CSE 544 Principles of Database Management Systems

Fall 2016

Lecture 5 – Datalog (1)

Announcements

Project

- Proposal due tomorrow
- Try to include **concrete deliverables** to keep track of progress

Homework 2 posted, due Friday, Nov. 4th

- SimpleDB

References

- R&G Chapter 24
- Phokion Kolaitis' tutorial on database theory at Simon's https://simons.berkeley.edu/sites/default/files/docs/5241/simons16-21.pdf
- Reading for Thursday:
  Joe Hellerstein, "The Declarative Imperative,"
  SIGMOD Record 2010

Datalog

- Alternative notation for queries
- Designed for *recursive* queries in the 80s
- Modern implementations: commercial (LogicBlox), networking (Overlog), programming languages, …

- Topics
  – Syntax of datalog
  – How to evaluate
  – Datalog semantics

Running Datalog

How to try out datalog quickly:

- Download DLV from: http://www.dlvsystem.com/dlvdb/
- Run DLV on this file:

```
parent(william, john).
parent(john, james).
parent(james, bill).
parent(sue, bill).
parent(james, carol).
parent(sue, carol).

male(john).
male(james).
female(sue).
male(bill).
female(carol).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).
brother(X, Y) :- parent(P, X), parent(P, Y), male(X), X != Y.
sister(X, Y)  :- parent(P, X), parent(P, Y), female(X), X != Y.
```

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

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)

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)

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)
```

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)

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

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

Terminology

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

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

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

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

```
U1(x,y) :- Movie(x,z,1994), y>1910
```

```
U2(x)   :- Movie(x,z,1994), not Casts(u,x)
```

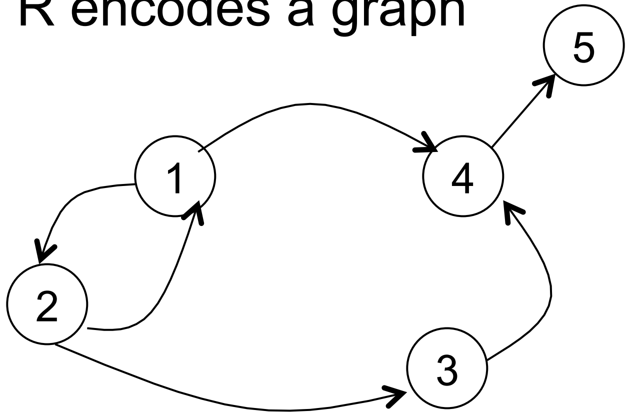
A datalog rule is *safe* if every variable appears in some positive relational atom

Datalog v.s. SQL

- Non-recursive datalog with negation is a cleaned-up, core of SQL
- You should be able to translate easily between non-recursive datalog with negation and SQL

Simple datalog programs

R encodes a graph



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

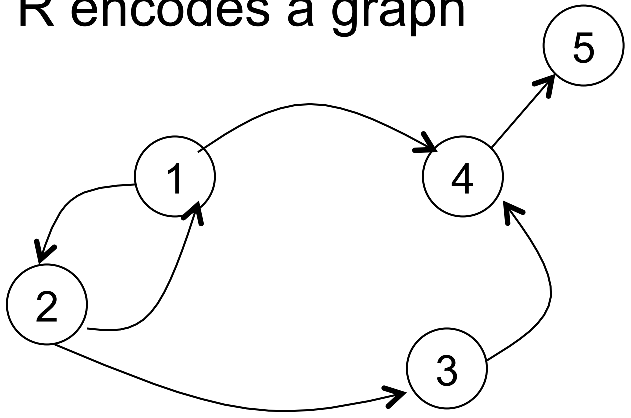
What does it compute?

R=

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

Simple datalog programs

R encodes a graph



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

What does it compute?

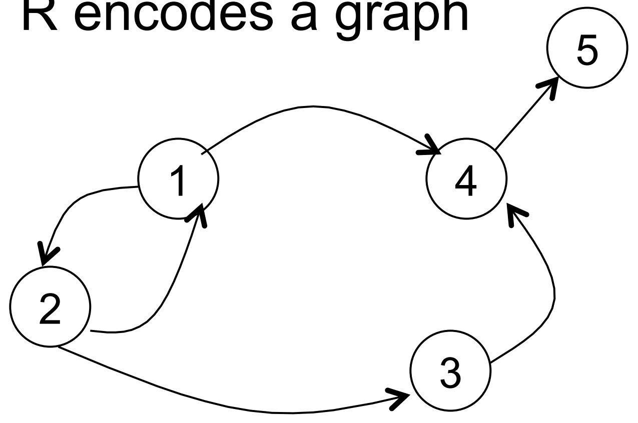
R=

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

Initially:
T is empty.

Simple datalog programs

R encodes a graph



```
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

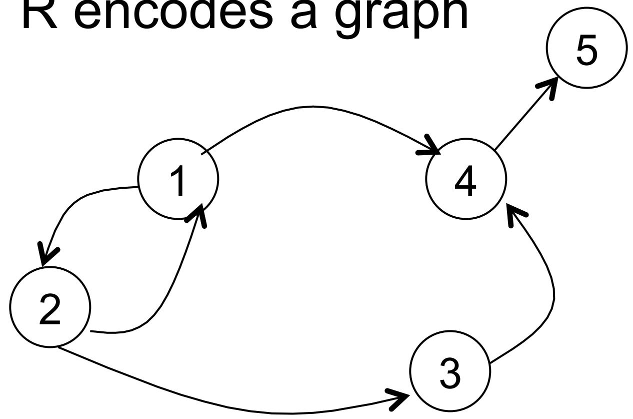
Initially:
T is empty.

First iteration:
T =

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

Simple datalog programs

R encodes a graph



```
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 =

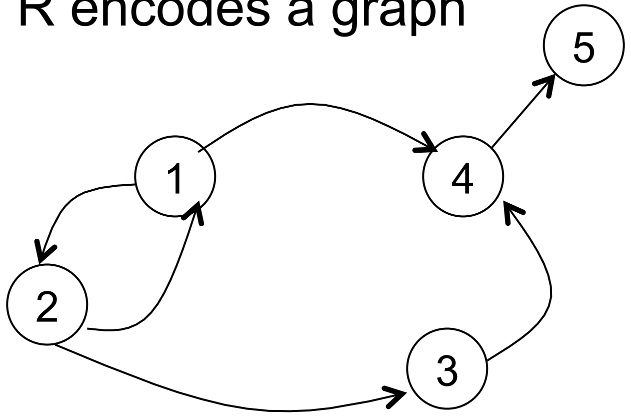
1	2
2	1
2	3
1	4
3	4
4	5

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

Simple datalog programs

R encodes a graph



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

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

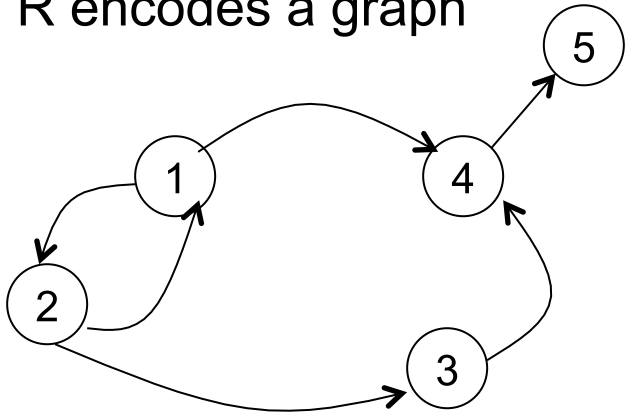
Third iteration:
T =

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

Done

Simple datalog programs

R encodes a graph



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

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

Third iteration:
T =

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

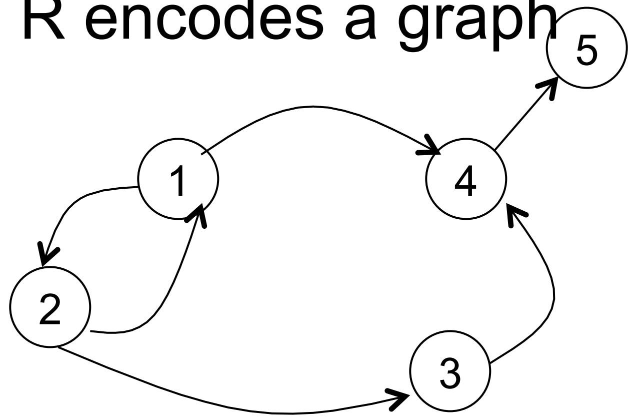
Discovered 3 times!

Discovered twice

Done

Simple datalog programs

R encodes a graph


$$\begin{tikzcd}
 & 1 \arrow[dl, bend right] \arrow[rr, bend left] & & 4 \arrow[r] & 5 \\
2 \arrow[ur, bend right] \arrow[rr, bend right] & & 3 \arrow[ur, bend right] & &
\end{tikzcd}$$

R=

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

Alternative ways to compute TC:

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

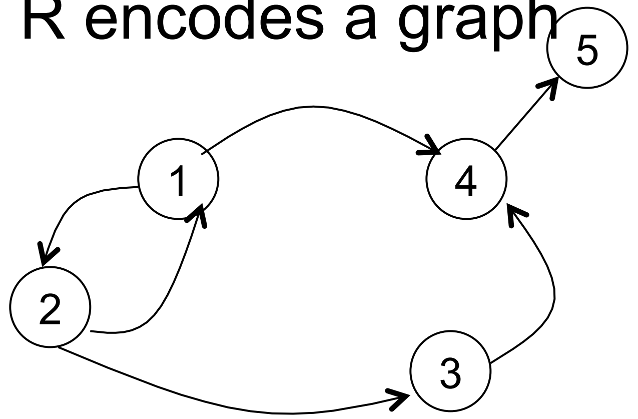
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

Non-linear

What are the pros / cons?

Other Interesting Programs

R encodes a graph



Non 2-colorability:

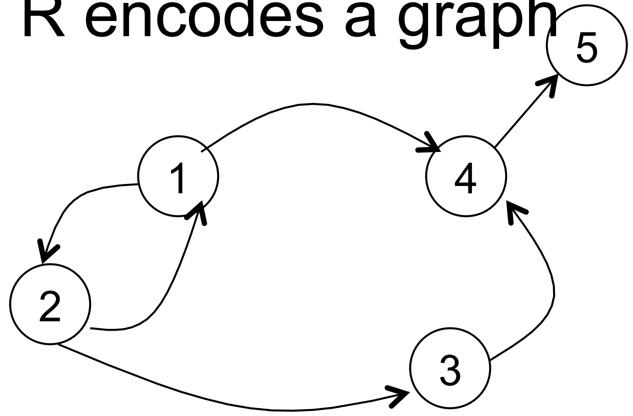
```
ODD(x,y) :- R(x,y)
ODD(x,y) :- R(x,z), EVEN(z,y)
EVEN(x,y) :- R(x,z),ODD(z,y)
Q :- ODD(x,x)
```

Path Systems (PTIME-complete)

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

Other Interesting Programs

R encodes a graph



Cousins:

```
Cousins(y,z) :- Parent(x,y),Parent(x,z)
Cousins(u,v) :- Cousins(x,y),Parent(x,u),Parent(y,v)
```

Syntax of Datalog Programs

The schema consists of two sets of relations:

- Extensional Database (EDB): R_1, R_2, …
- Intentional Database (IDB): P_1, P_2, …

A datalog program **P** has the form:

P:

$$\begin{array}{l} P_{i1}(x_{11},x_{12},\ldots) \text{ :- } body_1 \\ P_{i2}(x_{21},x_{22},\ldots) \text{ :- } body_2 \\ \quad\quad\quad\ldots. \end{array}$$

- Each head predicate P_i is an IDB
- Each body is a conjunction of IDB and/or EDB predicates

Note: no negation (yet)! Recursion OK.

Naïve Datalog Evaluation Algorithm

Datalog program:

```
P_i1 :-  body_1
P_i2 :-  body_2
    ….
```

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔

Group by
IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔

Group by
IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

➔

Each rule is a
Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- body_1
P_{i2} :- body_2
….

➔

Group by
IDB predicate

P_1 :- $\text{body}_{11} \cup \text{body}_{12} \cup \ldots$
P_2 :- $\text{body}_{21} \cup \text{body}_{22} \cup \ldots$
….

➔

Each rule is a
Select-Project-Join-Union query

P_1 :- SPJU_1
P_2 :- SPJU_2
….

Example:

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

➔ ?

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- body_1
P_{i2} :- body_2
….

➔

Group by
IDB predicate

P_1 :- $\text{body}_{11} \cup \text{body}_{12} \cup \ldots$
P_2 :- $\text{body}_{21} \cup \text{body}_{22} \cup \ldots$
….

➔

Each rule is a
Select-Project-Join-Union query

P_1 :- SPJU_1
P_2 :- SPJU_2
….

Example:

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

➔ $T(x,y) \text{ :- } R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$

Naïve Datalog Evaluation Algorithm

Datalog program:

$$\boxed{\begin{array}{l} P_{i1} :- \ body_1 \\ P_{i2} :- \ body_2 \\ \quad .... \end{array}} \xrightarrow{} \boxed{\begin{array}{l} P_1 :- \ body_{11} \cup body_{12} \cup \ldots \\ P_2 :- \ body_{21} \cup body_{22} \cup \ldots \\ .... \end{array}} \xrightarrow{} \boxed{\begin{array}{l} P_1 :- \ SPJU_1 \\ P_2 :- \ SPJU_2 \\ .... \end{array}}$$

Group by
IDB predicate

Each rule is a
<u>S</u>elect-<u>P</u>roject-<u>J</u>oin-<u>U</u>nion query

Naïve datalog evaluation algorithm:

```
$P_1 = P_2 = \ldots = \varnothing$
Loop
    $NewP_1 = SPJU_1$; $NewP_2 = SPJU_2$;  …
    if ($NewP_1 = P_1$ and $NewP_2 = P_2$ and …)
        then exit
    $P_1 = NewP_1$; $P_2 = NewP_2$; …
Endloop
```

Example:

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

$\rightarrow$ $T(x,y) :- R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

$\rightarrow$ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

$\rightarrow$ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve datalog evaluation algorithm:

```
P_1 = P_2 = … = ∅
Loop
    NewP_1 = SPJU_1; NewP_2 = SPJU_2; …
    if (NewP_1 = P_1 and NewP_2 = P_2 and …)
        then exit
    P_1 = NewP_1; P_2 = NewP_2; …
Endloop
```

Example:

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

$\rightarrow$ $T(x,y)$:- $R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$

```
T= ∅
Loop
    NewT(x,y) = $R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$
    if (NewT = T)
        then exit
    T = NewT
Endloop
```

Discussion

- A datalog program *always* terminates (why?)
- A datalog program *always* runs in PTIME in the size of the database (why?)

Problem with the Naïve Algorithm

- The same facts are discovered over and over again
- The *semi-naïve* algorithm tries to reduce the number of facts discovered multiple times

Background: Incremental View Maintenace

Let V be a view computed by one datalog rule (no recursion)

```
V :- body
```

If (some of) the relations are updated: $R_1 \leftarrow R_1 \cup \Delta R_1$, $R_1 \leftarrow R_2 \cup \Delta R_2$, …

Then the view is also modified as follows: $V \leftarrow V \cup \Delta V$

Incremental view maintenance:
Compute ΔV without having to recompute V

Background: Incremental View Maintenace

Example 1:

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

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

Background: Incremental View Maintenace

Example 1:

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

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

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

Background: Incremental View Maintenace

Example 2:

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

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

Background: Incremental View Maintenace

Example 2:

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

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

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

Background: Incremental View Maintenace

Example 3:

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

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

Background: Incremental View Maintenace

Example 3:

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

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

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

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.
Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

```
$P_1 = \Delta P_1 =$ non-recursive-$SPJU_1$,
$P_2 = \Delta P_2 =$ non-recursive-$SPJU_2$,
…
Loop
    $\Delta P_1 = \Delta SPJU_1 - P_1$;  $\Delta P_2 = \Delta SPJU_2 - P_2$;  …
    if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and  …)
        then break
    $P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$;  …
Endloop
```

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

```
$P_1 = \Delta P_1$ = non-recursive-$SPJU_1$, $P_2 = \Delta P_2$ = non-recursive-$SPJU_2$, …
Loop
    $\Delta P_1 = \Delta\, SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$;  …
    if ($\Delta P_1 = \varnothing$ and $\Delta P_2 = \varnothing$ and  …)
        then break
    $P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$;  …
Endloop
```

Example:

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

```
$T = \Delta T$ = ? (non-recursive rule)
Loop
    $\Delta T(x,y)$  =  ? (recursive Δ-rule)
    if ($\Delta T = \varnothing$)
        then break
    $T = T \cup \Delta T$
Endloop
```

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

```
$P_1 = \Delta P_1 = \text{non-recursive-}SPJU_1$, $P_2 = \Delta P_2 = \text{non-recursive-}SPJU_2$, …
Loop
    $\Delta P_1 = \Delta\, SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$;  …
    if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and  …)
        then break
    $P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$;  …
Endloop
```

Example:

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

```
$T(x,y) = R(x,y)$,   $\Delta T(x,y) = R(x,y)$
Loop
    $\Delta T(x,y) = R(x,z), \Delta T(z,y) - R(x,y)$
    if ($\Delta T = \emptyset$)
        then break
    $T = T \cup \Delta T$
Endloop
```

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part. Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

```
$P_1 = \Delta P_1$ = non-recursive-$SPJU_1$, $P_2 = \Delta P_2$ = non-recursive-$SPJU_2$, …
Loop
    $\Delta P_1 = \Delta SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$;  …
    if ($\Delta P_1 = \varnothing$ and $\Delta P_2 = \varnothing$ and  …)
        then break
    $P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$;  …
Endloop
```

Example:

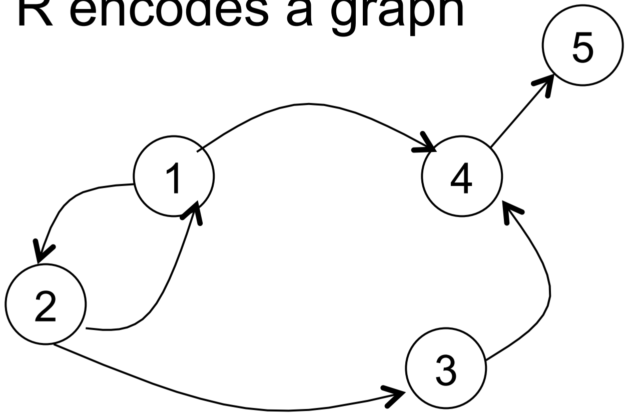
```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

```
$T(x,y) = R(x,y)$,   $\Delta T(x,y) = R(x,y)$
Loop
    $\Delta T(x,y) = R(x,z), \Delta T(z,y) - R(x,y)$
    if ($\Delta T = \varnothing$)
        then break
    $T = T \cup \Delta T$
Endloop
```

Note: for any linear datalog programs, the semi-naïve algorithm has only one Δ-rule for each rule!

Simple datalog programs

R encodes a graph



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

```
T= R,  ΔT = R
Loop
   ΔT(x,y)  = R(x,z),  ΔT(z,y)
                  -- R(x,y)
   if (ΔT = ∅)
          then break
   T = T∪ΔT
Endloop
```

R=

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

Initially:

ΔT=

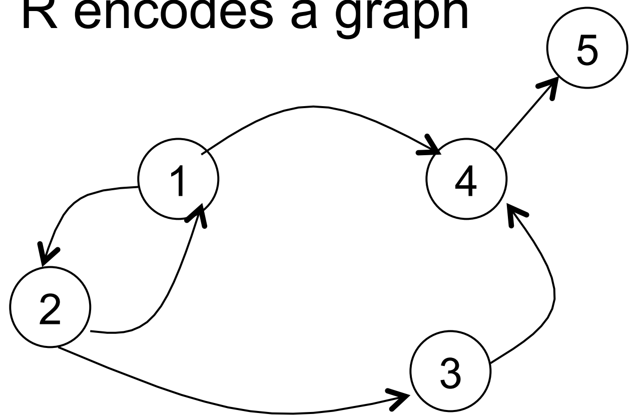
1	2
1	4
2	1
2	3
3	4
4	5

T=

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

Simple datalog programs

R encodes a graph



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

```
T= R,  ΔT = R
Loop
   ΔT(x,y)  = R(x,z),  ΔT(z,y)
                  -- R(x,y)
   if (ΔT = ∅)
          then break
   T = T∪ΔT
Endloop
```

R=

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

Initially:

ΔT=

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

T=

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

First iteration:

ΔT=
paths of length 2

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

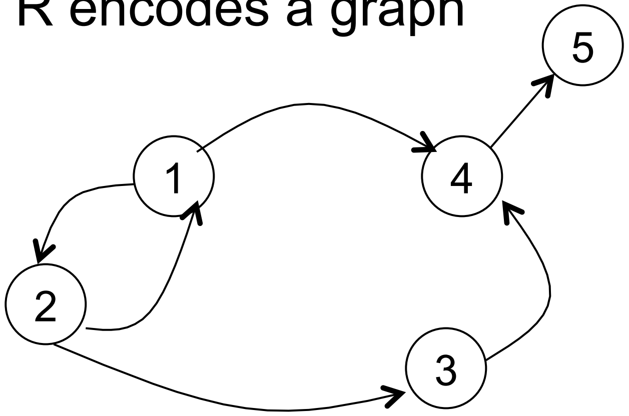
T=

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

Simple datalog programs

```
T= R,  ΔT = R
Loop
   ΔT(x,y)  = R(x,z),  ΔT(z,y)
                    -- R(x,y)
   if (ΔT = ∅)
          then break
   T = T∪ΔT
Endloop
```

R encodes a graph



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

R=

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

Initially:

ΔT=

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

T=

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

First iteration:

ΔT=
paths of length 2

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

T=

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

Second iteration:

ΔT=
paths of length 3

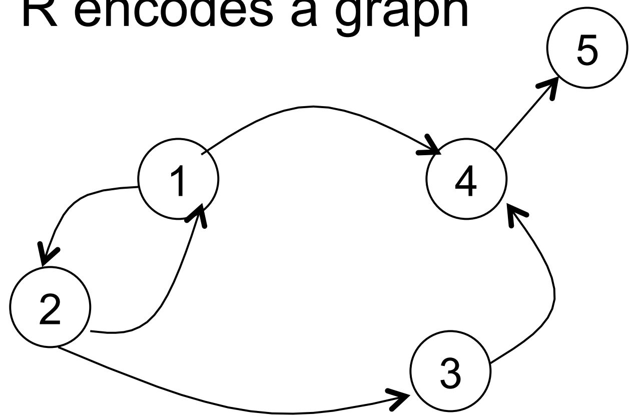
1	2
1	4
2	1
2	3
2	5

T=

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

Simple datalog programs

R encodes a graph



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

```
T= R,  ΔT = R
Loop
   ΔT(x,y)  = R(x,z), ΔT(z,y)
                  -- R(x,y)
   if (ΔT = ∅)
        then break
   T = T∪ΔT
Endloop
```

R=

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

Initially:

ΔT=

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

T=

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

First iteration:

ΔT= paths of length 2

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

T=

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

Second iteration:

ΔT= paths of length 3

1	2
1	4
2	1
2	3
2	5

T=

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

Third iteration:

ΔT= paths of length 4

Discussion of Semi-Naïve Algorithm

- Avoids re-computing some tuples, but not all tuples
- Easy to implement, no disadvantage over naïve

- A rule is called *linear* if its body contains only one recursive IDB predicate:
  - A linear rule always results in a single incremental rule
  - A non-linear rule may result in multiple incremental rules

Adding Negation: Datalog¬

Example: compute the complement of the transitive closure

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
CT(x,y) :- Node(x), Node(y), not T(x,y)
```

What does this mean??

Recursion and Negation Don't Like Each Other

EDB: I = { R(a) }

```
S(x) :- R(x), not T(x)
T(x) :- R(x), not S(x)
```

What are the possible outcomes of S and T?

Recursion and Negation Don't Like Each Other

EDB: I = { R(a) }

```
S(x) :- R(x), not T(x)
T(x) :- R(x), not S(x)
```

What are the possible outcomes of S and T?

J_1 = { } J_2 = {S(a)} J_3 = {T(a)} J_4 = {S(a), T(a) }

Adding Negation: datalog$^\neg$

- **Solution 1: Stratified Datalog$^\neg$**
  - Rules must be partitioned into strata
  - IDB predicates defined in strata ≤ k may be negated in strata ≥ k+1
- **Solution 2: Inflationary-fixpoint Datalog$^\neg$**
  - Fire rules and always add facts (never retract)
  - Stop when nothing new is added
  - Always terminates (why ?)
- **Solution 3: Partial-fixpoint Datalog$^{\neg,*}$**
  - Fire rules, adding/retracting facts as needed
  - Stop when reaching a fixpoint
  - May not terminate
- **Solution 4: Well-founded semantics** (next lecture)

What semantics does the paper use?