Advanced Topics in Data Management

Lecture 5

Datalog

Motivation

- SQL can expression *relational queries;* Iteration/recursion need CTE construct

- Data processing today require iteration. Common solution: external driver

- Datalog is a language that allows both recursion and relational queries

Datalog

- Designed in the 80's
- Simple, concise, elegant
- Today is a hot topic: network protocols, static program analysis, DB+ML
- No standard, no reference implementation

Agenda

- $\boxed{\text{Definition}}$
- Naïve Algorithm
- Termination
- Semi-naïve Algorithm

Datalog program

- A datalog program = several rules
- Rules may be recursive
- Set semantics only

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

Schema

Datalog: Facts and Rules

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, 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, name, 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, name, 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'.
```

Find Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, 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, name, 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 Actors who acted in Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, 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, name, 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, name, 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)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Anatomy of a Rule

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

f, l = head variables
x,y,z = existential variables

Anatomy of a Rule

$$\text{Q2(f, l) :- } \overbrace{\text{Actor(z,f,l)}}^{\text{atom}}\text{, Casts(z,x), Movie(x,y,'1940').}$$

f, l = head variables
x,y,z = existential variables

Anatomy of a Rule

$$Q2(f, l) \text{ :- } \overbrace{\text{Actor}(z,f,l)}^{\text{atom}}, \overbrace{\text{Casts}(z,x)}^{\text{atom}}, \text{Movie}(x,y,\text{'1940'}).$$

f, l = head variables
x,y,z = existential variables

Anatomy of a Rule

$$\text{Q2(f, l) :- } \overbrace{\text{Actor(z,f,l)}}^{\text{atom}}\text{, }\overbrace{\text{Casts(z,x)}}^{\text{atom}}\text{, }\overbrace{\text{Movie(x,y,'1940')}}^{\text{atom (aka predicate)}}\text{.}$$

f, l = head variables
x,y,z = existential variables

Anatomy of a Rule

body

atom atom atom (aka predicate)

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

f, l = head variables
x,y,z = existential variables

Anatomy of a Rule

$$\overbrace{\text{Q2(f, l)}}^{\text{head}} \text{ :- } \overbrace{\overbrace{\text{Actor(z,f,l)}}^{\text{atom}}\text{, }\overbrace{\text{Casts(z,x)}}^{\text{atom}}\text{, }\overbrace{\text{Movie(x,y,'1940')}}^{\text{atom (aka predicate)}}}^{\text{body}}\text{.}$$

f, l = head variables

x,y,z = existential variables

Discussion

- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z) -- join
  - Q2(x,y) :- R(x,y), y > 5 -- selection
  - Q3(x) :- R(x,y) -- projection
  - Q4(x,z) :- R(x,y),S(y,5),T(y,z) -- SPJ

Discussion

- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z) -- join
  - Q2(x,y) :- R(x,y), y > 5 -- selection
  - Q3(x) :- R(x,y) -- projection
  - Q4(x,z) :- R(x,y),S(y,5),T(y,z) -- SPJ
- Unions can be obtained by writing two rules:
  - Q5(x,y) :- R(x,y)
    Q5(x,y) :- S(x,y)

Discussion

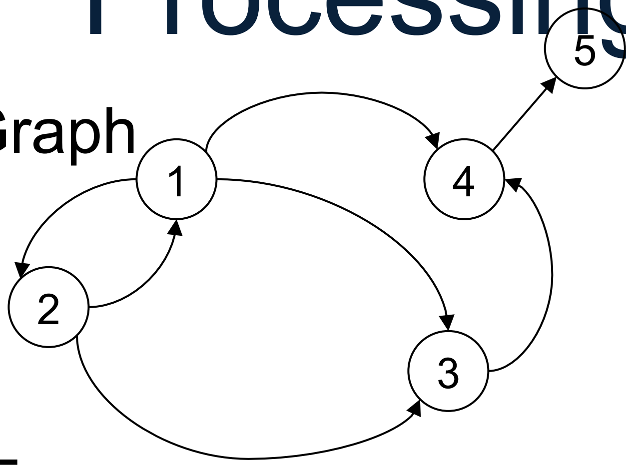
- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z) -- join
  - Q2(x,y) :- R(x,y), y > 5 -- selection
  - Q3(x) :- R(x,y) -- projection
  - Q4(x,z) :- R(x,y),S(y,5),T(y,z) -- SPJ
- Unions can be obtained by writing two rules:
  - Q5(x,y) :- R(x,y)
    Q5(x,y) :- S(x,y)
- Difference: it's getting complicated; more on this later

Discussion

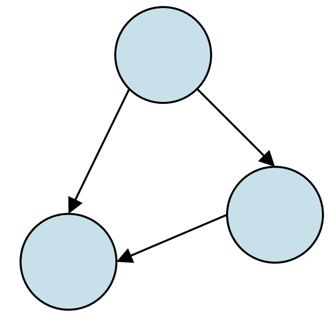
- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z)   -- join
  - Q2(x,y) :- R(x,y), y > 5   -- selection
  - Q3(x) :- R(x,y)   -- projection
  - Q4(x,z) :- R(x,y),S(y,5),T(y,z)   -- SPJ
- Unions can be obtained by writing two rules:
  - Q5(x,y) :- R(x,y)
    Q5(x,y) :- S(x,y)
- Difference: it's getting complicated; more on this later
- New! In datalog we can write *recursive* rules!

Processing Graphs in Datalog

Graph



Pattern Matching

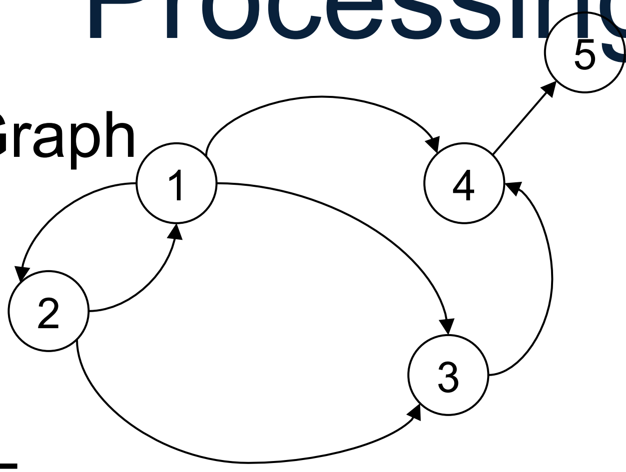


R=

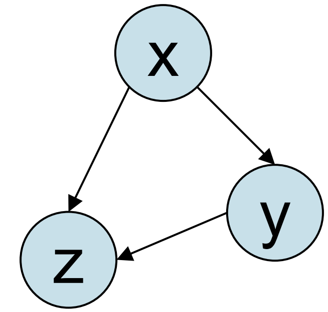
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in Datalog

Graph



Pattern Matching



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

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

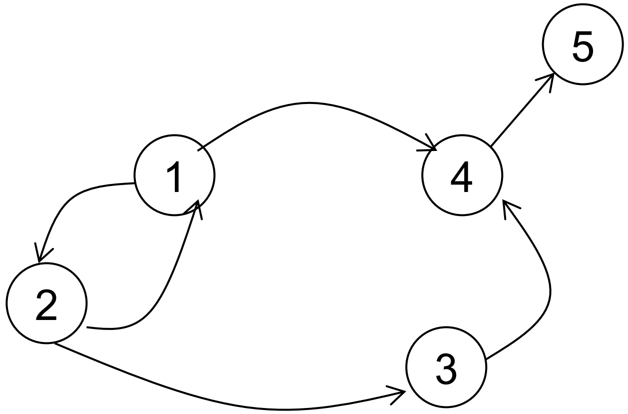
Discussion

- SQL uses the ***named*** perspective:
  - Each attribute has a name
  - The order of the attributes is irrelvant
  - **Person.address** or **Person.ssn** or **Person.age**
  - Need to know the names of an attribute

Discussion

- SQL uses the *named* perspective:
  - Each attribute has a name
  - The order of the attributes is irrelvant
  - **Person.address** or **Person.ssn** or **Person.age**
  - Need to know the names of an attribute
- Datalog uses the *unnamed* perspective:
  - The order of the attributes is important
  - Attributes do not have names
  - **Person(x,y,z,u,v)**: here x is ssn, y is address, etc
  - Need to know the position of an attribute

Recursion

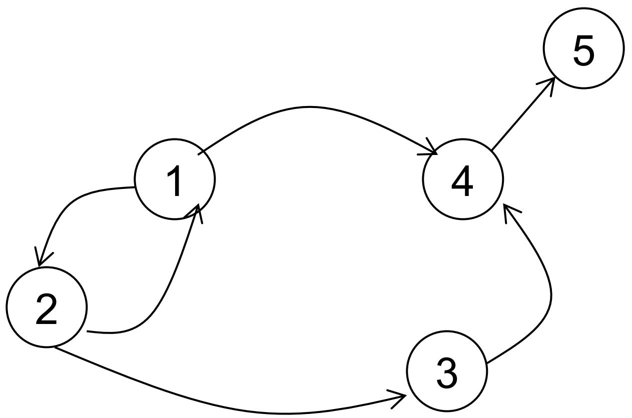


Descendants of node 2

R=

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

Recursion



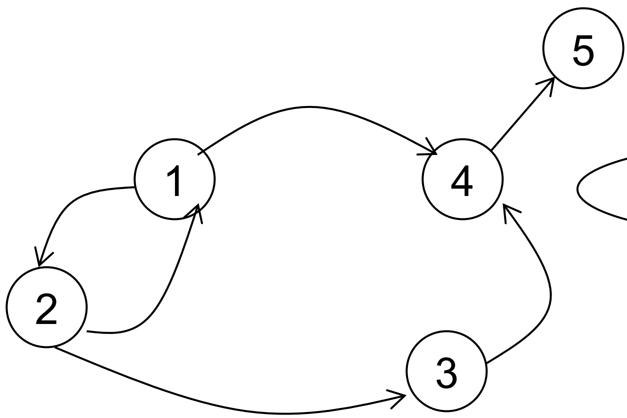
Descendants of node 2

R=

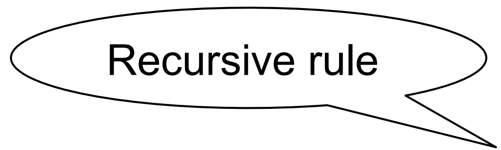
1	2
2	1
2	3
1	4
3	4
4	5

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2



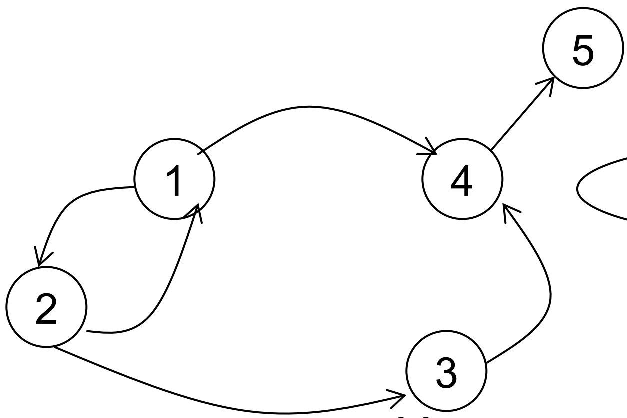
D(x) :- R(2, x)

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

R=

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

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

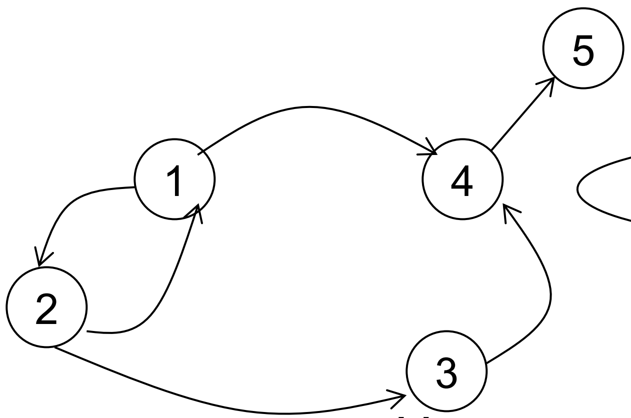
R=

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

How recursion works in datalog:

Initially D = empty

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

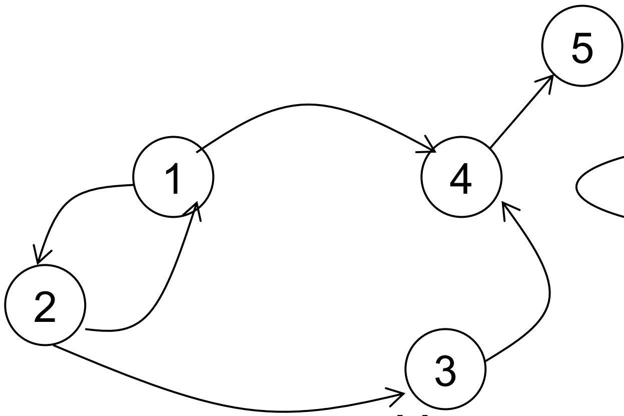
How recursion works in datalog:

Initially D = empty

- Compute both rules:

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

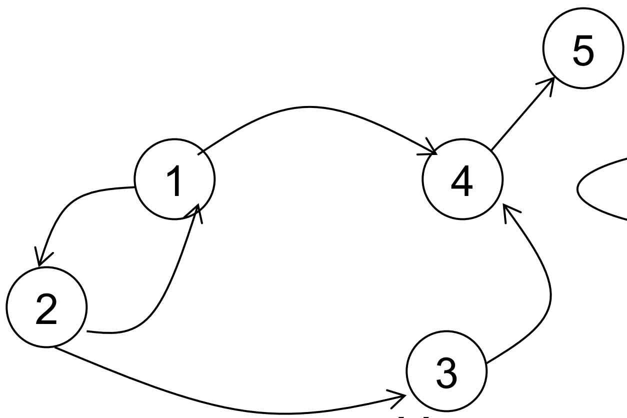
How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}

{1,3}
D(x) :- R(2, x)
{ }
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:

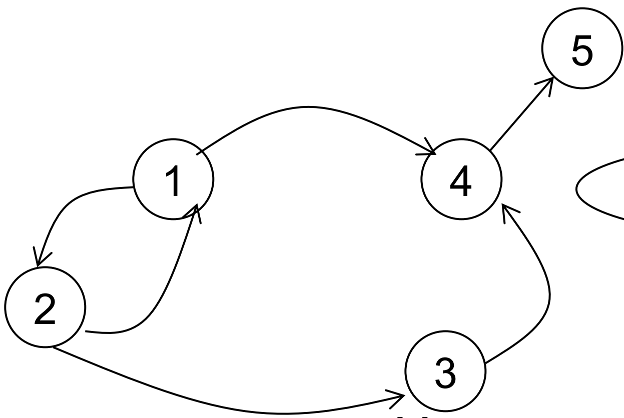
{1,3}

{ }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)

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

R=

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

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}

{1,3}

{ }

D(x) :- R(2, x)

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

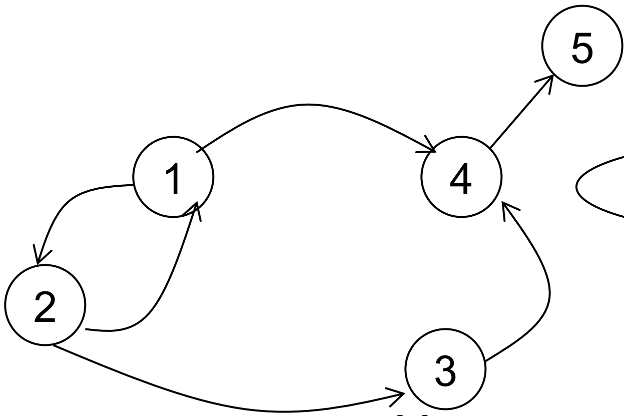
{1,3}

{2,4 }

D(x) :- R(2, x)

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

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

How recursion works in datalog:

Initially D = empty

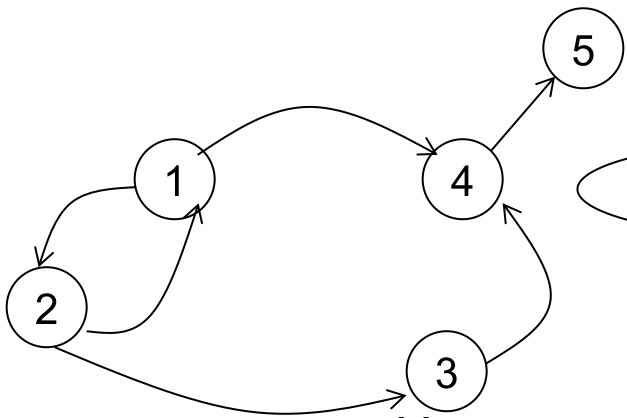
- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:

{1,3}
D(x) :- R(2, x)
{ }
D(y) :- D(x), R(x,y)

{1,3}
D(x) :- R(2, x)
{2,4 }
D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

How recursion works in datalog:

Initially D = empty

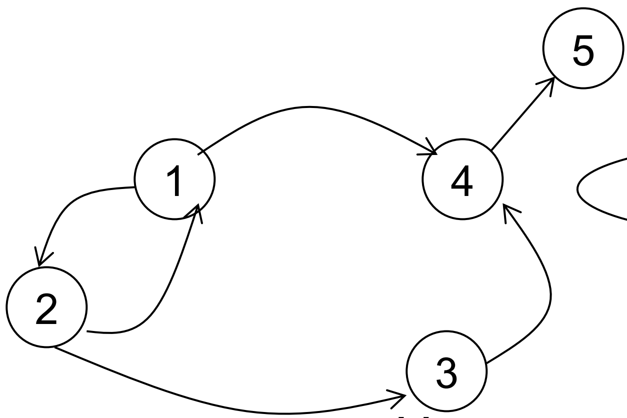
- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}

{1,3}
D(x) :- R(2, x)
{ }
D(y) :- D(x), R(x,y)

{1,3}
D(x) :- R(2, x)
{2,4 }
D(y) :- D(x), R(x,y)

{1,3}
D(x) :- R(2, x)
{2,4,1,3,5 }
D(y) :- D(x), R(x,y)

Recursion



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

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

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}
- Compute both rules:
  …nothing new. STOP

{1,3}
D(x) :- R(2, x)
{ }
D(y) :- D(x), R(x,y)

{1,3}
D(x) :- R(2, x)
{2,4 }
D(y) :- D(x), R(x,y)

{1,3}
D(x) :- R(2, x)
{2,4,1,3,5 }
D(y) :- D(x), R(x,y)

Discussion

- Datalog is designed for recursion
- Will prove that it always terminates
- To prove that, first need to define the semantics: **Naïve Algorithm**

Agenda

- Definition
- Naïve Algorithm
- Termination
- Semi-naïve Algorithm

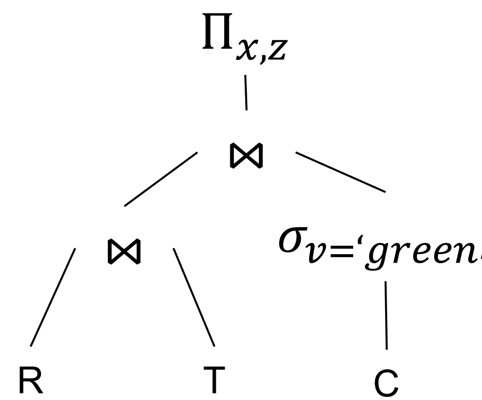
Naïve Evaluation Algorithm

- Convert all datalog rules into a set of Union-Select-Project-Join (USPJ)
- Called: *Immediate Consequence Operator*
- Naïve Evaluation Algorithm:
  Repeatedly apply the **ICO** until fixpoint

Immediate Consequence Operator

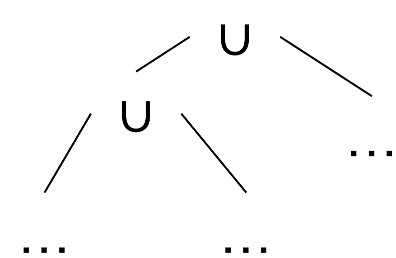
Every rule → SPJ query

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



Multiple rules same IDB head → USPJ

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



The **ICO** consists of all USPJ's for all IDBs

Naïve Evaluation Algorithm

$\mathrm{IDB}_0 \ := \emptyset$
for $\mathrm{t} = 1, \ldots, \infty$ **do**
  $\mathrm{IDB}_\mathrm{t} \ := \mathbf{ICO}(\mathrm{IDB}_{\mathrm{t}-1})$
  if $\mathrm{IDB}_\mathrm{t} = \mathrm{IDB}_{\mathrm{t}-1}$ **then** break

Naïve Evaluation Algorithm

```
D(x) :- R(2,x)
D(y) :- D(x),R(x,y)
```

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R))$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)

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

$D := \emptyset$;

repeat

$$D := \Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

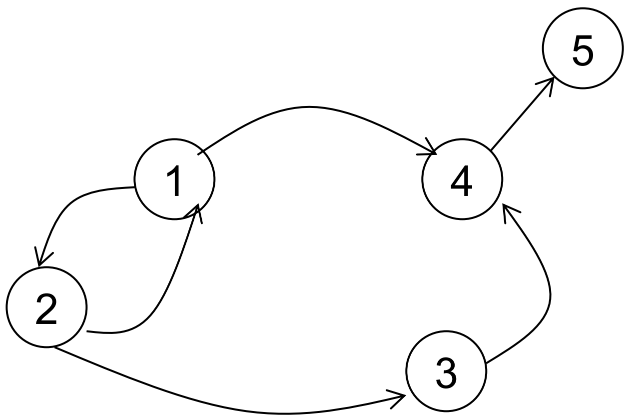
until [no more change]

Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Example



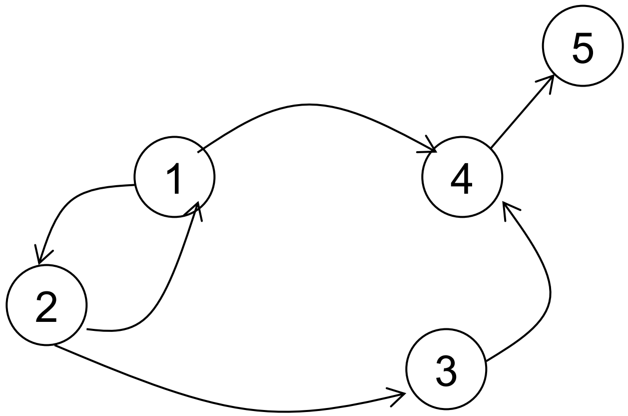
R=

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?

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

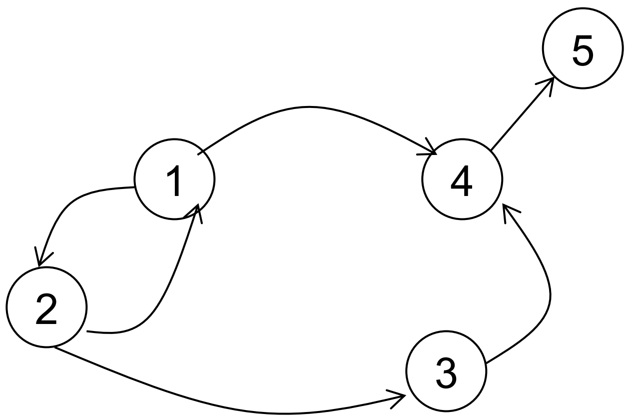
Initially:
T is empty.

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:

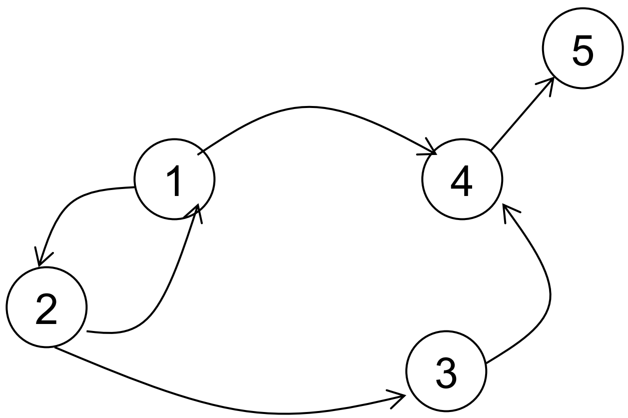
T =

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



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

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 =

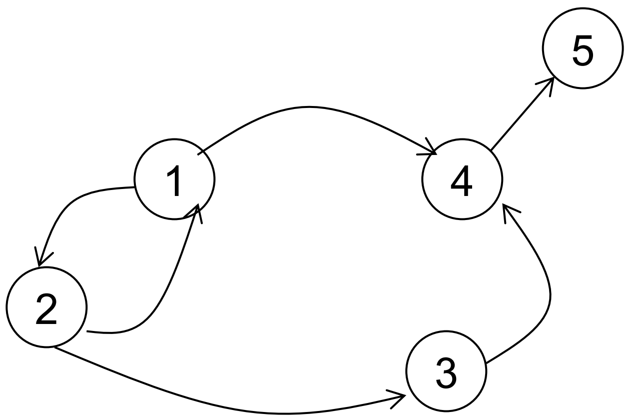
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

First rule generates this

Second rule generates this

New facts

Example



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

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

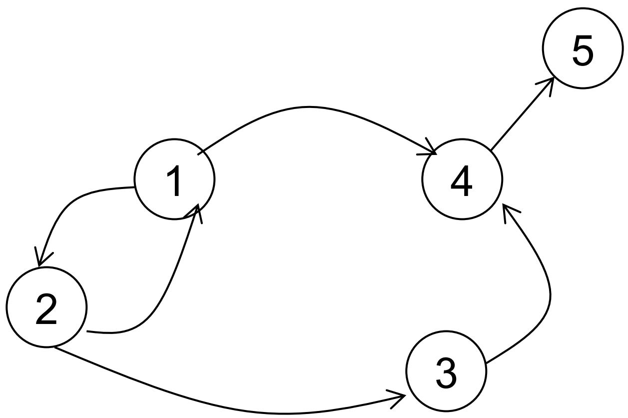
Second iteration:
T =

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	Both rules
2	1	
2	3	First rule
1	4	
3	4	
4	5	
1	1	
2	2	Second rule
1	3	
2	4	
1	5	
3	5	
2	5	New fact

Example



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

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

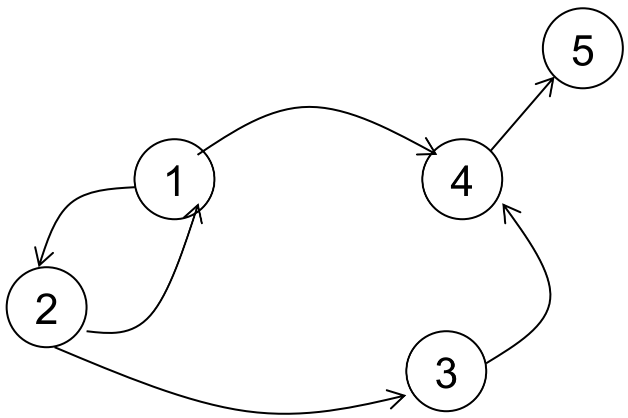
Third iteration:
T =

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

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 =

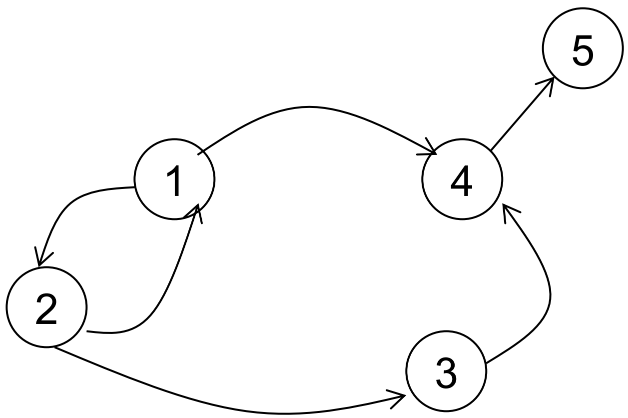
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
[illegible]	5

Fourth iteration
T =
(same)

No new facts.
DONE

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

Three Equivalent Programs



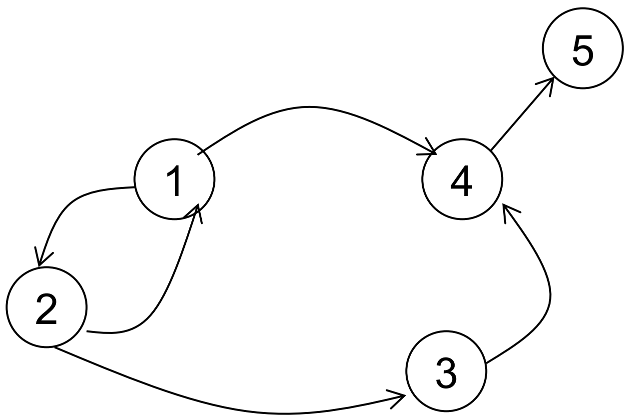
R=

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

Three Equivalent Programs



R=

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

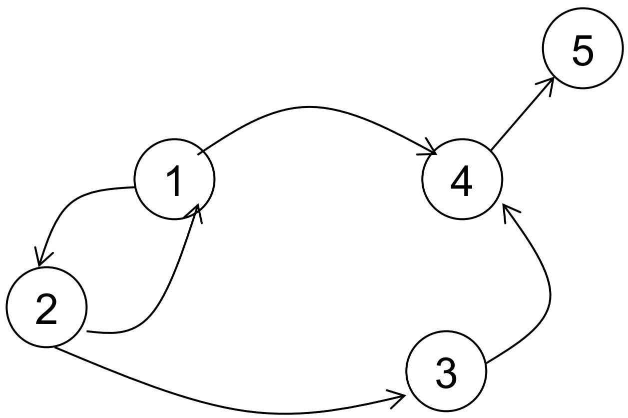
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

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

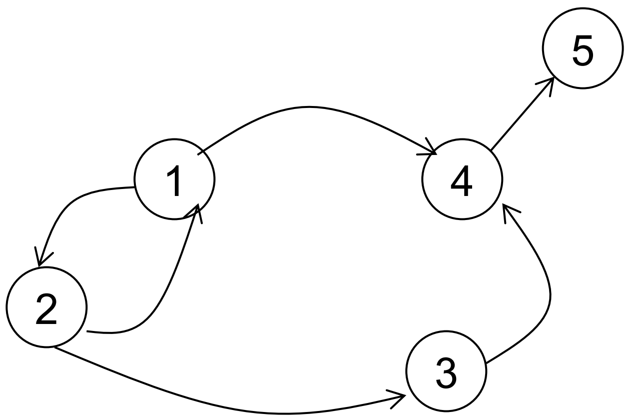
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)

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)

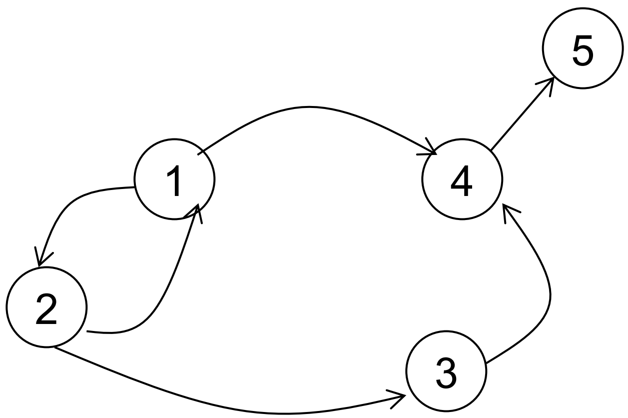
Left linear

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

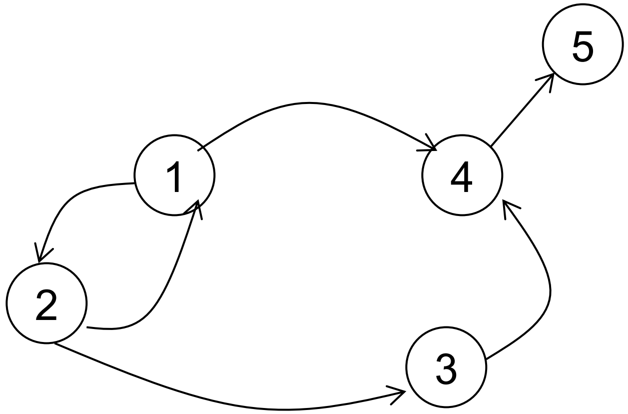
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

Non-linear

#iterations = log(diameter)

Question: how many iterations does each require?

Multiple IDBs

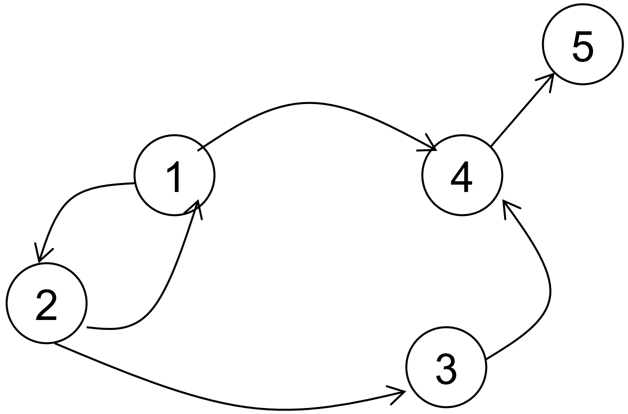


Find pairs of nodes (x,y) connected by a path of *even* length

R=

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

Multiple IDBs



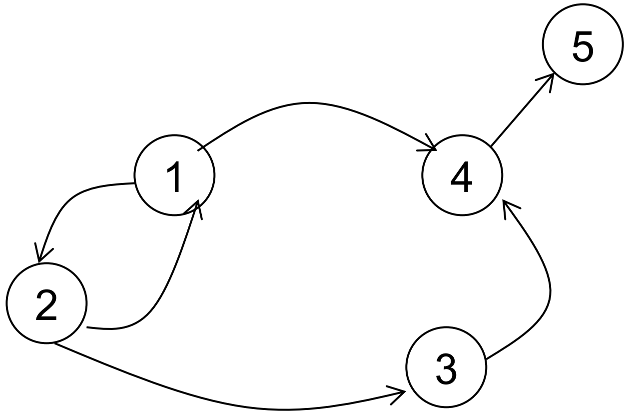
Find pairs of nodes (x,y) connected by a path of *even* length

R=

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

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

Multiple IDBs



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

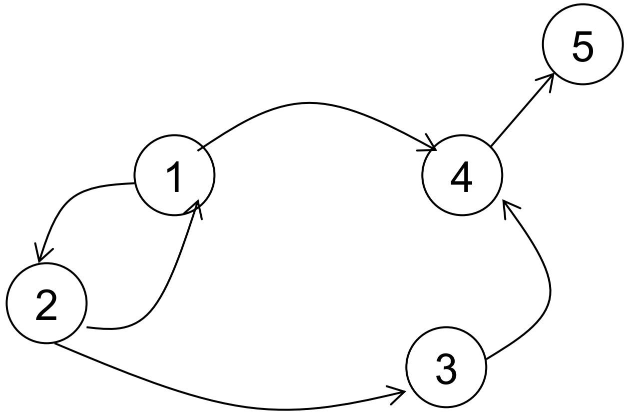
R=

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)

Multiple IDBs



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

R=

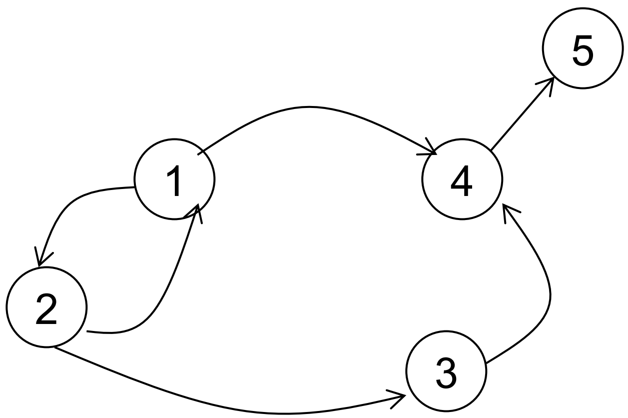
1	2
2	1
2	3
1	4
3	4
4	5

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

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

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

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)

We have two IDBs: Odd(x,y) and Even(x,y)

Naïve Evaluation Algorithm

When multiple IDBs:
need to compute their
new values *together*:

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

$$\left.\begin{array}{l} Even_{new} := \Pi_{x,y}(Odd \bowtie R); \\ Odd_{new} := R \cup \Pi_{x,y}(Even \bowtie R); \end{array}\right.$$

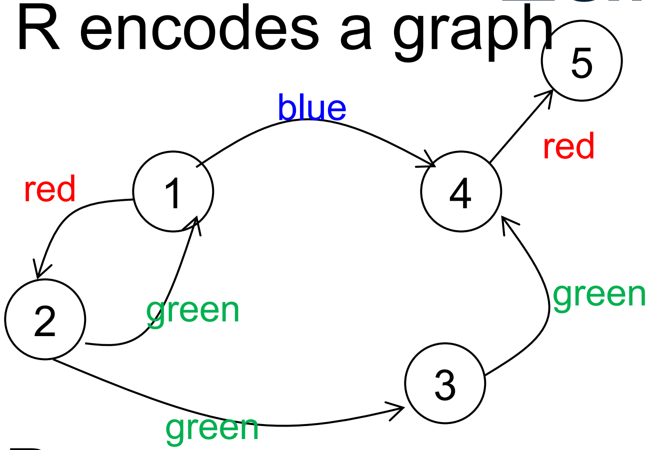
if $Odd = Odd_{new} \wedge Even = Even_{new}$
then break

$Odd := Odd_{new}$

$Even := Even_{new}$

Labeled Graphs

R encodes a graph



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

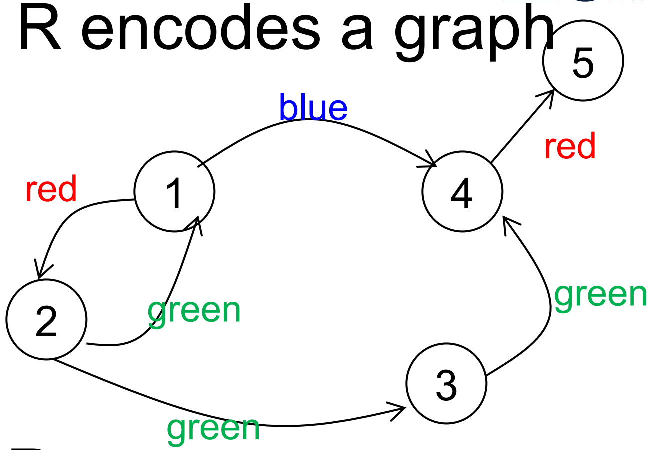
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

```
GreenP(x,y) :- R(x,y,'green')
GreenP(x,y) :- R(x,z,'green'),GreenP(z,y)
```

Labeled Graphs

R encodes a graph



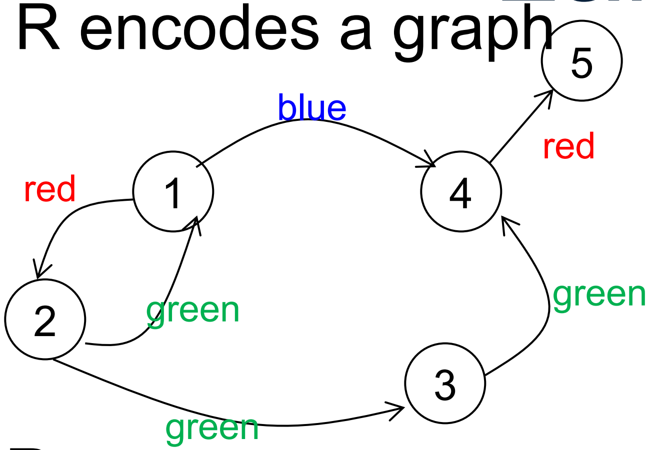
Find pairs of nodes (x,y) connected by a *monochromatic* path

R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Labeled Graphs

R encodes a graph



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

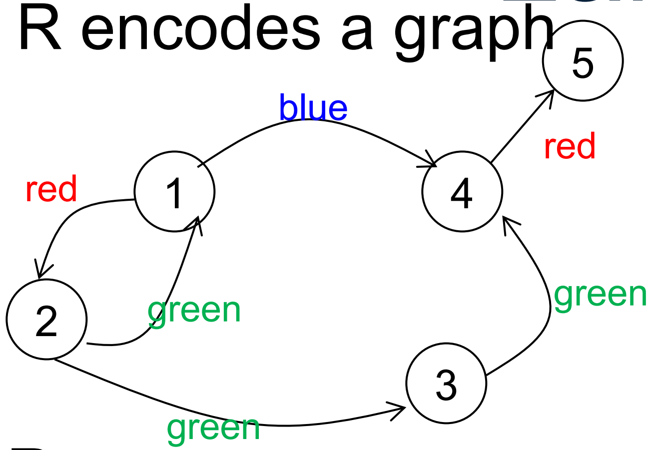
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

P(x,y,c) :- R(x,y,c)

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a <u>*monochromatic*</u> path

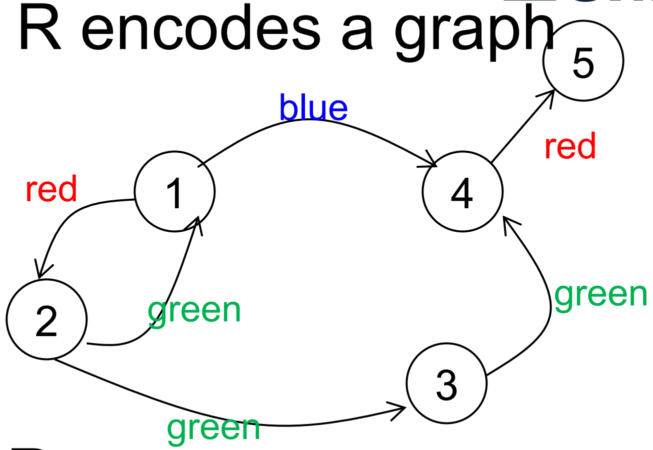
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)

Labeled Graphs

R encodes a graph



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

R=

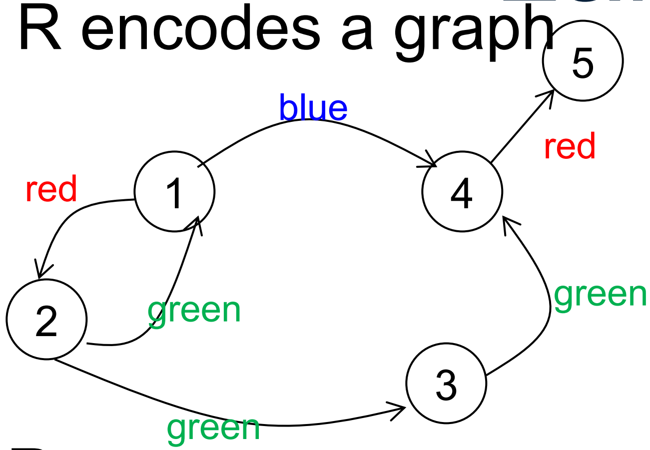
1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

We join on both the node z, *and* the color c

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)

Labeled Graphs

R encodes a graph



R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

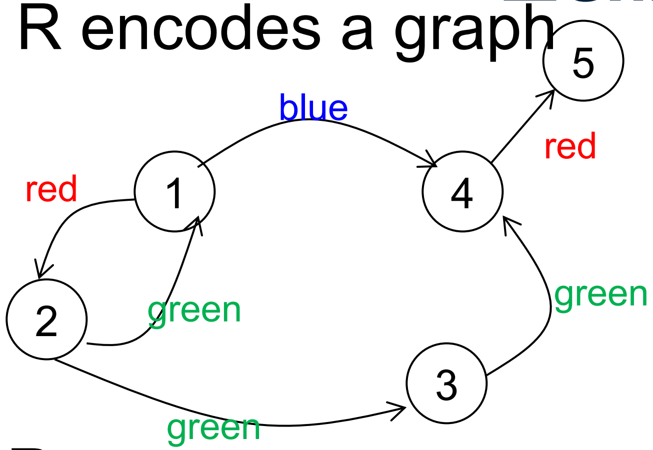
Find pairs of nodes (x,y) connected by a *monochromatic* path

We join on both the node z, *and* the color c

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)
Answer(x,y) :- P(x,y,c)

Labeled Graphs

R encodes a graph



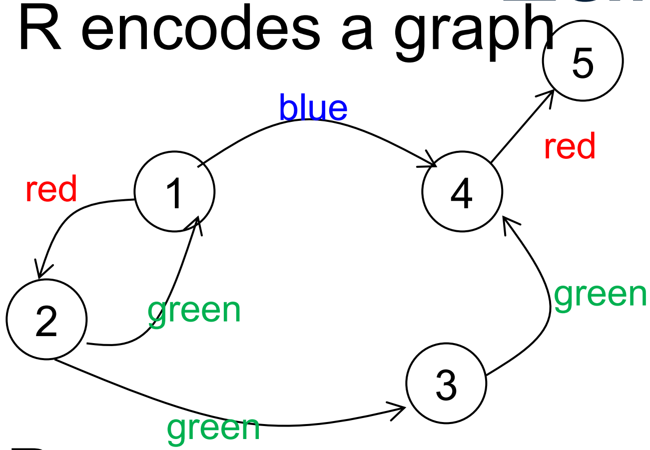
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Labeled Graphs

R encodes a graph

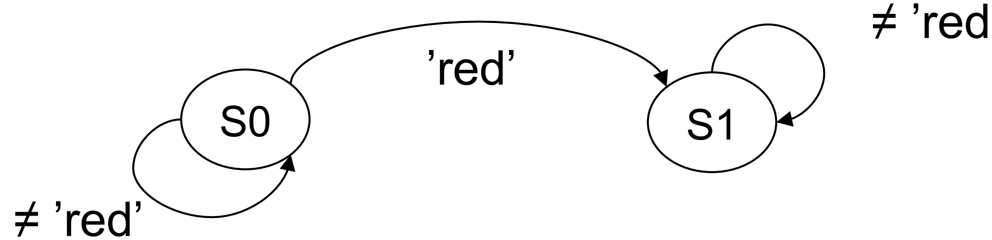


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

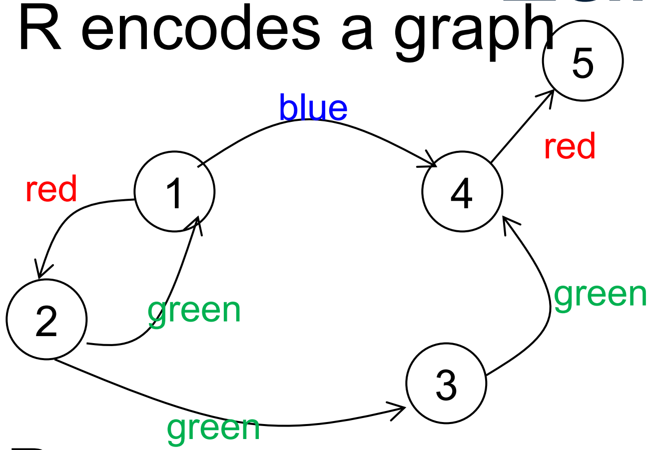
Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:



Labeled Graphs

R encodes a graph

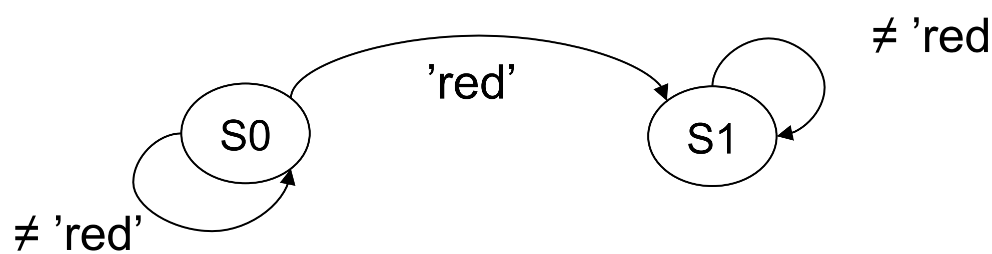


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

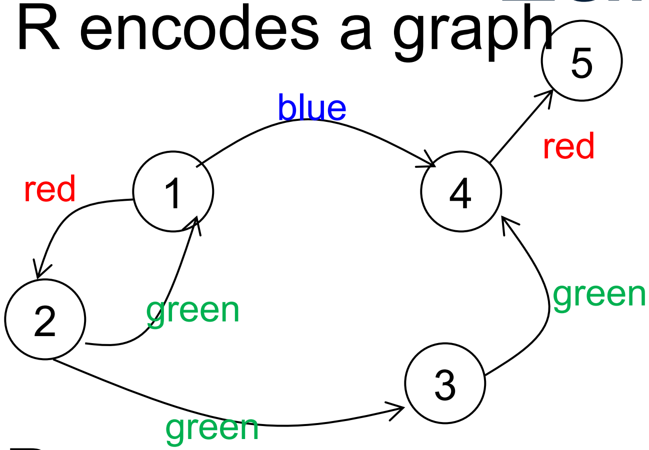
Automaton:



S0(2) :- .

Labeled Graphs

R encodes a graph

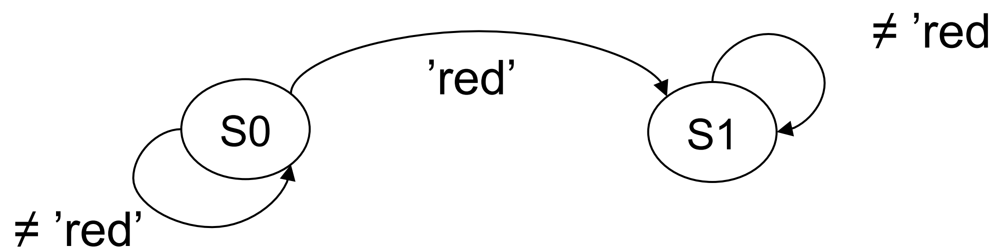


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

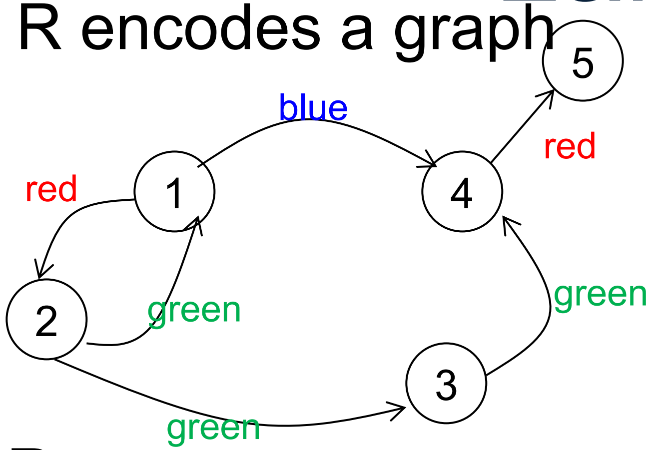
Automaton:




```
S0(2)    :- .
S0(y)    :- S0(x), R(x,y,c), c!='red'.
```

Labeled Graphs

R encodes a graph

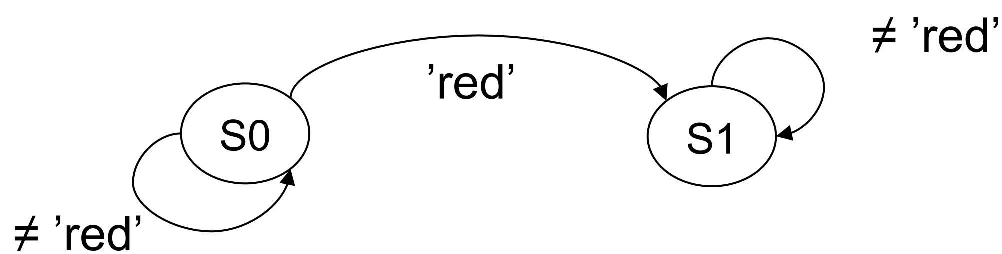


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

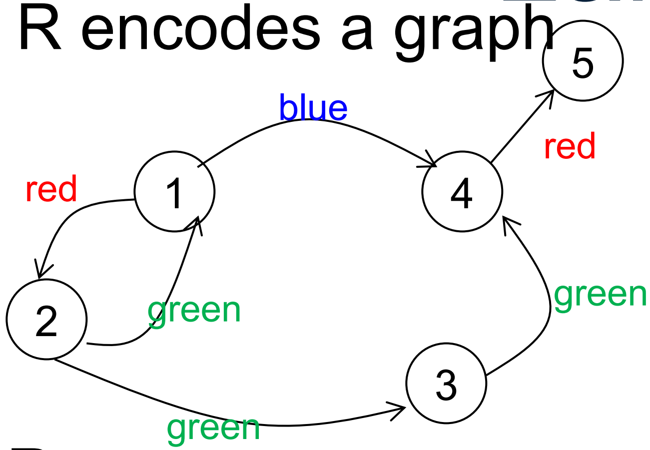
Automaton:




```
S0(2)   :- .
S0(y)   :- S0(x), R(x,y,c), c!='red'.
S1(y)   :- S0(x), R(x,y,'red').
```

Labeled Graphs

R encodes a graph

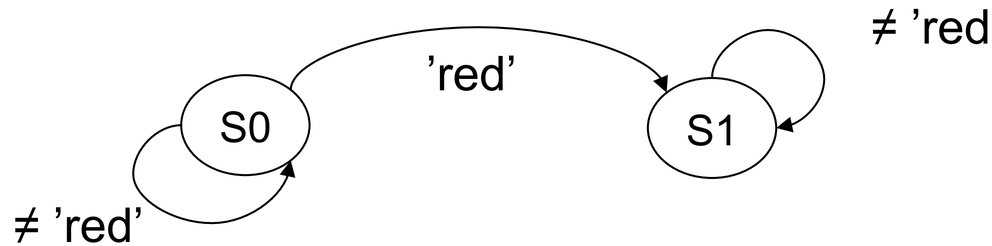


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:




```
S0(2)   :- .
S0(y)   :- S0(x), R(x,y,c), c!='red'.
S1(y)   :- S0(x), R(x,y,'red').
S1(y)   :- S1(x), R(x,y,c), c!='red'.
```

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc
- Linear query only
  - Cannot write T(x,y) :- T(x,z), T(z,y)

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc
- Linear query only
  - Cannot write T(x,y) :- T(x,z), T(z,y)
- Has bag semantics (really???)
  - May not terminate!

Discussion: Recursion in SQL

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

Discussion: Recursion in SQL

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

```sql
with recursive T as(
    select * from R
    union
    select distinct R.x, T.y
    from R, T
    where R.y=T.x
)
select * from T;
```

Discussion: Recursion in SQL

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

Relation T is called a
Common Table Expression CTE

```
with recursive T as(
    select * from R
    union
    select distinct R.x, T.y
    from R, T
    where R.y=T.x
)
select * from T;
```

Agenda

- Definition
- Naïve Algorithm
- Termination
- Semi-naïve Algorithm

Runtime of a Datalog Program

Theorem The Naïve Algorithm:

- Always terminates
- Terminates in a number of steps that is polynomial in the size of the database

Assumptions:

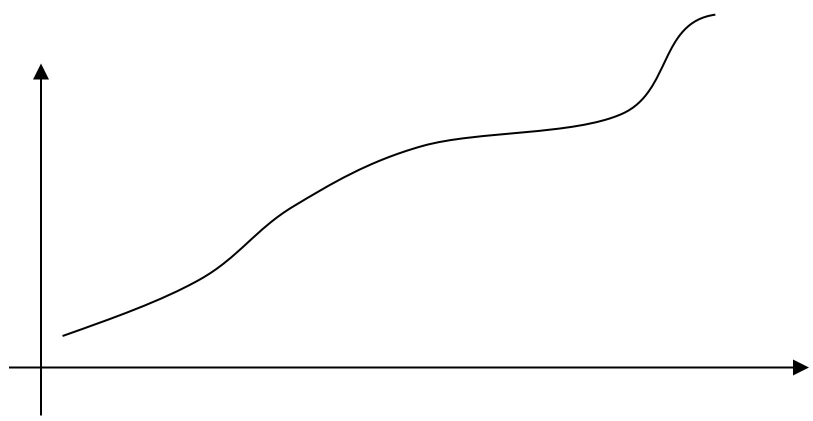
- Set semantics only
- Monotone rules only
- No “value invention”

This is cool!
Compare with
java, python, etc

Detour: Monotone Functions

- A function f(x) is called ***monotonically increasing***, or just ***monotone*** if:

$$\textbf{If} \quad x \leq y \quad \textbf{then} \quad f(x) \leq f(y)$$



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

Which Ops are Monotone?

- Selection: σ_{pred}
- Projection: $\Pi_{A,B,\ldots}$
- Join: $\bowtie$
- Union: $\cup$
- Difference: $-$
- Group-by-sum: $\gamma_{A,B,sum(C)}$

Which Ops are Monotone?

- Selection: σ_{pred} **MONOTONE**
- Projection: $\Pi_{A,B,\ldots}$ **MONOTONE**
- Join: $\bowtie$ **MONOTONE**
- Union: ∪ **MONOTONE**
- Difference: $-$ **NON-MONOTONE**
- Group-by-sum: $\gamma_{A,B,sum(C)}$

  NON-MONOTONE

Back to Datalog

Naïve Algorithm:

- Always terminates
- Terminates in a number of steps that is polynomial in the size of the database
- This is cool! Compare with java, python, etc

Assumptions:

- Set semantics only
- Monotone rules only
- No "value invention"

Will show this next

$\mathrm{IDB}_0 := \emptyset; \quad t := 0$
repeat $\mathrm{IDB}_{t+1} := \mathrm{USPJ}(\mathrm{IDB}_t); \ t := t + 1$
until no more change

Proof

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

$\mathrm{IDB}_0 := \emptyset; \quad \mathrm{t} := 0$
repeat $\mathrm{IDB}_{t+1} := \mathrm{USPJ}(\mathrm{IDB}_t); \ \mathrm{t} := \mathrm{t} + 1$
until no more change

Proof

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

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

Proof: by induction

$\mathrm{IDB}_0 := \emptyset; \quad \mathrm{t} := 0$
repeat $\mathrm{IDB}_{\mathrm{t}+1} := \mathrm{USPJ}(\mathrm{IDB}_{\mathrm{t}}); \ \mathrm{t} := \mathrm{t} + 1$
until no more change

Proof

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

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

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

$\mathrm{IDB}_0 := \emptyset; \quad \mathrm{t} := 0$
repeat $\mathrm{IDB}_{\mathrm{t}+1} := \mathrm{USPJ}(\mathrm{IDB}_{\mathrm{t}}); \ \mathrm{t} := \mathrm{t} + 1$
until no more change

Proof

Fact: every USPJ query is monotone
Proof: 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:

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

$\mathrm{IDB}_0 := \emptyset; \quad \mathrm{t} := 0$
repeat $\mathrm{IDB}_{\mathrm{t}+1} := \mathrm{USPJ}(\mathrm{IDB}_{\mathrm{t}}); \ \mathrm{t} := \mathrm{t} + 1$
until no more change

Proof

Fact: every USPJ query is monotone
Proof: 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:
$IDB_{t+1} = \mathrm{USPJ}(IDB_t) \subseteq \mathrm{USPJ}(IDB_{t+1}) = IDB_{t+2}$

Naïve Evaluation Algorithm

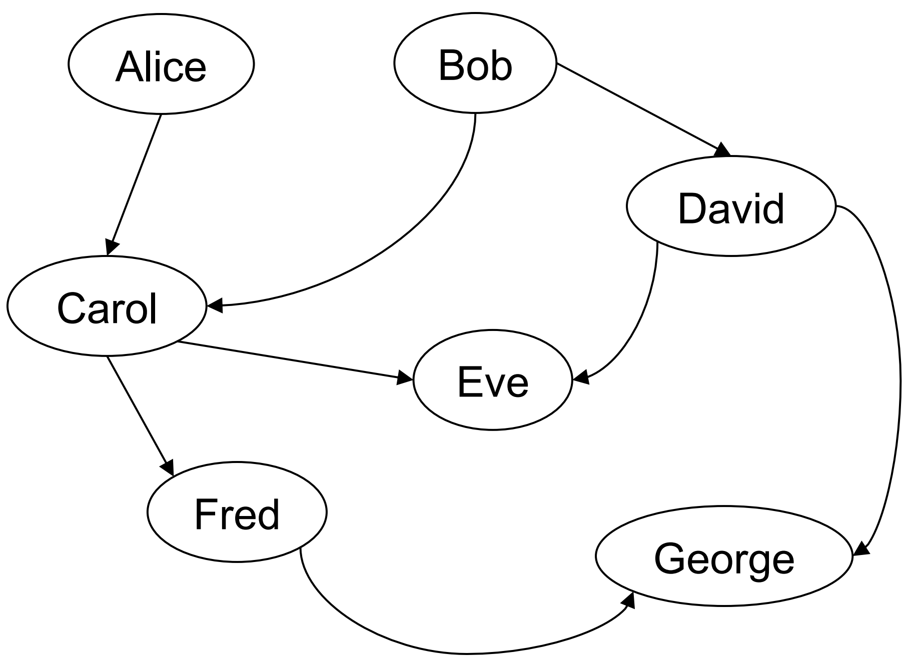
Consequence: The naïve algorithm *terminates*, in $O(n^k)$ steps, where:

- n = number of distinct values in the DB
- k = arity of widest IDB relation

Proof: IDBs increases to $\leq O(n^k)$ facts

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)
```

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
```

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)
```

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
```

Agenda

- Definition
- Naïve Algorithm
- Termination
- Semi-naïve Algorithm

Main Idea

- The naïve algorithm re-re-computes the same facts over and over again

- Main idea: compute only the new facts of the IDBs, from the previous new facts

- Need: Incremental View Maintenance

Incremental View Maintenance

- A *view* is a relation defined by a query

```
CREATE VIEW PathsLengthTwo AS
   SELECT e1.src as src, e2.dst as dst
   FROM Edge e1, Edge e2
   WHERE e1.dst = e2.src
```

- In datalog: every IDB is a *view*

```
PathsLengthTwo(x,y) :- Edge(x,z),Edge(z,y)
```

Incremental View Maintenance

- Suppose we have computed the view:

$$V := [\text{monotone query here}]$$

Incremental View Maintenance

- Suppose we have computed the view:

$$V := [\text{monotone query here}]$$

- One, or several of the base relations is updated: e.g. we insert new tuples:

$$R := R \cup \Delta R$$

Incremental View Maintenance

- Suppose we have computed the view:
$$V := [\text{monotone query here}]$$

- One, or several of the base relations is updated: e.g. we insert new tuples:
$$R := R \cup \Delta R$$

- Then the view needs to be incremented with some new tuples:
$$V := V \cup \Delta V$$

Incremental View Maintenance

- Suppose we have computed the view:

$$\mathrm{V} \coloneqq [\text{monotone query here}]$$

- One, or several of the base relations is updated: e.g. we insert new tuples:

$$\mathrm{R} \coloneqq \mathrm{R} \cup \Delta\mathrm{R}$$

- Then the view needs to be incremented with some new tuples:

$$\mathrm{V} \coloneqq \mathrm{V} \cup \Delta\mathrm{V}$$

- Problem: compute $\Delta\mathrm{V}$ without recomputing V

Incremental View Maintenance

Fix a USPJ query Q with inputs R_1, R_2, …:

$$\boxed{Q(R_1, R_2, \ldots)}$$

The delta query ΔQ is any query that has the property:

$$\boxed{Q(R_1 \cup \Delta R_1, R_2 \cup \Delta R_2, \ldots) = Q(R_1, R_2, \ldots) \cup \Delta Q(R_1, \Delta R_1, R_2, \Delta R_2, \ldots)}$$

Incremental View Maintenance

Example 1:

$\boxed{V(x,y) \text{ :- } R(x,z),S(z,y)}$

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

Incremental View Maintenance

Example 1:

$\boxed{V(x,y) \text{ :- } R(x,z),S(z,y)}$

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

$\boxed{\Delta V(x,y) \text{ :- } \Delta R(x,z),S(z,y)}$

Incremental View Maintenance

Example 2:

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

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

Incremental View Maintenance

Example 2:

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

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

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

Incremental View Maintenance

Example 3:

$\boxed{V(x,y) \text{ :- } T(x,z),T(z,y)}$

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

Incremental View Maintenance

Example 3:

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

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

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

Incremental View Maintenance

Fix a USPJ query Q with inputs R_1, R_2, …:

$$\boxed{Q(R_1, R_2, \ldots)}$$

The delta query ΔQ is any query that has the property:

$$\boxed{Q(R_1 \cup \Delta R_1, R_2 \cup \Delta R_2, \ldots) = Q(R_1, R_2, \ldots) \cup \Delta Q(R_1, \Delta R_1, R_2, \Delta R_2, \ldots)}$$

Semi-Naïve Evaluation Algorithm

IDB = SPJU(∅)

ΔIDB = IDB

Apply only
the non-recursive
rules

Loop

    ΔIDB := **Δ**SPJU(IDB, **Δ**IDB) -- IDB

    IDB := IDB ∪ **Δ**IDB

  if (**Δ**IDB= ∅) then break

End Loop

From Naïve to Semi-Naive

Naïve Algorithm:

$$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$$

```
IDB_0 := ∅
t := 0
Loop
    IDB_{t+1} := SPJU(IDB_t)
    if (IDB_{t+1} = IDB_t) then break
    t := t+1
End Loop
```

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

```
IDB_0 := ∅
t := 0
Loop
    IDB_{t+1} := SPJU(IDB_t)
    if (IDB_{t+1} = IDB_t) then break
    t := t+1
End Loop
```

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

```
IDB_0 := ∅
t := 0
Loop
    IDB_{t+1} := SPJU(IDB_t)
    if (IDB_{t+1} = IDB_t) then break
    t := t+1
End Loop
```

```
IDB_0 := ∅;   ΔIDB_0 := ∅;
t := 0
Loop
    ΔIDB_{t+1} := SPJU(IDB_t) -- IDB_t
    IDB_{t+1} := IDB_t ∪ ΔIDB_{t+1}
    if (ΔIDB_{t+1} = ∅) then break
    t := t+1
End Loop
```

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

IDB$_0$:= ∅
t := 0
Loop
    IDB$_{t+1}$:= SPJU(IDB$_t$)
    if (IDB$_{t+1}$ = IDB$_t$) then break
    t := t+1
End Loop

IDB$_0$:= ∅; **Δ**IDB$_0$:= ∅;
t := 0
Loop
    ΔIDB$_{t+1}$:= SPJU(IDB$_t$) -- IDB$_t$
    IDB$_{t+1}$:= IDB$_t$ ∪ **Δ**IDB$_{t+1}$
    if (**Δ**IDB$_{t+1}$ = ∅) then break
    t := t+1
End Loop

$\mathbf{SPJU(IDB_{t-1})}$

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

IDB$_0$:= ∅
t := 0
Loop
    IDB$_{t+1}$:= SPJU(IDB$_t$)
    if (IDB$_{t+1}$ = IDB$_t$) then break
    t := t+1
End Loop

IDB$_0$:= ∅;   **Δ**IDB$_0$:= ∅;
t := 0
Loop
    ΔIDB$_{t+1}$:= SPJU(IDB$_t$) -- IDB$_t$
    IDB$_{t+1}$:= IDB$_t$ ∪ **Δ**IDB$_{t+1}$
    if (**Δ**IDB$_{t+1}$ = ∅) then break
    t := t+1
End Loop

$\mathbf{SPJU(IDB_{t-1} \cup \Delta IDB_t)}$

$\mathbf{SPJU(IDB_{t-1})}$

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

$\mathrm{IDB}_0 := \emptyset$; **Δ**$\mathrm{IDB}_0 := \emptyset$;

t := 0

Loop

$\Delta\mathrm{IDB}_{t+1} := \mathrm{SPJU}(\mathrm{IDB}_t)$ -- IDB_t

$\mathrm{IDB}_{t+1} := \mathrm{IDB}_t \cup \Delta\mathrm{IDB}_{t+1}$

if ($\Delta\mathrm{IDB}_{t+1} = \emptyset$) then break

t := t+1

End Loop

$\mathbf{SPJU(IDB_{t-1} \cup \Delta IDB_t)}$

$\mathbf{\Delta SPJU(IDB_{t-1}, \Delta IDB_t)}$

$\mathbf{SPJU(IDB_{t-1})}$

From Naïve to Semi-Naive

Naïve Algorithm:

$\mathrm{IDB}_0 \subseteq \mathrm{IDB}_1 \subseteq \mathrm{IDB}_2 \subseteq \cdots$

$\Delta IDB_t = IDB_t - IDB_{t-1}$

IDB_0 := ∅; **Δ**IDB_0:= ∅;
t := 0
Loop
    ΔIDB_{t+1} := SPJU(IDB_t) -- IDB_t
    IDB_{t+1} := IDB_t ∪ **Δ**IDB_{t+1}
    if (**Δ**IDB_{t+1} = ∅) then break
    t := t+1
End Loop

$\mathbf{SPJU(IDB_{t-1} \cup \Delta IDB_t)}$

$\mathbf{\Delta SPJU(IDB_{t-1}, \Delta IDB_t)}$

$\mathbf{SPJU(IDB_{t-1})}$

But need to start from t=1, since IDB_{-1} undefined

Semi-Naïve Algorithm

IDB_1 :=SPJU(∅); ΔIDB_1:=IDB_1;

t := 1

Loop

  $\Delta IDB_{t+1} := \Delta SPJU(IDB_{t-1}, \Delta IDB_t) -- IDB_t$

  $IDB_{t+1} := IDB_t \cup \Delta IDB_{t+1}$

  if ($\Delta IDB_{t+1} = \emptyset$) then break

  t := t+1

End Loop

Semi-Naïve Evaluation Algorithm

IDB = SPJU(∅)

ΔIDB = IDB

Apply only the non-recursive rules

Loop

    ΔIDB := **Δ**SPJU(IDB, **Δ**IDB) -- IDB

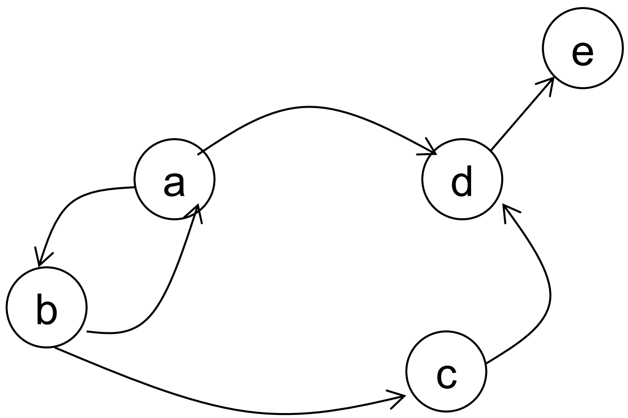
    IDB := IDB ∪ **Δ**IDB

  if (**Δ**IDB= ∅) then break

End Loop

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

Example: Linear Recursion



R=

a	b
b	a
b	c
a	d
c	d
d	e

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

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

Example

$\mathrm{T} \coloneqq \mathrm{R} \quad \Delta\mathrm{T} \coloneqq \mathrm{T}$

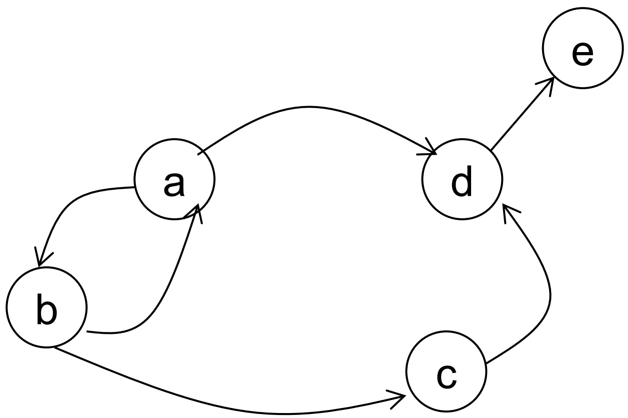
Loop

$\Delta\mathrm{T}(\mathrm{x},\mathrm{y}) \coloneqq \Pi_{x,y}(\mathrm{R}(\mathrm{x},\mathrm{z}) \bowtie \Delta\mathrm{T}(\mathrm{z},\mathrm{y})) - \mathrm{T}(\mathrm{x},\mathrm{y})$

$\mathrm{T} \coloneqq \mathrm{T} \cup \Delta\mathrm{T}$

if ($\Delta\mathrm{T} = \emptyset$) then break

End Loop



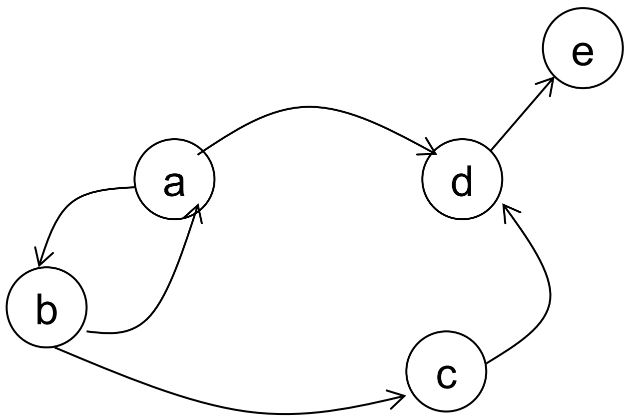
R=

a	b
b	a
b	c
a	d
c	d
d	e

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

$T := R \quad \Delta T := T$
Loop
$\Delta T(x, y) := \Pi_{x,y}(R(x, z) \bowtie \Delta T(z, y)) - T(x, y)$
$T := T \cup \Delta T$
if ($\Delta T = \emptyset$) then break
End Loop

Example



R=

a	b
b	a
b	c
a	d
c	d
d	e

Step 0:

ΔT

a	b
b	a
b	c
a	d
c	d
d	e

T

a	b
b	a
b	c
a	d
c	d
d	e

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

Example

$\mathrm{T} \coloneqq \mathrm{R} \quad \Delta\mathrm{T} \coloneqq \mathrm{T}$

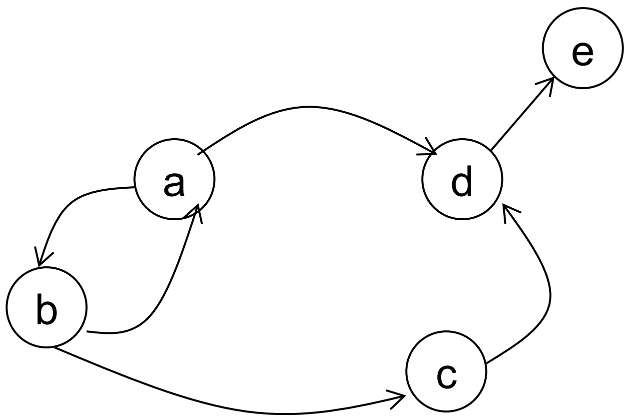
Loop

$$\Delta\mathrm{T}(x, y) \coloneqq \Pi_{x,y}(\mathrm{R}(x, z) \bowtie \Delta\mathrm{T}(z, y)) - \mathrm{T}(x, y)$$

$$\mathrm{T} \coloneqq \mathrm{T} \cup \Delta\mathrm{T}$$

if ($\Delta\mathrm{T} = \emptyset$) then break

End Loop



R=

a	b
b	a
b	c
a	d
c	d
d	e

Step 0:

ΔT

a	b
b	a
b	c
a	d
c	d
d	e

T

a	b
b	a
b	c
a	d
c	d
d	e

Step 1:

ΔT

a	a
b	b
a	c
b	d
a	e
c	e

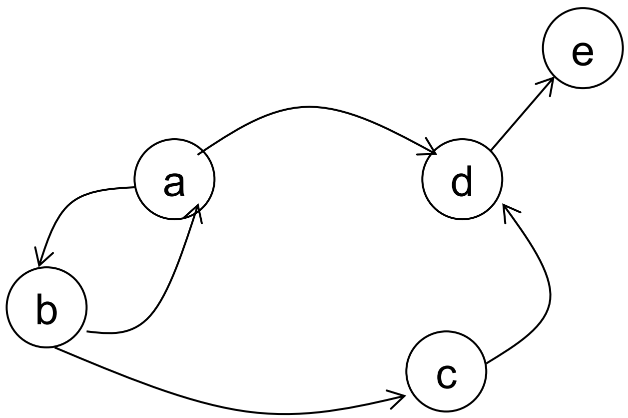
T

a	b
b	a
b	c
a	d
c	d
d	e
a	a
b	b
a	c
b	d
a	e
c	e

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

Example

$T := R \quad \Delta T := T$
Loop
$\quad \Delta T(x,y) := \Pi_{x,y}(R(x,z) \bowtie \Delta T(z,y)) - T(x,y)$
$\quad T := T \cup \Delta T$
$\quad$ if ($\Delta T = \emptyset$) then break
End Loop



R=

a	b
b	a
b	c
a	d
c	d
d	e

Step 0:

ΔT

a	b
b	a
b	c
a	d
c	d
d	e

T

a	b
b	a
b	c
a	d
c	d
d	e

Step 1:

ΔT

a	a
b	b
a	c
b	d
a	e
c	e

T

a	b
b	a
b	c
a	d
c	d
d	e
a	a
b	b
a	c
b	d
a	e
c	e

Step 2:

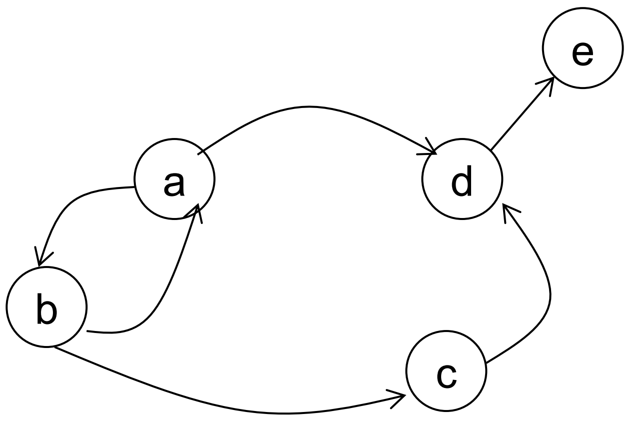
ΔT

d	e

T

a	b
b	a
b	c
a	d
c	d
d	e
a	a
b	b
a	c
b	d
a	e
c	e
d	e

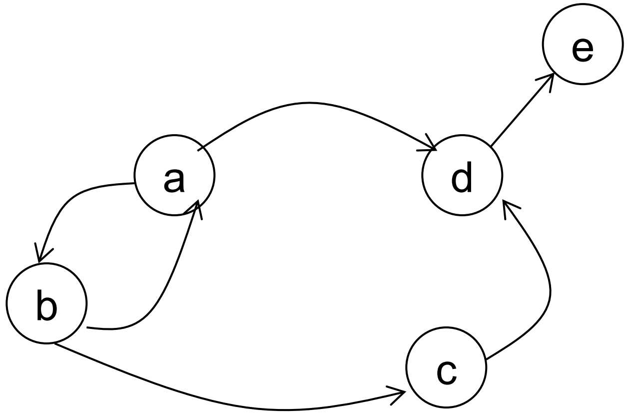
Example: Non-linear Recursion



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

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

Example: Non-linear Recursion



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

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

Semi-naïve algorithm:

$\mathrm{T} \coloneqq \mathrm{R} \quad \Delta\mathrm{T} \coloneqq \mathrm{T}$

Loop

$\Delta\mathrm{T}(\mathrm{x},\mathrm{y}) \coloneqq (\Pi_{\mathrm{x,y}}(\mathrm{T}(\mathrm{x},\mathrm{z}) \bowtie \Delta\mathrm{T}(\mathrm{z},\mathrm{y})) \cup \Pi_{\mathrm{x,y}}(\Delta\mathrm{T}(\mathrm{x},\mathrm{z}) \bowtie \mathrm{T}(\mathrm{z},\mathrm{y})) \cup \Pi_{\mathrm{x,y}}(\Delta\mathrm{T}(\mathrm{x},\mathrm{z}) \bowtie \Delta\mathrm{T}(\mathrm{z},\mathrm{y}))) - \mathrm{T}(\mathrm{x},\mathrm{y})$

$\mathrm{T} \coloneqq \mathrm{T} \cup \Delta\mathrm{T}$

if ($\Delta\mathrm{T} = \emptyset$) then break

End Loop

Example: Same Generation

```
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
SG(x,y) :- ParentChild(p,x), SG(p,q), ParentChild(q,y)
```

Example: Same Generation

SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
SG(x,y) :- ParentChild(p,x), SG(p,q), ParentChild(q,y)

$\mathrm{SG} \coloneqq \mathrm{ParentChild} \quad \Delta\mathrm{SG} \coloneqq \mathrm{SG}$

Loop

$\Delta\mathrm{SG}(\mathrm{x},\mathrm{y}) \coloneqq \Pi_{\mathrm{x},\mathrm{y}}\big(\mathrm{ParentChild}(\mathrm{p},\mathrm{x}) \bowtie \Delta\mathrm{SG}(\mathrm{p},\mathrm{q}) \bowtie \mathrm{ParentChild}(\mathrm{q},\mathrm{y})\big) - \mathrm{SG}(\mathrm{x},\mathrm{y})$

$\mathrm{SG} \coloneqq \mathrm{SG} \cup \Delta\mathrm{SG}$

if ($\Delta\mathrm{SG} = \emptyset$) then break

End Loop

Discussion

- Most datalog engines implement the semi-naïve algorithm

- Notice: it only works when the recursion has a monotone body

Summary

- Datalog = light-weight syntax, recursion
- Several optimizations
- Limitations:
  - Monotone queries work great
  - Non-monotone queries: stratification
- SQL: supports limited recursion only