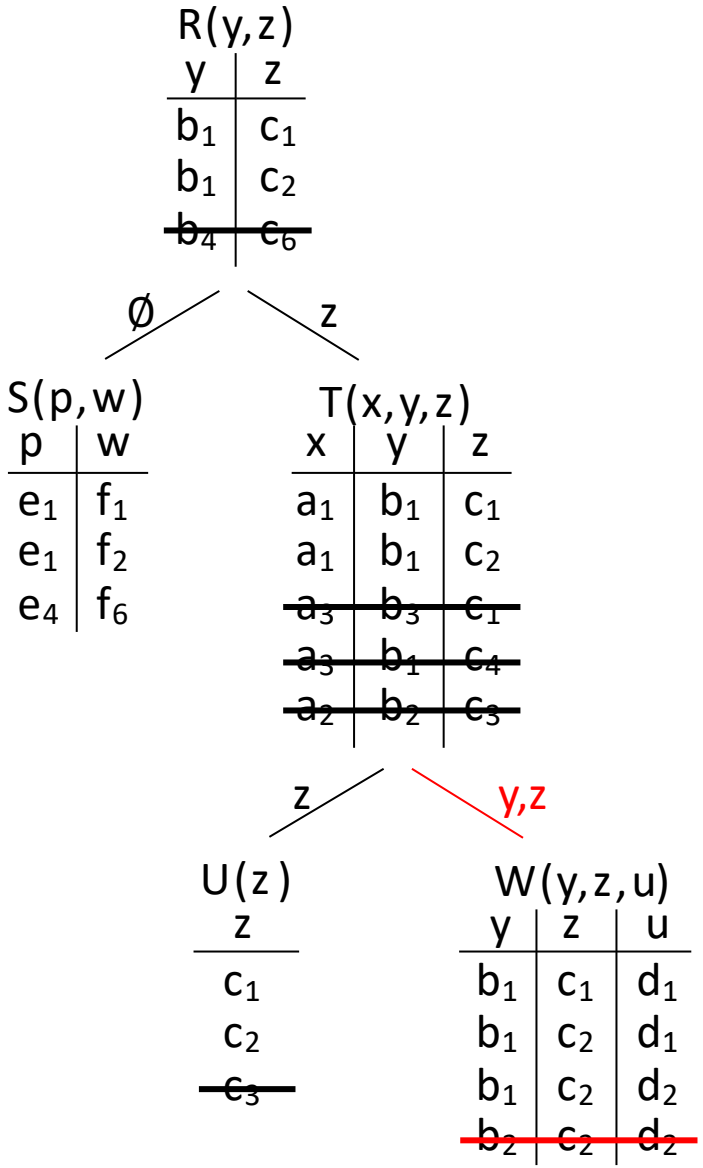
y	z	p	w	x
b_1	c_1	e_1	f_1	a_1
b_1	c_1	e_1	f_2	a_1
b_1	c_1	e_4	f_6	a_1
b_1	c_2	e_1	f_1	a_1
b_1	c_2	e_1	f_2	a_1
b_1	c_2	e_4	f_6	a_1



Yannakakis Algorithm – Example

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

y	z	p	w	x	u
b_1	c_1	e_1	f_1	a_1	d_1
b_1	c_1	e_1	f_2	a_1	d_1
b_1	c_1	e_4	f_6	a_1	d_1
b_1	c_2	e_1	f_1	a_1	d_1
b_1	c_2	e_1	f_2	a_1	d_1
b_1	c_2	e_4	f_6	a_1	d_1
b_1	c_2	e_1	f_1	a_1	d_2
b_1	c_2	e_1	f_2	a_1	d_2
b_1	c_2	e_4	f_6	a_1	d_2



Recap

- The notions of query equivalence and containment
- The Homomorphism Theorem
- The complexity of query equivalence and containment
- Minimization of conjunctive queries
- Acyclic conjunctive queries
- The Yannakakis algorithm

Yet to be seen:

- What do we do with cycles?
- What about ranked retrieval?

Pointers to related work

- Abiteboul, Hull, Vianu. *Foundations of Databases*. Addison Wesley, 1995. http://webdam.inria.fr/Alice/, Ch 6.4: Acyclic joins, semi-join reduction, Yannakakis, GYO algorithm.
- The original GYO algorithm was developed concurrently in 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.
- 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*. 1983 https://doi.org/10.1145/2402.322389