CSE544
Data Management

Lectures 17-19: Datalog

Announcements

- Project milestones due on Friday
- HW4 is posted and due on Dec. 5
  - Part 1 Theory. You can start now
  - Part 2 Datalog. Discussed today+Monday

Motivation

- Data processing today require iteration.
  - Common solution: external driver
  - The adventurous: SQL WITH RECURSIVE

- Datalog is a language designed for recursive queries

Datalog

- Designed in the 80's: simple, concise
- Used for research on recursive queries
- Only research prototypes exists…
- …in HW4 we use Souffle

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

← Schema

Datalog: Facts and Rules

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, ‘A Night in Armour’, 1910).
Movie(29000, ‘Arizona’, 1940).
Movie(29445, ‘Ave Maria’, 1940).
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

Find titles of movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Find first and last names of Actors who acted in Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x),
             Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940)
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

```
Q1(y) :- Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :- Actor(z,f,l), Casts(z,x),
            Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940)
```

Find Actors who acted in a Movie in 1940 and in one in 1910

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

```
Q1(y) :- Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :- Actor(z,f,l), Casts(z,x),
            Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

```
Q1(y) :- Movie(x,y,z), z='1940'.
```

```
Q2(f, l) :- Actor(z,f,l), Casts(z,x),
            Movie(x,y,'1940').
```

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Semantics of One Rule

SQL and single Datalog rule have similar semantics:

- SQL: nested loops of *tuples*
- Datalog rule: nested loops of *variables*:
  - Map variables to values in the database
  - Check that all atoms occur in the input

Semantics of One Rule

Actor

id	fname	lname
344759	Douglas	Fowley

Casts

pid	mid
344759	29851
355713	29000

Movie

id	title	year
7909	A Night in Armour	1910
29000	Arizona	1940
29445	Ave Maria	1940

Q1(y) :- Movie(x,y,z), z='1940'.

Semantics of One Rule

Actor

id	fname	lname
344759	Douglas	Fowley

Casts

pid	mid
344759	29851
355713	29000

Movie

id	title	year
7909	A Night in Armour	1910
29000	Arizona	1940
29445	Ave Maria	1940

Q1(y) :- Movie(x,y,z), z='1940'.

x	y	z
29000	Arizona	1940
29445	Ave Maria	1940

Semantics of One Rule

Actor

id	fname	lname
344759	Douglas	Fowley

Casts

pid	mid
344759	29851
355713	29000

Movie

id	title	year
7909	A Night in Armour	1910
29000	Arizona	1940
29445	Ave Maria	1940

Q1(y) :- Movie(x,y,z), z='1940'.

x	y	z
29000	Arizona	1940
29445	Ave Maria	1940

Answer

y
Arizona
Ave Maria

Semantics of One Rule

Actor

id	fname	lname
344759	Douglas	Fowley

Casts

pid	mid
344759	29851
355713	29000

Movie

id	title	year
7909	A Night in Armour	1910
29000	Arizona	1940
29445	Ave Maria	1940

Q1(y) :- Movie(x,y,z), z='1940'.

x	y	z
29000	Arizona	1940
29445	Ave Maria	1940

Answer

y
Arizona
Ave Maria

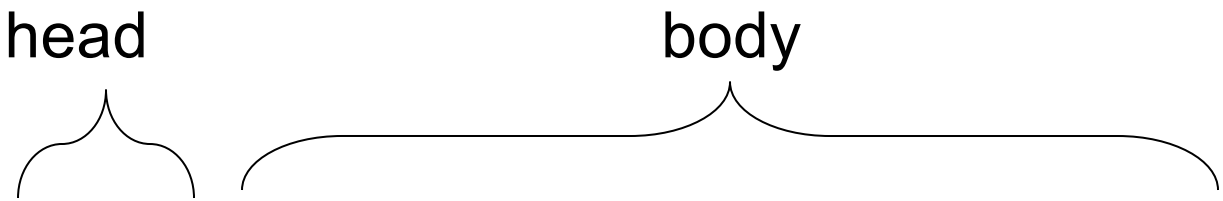
Usually displayed like this:

Q1('Arizona')
Q1('Ave Maria')

Anatomy of a Rule

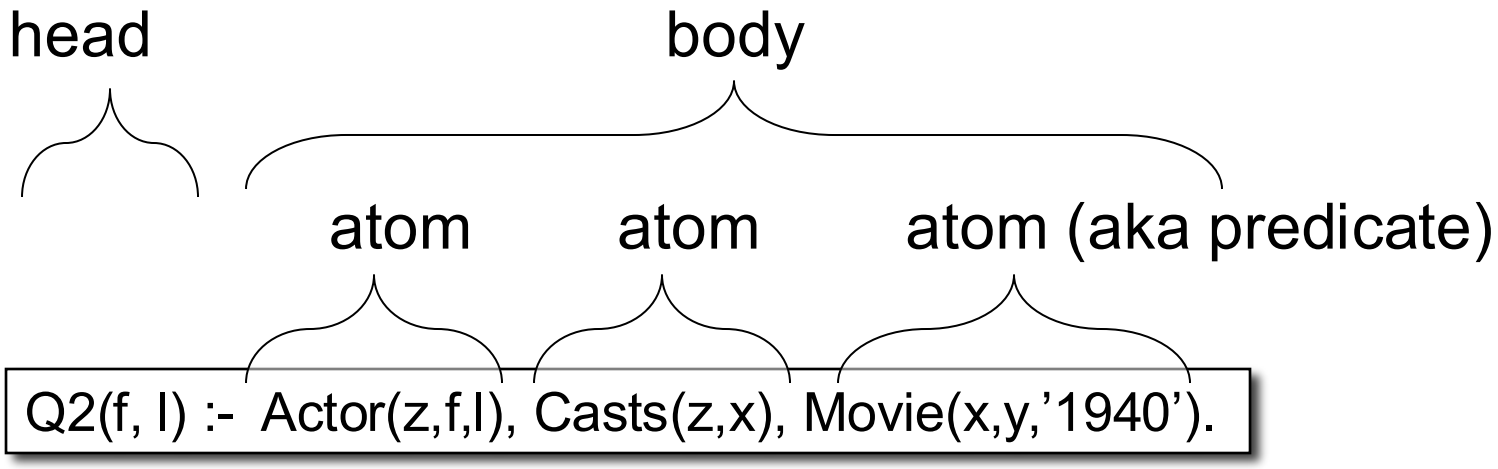
```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
```

Anatomy of a Rule

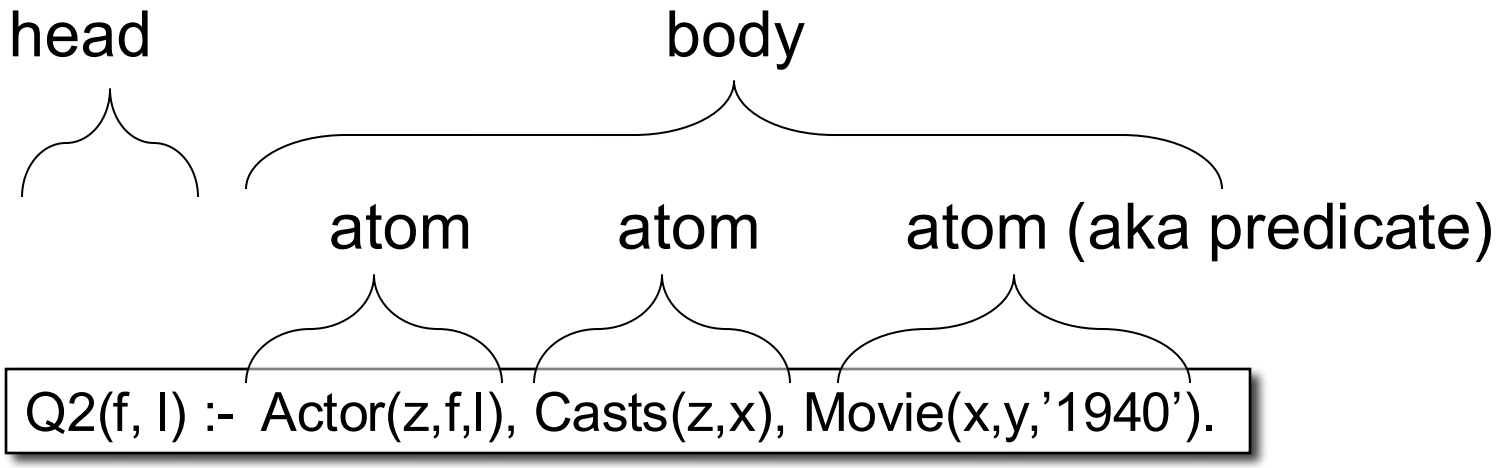



```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
```

Anatomy of a Rule



Anatomy of a Rule



f, l = head variables

x,y,z = existential variables

Discussion

- Rule body: a select-join expression
- Rule head specifies what to return
- Datalog program = multiple rules

Outline

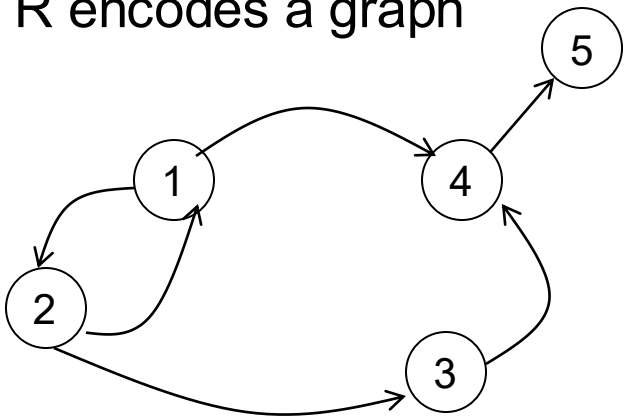
- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Multiple Rules

- Rule body may have EDBs and IDBs
- Same IDB may be in the head of multiple rules

Multiple Rules

R encodes a graph

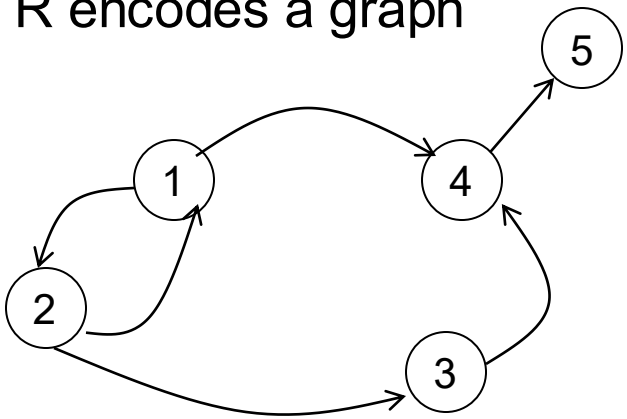


R=

1	2
2	1
2	3
1	4
3	4
4	5

Multiple Rules

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

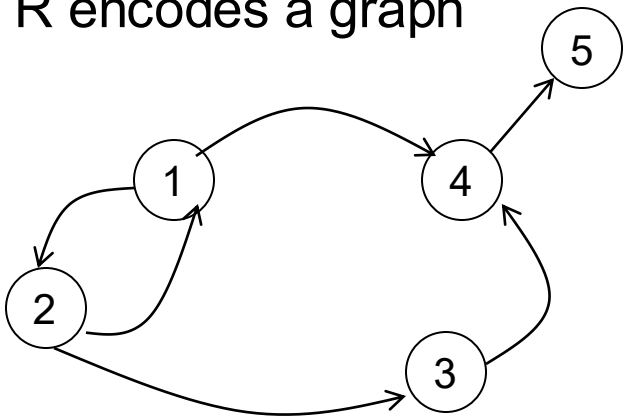
Entering data

```
R(1,2).
R(2,1).
…
R(4,5).
```

As facts

Multiple Rules

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Entering data

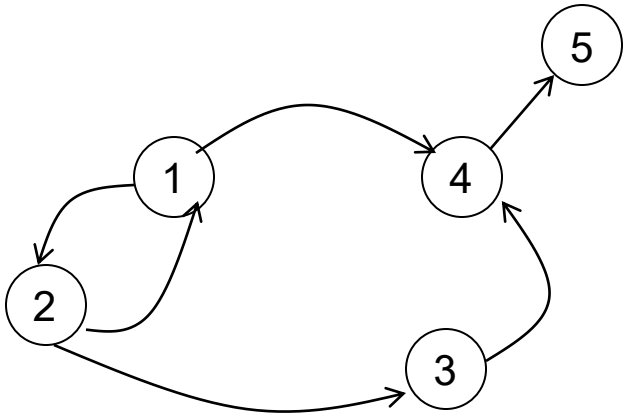
```
R(1,2).
R(2,1).
…
R(4,5).
```

As facts

```
.input  R("file-name")
```

From a file

Multiple Rules



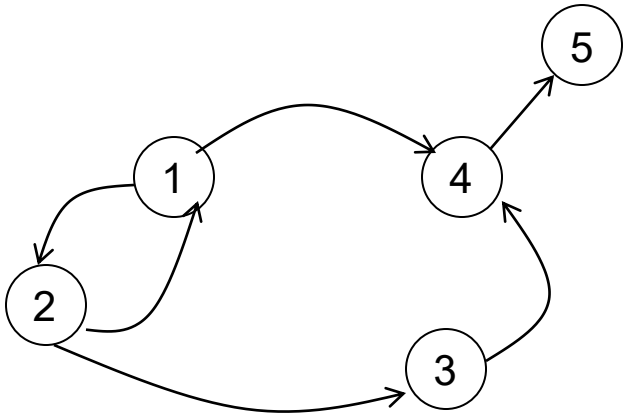
A(x) :- R(1,z),R(z,x)

R=

1	2
2	1
2	3
1	4
3	4
4	5

Answer??

Multiple Rules



A(x) :- R(1,z),R(z,x)

R=

1	2
2	1
2	3
1	4
3	4
4	5

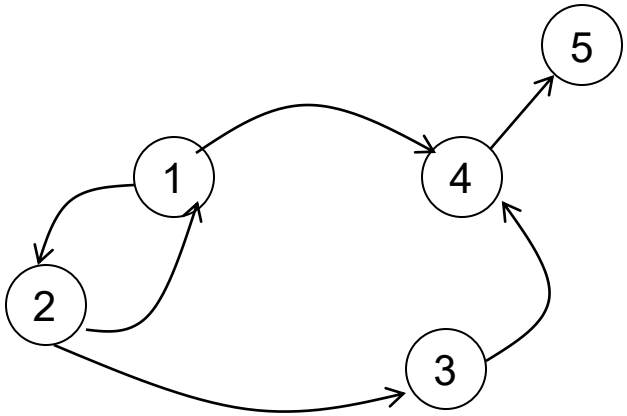
Answer:

A(1)

A(3)

A(5)

Multiple Rules



R=

1	2
2	1
2	3
1	4
3	4
4	5

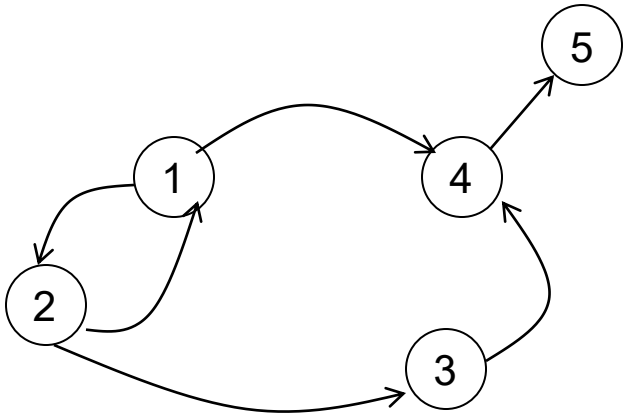
A(x) :- R(1,z),R(z,x)
B(x) :- A(z),R(z,x)

Answer:

Answer??

A(1)
A(3)
A(5)

Multiple Rules



R=

1	2
2	1
2	3
1	4
3	4
4	5

A(x) :- R(1,z),R(z,x)
B(x) :- A(z),R(z,x)

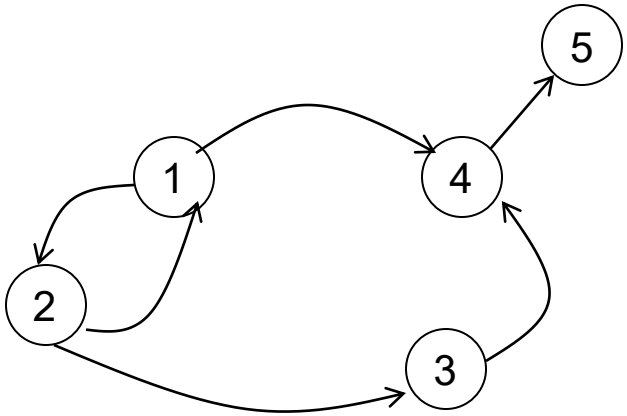
Answer:

Answer??

A(1)
A(3)
A(5)

B(2)
B(4)

Multiple Rules



R=

1	2
2	1
2	3
1	4
3	4
4	5

A(x) :- R(1,z),R(z,x)
B(x) :- A(z),R(z,x)

Answer:

A(1)
A(3)
A(5)

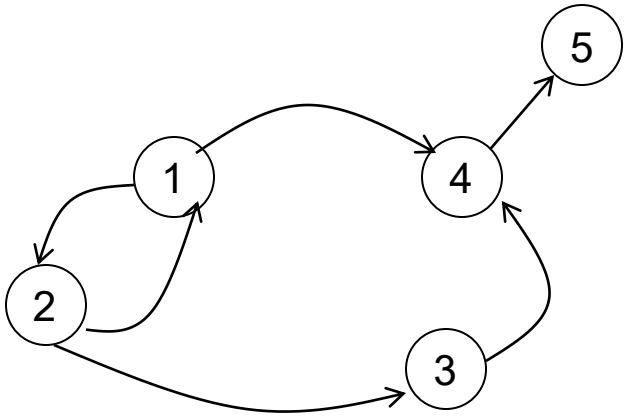
B(2)
B(4)

Answer??

Set semantics:
B(4) occurs
only once

Multiple Rules

Order of rules does not matter



R=

1	2
2	1
2	3
1	4
3	4
4	5

B(x) :- A(z),R(z,x)

A(x) :- R(1,z),R(z,x)

Answer:

A(1)

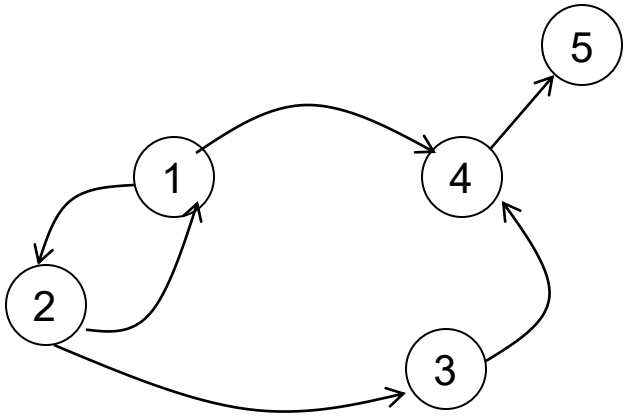
A(3)

A(5)

B(2)

B(4)

Multiple Rules

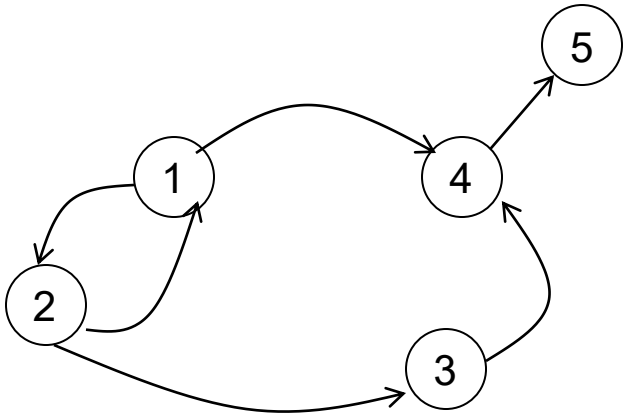


C(x) :- R(1,x),R(x,3)
C(x) :- R(1,x),R(3,x)

R=

1	2
2	1
2	3
1	4
3	4
4	5

Multiple Rules



R=

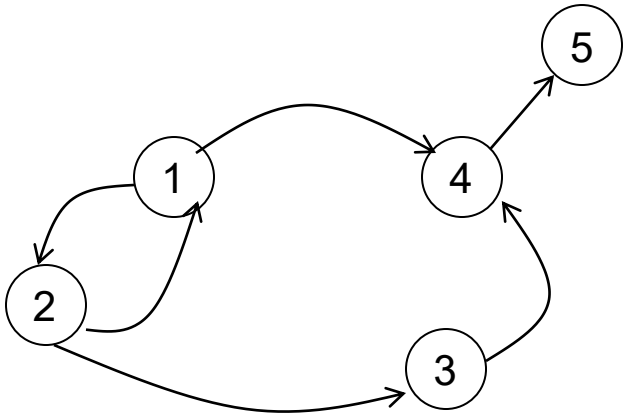
1	2
2	1
2	3
1	4
3	4
4	5

Multiple rules with same head means “union”

```
C(x) :- R(1,x),R(x,3)
C(x) :- R(1,x),R(3,x)
```

Multiple Rules

Multiple rules with same head means "union"



C(x) :- R(1,x),R(x,3)
C(x) :- R(1,x),R(3,x)

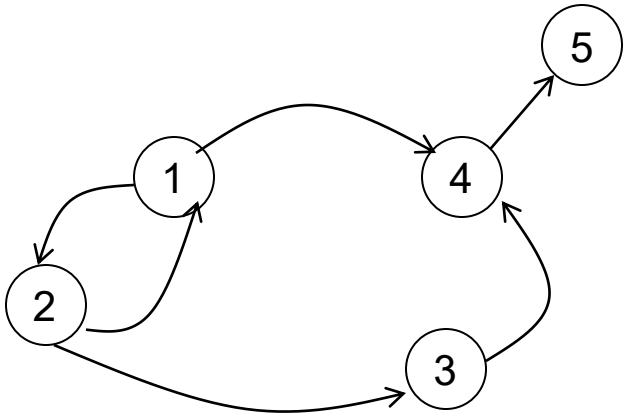
R=

1	2
2	1
2	3
1	4
3	4
4	5

Answer??

Multiple Rules

Multiple rules with same head means "union"




```
C(x) :- R(1,x),R(x,3)
C(x) :- R(1,x),R(3,x)
```

R=

1	2
2	1
2	3
1	4
3	4
4	5

Answer??

C(2)

C(4)

Discussion

- Rule body: EDBs and/or IDBs
- Rule heads w/ the same IDB: union
- Rule order does not matter
- Rules may be recursive: next

Outline

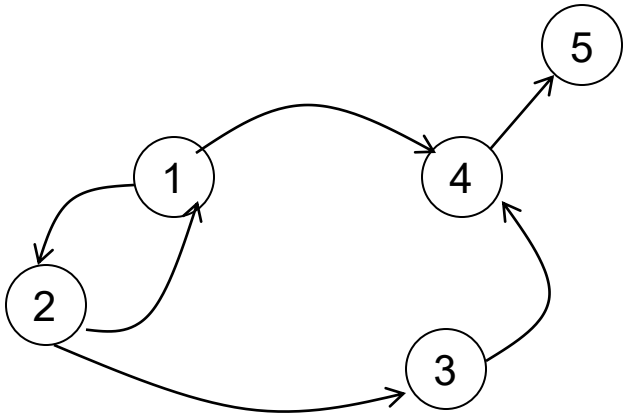
- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Recursion in Datalog

- Rules may be recursive

- Bottom up evaluation:
  - Start with empty IDBs
  - Compute rules repeatedly, increasing IDBs
  - Stop when no more change

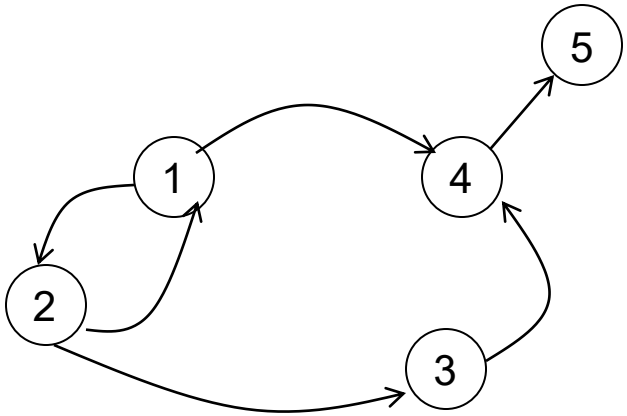
Example



R=

1	2
2	1
2	3
1	4
3	4
4	5

Example



T(x,y) :- R(x,y)

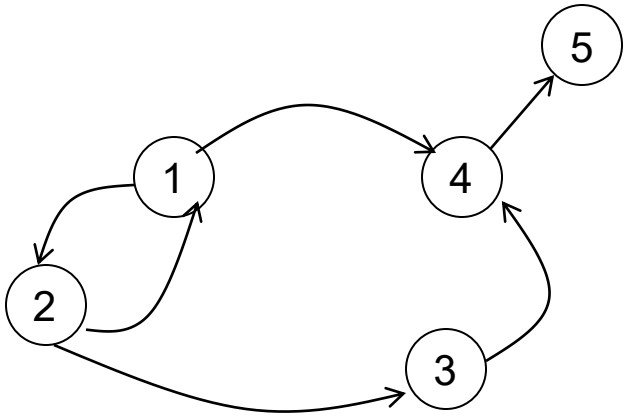
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example



R=

1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)

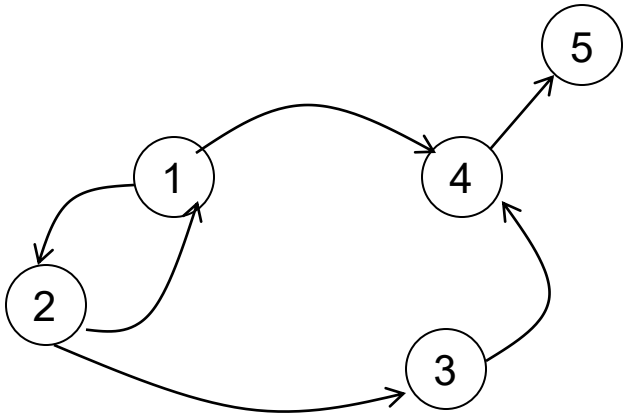
T(x,y) :- R(x,z), T(z,y)

What does it compute?

Initially:

T is empty.

Example



$$T(x,y) :- R(x,y)$$
$$T(x,y) :- R(x,z), T(z,y)$$

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

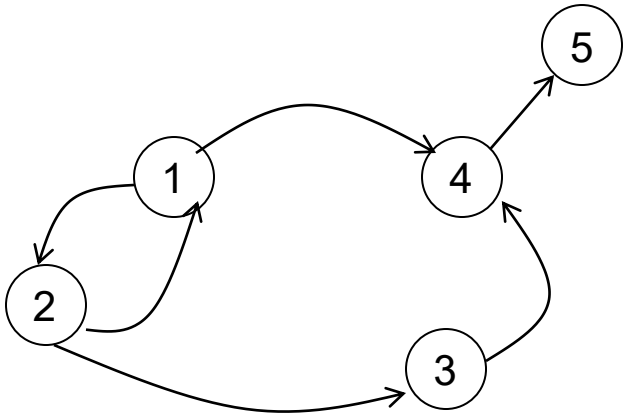
First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule generates nothing (because T is empty)

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

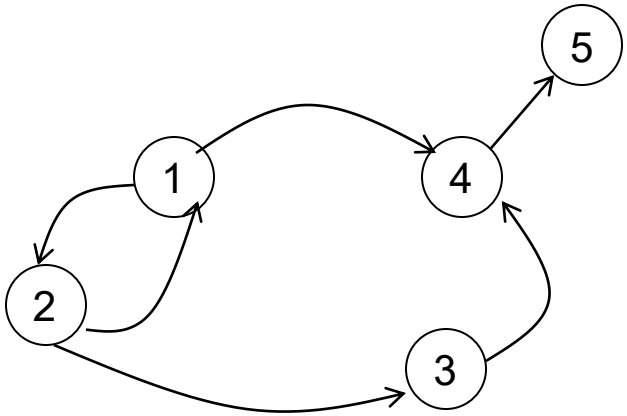
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

First rule generates this

Second rule generates this

New facts

Example



$$T(x,y) :- R(x,y)$$
$$T(x,y) :- R(x,z), T(z,y)$$

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

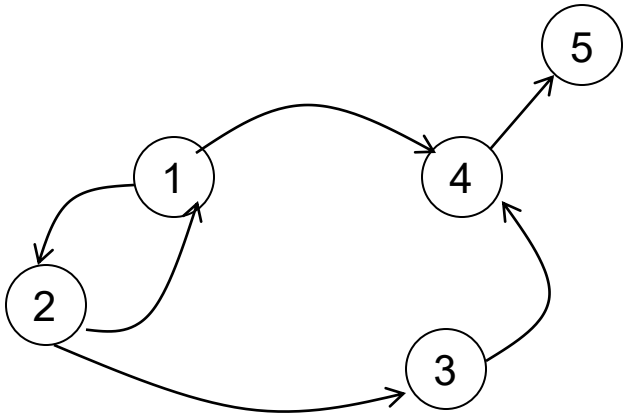
Both rules

First rule

Second rule

New fact

Example



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

$T(x,y) :- R(x,y)$
$T(x,y) :- R(x,z), T(z,y)$

What does it compute?

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

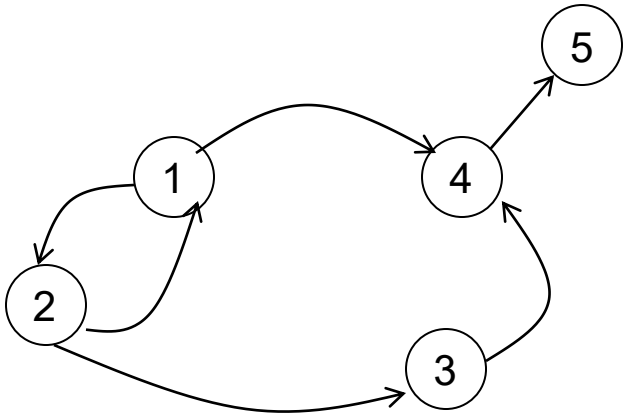
Third iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

Fourth iteration
T = (same)

No new facts. DONE

Example



$$T(x,y) :- R(x,y)$$
$$T(x,y) :- R(x,z), T(z,y)$$

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

Fourth iteration
T =
(same)

No new facts. DONE

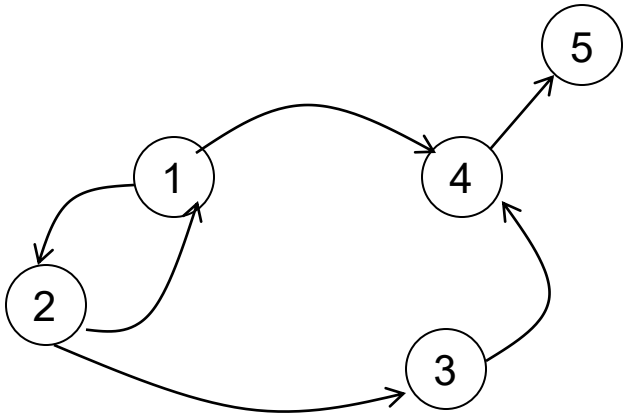
Iteration k computes pairs (x,y) connected by path of length ≤ k

Example

Computes the transitive closure of R

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

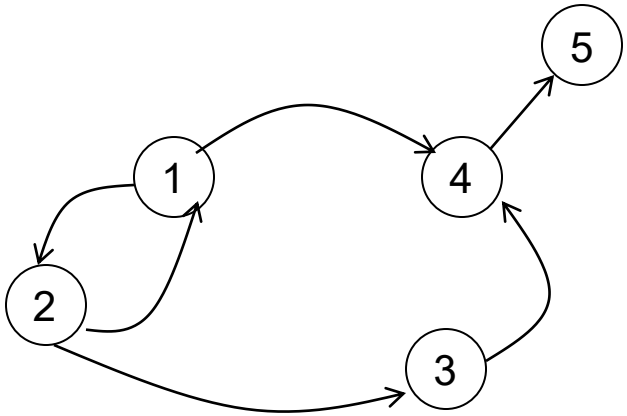
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

Fourth iteration
T =
(same)

No new facts. DONE

Iteration k computes pairs (x,y) connected by path of length ≤ k

Three Equivalent Programs



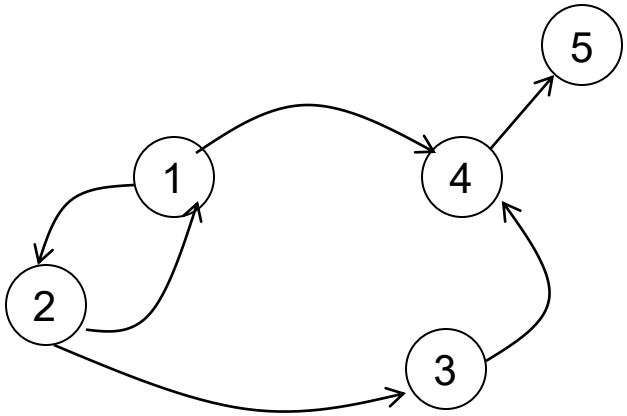

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

R=

1	2
2	1
2	3
1	4
3	4
4	5

Three Equivalent Programs



R=

1	2
2	1
2	3
1	4
3	4
4	5

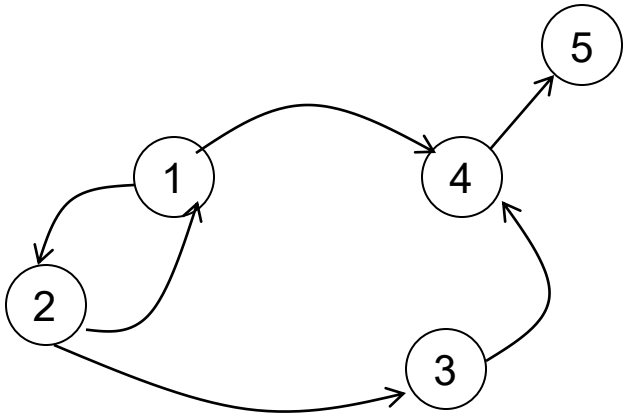
```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

Left linear

Three Equivalent Programs



R=

1	2
2	1
2	3
1	4
3	4
4	5

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

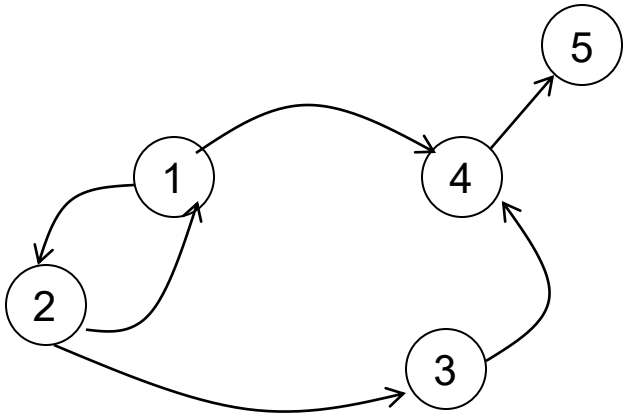
```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

Left linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)
```

Non-linear

Three Equivalent Programs



R=

1	2
2	1
2	3
1	4
3	4
4	5

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

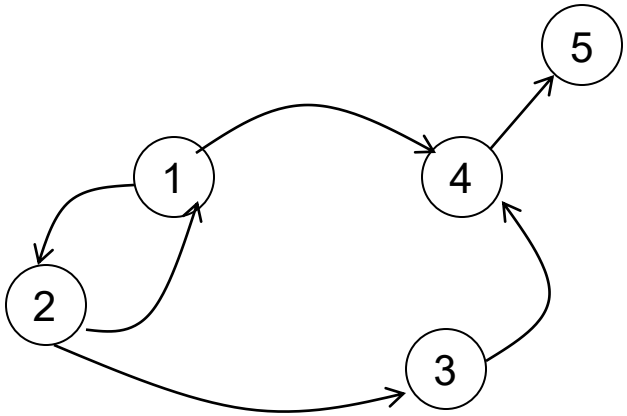
Left linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)
```

Non-linear

Question: how many iterations does each require?

Three Equivalent Programs



R=

1	2
2	1
2	3
1	4
3	4
4	5

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Right linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

Left linear

#iterations = diameter

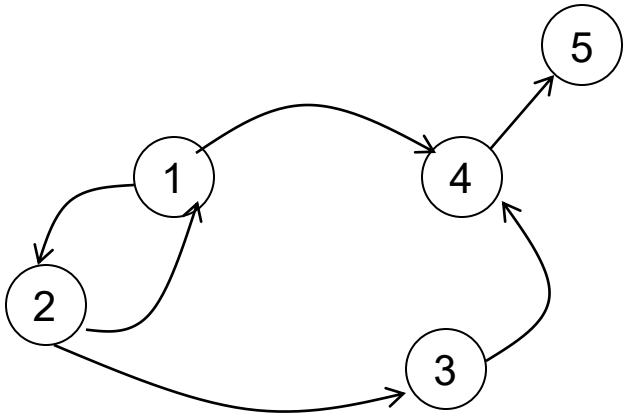
```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)
```

Non-linear

#iterations = log(diameter)

Question: how many iterations does each require?

Multiple IDBs

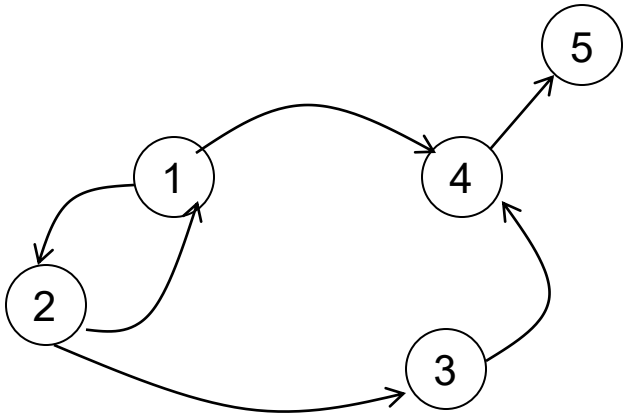


Find pairs of nodes (x,y) connected by a path of *even* length

R=

1	2
2	1
2	3
1	4
3	4
4	5

Multiple IDBs



Find pairs of nodes (x,y) connected by a path of *even* length

R=

1	2
2	1
2	3
1	4
3	4
4	5

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z), R(z,y)
Odd(x,y) :- Even(x,z), R(z,y)
```

Two IDBs: Odd(x,y) and Even(x,y)

Recap

- Rules may be recursive
- If every rule body has at most one recursive IDB, then the program is called linear. Otherwise: non-linear.
- Evaluation is bottom-up
  This is called the Naïve Algorithm (next)

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Naïve Evaluation Algorithm

- Every rule* → SPJ query

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule* → SPJ query

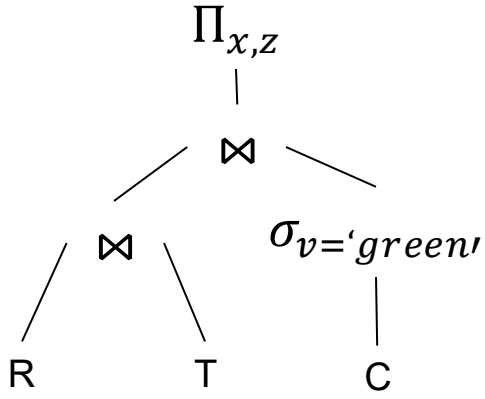
```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule* → SPJ query

T(x,z) :- R(x,y), T(y,z), C(y,'green')

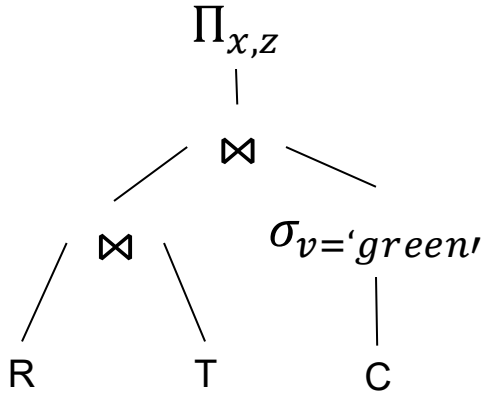


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

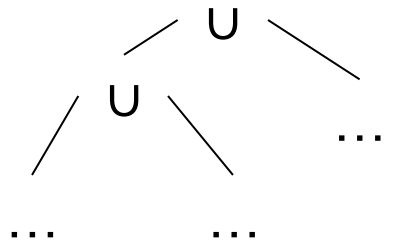
- Every rule* → SPJ query

```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```



- Multiple rules same head+ → USPJ

```
T(x,y) :-  …
T(x,y) :-  …
    …
```

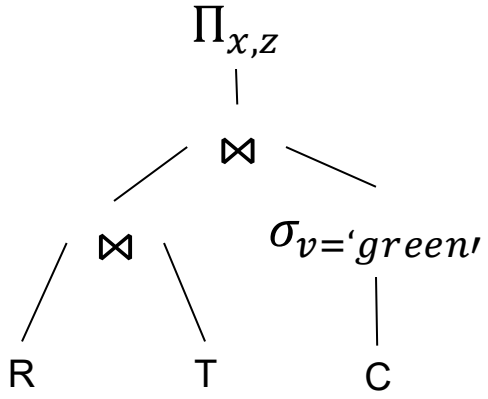


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

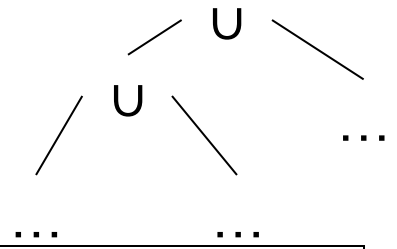
- Every rule* → SPJ query

T(x,z) :- R(x,y), T(y,z), C(y,'green')



- Multiple rules same head+ → USPJ

T(x,y) :- …
T(x,y) :- …
…



- Naïve Algorithm:

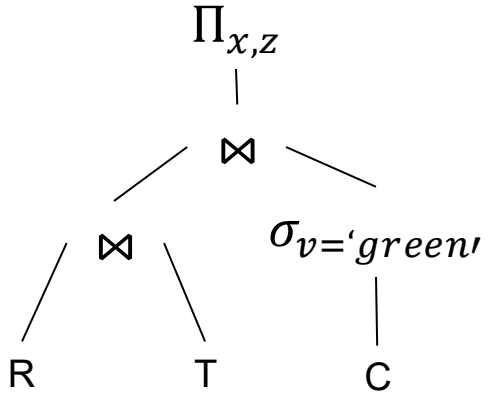
$IDB_0 := \emptyset$
repeat IDB_{t+1} :=USPJ(IDB_t)
until no more change

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule* → SPJ query

T(x,z) :- R(x,y), T(y,z), C(y,'green')



SQL optimizes the plan.

Souffle does not optimize the plan ☹

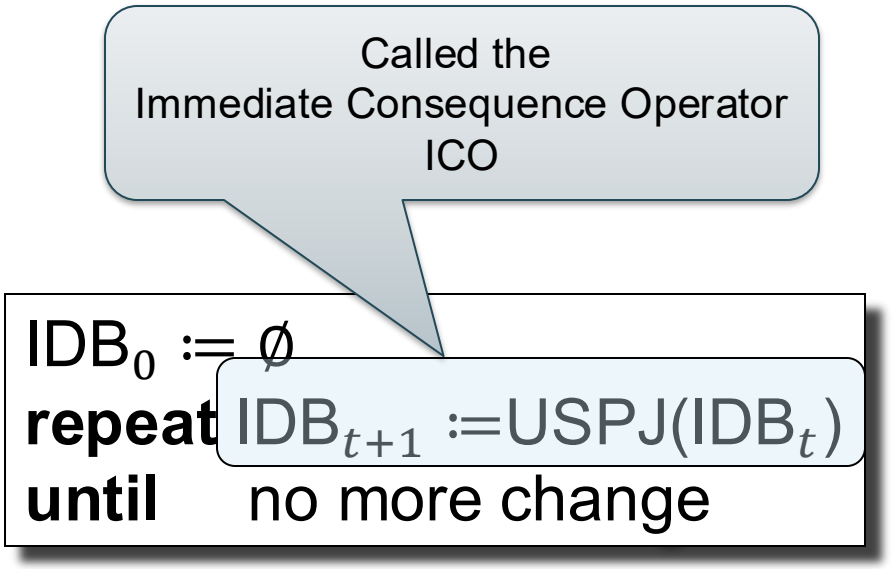
Computes joins left-to-right.

Equivalently, left-deep plan.

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Naïve Algorithm:



*SPJ = select-project-join
+USPJ = union-select-project-join

Example

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

Example

T(x,y) :- R(x,y)

T(x,y) :- R(x,z), T(z,y)

$T := \emptyset$;

repeat

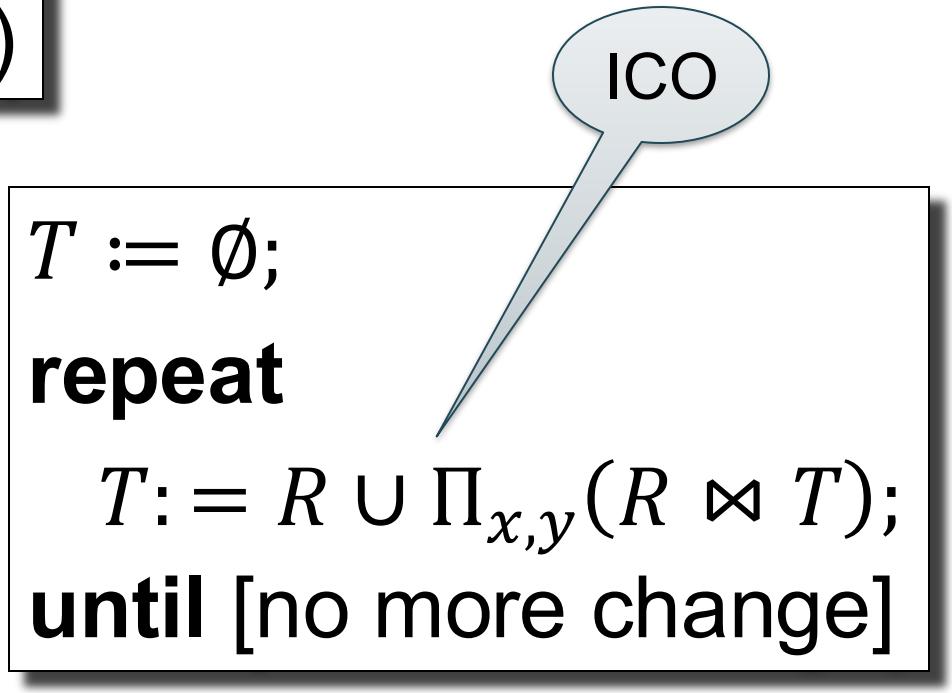
$T := R \cup \Pi_{x,y}(R \bowtie T)$;

until [no more change]

Example

T(x,y) :- R(x,y)

T(x,y) :- R(x,z), T(z,y)



Example

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Example

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Odd := ∅; Even := ∅;

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R)$;

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R)$;

Example

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Odd := ∅; Even := ∅;

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R)$;

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R)$;

$\text{Odd} := \text{Odd}_{\text{new}}$

$\text{Even} := \text{Even}_{\text{new}}$

Example

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Odd := ∅; Even := ∅;

repeat

  $\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R)$;

  $\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R)$;

  if $\text{Odd}=\text{Odd}_{\text{new}} \wedge \text{Even}=\text{Even}_{\text{new}}$

    then break

  $\text{Odd}:=\text{Odd}_{\text{new}}$

  $\text{Even}:=\text{Even}_{\text{new}}$

Example

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

ICO

Odd := ∅; Even := ∅;

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R);$

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R);$

if $\text{Odd}=\text{Odd}_{\text{new}} \wedge \text{Even}=\text{Even}_{\text{new}}$

then break

$\text{Odd}:=\text{Odd}_{\text{new}}$

$\text{Even}:=\text{Even}_{\text{new}}$

Example

Theorem The Naïve Algorithm terminates after a number of iteration that is at most a polynomial in the size of the database

Before we prove this, a review of **monotone queries**

Monotone Queries

- A query with input relations R, S, T, … is called *monotone* if, whenever we increase a relation, the query answer also increases (or stays the same)
- *Increase* here means *larger set*

Monotone Queries

- A query with input relations R, S, T, … is called *monotone* if, whenever we increase a relation, the query answer also increases (or stays the same)
- *Increase* here means *larger set*
- Mathematically

If $R \subseteq R', S \subseteq S', \ldots$ **then** $Q(R, S, \ldots) \subseteq Q(R', S', \ldots)$

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

NON-MONOTONE

Back to Datalog

Theorem The Naïve Algorithm terminates after a number of iteration that is at most a polynomial in the size of the database

Proof next

Naïve Evaluation Algorithm

The ICO is monotone (uses only $\sigma, \Pi, \bowtie, \cup$)

Naïve Evaluation Algorithm

The ICO is monotone (uses only $\sigma, \Pi, \bowtie, \cup$)

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction

Naïve Evaluation Algorithm

The ICO is monotone (uses only $\sigma, \Pi, \bowtie, \cup$)

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Naïve Evaluation Algorithm

The ICO is monotone (uses only $\sigma, \Pi, \bowtie, \cup$)

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:

$$\text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1})$$

Naïve Evaluation Algorithm

The ICO is monotone (uses only $\sigma, \Pi, \bowtie, \cup$)

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:
$IDB_{t+1} = \text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1}) = IDB_{t+2}$

Naïve Evaluation Algorithm

Number of iterations of naïve algorithm:

$$\leq \sum_{R \in IDB} n^{arity(R)} = O\left(n^{\max(arity)}\right)$$

- n = number of distinct values in EDBs

Proof: there are $\leq \sum_{R \in IDB} n^{arity(R)}$ possible facts that can be inserted to IDBs

Take-Away

- Datalog *always* terminates in PTIME
- Only holds for monotone programs
- And cannot compute new values:

  T(x, v) :- T(x,w), v=w+1

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Problem with the Naïve Algorithm

- Same facts are discovered repeatedly
- The *semi-naïve* algorithm reduces the number of rediscovered facts
- Uses *incremental view maintenance*

Incremental View Maintenance

- Materialized view: $V = Q(R, S, T, \ldots)$
- A table gets updated: $R \leftarrow R \cup \Delta R$
- Want to update the view: $V \leftarrow V \cup \Delta V$

IVM: compute $\Delta V = \Delta Q(\Delta R, R, S, T, \ldots)$

IVM

Example 1:

V(x,y) :- R(x,z),S(z,y)

If $R \leftarrow R \cup \Delta R$
then what is $\Delta V(x,y)$?

IVM

Example 1:

$V(x,y) :- R(x,z),S(z,y)$

If $R \leftarrow R \cup \Delta R$
then what is $\Delta V(x,y)$?

$\Delta V(x,y) :- \Delta R(x,z),S(z,y)$

IVM

Example 2:

V(x,y) :- R(x,z),S(z,y)

If R ← R ∪ΔR and S ← S ∪ΔS
then what is ΔV(x,y) ?

IVM

Example 2:

V(x,y) :- R(x,z),S(z,y)

If R ← R ∪ΔR and S ← S ∪ΔS
then what is ΔV(x,y) ?

ΔV(x,y) :- ΔR(x,z),S(z,y)
ΔV(x,y) :- R(x,z), ΔS(z,y)
ΔV(x,y) :- ΔR(x,z), ΔS(z,y)

We use datalog convention
to represent the union
of three queries

IVM

Example 3:

V(x,y) :- T(x,z),T(z,y)

If $T \leftarrow T \cup \Delta T$
then what is $\Delta V(x,y)$?

IVM

Example 3:

V(x,y) :- T(x,z),T(z,y)

If T ← T ∪ΔT
then what is ΔV(x,y) ?

ΔV(x,y) :- ΔT(x,z),T(z,y)
ΔV(x,y) :- T(x,z), ΔT(z,y)
ΔV(x,y) :- ΔT(x,z), ΔT(z,y)

Semi-Naïve Evaluation Algorithm

Key idea:

- Use IVM in the inner loop of the naïve algorithm

Applies *only* to a monotone program

Semi-Naïve Evaluation Algorithm

$IDB := \emptyset$
repeat
  $IDB := USPJ(IDB)$
until no more change

Naive

Semi-Naïve Evaluation Algorithm

$IDB := \emptyset$
repeat
  $IDB := USPJ(IDB)$
until no more change

Naive

Semi-naive

$IDB := \Delta := NonRecursiveUSPJ$
repeat
  $\Delta := \Delta USPJ(IDB, \Delta) - IDB$
  if $\Delta = \emptyset$ then break
  $IDB := IDB \cup \Delta$

Example

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

$T := \emptyset$;

repeat

$T := R \cup \Pi_{x,y}(R \bowtie T)$;

until [no more change]

Naïve

Example

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \emptyset;$

repeat

$T := R \cup \Pi_{x,y}(R \bowtie T);$

until [no more change]

Naïve

Semi-naïve

$T := \Delta T := R;$

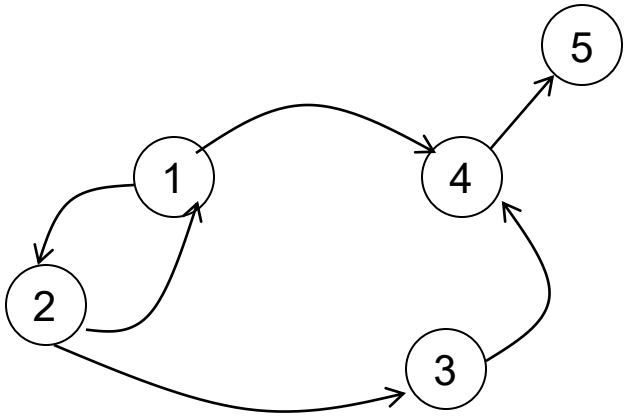
repeat

$\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$

if $\Delta T = \emptyset$ **then** break

$T := T \cup \Delta T;$

Example

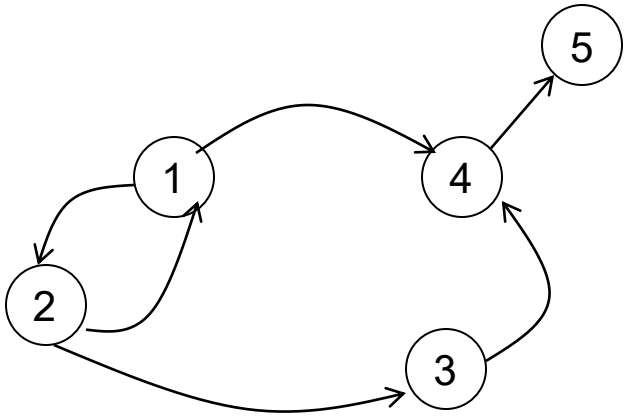


T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

R=

1	2
1	4
2	1
2	3
3	4
4	5

Example



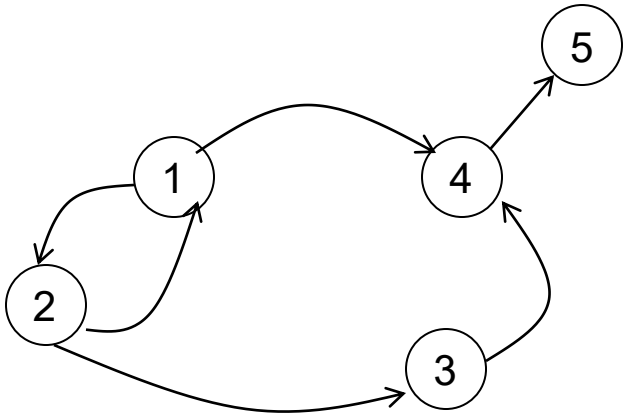
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

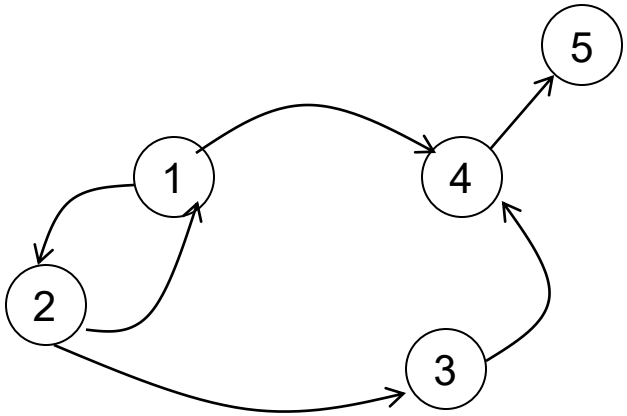
ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

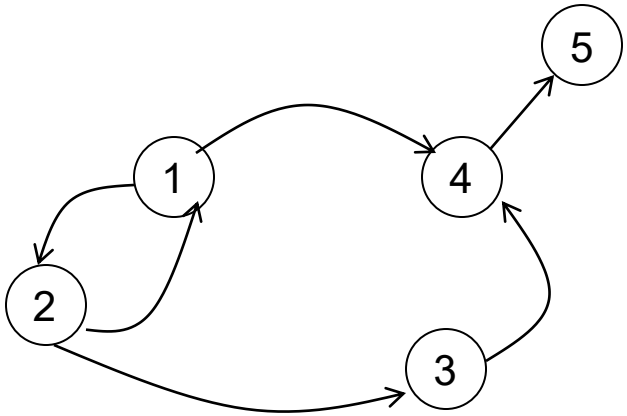
ΔT=
paths of
length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

ΔT= paths of length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

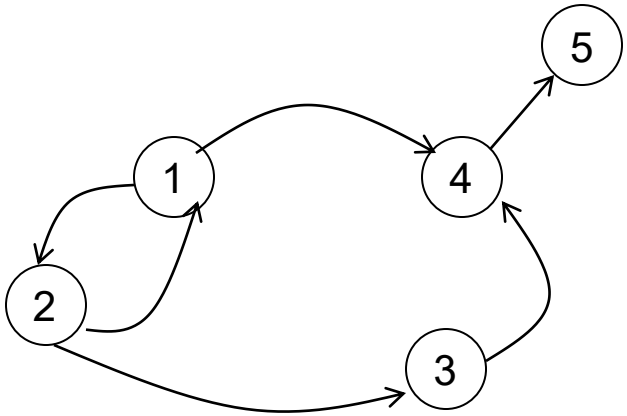
1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

Second iteration:

ΔT= paths of length 3

1	2
1	4
2	1
2	3
2	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
$\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
if $\Delta T = \emptyset$ **then** break
$T := T \cup \Delta T;$

…this difference

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

ΔT= paths of length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

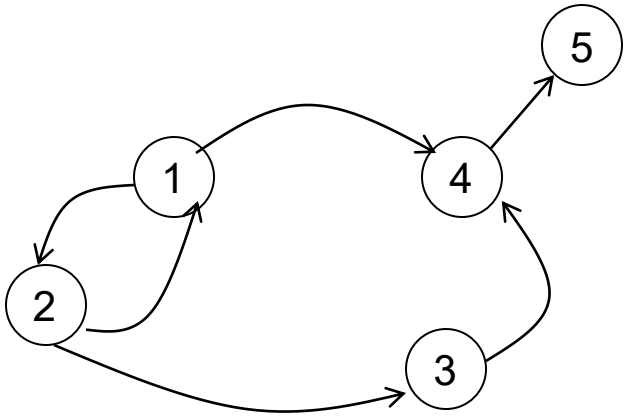
Second iteration:

ΔT= paths of length 3

1	2
1	4
2	1
2	3
2	5

Removed by the difference

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

ΔT= paths of length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

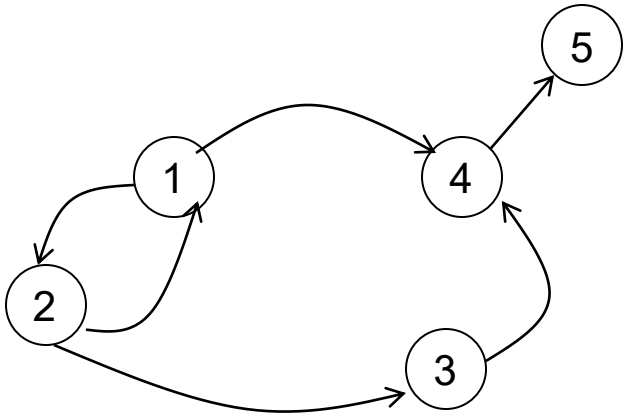
1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

Second iteration:

ΔT= paths of length 3

2	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

ΔT= paths of length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

Second iteration:

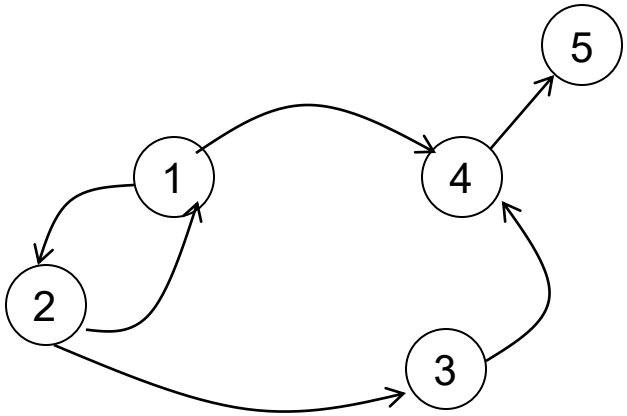
ΔT= paths of length 3

2	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5
2	5

Example



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

$T := \Delta T := R;$
repeat
  $\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
  if $\Delta T = \emptyset$ **then** break
  $T := T \cup \Delta T;$

R=

1	2
1	4
2	1
2	3
3	4
4	5

Initially:

ΔT=

1	2
1	4
2	1
2	3
3	4
4	5

T=

1	2
1	4
2	1
2	3
3	4
4	5

First iteration:

ΔT= paths of length 2

1	1
1	3
1	5
2	2
2	4
3	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5

Second iteration:

ΔT= paths of length 3

2	5

T=

1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5
2	5

Third iteration:

ΔT= paths of length 4

Discussion

- Datalog systems, SQL run semi-naïve
- When rule is non-linear, IVM more complicated:

  T(x,y) :- T(x,z),T(z,y)

  becomes:

  ΔT(x,y) :- T(x,z), ΔT(z,y)

  ΔT(x,y) :- ΔT(x,z),T(z,y)

  ΔT(x,y) :- ΔT(x,z), ΔT(z,y)
- SQL allows only linear recursion
- Postgres does not implement yet mutual recursion

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

Non-Montone Operations

Non-monotone queries are the Achille's heel of datalog. Our plan:

- Non-monotone queries in Souffle
- Why recursion and negation don't mix

Non-monotone Queries in Souffle

Non-monotone Extensions

- Aggregates
- Grouping

No standard syntax
We will follow Souffle

- Negation

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Aggregates

Syntax: min x : { Actor(x, y, _), y = 'John' }

```
Q(m) :- m = min x : { Actor(x, y, _), y = 'John' }
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Aggregates

Syntax: min x : { Actor(x, y, _), y = 'John' }

```
Q(m) :- m = min x : { Actor(x, y, _), y = 'John' }
```

Meaning (in SQL)

```
SELECT min(id) as m
FROM Actor as a
WHERE a.name = 'John'
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Aggregates

Syntax: min x : { Actor(x, y, _), y = 'John' }

```
Q(m) :- m = min x : { Actor(x, y, _), y = 'John' }
```

Meaning (in SQL)

```
SELECT min(id) as m
FROM Actor as a
WHERE a.name = 'John'
```

Aggregates in Souffle:

- count
- min
- max
- sum

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Grouping

```
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }
```

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Grouping

```
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }
```

Here we bind the year y

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, title, year)

Grouping

```
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }
```

Here we bind the year y

We count movies with year = y

Meaning (in SQL)

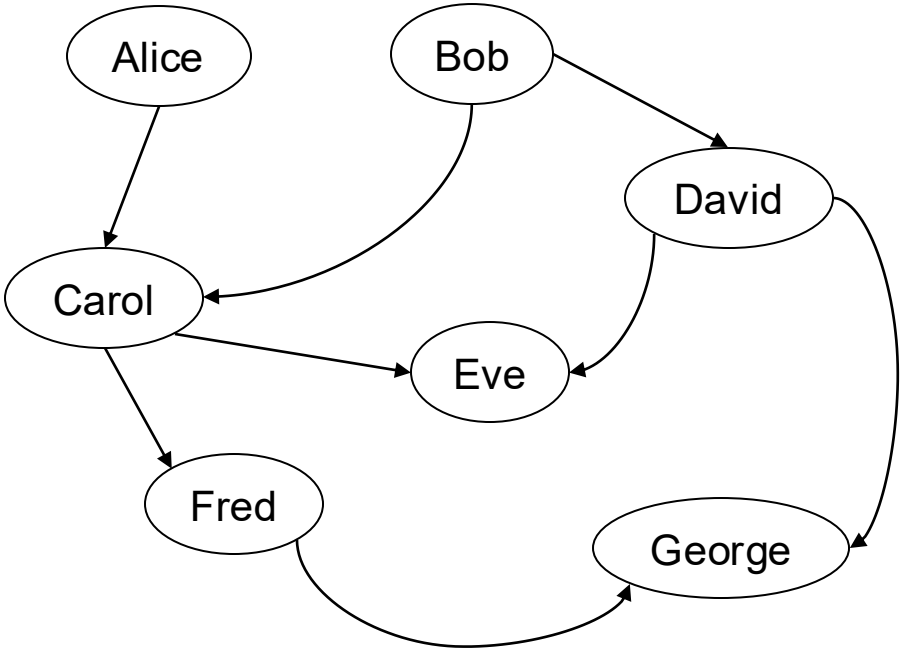
```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```

Stratified datalog

- The (semi-)naïve algorithm does not work with non-monotone rules

- Souffle programs must be stratified:
  - Rules can be grouped into strata…
  - …such that IDBs that occur in a body have been defined in earlier strata

Genealogy Database



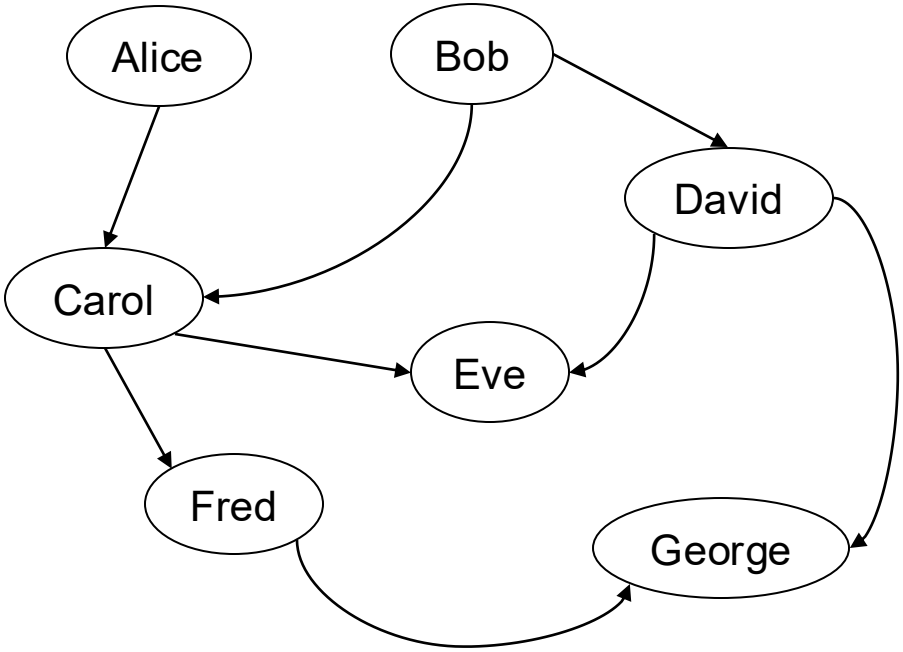
ParentChild

p	c
Alice	Carol
Bob	Carol
Bob	David
Carol	Eve
…	

ParentChild(p,c)

Count Descendants

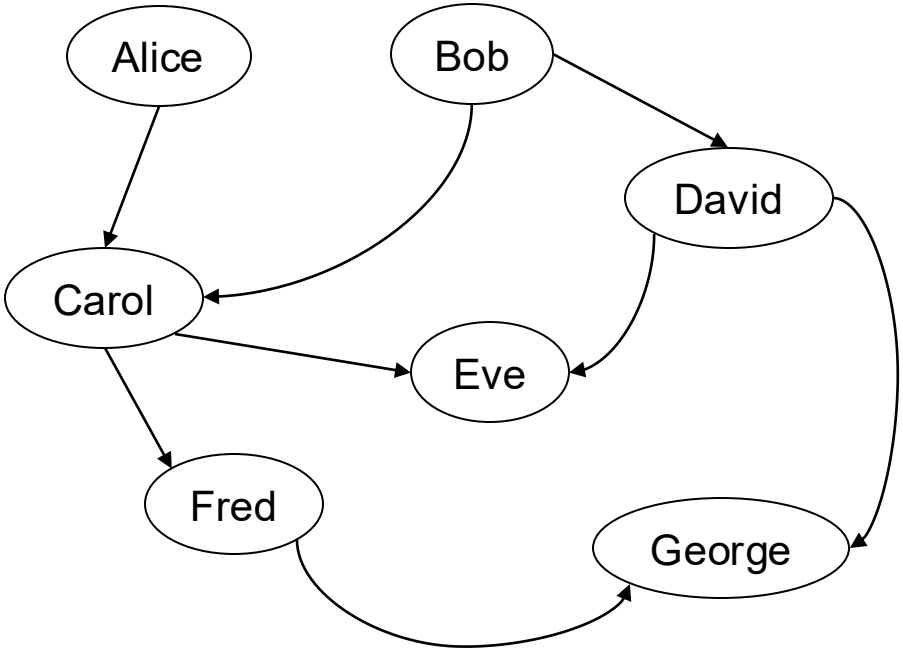
For each person, count his/her descendants



ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



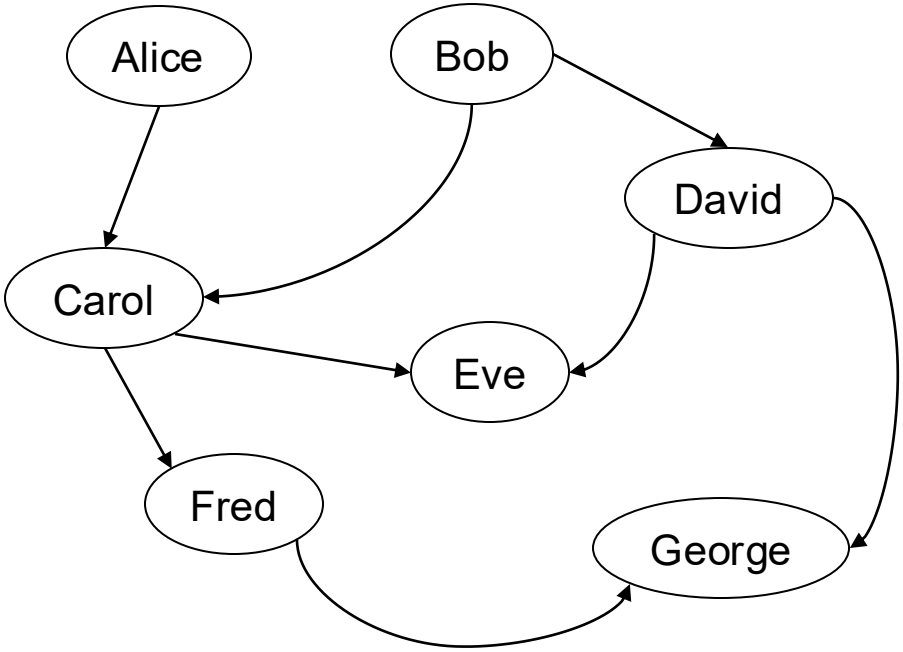
Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

Note: Eve and George do not appear in the answer (why?)

Count Descendants

Compute transitive closure of ParentChild

```
// for each person, compute his/her descendants
```

Count Descendants

Compute transitive closure of ParentChild

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Count Descendants

For each person, compute the total number of descendants

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Count Descendants

For each person, compute the total number of descendants

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.
```

Count Descendants

How many descendants does Alice have?

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.
```

Count Descendants

How many descendants does Alice have?

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.

// Find the number of descendants of Alice
Q(d) :- T(p,d), p = “Alice”.
```

Count Descendants

How many descendants does Alice have?

Stratum 1

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

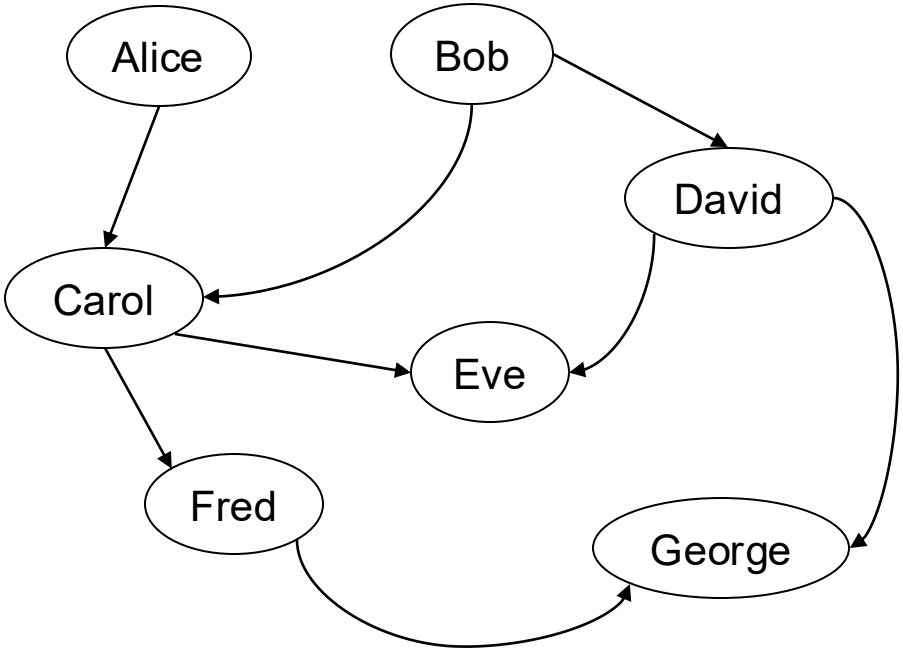
```
// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.

// Find the number of descendants of Alice
Q(d) :- T(p,d), p = “Alice”.
```

Stratum 2

ParentChild(p,c)

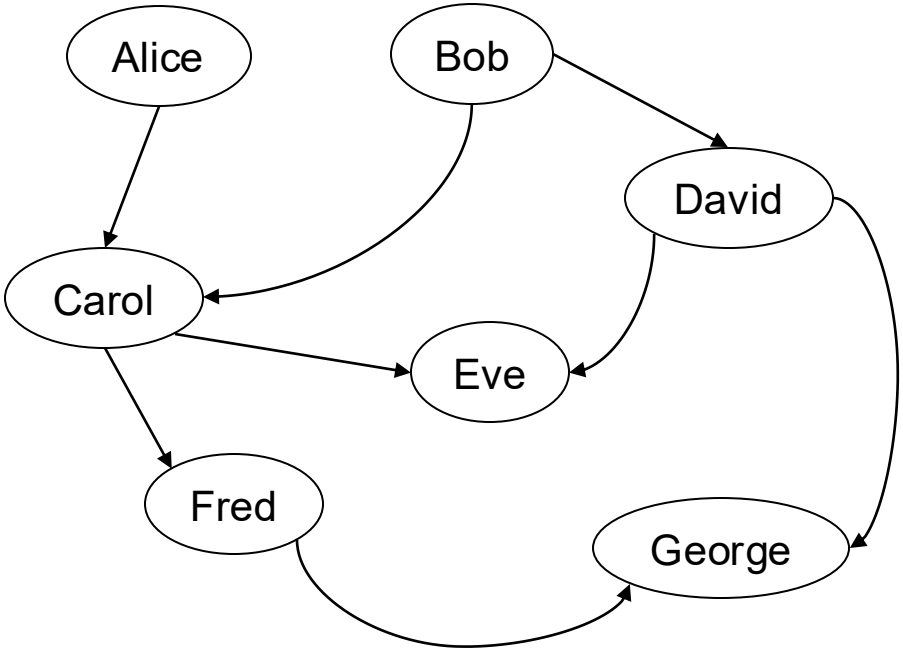
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice



Negation: use “!”

Find all descendants of Bob that are not descendants of Alice



Answer

x
David

ParentChild(p,c)

Negation: use "!"

Find all descendants of Bob that are not descendants of Alice

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Negation: use “!”

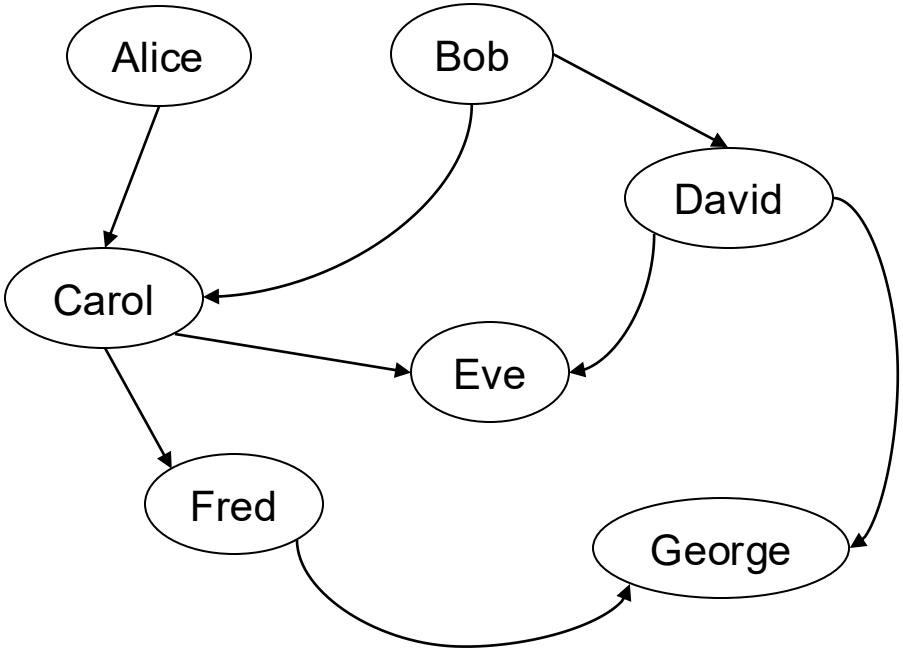
Find all descendants of Bob that are not descendants of Alice

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// Compute the answer: notice the negation
Q(x) :- D(“Bob”,x), !D(“Alice”,x).
```

ParentChild(p,c)

Same Generation

Two people are in the *same generation* if they are descendants at the same generation of some common ancestor



SG

p1	p2
Carol	David
Eve	George
Fred	George
Fred	Eve

Same Generation

Compute pairs of people at the same generation

```
// common parent
```

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
```

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
```

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
```

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
```

Problem: this includes answers like SG(Carol, Carol)

And also SG(Eve, George), SG(George, Eve)

How to fix?

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y), x < y

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y),
           SG(p,q),  x < y
```

Outline

- Syntax
- Single rule
- Multiple rules
- Recursion
- Naïve algorithm
- Semi-naïve algorithm
- Non-monotone operations
- Recursion in SQL

SQL: WITH RECURSIVE

Warning

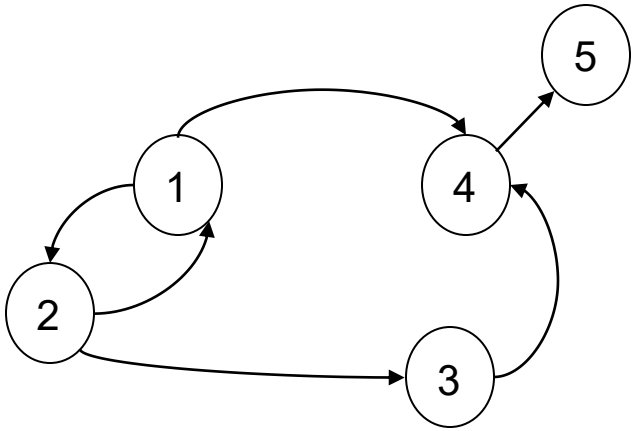
- SQL Recursion: notoriously bad design
- Right design: datalog (in a few weeks)
- Until then, fasten your seat-belts!

WITH RECURSIVE

```
WITH RECURSIVE TBL AS (
            SELECT…FROM…                    -- non-recursive rule
               UNION
            SELECT … FROM …[TBL]…)          -- recursive rule
SELECT … FROM …[TBL] …
```

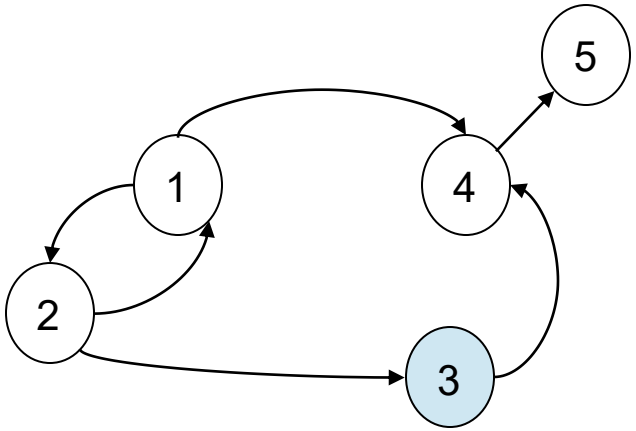
Example

Find all nodes x that have a path to node 3



Example

Find all nodes x that have a path to node 3

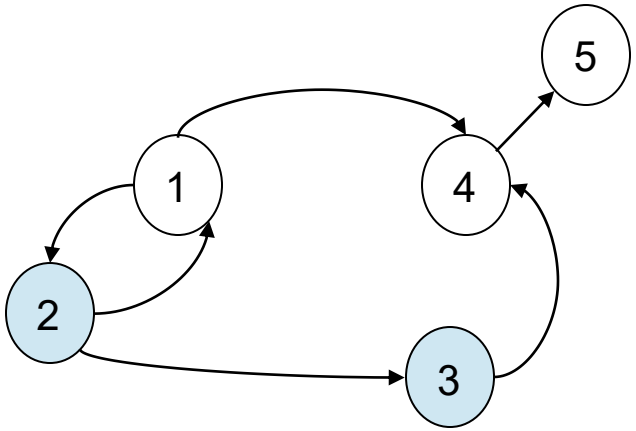


3 itself

```
SELECT 3 as v;
```

Example

Find all nodes x that have a path to node 3



3 itself

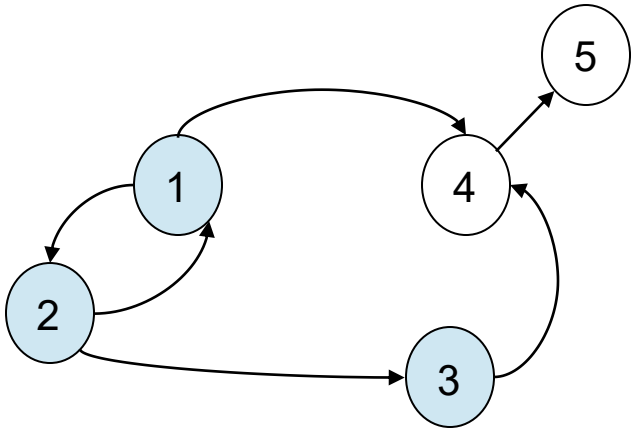
```
SELECT 3 as v;
```

Nodes connected to 3

```
… UNION
SELECT x.src as v
FROM Edge x
WHERE x.dst = 3;
```

Example

Find all nodes x that have a path to node 3



3 itself

```
SELECT 3 as v;
```

Nodes connected to 3

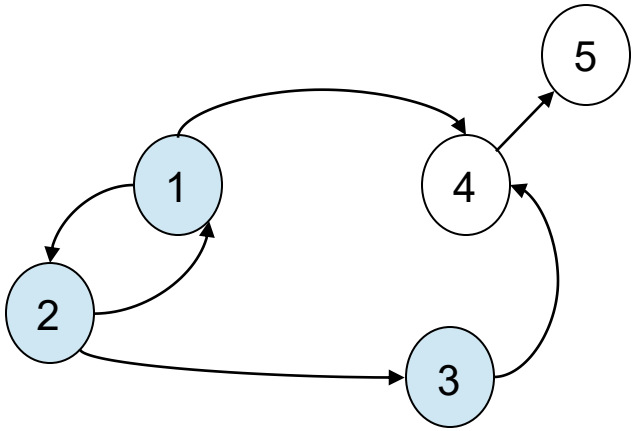
```
… UNION
SELECT x.src as v
FROM Edge x
WHERE x.dst = 3;
```

Nodes connected to them

```
… UNION
SELECT x.src as v
FROM Edge x, Edge y
WHERE x.dst=y.src
      and y.dst = 3;
```

Example

Find all nodes x that have a path to node 3



3 itself

```
SELECT 3 as v;
```

Nodes connected to 3

```
… UNION
SELECT x.src as v
FROM Edge x
WHERE x.dst = 3;
```

Need recursion to answer this query

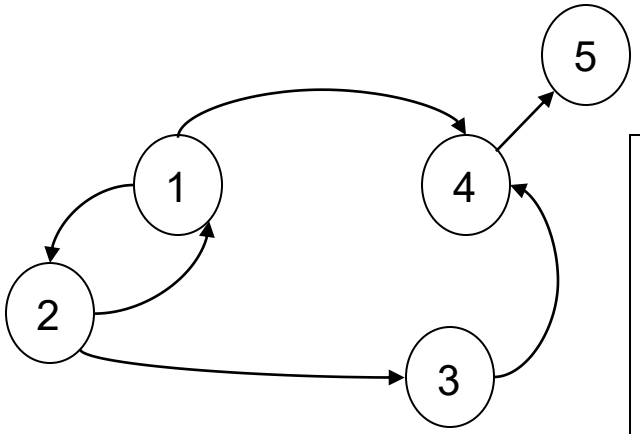
Nodes connected to them

```
… UNION
SELECT x.src as v
FROM Edge x, Edge y
WHERE x.dst=y.src
      and y.dst = 3;
```

■ ■ ■

Example

Find all nodes x that have a path to node 3




```
WITH RECURSIVE
   Answ AS (SELECT 3 as v
                  UNION
                  SELECT x.src as v
                  FROM Edge x, Answ a
                  WHERE x.dst = a.v)
SELECT * FROM Answ;
```

Semantics

Semi-naïve plus IVM:

- Initially TBL = non-recursive query
- Repeatedly evaluate the recursive query, add new tuples to TBL
- Stop when no more change
- Finally, evaluate the main query

Semantics

```
WITH RECURSIVE TBL AS (
        SELECT…FROM…                          -- non-recursive rule
          UNION
        SELECT … FROM …[TBL]…)                -- recursive rule
SELECT … FROM …[TBL] …
```

ΔTBL_0 := SELECT…FROM… -- non-recursive rule

$TBL_0 := \Delta TBL_0$

t := 0

REPEAT t := t+1

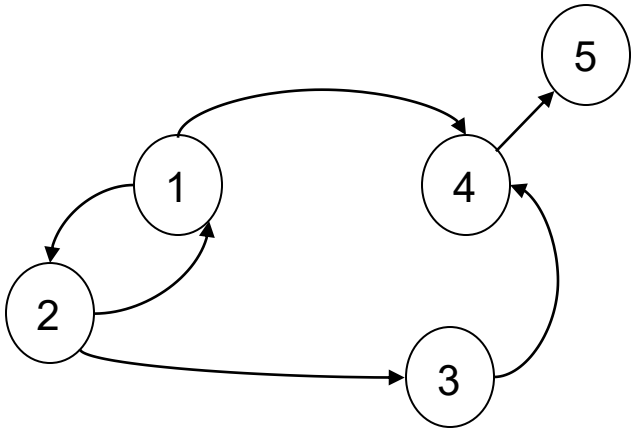
ΔTBL_t := SELECT … FROM …[ΔTBL_{t-1}]… -- recursive rule

$TBL_t := TBL_{t-1}$ UNION ΔTBL_t

UNTIL no more change

Example

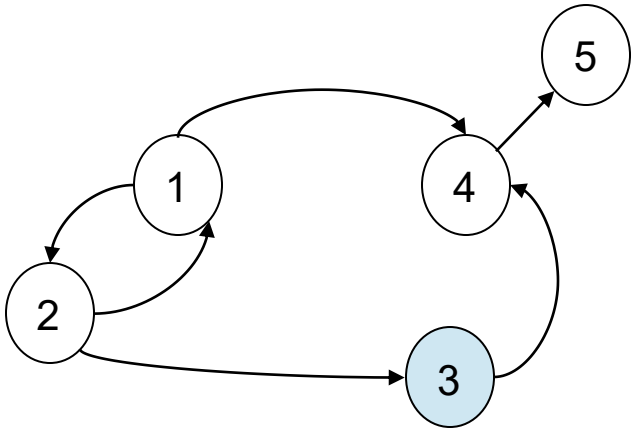
Find all nodes x that have a path to node 3




```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
                UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

Example

Find all nodes x that have a path to node 3

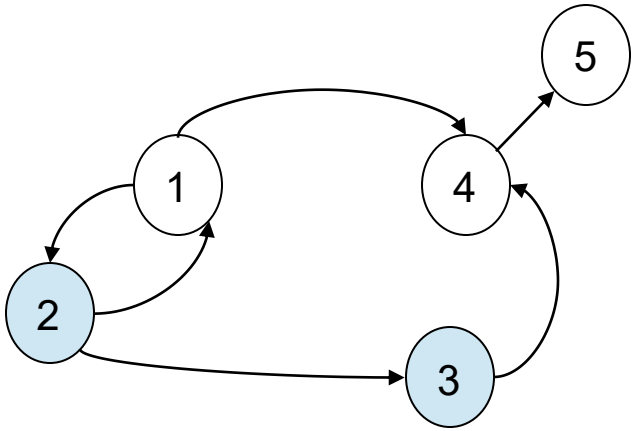



```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
             UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

t	$\Delta Answ_t$	$Answ_t$
0	3	3

Example

Find all nodes x that have a path to node 3

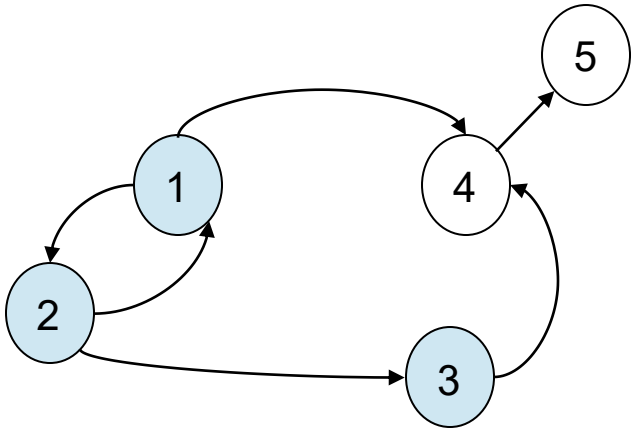



```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
             UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

t	$\Delta Answ_t$	$Answ_t$
0	3	3
1	2	3, 2

Example

Find all nodes x that have a path to node 3

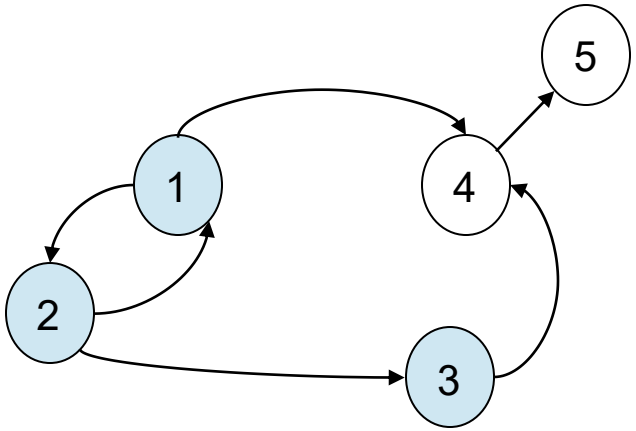



```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
                UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

t	$\Delta Answ_t$	$Answ_t$
0	3	3
1	2	3, 2
2	1	3,2,1

Example

Find all nodes x that have a path to node 3




```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
                UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

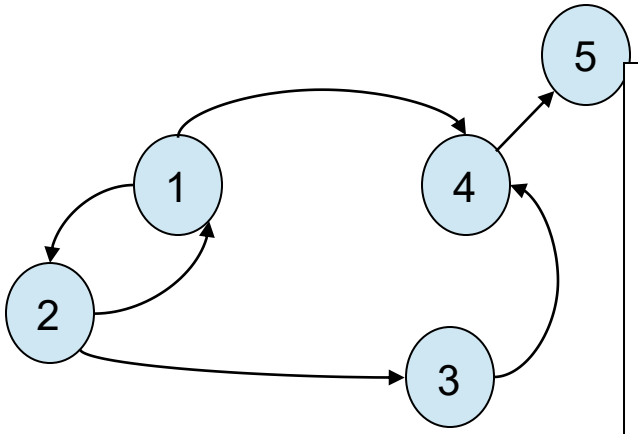
t	$\Delta Answ_t$	$Answ_t$
0	3	3
1	2	3, 2
2	1	3,2,1
3	2	3,2,1

Limitations

- Strict syntax:
  - One non-recursive rule
  - UNION one recursive rule
- May use UNION ALL, but that is often leads to non-termination
- No aggregates in the recursion
- Recursive relation may occur only once

Strict Syntax

Find all nodes x that have undirected a path to 3




```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
           UNION
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v
             UNION
           SELECT x.dst as v
           FROM Answ a, Edge x
           WHERE a.v = x.src
)
SELECT * FROM Answ;
```

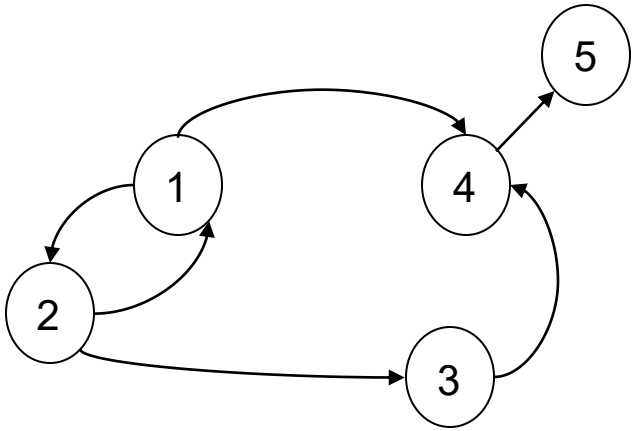
Backwards

Forward

Syntax Error

Union All is Dangerous

Find all nodes x that have a path to node 3



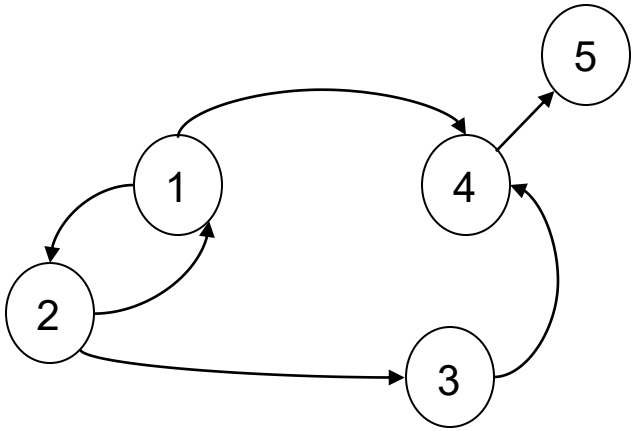
t	ΔAnsw$_t$	Answ$_t$
0	3	3
1	2	3, 2
2	1	3,2,1
3	2	3,2,1,2
4	1	3,2,1,2,1
5	2	3,2,1,2,1,2

```
WITH RECURSIVE
  Answ AS (SELECT 3 as v
           UNION ALL
           SELECT x.src as v
           FROM Edge x, Answ a
           WHERE x.dst = a.v)
SELECT * FROM Answ;
```

Does not terminate

No Aggregates in Recursion

Find all nodes x find the shortest path to node 3



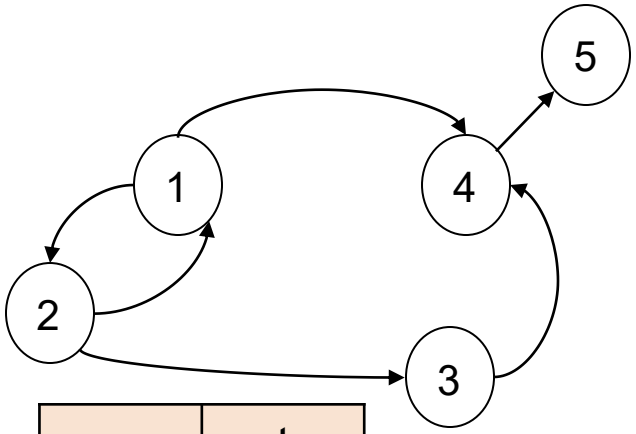

```
WITH RECURSIVE
  Answ AS (SELECT 3 as v, 0 as l
                UNION
           SELECT x.src as v, 1+a.l as l
           FROM Edge x, Answ a
           WHERE x.dst = a.v
                 and a.l < (SELECT count(*)
                            FROM Edge))
SELECT v, min(l) as l
FROM Answ
GROUP BY l;
```

v	l
3	0
2	1
1	2

Debugging

Debugging

Find all nodes x that have a path to node 3



v	t
3	0
2	1
1	2
2	3
1	4
2	5

```
WITH RECURSIVE
  Answ AS (SELECT 3 as v, 0 as t
           UNION
           SELECT x.src as v, a.t+1 as t
           FROM Edge x, Answ a
           WHERE x.dst = a.v and a.t<5)
SELECT * FROM Answ ORDER BY t;
```

t	$\Delta Answ_t$	$Answ_t$
0	3	3
1	2	3, 2
2	1	3,2,1
3	2	3,2,1
4	1	3,2,1
…		

Summary

- SQL supports recursion using WITH RECURSIVE
- Semi-naïve evaluation; hardwired IVM
- Consequence:
  - Only linear recursion (why??)
  - No stratification, no aggregates
- Limited optimizations

Recursion and Negation Don’t Mix

Monotone Queries

- A set function $F(R_1, R_2, \dots)$ is monotone if

$$R_1 \subseteq R_1', R_2 \subseteq R_2', \dots \Rightarrow F(R_1, R_2, \dots) \subseteq F(R_1', \dots)$$

Monotone Queries

- A set function $F(R_1, R_2, \dots)$ is monotone if

$$R_1 \subseteq R_1', R_2 \subseteq R_2', \dots \Rightarrow F(R_1, R_2, \dots) \subseteq F(R_1', \dots)$$

- Set difference is not monotone:

$$R_1 - R_2 \not\subseteq R_1 - R_2'$$

Monotone Queries

- A set function $F(R_1, R_2, \dots)$ is monotone if

$$R_1 \subseteq R_1', R_2 \subseteq R_2', \dots \Rightarrow F(R_1, R_2, \dots) \subseteq F(R_1', \dots)$$

- Set difference is not monotone:

$$R_1 - R_2 \not\subseteq R_1 - R_2'$$

- Aggregates are not montone:

$$\{1 + 2\} \not\subseteq \{1 + 2 + 3\}$$

Fixpoint Theorems

On the whiteboard:

- Knaster-Tarski
- Kleene

Non-Monotone Features

- Negation
- Aggregates/group-by

Three Useful Queries w/ Negation

Transitive closure of the complement

$$\mathrm{NR}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,z), \mathrm{T}(z,y)$$

Three Useful Queries w/ Negation

Transitive closure of the complement

$$\mathrm{NR}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,z), \mathrm{T}(z,y)$$

Complement of the transitive closure

$$\mathrm{T}(x,y) :- \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{R}(x,z), \mathrm{T}(z,y)$$
$$\mathrm{Answ}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{T}(x,y)$$

Three Useful Queries w/ Negation

Transitive closure of the complement

$$\mathrm{NR}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,z), \mathrm{T}(z,y)$$

Complement of the transitive closure

$$\mathrm{T}(x,y) :- \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{R}(x,z), \mathrm{T}(z,y)$$
$$\mathrm{Answ}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{T}(x,y)$$

The Win-Move game (will discuss later on the whiteboard)

$$\mathrm{W}(x) :- \mathrm{R}(x,y), \neg \mathrm{W}(y)$$

Recursion+Negation Don't Mix

Recall a simple fact:

A relation A of arity 0 is a Boolean value:

- $A = \emptyset$ means that A is false, or A=0
- $A = \{()\}$ means that A is true, or A=1
- FALSE < TRUE

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are
the models?

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

What does the naïve algorithm compute?

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

What does the naïve algorithm compute?

$(\mathrm{A}_0, \mathrm{B}_0) = (0,0);$

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

What does the naïve algorithm compute?

$(\mathrm{A}_0, \mathrm{B}_0) = (0,0);\ (\mathrm{A}_1, \mathrm{B}_1) = (1,1);$

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

What does the naïve algorithm compute?

$$(\mathrm{A}_0, \mathrm{B}_0) = (0,0);\ (\mathrm{A}_1, \mathrm{B}_1) = (1,1);\ (\mathrm{A}_2, \mathrm{B}_2) = (0,0); \cdots$$

Recursion+Negation Don't Mix

$$B() :- \neg A()$$
$$A() :- \neg B()$$

What are the models?

A=False, B=True

A=True, B=False

A=True, B=True

No minimal model

What are the fixpoints?

(False,True), (True, False)

No least fixpoint

What does the naïve algorithm compute?

$$(\mathrm{A}_0, \mathrm{B}_0) = (0,0);\ (\mathrm{A}_1, \mathrm{B}_1) = (1,1);\ (\mathrm{A}_2, \mathrm{B}_2) = (0,0); \cdots$$

Does not converge

Approaches to Negation

- Semi-positive datalog
- Stratified datalog
- Sophisticated model-theoretic notions:
  - Stable models (will not discuss)
  - Well founded semantics

Semi-positive Datalog

- EDB atoms may be positive or negated
- IDB atoms are positive
- ICO is monotone.
- Semantics: least fixpoint of ICO

Semi-positive Datalog

- E.g. transitive closure of complement

$$\mathrm{NR}(x,y) :- \mathrm{V}(x), \mathrm{V}(y), \neg \mathrm{R}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,y)$$
$$\mathrm{T}(x,y) :- \mathrm{NR}(x,z), \mathrm{T}(z,y)$$

Stratified Datalog

- Assign IDBs to strata 1, 2, 3, …
- IDBs computed in stratum s, may use non-monotone occurrences of IDBs at strata < s

Stratified Datalog

Formally: assign a stratum $s(R) \in \mathbb{N}$ to each IDB predicate R

The program is stratified if there exists a stratification such that:

Stratified Datalog

Formally: assign a stratum $s(R) \in \mathbb{N}$ to each IDB predicate R

The program is stratified if there exists a stratification such that:

- Positive atoms: $A(X) :- \cdots B(Y) \cdots$ $\quad s(A) \geq s(B)$

Stratified Datalog

Formally: assign a stratum $s(R) \in \mathbb{N}$ to each IDB predicate R

The program is stratified if there exists a stratification such that:

- Positive atoms: $A(X) :- \cdots B(Y) \cdots$ $\quad s(A) \geq s(B)$
- Negative atoms: $A(X) :- \cdots \neg B(Y) \cdots$ $\quad s(A) > s(B)$

Stratified Datalog

Formally: assign a stratum $s(R) \in \mathbb{N}$ to each IDB predicate R

The program is stratified if there exists a stratification such that:

- Positive atoms: $A(X) :- \cdots B(Y) \cdots$ $\quad s(A) \geq s(B)$
- Negative atoms: $A(X) :- \cdots \neg B(Y) \cdots$ $\quad s(A) > s(B)$
- Aggregates: $A(agg(\cdots)) :- body$ $\quad \forall B \in body: s(A) > s(B)$

Well-founded Semantics

- On the whiteboard in class

Summary of Datalog

- Simple, concise syntax
- (Semi)-naïve algorithm for monotone programs; otherwise, need stratification
- Ad-hoc restrictions in SQL:
  - Linear recursion only
  - Postgres: no mutual recursion

Grand Finale in 544

Takeaways from 544

- SQL: set-at-a-time, data independence
- Query execution/optimization, recursion

What I wished we had time to cover:

- Parallel query execution
- Transactions
- Provenance