

DATA516/CSED516
Scalable Data Systems and
Algorithms

Lecture 6

Datalog

Announcements

- HW2 => Discarded
- HW3 Part 6 is posted (git pull upstream)
- Project Milestone due on Nov. 25
- Project Feedback EOW

Case study: Snowflake

Snowflake

- It is an SaaS – what is this? Give other examples of types of cloud services...

Snowflake

- It is an SaaS – what is this? Give other examples of types of cloud services...
- SaaS = software as a service
- Other examples:
 - Platform as a service (PaaS): e.g. Amazon's EC
 - Infrastructure as a service (virtual machines)
 - Function as a Service: Amazon's Lambda

Snowflake

- Describe Snowflake's Data Storage

Snowflake

- Describe Snowflake's Data Storage

In class:

- S3:PUT/GET/DELETE
- Table → horizontal partition in files
- Blobs+PAX
- Temp storage → S3

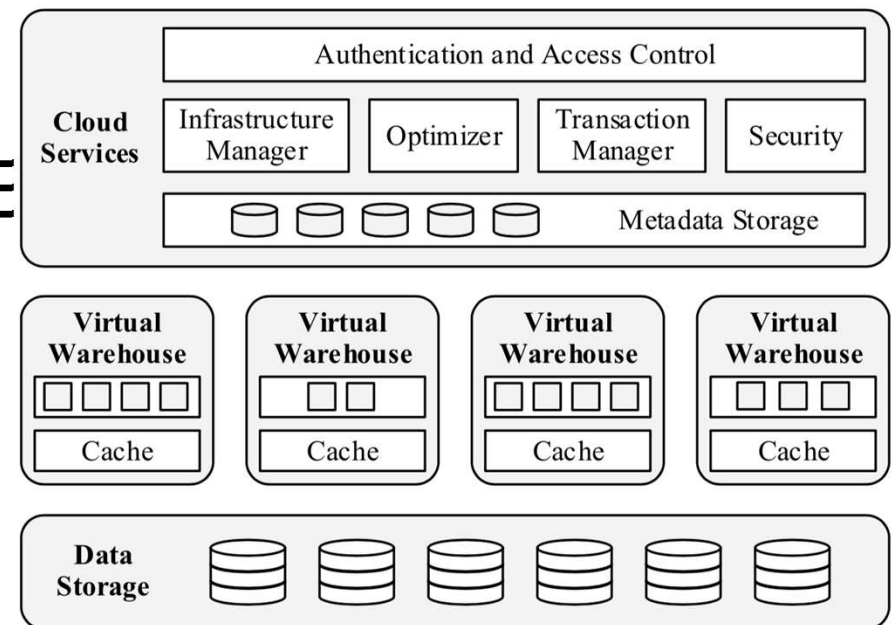


Figure 1: Multi-Cluster, Shared Data Architecture

Snowflake

- Describe Elasticity in Snowflake
- Describe failure handling in Snowflake

Snowflake

- Describe Elasticity in Snowflake
 - Virtual Warehouse (VW) serves one user
 - T-Shirt sizes: X-Small ... XX-Large
 - Small query may run on subset of VW
- Describe failure handling in Snowflake

Snowflake

- Describe Elasticity in Snowflake
 - Virtual Warehouse (VW) serves one user
 - T-Shirt sizes: X-Small ... XX-Large
 - Small query may run on subset of VW
- Describe failure handling in Snowflake
 - Restart the query
 - No partial retries (like MapReduce or Spark)

Snowflake

- Describe its execution engine

Snowflake

- Describe its execution engine
 - Column-oriented (in class)
 - Vectorized (“tuple batches” – in class)
 - Push-based (in class)

Snowflake

- What does Snowflake use instead of indexes?

Snowflake

- What does Snowflake use instead of indexes?
 - “Pruning”: for each file (recall: this is a horizontal partition of a table) and each attribute, it stores the min/max values in that column in that file; may skip files when not needed.

Outline

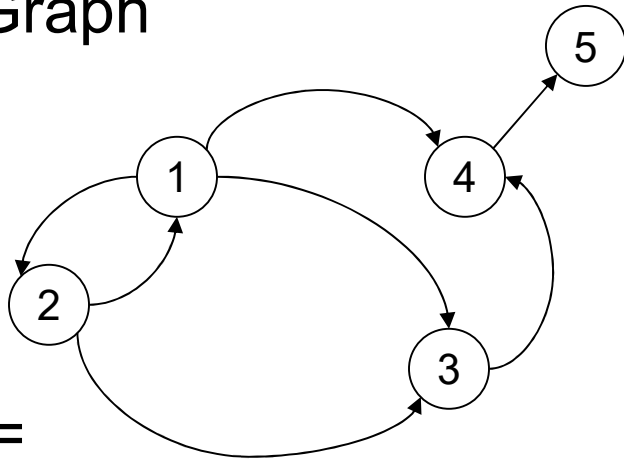
- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Datalog program

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

Processing Graphs in Datalog

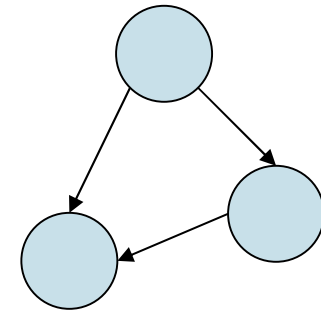
Graph



R=

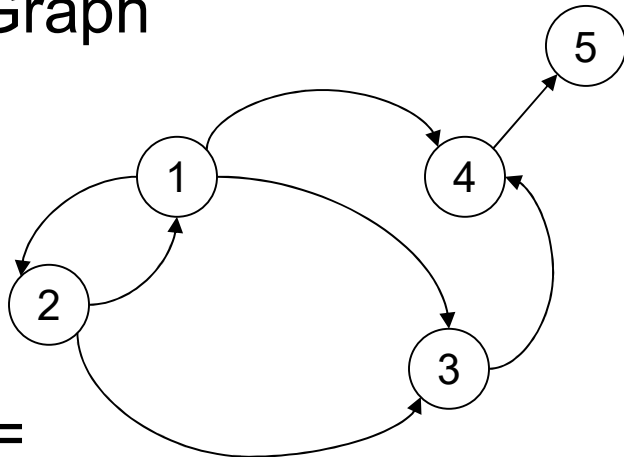
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Pattern Matching



Processing Graphs in Datalog

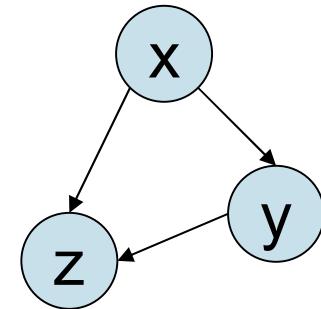
Graph



R=

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

Pattern Matching

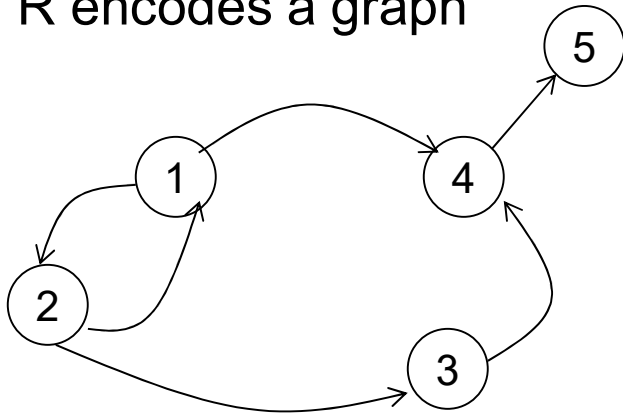


Answer(x,y,z) :- R(x,y), R(x,z), R(y,z)

Example

Descendants of node 2

R encodes a graph

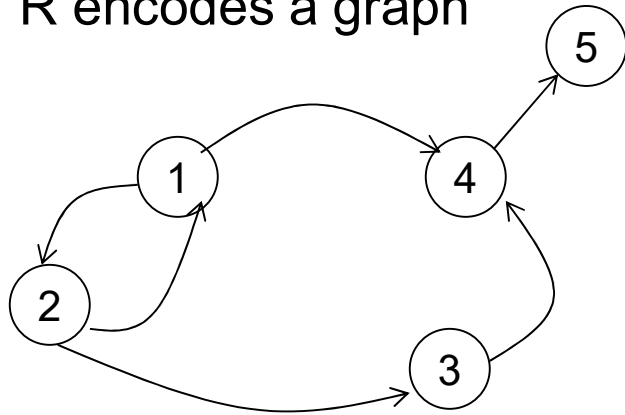


R=

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

Example

R encodes a graph



R=

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

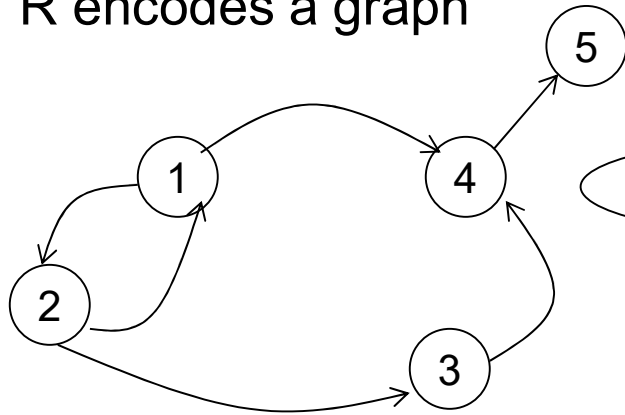
Descendants of node 2

$D(x) \text{ :- } R(2, x)$

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

Example

R encodes a graph



Descendants of node 2

Recursive rule

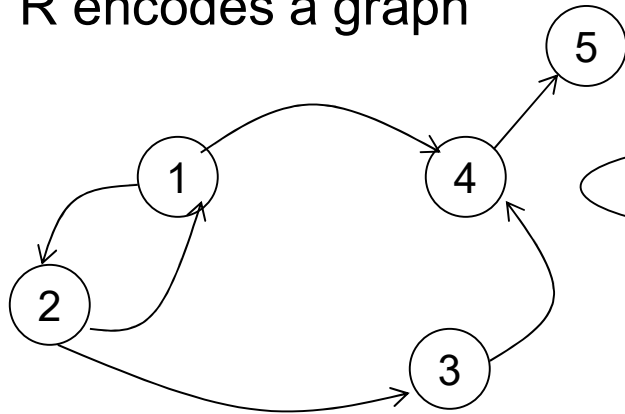
$$D(x) \text{ :- } R(2, x)$$
$$D(y) \text{ :- } D(x), R(x, y)$$

R=

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

Example

R encodes a graph



Descendants of node 2

Recursive rule

$D(x) \text{ :- } R(2, x)$

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

R=

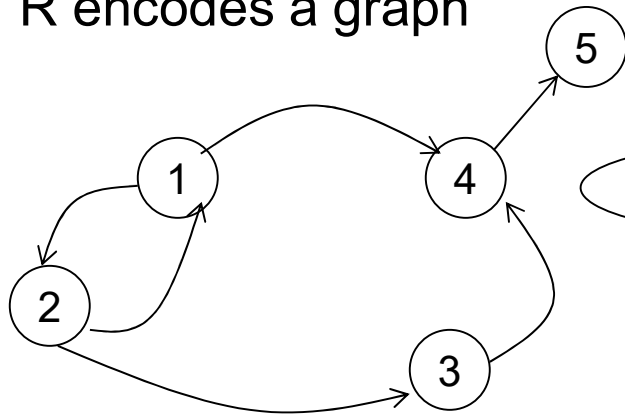
1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

Example

R encodes a graph



Descendants of node 2

Recursive rule

$D(x) \text{ :- } R(2, x)$
 $D(y) \text{ :- } D(x), R(x, y)$

R=

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

How recursion works in datalog:

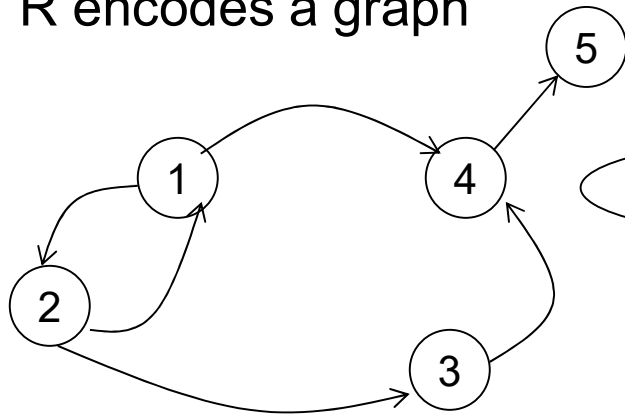
Initially D = empty

- Compute both rules:

$D(x) \text{ :- } R(2, x)$
 $D(y) \text{ :- } D(x), R(x, y)$

Example

R encodes a graph



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}

Descendants of node 2

Recursive rule

$D(x) \text{ :- } R(2, x)$

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

{1,3}

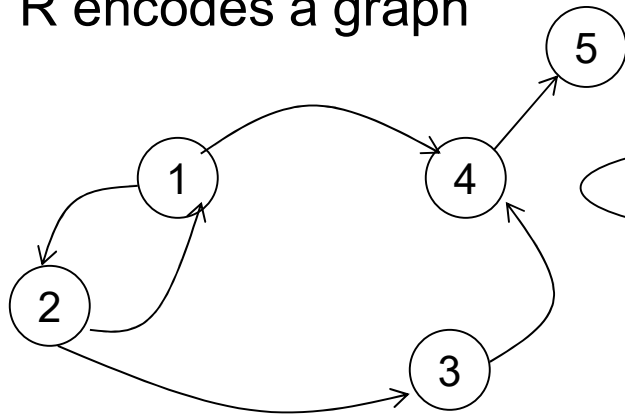
$D(x) \text{ :- } R(2, x)$

{}

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

Example

R encodes a graph



R=

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

Descendants of node 2

Recursive rule

$$D(x) \text{ :- } R(2, x)$$

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

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)

Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Naïve Evaluation Algorithm

- Every rule \rightarrow SPJ* query

*SPJ = select-project-join

+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule \rightarrow SPJ* query

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

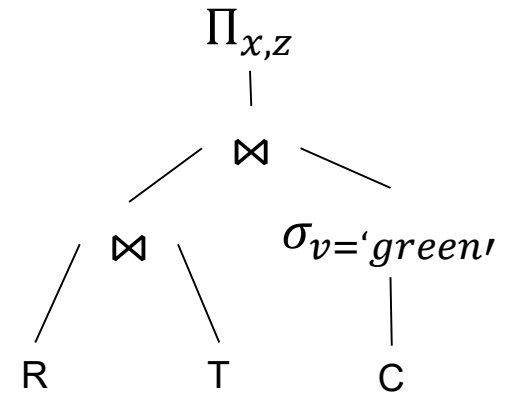
*SPJ = select-project-join

+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule \rightarrow SPJ* query

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



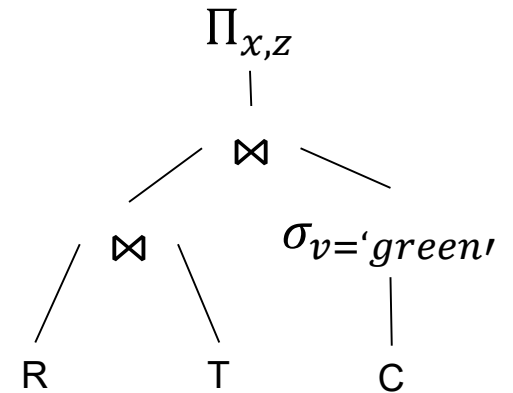
*SPJ = select-project-join

+USPJ = union-select-project-join

Naïve Evaluation Algorithm

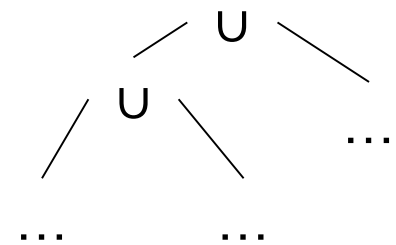
- Every rule \rightarrow SPJ* query

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



- Multiple rules same head \rightarrow USPJ+

$T(x,y) :- \dots$
 $T(x,y) :- \dots$
 \dots



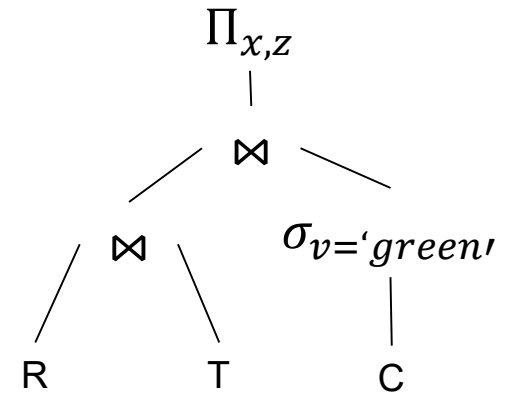
*SPJ = select-project-join

+USPJ = union-select-project-join

Naïve Evaluation Algorithm

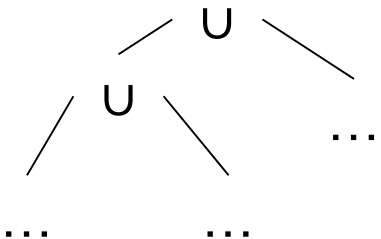
- Every rule \rightarrow SPJ* query

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



- Multiple rules same head \rightarrow USPJ+

$T(x,y) :- \dots$
 $T(x,y) :- \dots$
 \dots



- Naïve Algorithm:

$IDBs := \emptyset$
repeat $IDBs := USPJs$
until no more change

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

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]

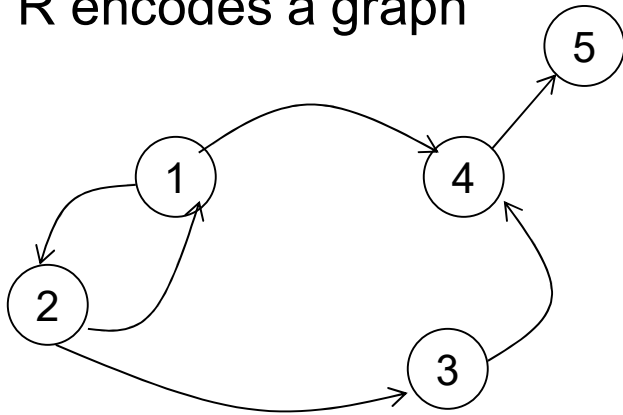
Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

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

Example

R encodes a graph



R=

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

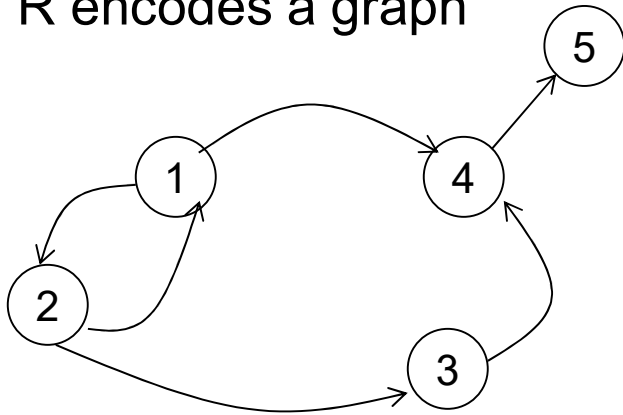
$T(x,y) :- R(x,y)$

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

What does it compute?

Example

R encodes a graph



R=

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

Initially:
T is empty.



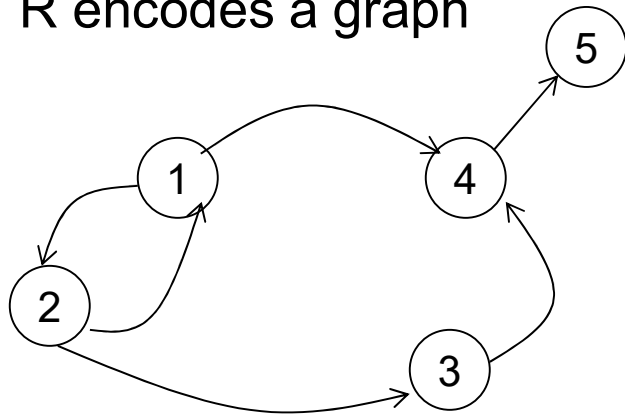
$T(x,y) :- R(x,y)$
 $T(x,y) :- R(x,z), T(z,y)$

What does
it compute?

Example

What does it compute?

R encodes a graph



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

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

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

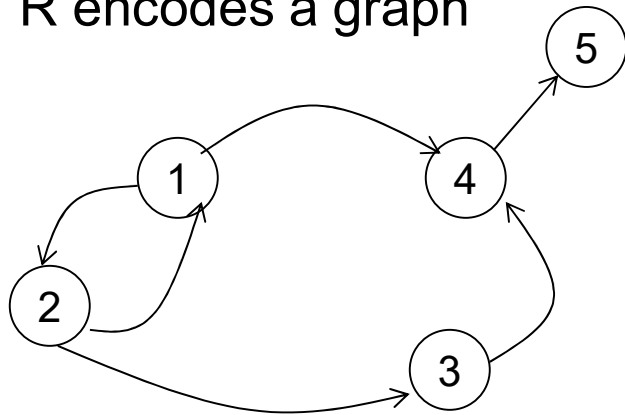
First rule generates this

Second rule
generates nothing
(because T is empty)

Example

What does it compute?

R encodes a graph

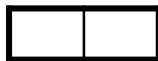


$T(x,y) :- R(x,y)$
 $T(x,y) :- R(x,z), T(z,y)$

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

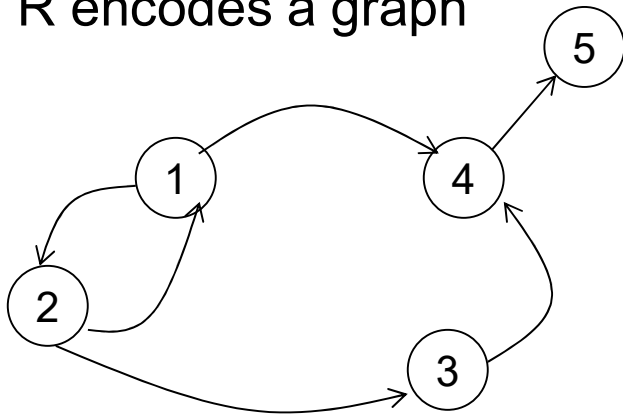
First rule generates this

Second rule generates this

New facts

Example

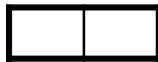
R encodes a graph



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 =

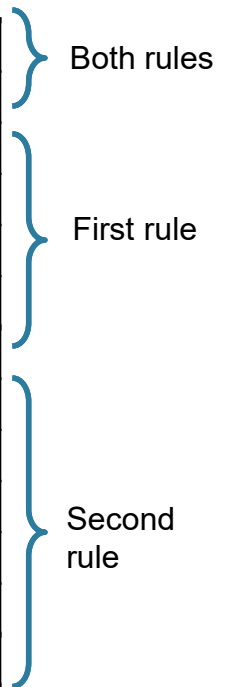
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

New fact

Third iteration:

T =

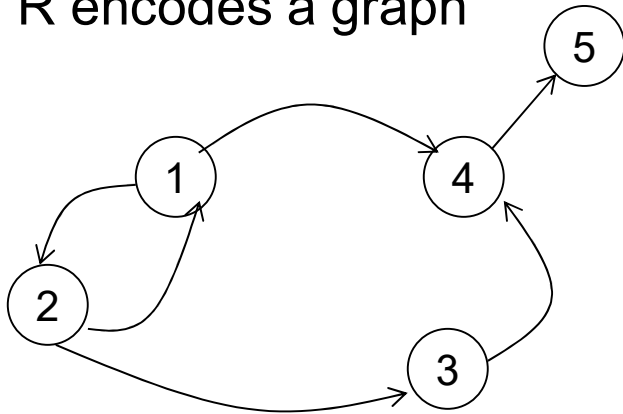
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5



What does it compute?

Example

R encodes a graph



R =

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

Initially:
T is empty.

--	--

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

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

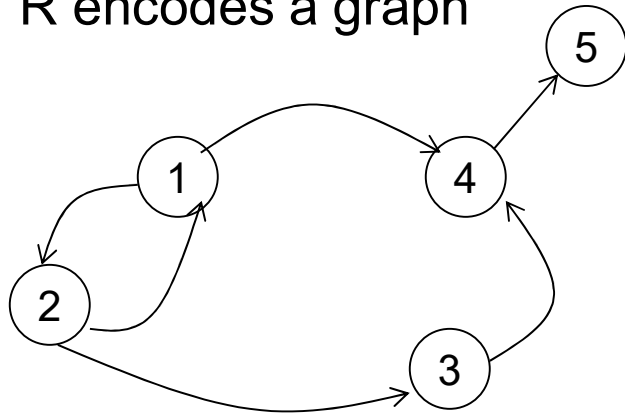
Fourth iteration
T =
(same)

No new facts.
DONE

What does it compute?

Example

R encodes a graph



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

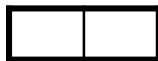
$$T(x,y) \text{ :- } R(x,z), T(z,y)$$

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

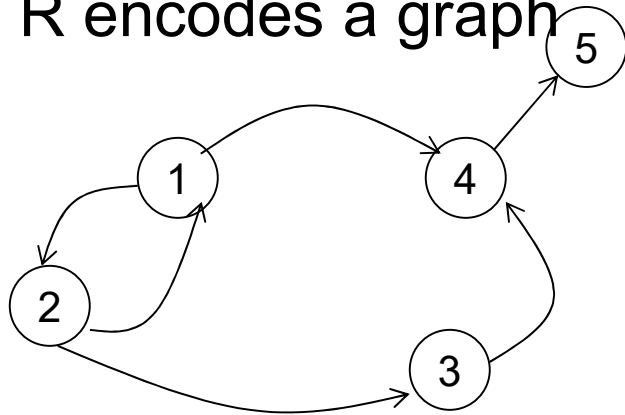
Fourth iteration
T =
(same)

No new facts.
DONE

Iteration k computes pairs (x,y) connected by path of length $\leq k$

Three Equivalent Programs

R encodes a graph



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

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

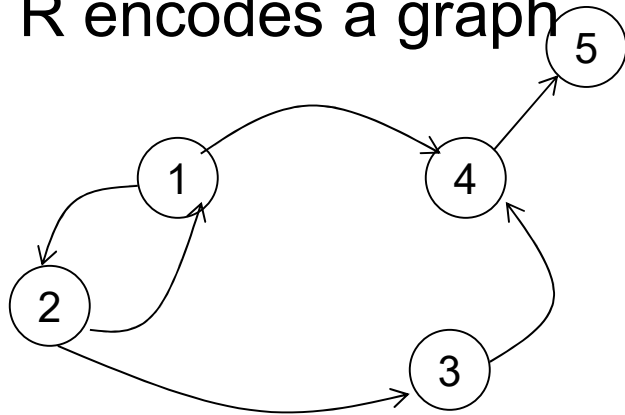
Right linear

R=

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

Three Equivalent Programs

R encodes a graph



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

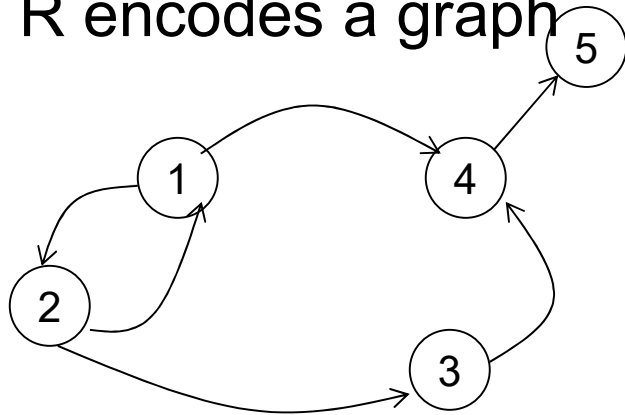
$T(x,y) :- R(x,y)$

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

Left linear

Three Equivalent Programs

R encodes a graph



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

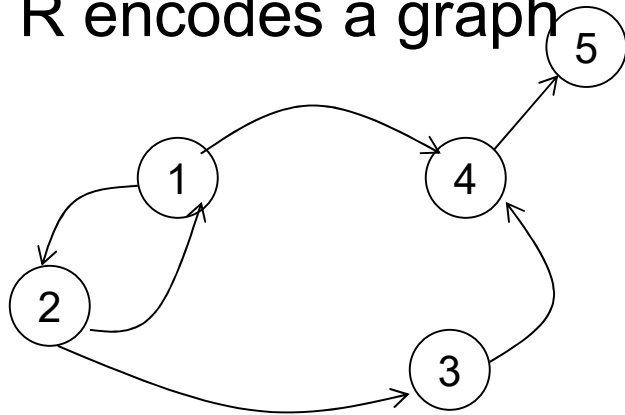
$T(x,y) :- R(x,y)$

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

Non-linear

Three Equivalent Programs

R encodes a graph



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

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

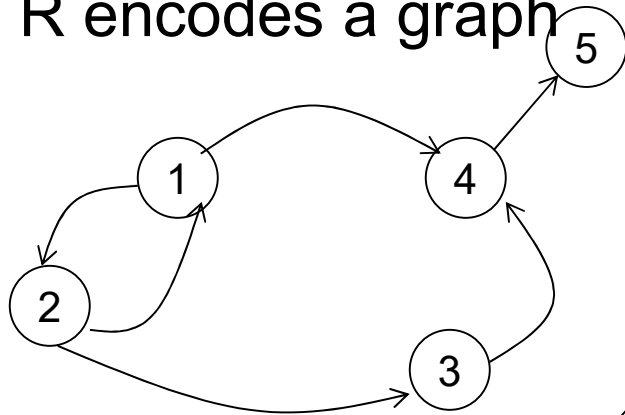
$$T(x,y) :- T(x,z), T(z,y)$$

Non-linear

Question: how many iterations does each require?

Three Equivalent Programs

R encodes a graph



R=

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

#iterations =
diameter

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

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

Right linear

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

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

Left linear

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

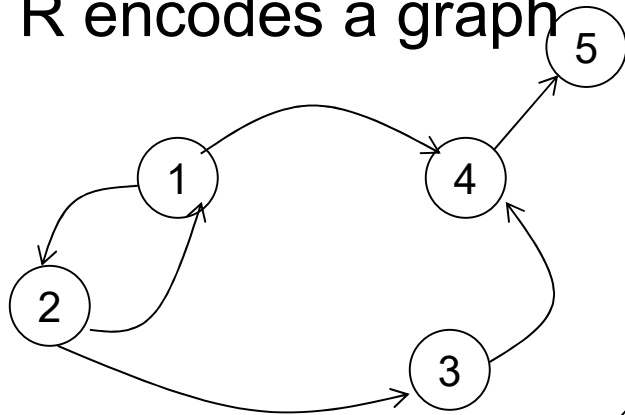
$T(x,y) :- T(x,z), T(z,y)$

Non-linear

Question: how many iterations does each require?

Three Equivalent Programs

R encodes a graph



R=

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

#iterations =
diameter

#iterations =
log(diameter)

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

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

Right linear

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

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

Left linear

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

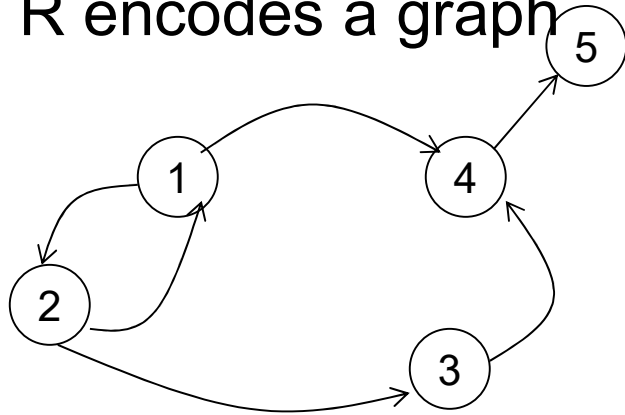
$T(x,y) :- T(x,z), T(z,y)$

Non-linear

Question: how many iterations does each require?

Multiple IDBs

R encodes a graph



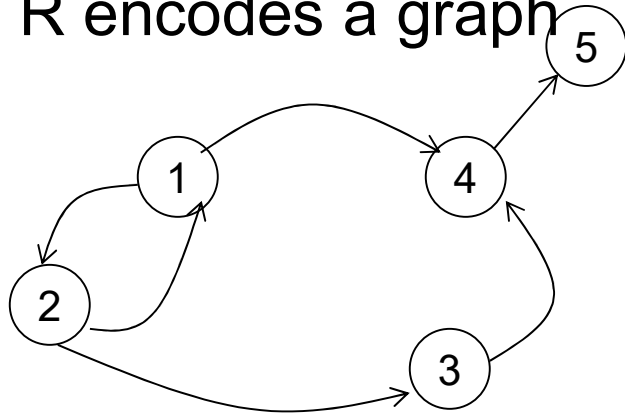
Find pairs of nodes (x,y) connected by a path of even length

R=

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

Multiple IDBs

R encodes a graph



R=

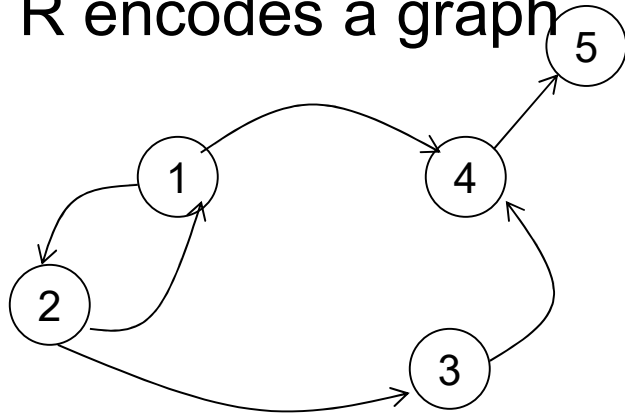
1	2
2	1
2	3
1	4
3	4
4	5

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

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

Multiple IDBs

R encodes a graph



R=

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

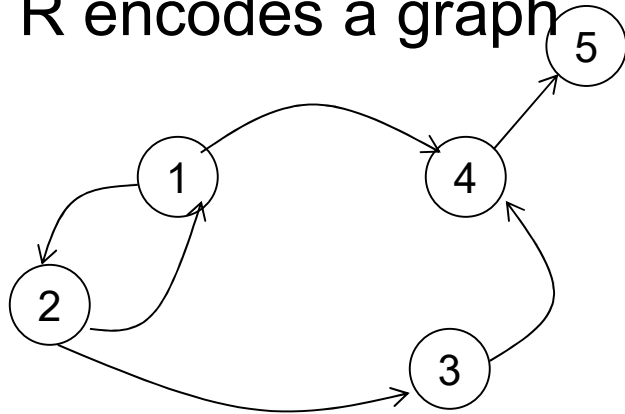
Find pairs of nodes (x,y)
connected by a path of even length

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

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

Multiple IDBs

R encodes a graph



R=

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

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

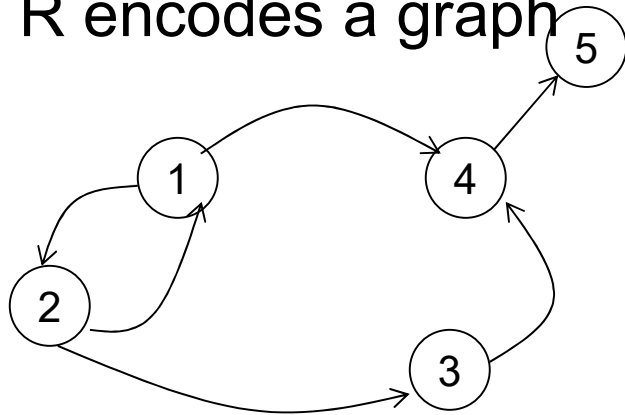
Odd(x,y) :- R(x,y)

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

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

Multiple IDBs

R encodes a graph



R=

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

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

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

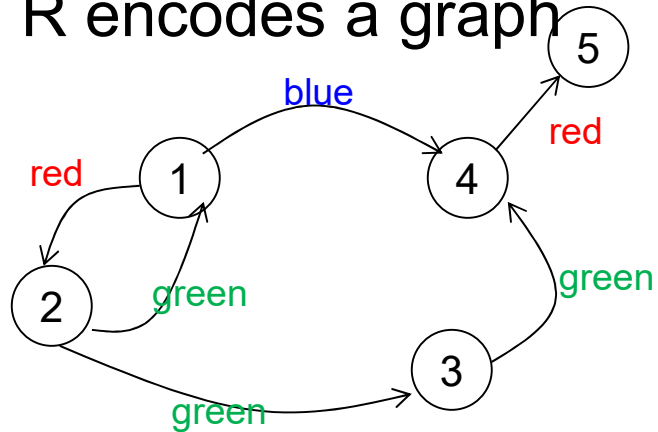
Even(x,y) :- Odd(x,z), R(z,y)

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

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

Labeled Graphs

R encodes a graph



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

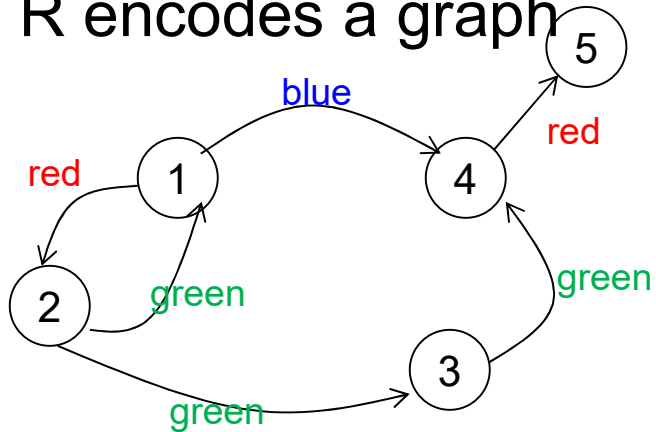
R=

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

GreenP(x,y) :-

Labeled Graphs

R encodes a graph



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

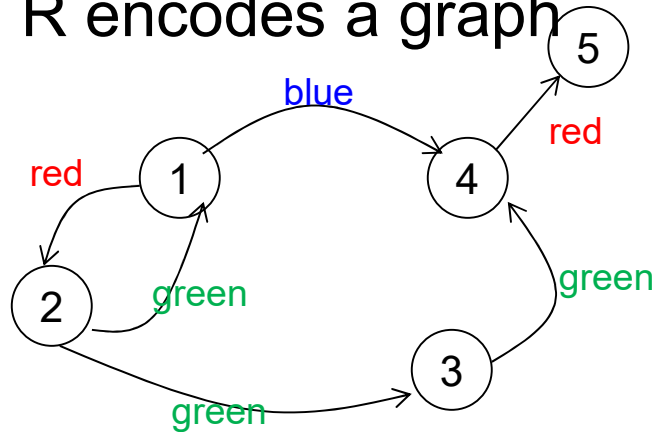
R=

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')

Labeled Graphs

R encodes a graph



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

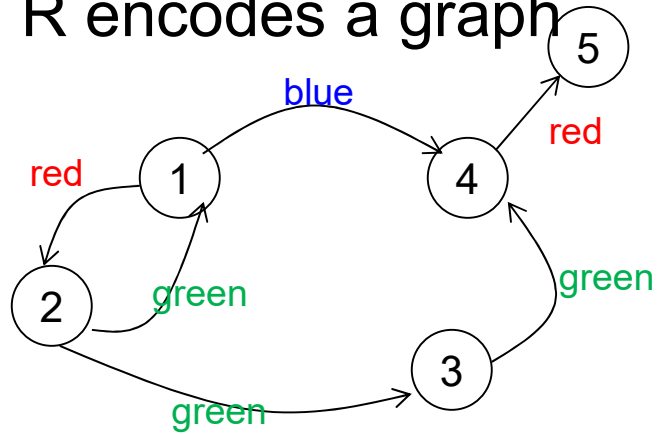
R=

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



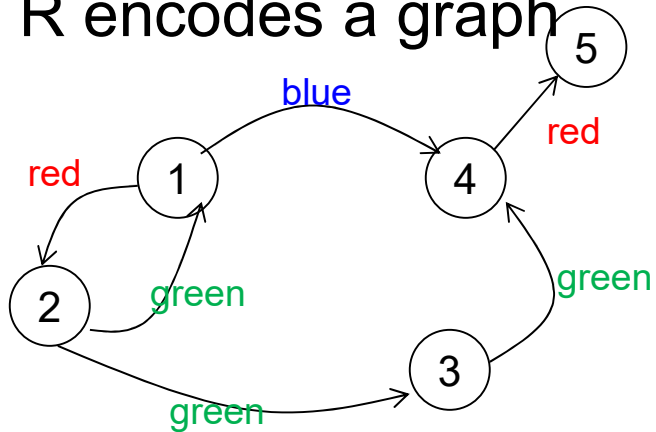
Find pairs of nodes (x,y) connected by a monochromatic path

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



R=

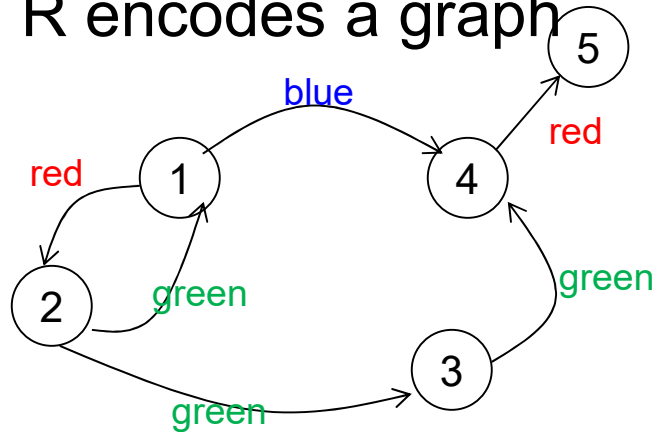
1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

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

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

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

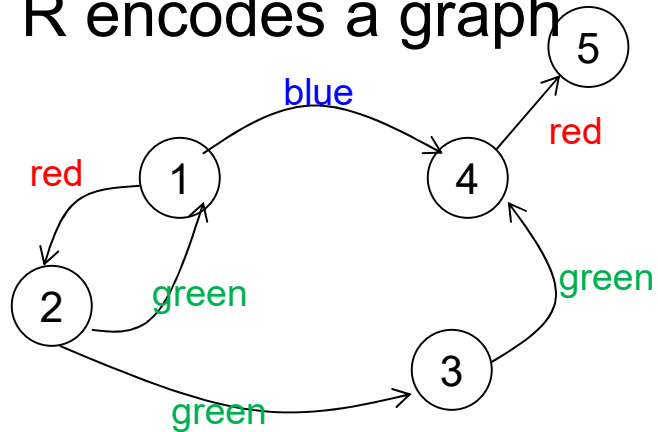
Find pairs of nodes (x,y) connected by a monochromatic path

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

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

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

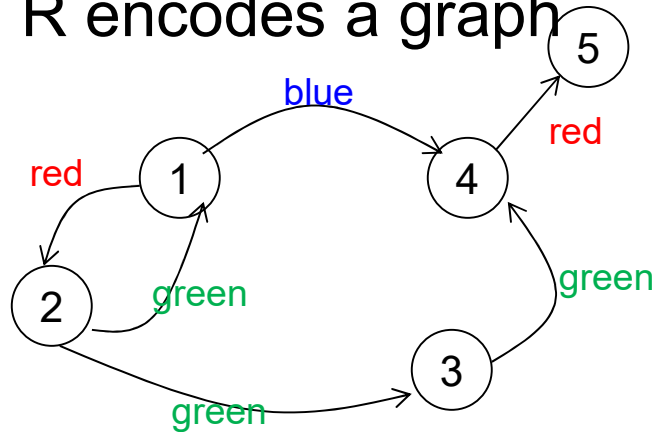
We join on
both the node z,
and the color c

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

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

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

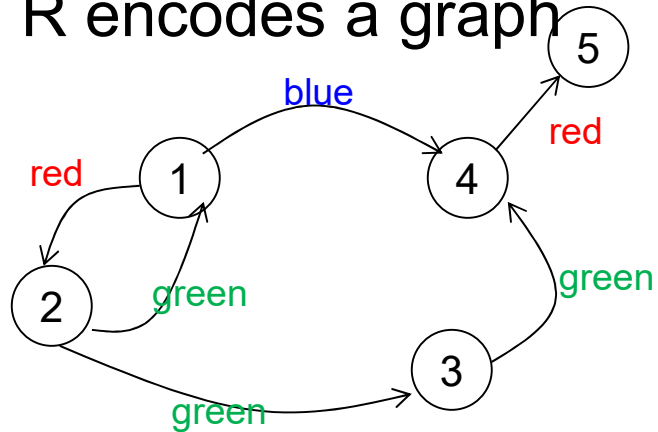
$P(x,y,c) :- R(x,y,c)$

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

Answer(x,y) :- P(x,y,c) – why needed?

Labeled Graphs

R encodes a graph



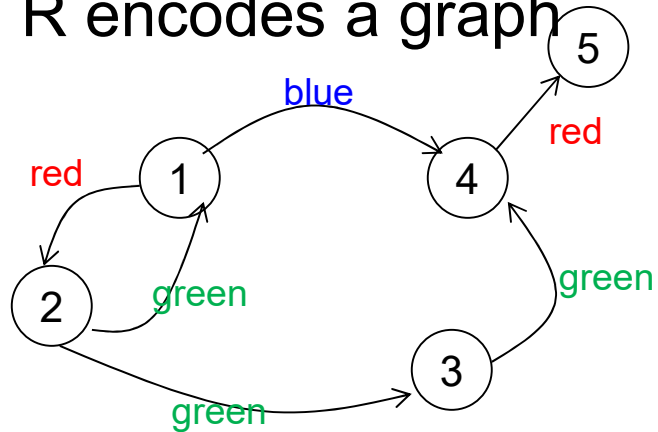
Find all nodes reachable from node 2 by a path containing exactly one red edge.

R=

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

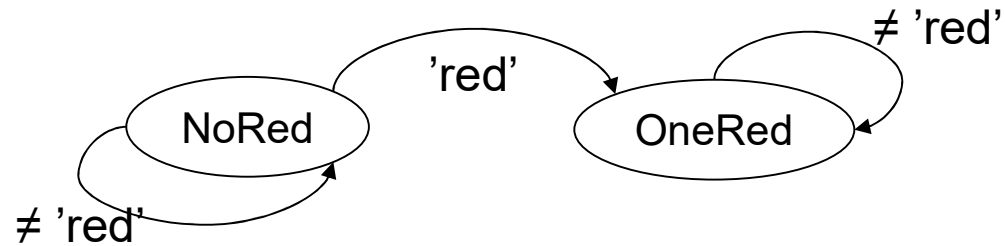
Labeled Graphs

R encodes a graph



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

Automaton:

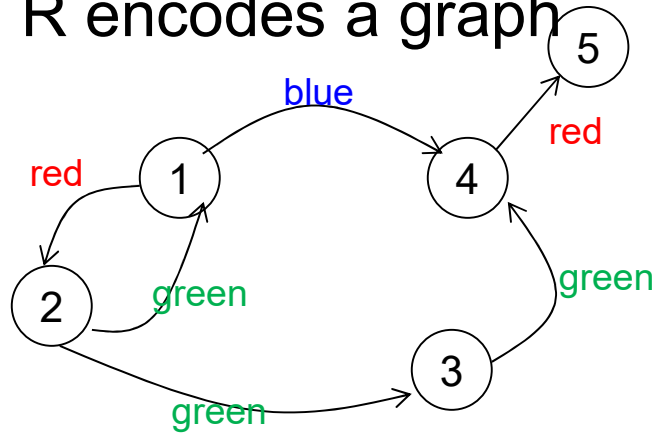


R=

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

Labeled Graphs

R encodes a graph

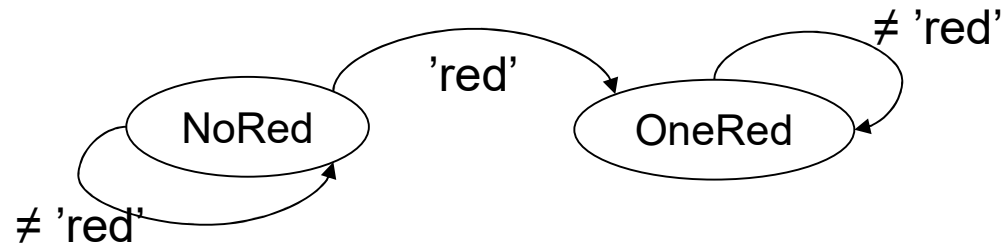


R=

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.

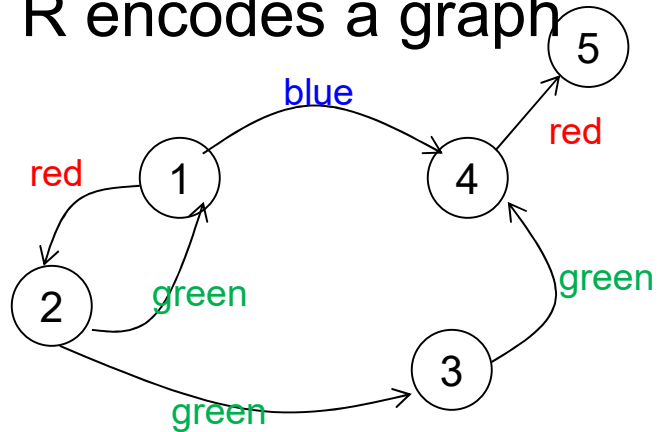
Automaton:



NoRed(2). :- .

Labeled Graphs

R encodes a graph

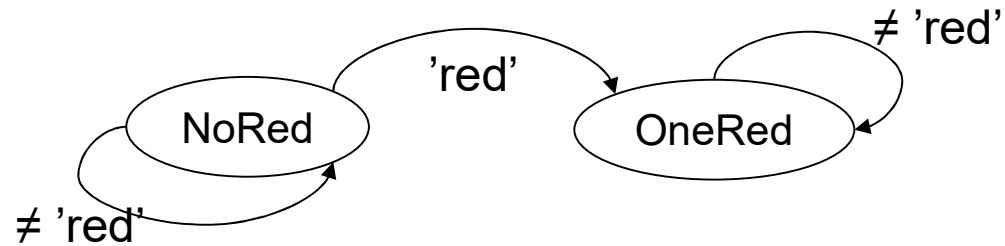


R=

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:

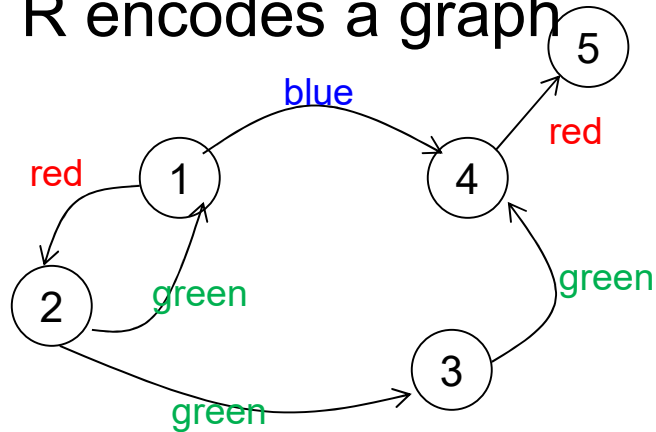


NoRed(2). :- .

NoRed(y) :- NoRed(x), R(x,y,c), c!='red'.

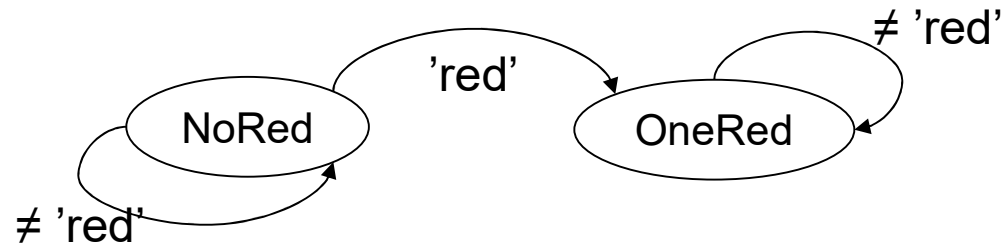
Labeled Graphs

R encodes a graph



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

Automaton:



R=

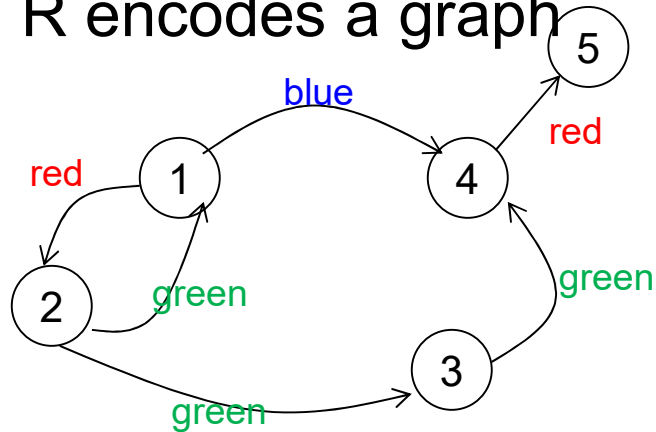
1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

```

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

Labeled Graphs

R encodes a graph

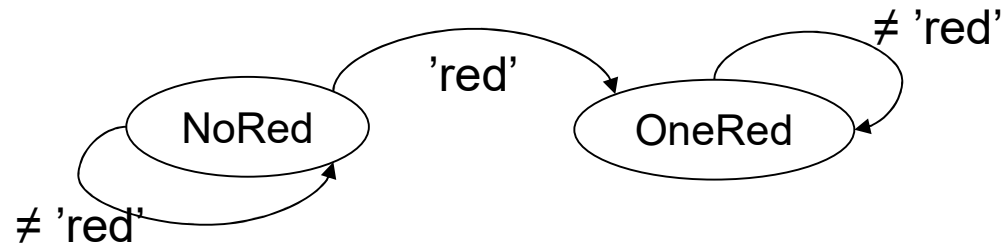


Find all nodes reachable from node 2 by a path containing exactly one red edge.

R=

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

Automaton:



```

NoRed(2). :- .
NoRed(y) :- NoRed(x), R(x,y,c), c!='red'.
OneRed(y) :- NoRed(x), R(x,y,'red').
OneRed(y) :- OneRed(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

Relation T is called a
Common Table Expression
CTE

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

```
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;
```

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)

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)

Odd := \emptyset ; Even := \emptyset ;

repeat

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

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

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)

Odd := \emptyset ; Even := \emptyset ;

repeat

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

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

Odd := Odd_{new}

Even := Even_{new}

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)
```

```
Odd :=  $\emptyset$ ; Even :=  $\emptyset$ ;
```

```
repeat
```

```
  Evennew :=  $\Pi_{x,y}(\text{Odd} \bowtie R)$ ;
```

```
  Oddnew :=  $R \cup \Pi_{x,y}(\text{Even} \bowtie R)$ ;
```

```
  if Odd=Oddnew  $\wedge$  Even=Evennew
```

```
    then break
```

```
  Odd:=Oddnew
```

```
  Even:=Evennew
```

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

Before we show this, a digression: **monotone queries**

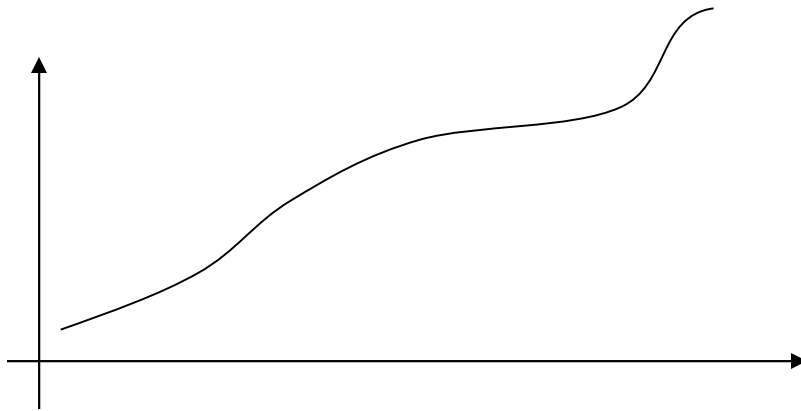
Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Review: Monotone Functions

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

If $x \leq y$ then $f(x) \leq f(y)$



Monotone Queries

- A query with input relations R, S, T, \dots 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, \dots 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 $R \subseteq R', S \subseteq S', \dots$ then $Q(R, S, \dots) \subseteq Q(R', S', \dots)$

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

Which Ops are Monotone?

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

Which Ops are Monotone?

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

Fun Fact

- A SELECT-FROM-WHERE query (without aggregates or subqueries) is monotone

```
SELECT [DISTINCT] ...  
FROM R1 x1, R2 x2, ...  
WHERE ...
```


Fun Fact

- A **SELECT-FROM-WHERE** query (without aggregates or subqueries) is monotone

```
SELECT [DISTINCT] ...  
FROM R1 x1, R2 x2, ...  
WHERE ...
```

- **Proof: the nested loop semantics!**
When we add tuples to one relation, we cannot lose answers:

```
for x1 in R1 do:  
  for x2 in R2 do:  
    ...
```

Tips for Writing SQL Queries

- If the English formulation of a query is non-monotone, then you need to use a subquery OR aggregate in SQL

Return SUPPLIERS who supply
some product with price > \$10000

Return SUPPLIERS who supply
only products with price > \$10000

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

```
 $IDB_0 := \emptyset; t := 0$   
repeat  $IDB_{t+1} := USPJ(IDB_t); t := t + 1$   
until no more change
```

Naïve Evaluation Algorithm

Fact: Every USPJ query is monotone

Proof: Uses only $\sigma, \Pi, \bowtie, \cup$

```
 $IDB_0 := \emptyset; t := 0$   
repeat  $IDB_{t+1} := USPJ(IDB_t); t := t + 1$   
until no more change
```

Naïve Evaluation Algorithm

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

```
 $IDB_0 := \emptyset; t := 0$   
repeat  $IDB_{t+1} := USPJ(IDB_t); t := t + 1$   
until no more change
```

Naïve Evaluation Algorithm

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 $IDB_0 (= \emptyset) \subseteq IDB_1$

```
 $IDB_0 := \emptyset; \quad t := 0$   
repeat  $IDB_{t+1} := USPJ(IDB_t); \quad t := t + 1$   
until no more change
```

Naïve Evaluation Algorithm

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

```
 $IDB_0 := \emptyset; t := 0$   
repeat  $IDB_{t+1} := USPJ(IDB_t); t := t + 1$   
until no more change
```

Naïve Evaluation Algorithm

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

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

Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Non-monotone Extensions

- Aggregates
 - Grouping
 - Negation
- No standard syntax
We will follow Souffle

Actor(id, fname, lname)

Casts(pid, mid)

Movie(id, name, year)

Aggregates

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

Actor(id, fname, lname)

Casts(pid, mid)

Movie(id, name, year)

Aggregates

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

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, name, year)

Aggregates

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

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, name, year)

Grouping

$Q(y,c) :- \text{Movie}(_,_,y), c = \text{count} : \{ \text{Movie}(_,_,y) \}$

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

Grouping

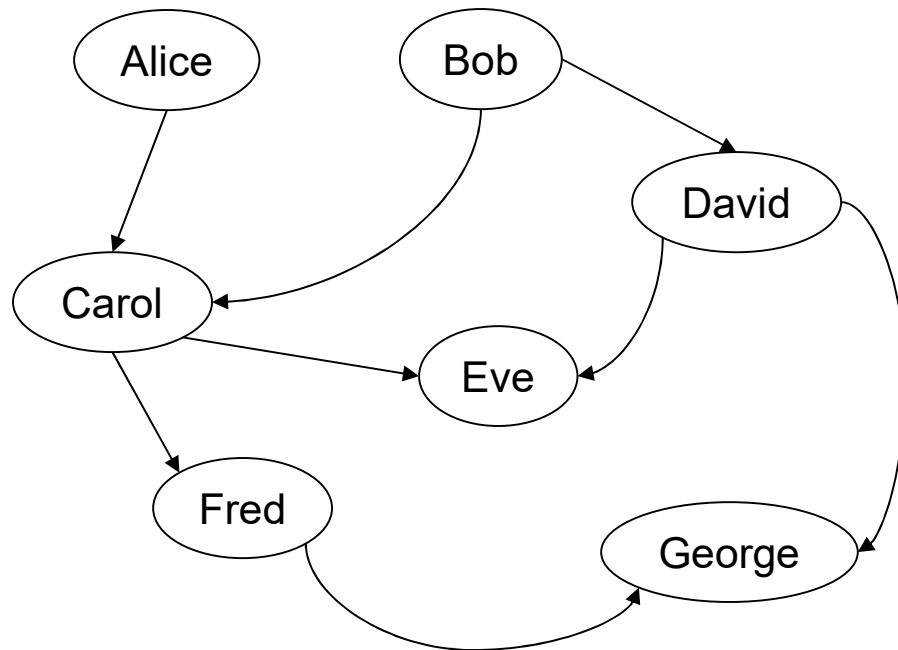
$Q(y,c) :- \text{Movie}(_,_,y), c = \text{count} : \{ \text{Movie}(_,_,y) \}$

Meaning (in SQL)

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


Examples

A genealogy database (parent/child)



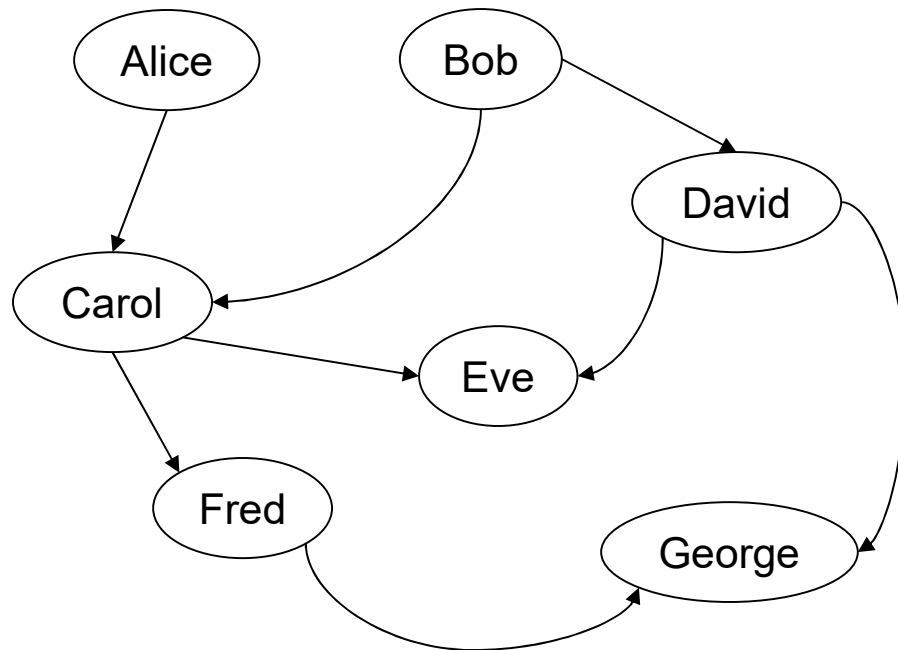
ParentChild

p	c
Alice	Carol
Bob	Carol
Bob	David
Carol	Eve
...	

ParentChild(p,c)

Count Descendants

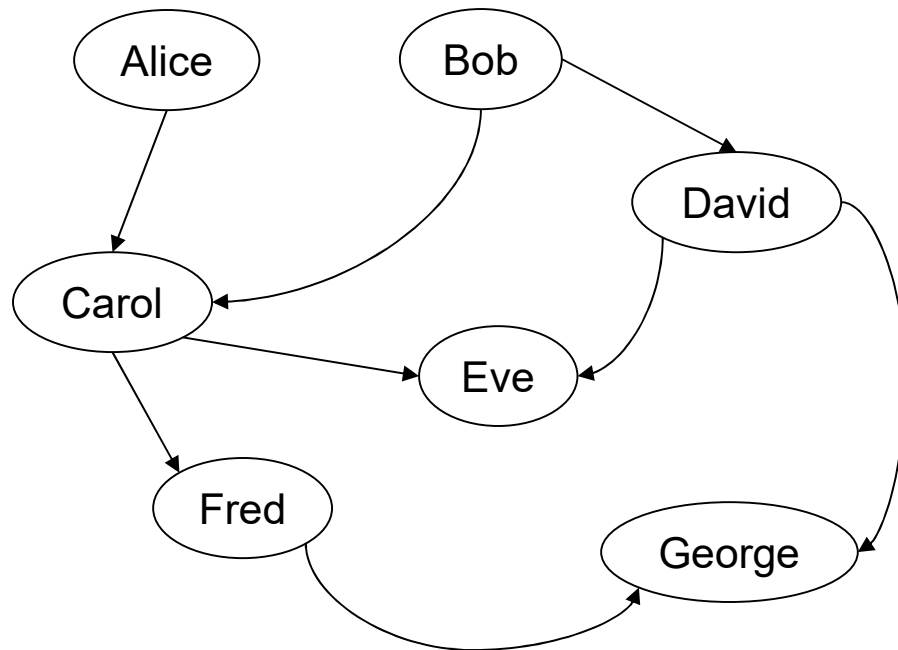
For each person, count his/her descendants



ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



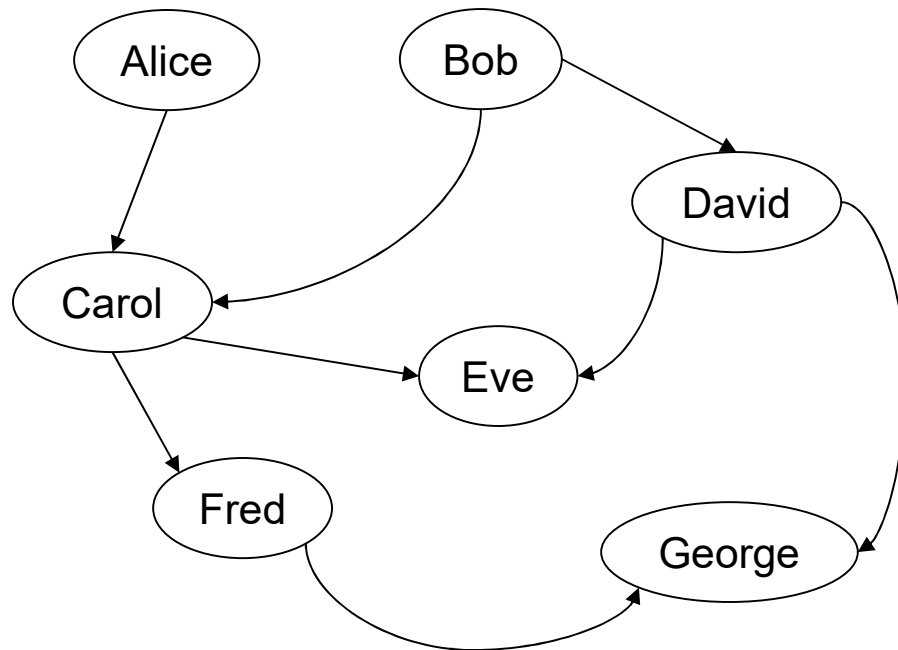
Answer

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?)

ParentChild(p,c)

Count Descendants

Compute transitive closure of ParentChild

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

ParentChild(p,c)

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).
```

ParentChild(p,c)

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).

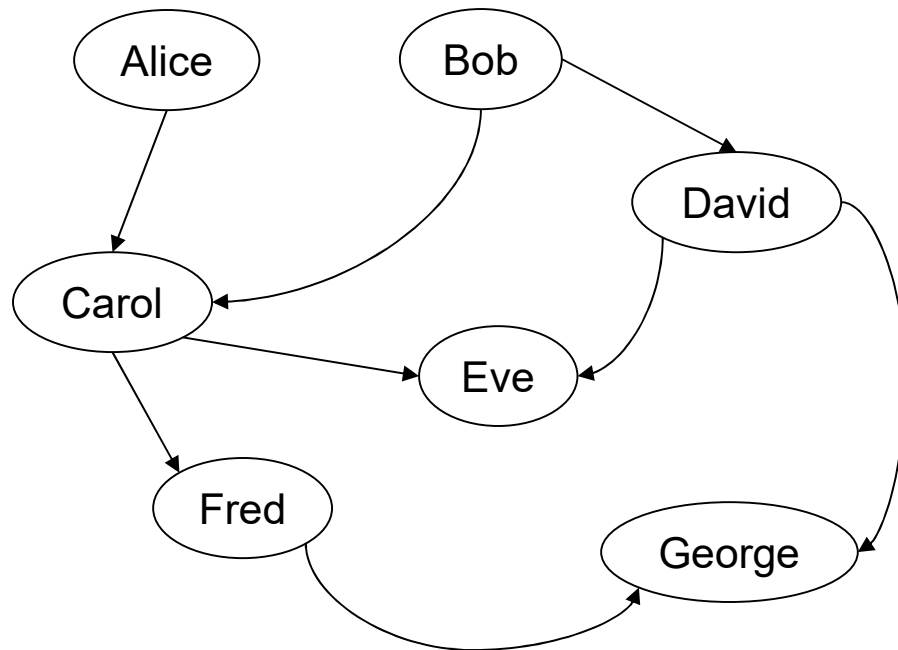
// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.

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

ParentChild(p,c)

Negation: use “!”

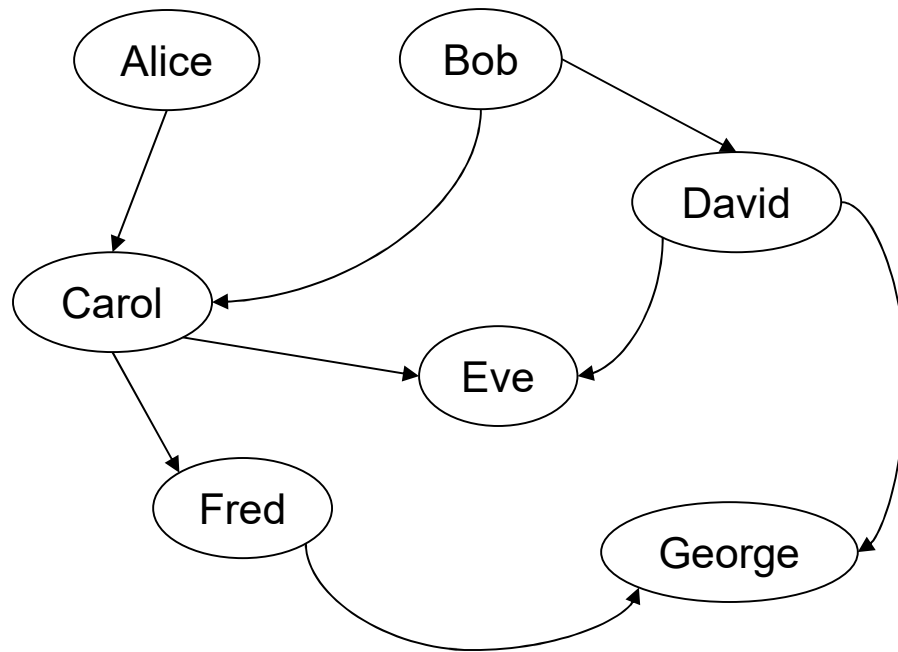
Find all descendants of Bob that are not descendants of Alice



ParentChild(p,c)

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).
```

ParentChild(p,c)

Negation: use “!”

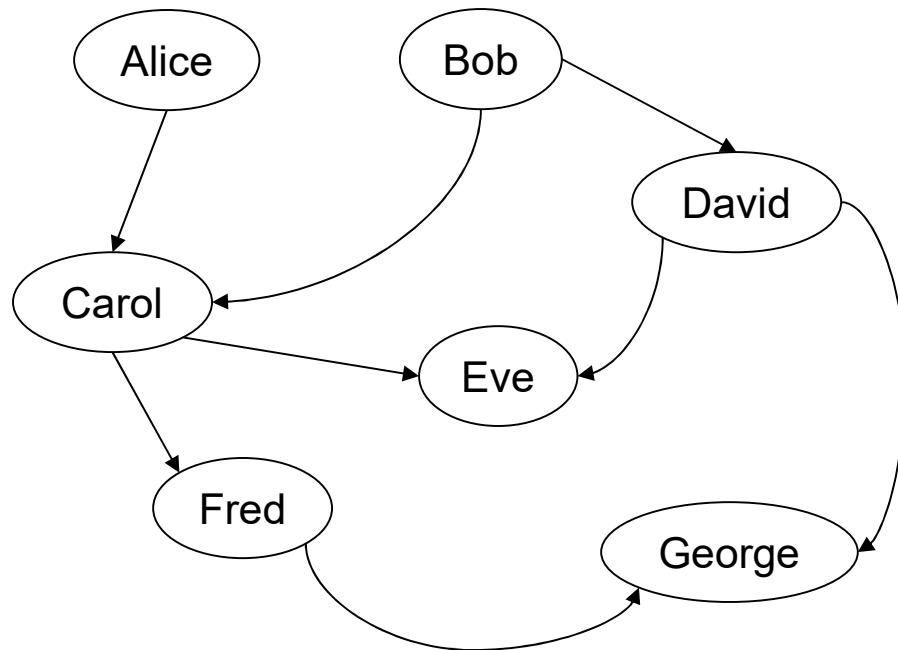
Find all descendants of Bob that are not descendants of Alice

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

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

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

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
```


ParentChild(p,c)

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)
```

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?

ParentChild(p,c)

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
```

Stratified Datalog

Recursion conflicts with non-monotone queries

- Example: what does this mean?

```
Happy(Bob):- !Happy(Alice).  
Happy(Alice) :- !Happy(Bob).
```

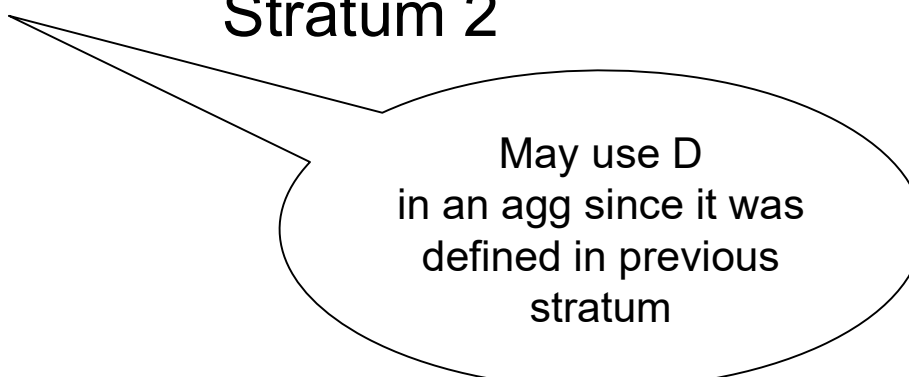
- A program is *stratified* if it can be partitioned into *strata*, such that every IDB predicate in a non-monotone position has been defined in an earlier stratum

Stratified Datalog

D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
T(p,c) :- D(p,_), c = count : { D(p,_) }.
Q(d) :- T(p,d), p = "Alice".

Stratum 1

Stratum 2



May use D
in an agg since it was
defined in previous
stratum

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).  
D(x,z) :- D(x,y), ParentChild(y,z).  
T(p,c) :- D(p,_), c = count : { D(p,_) }.  
Q(d) :- T(p,d), p = "Alice".
```

Stratum 1

Stratum 2

```
D(x,y) :- ParentChild(x,y).  
D(x,z) :- D(x,y), ParentChild(y,z).  
Q(x) :- D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use D
in an agg since it was
defined in previous
stratum

May use !D

```
Happy(Bob):- !Happy(Alice).  
Happy(Alice) :- !Happy(Bob).
```

Non-stratified

Some Datalog Optimizations

- Every USPJ optimized traditionally
- Semi-naïve evaluation
- Magic sets
- Asynchronous execution

Summary

- Datalog = light-weight syntax, recursion
- Data independence, optimizations
- Limitations:
 - Monotone queries work great
 - Non-monotone queries: various restrictions