CSE544
Data Management

Lectures 11-12

Datalog

Announcement

- HW3 due this Friday
- I will contact some of you to meet this Friday about the project
- No lecture on Monday: Presidents day

Motivation

- SQL can expression *relational queries;* Cannot express iteration/recursion
- 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
- In HW3 we will use Souffle

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation

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

head

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

More Datalog Terminology

Q(args) :- R1(args), R2(args), ...

- $R_i(args_i)$ called an *atom*, or a *relational predicate*

More Datalog Terminology

```
Q(args) :- R1(args), R2(args), ....
```

- $R_i(args_i)$ called an *atom*, or a *relational predicate*
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, 'Douglas', 'Fowley') is true

More Datalog Terminology

```
Q(args) :- R1(args), R2(args), ....
```

- $R_i(args_i)$ called an ***atom***, or a ***relational predicate***
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, 'Douglas', 'Fowley') is true
- In addition we can also have arithmetic predicates
  - Example: z > '1940'.

More Datalog Terminology

```
Q(args) :- R1(args), R2(args), ....
```

- $R_i(args_i)$ called an ***atom***, or a ***relational predicate***
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, ‘Douglas’, ‘Fowley’) is true
- In addition we can also have arithmetic predicates
  - Example: z > ‘1940’.
- Some systems use <-

```
Q(args) <-  R1(args), R2(args), ....
```

More Datalog Terminology

```
Q(args) :- R1(args), R2(args), ....
```

- $R_i(args_i)$ called an ***atom***, or a ***relational predicate***
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, ‘Douglas’, ‘Fowley’) is true
- In addition we can also have arithmetic predicates
  - Example: z > ‘1940’.
- Some systems use <-
- Some use AND

```
Q(args) <-  R1(args), R2(args), ....
```

```
Q(args) :- R1(args) AND R2(args) ....
```

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- If (x,y,z) ∈ Movies and z = '1940' then y is in answer

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- If (x,y,z) ∈ Movies and z = '1940' then y is in answer

$\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- If (x,y,z) ∈ Movies and z = '1940' then y is in answer

  $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') $\Rightarrow$ Q1(y)]

- We want ***smallest*** answer with this property (why?)

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- If (x,y,z) ∈ Movies and z = '1940' then y is in answer

$\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]

- We want *smallest* answer with this property (why?)
- Logically equivalent:

$\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') ⇒ Q1(y)]

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- If (x,y,z) ∈ Movies and z = '1940' then y is in answer

$\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') $\Rightarrow$ Q1(y)]

- We want ***smallest*** answer with this property (why?)
- Logically equivalent:

$\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') $\Rightarrow$ Q1(y)]

- Non-head variables are called "existential variables"

Outline

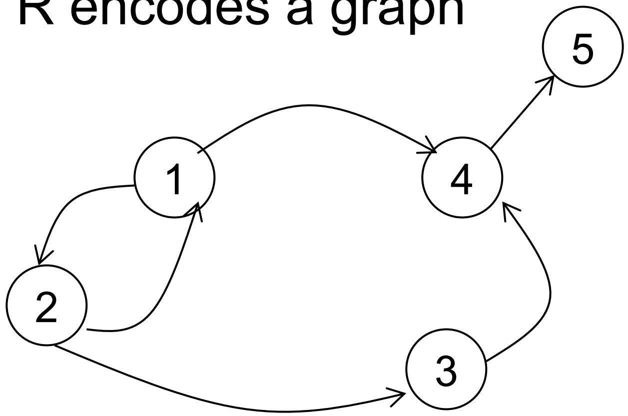
- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation

Datalog program

- A datalog program consists of several rules
- Importantly, rules may be recursive!
- Usually there is one distinguished predicate that’s the final answer
- We will show an example first, then give the general semantics.

Example

R encodes a graph

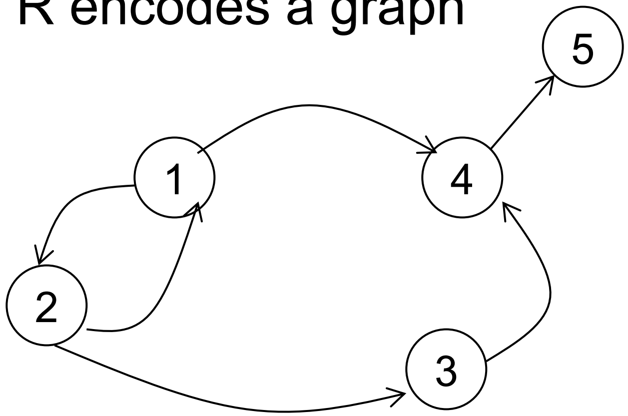


R=

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

Example

R encodes a graph



R=

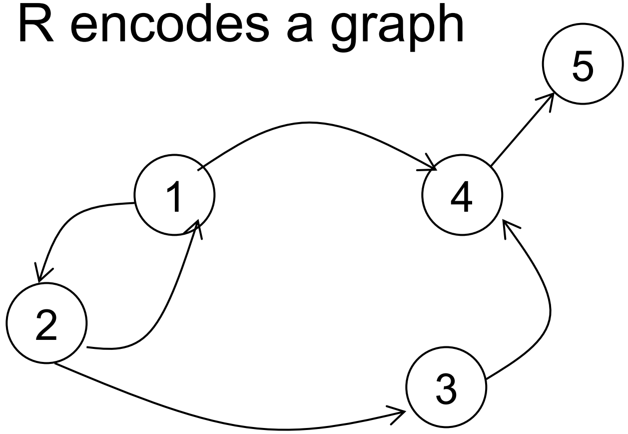
1	2
2	1
2	3
1	4
3	4
4	5

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

What does it compute?

Example

R encodes a graph



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

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

What does it compute?

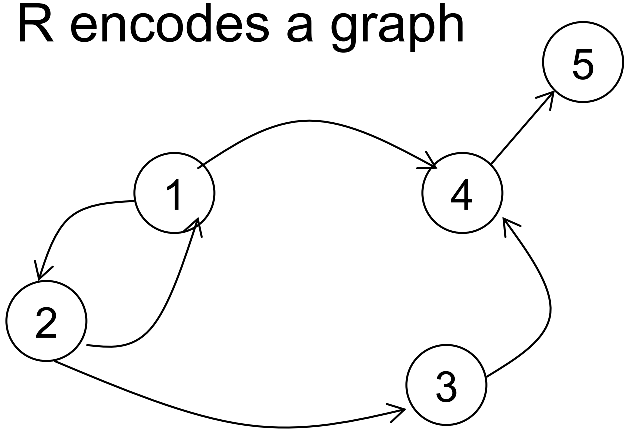
R=

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

Initially:
T is empty.

Example

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 =

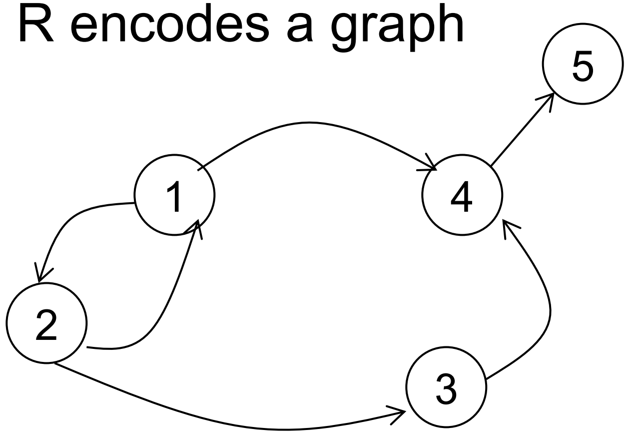
1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule generates nothing (because T is empty)

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

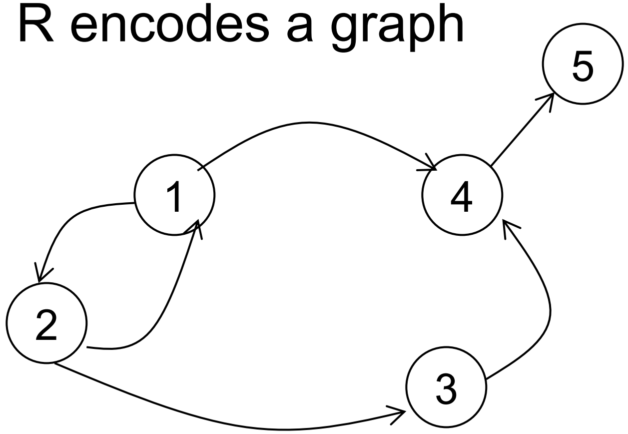
First rule generates this (rows 1–6)

Second rule generates this (rows 7–12)

New facts

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?

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

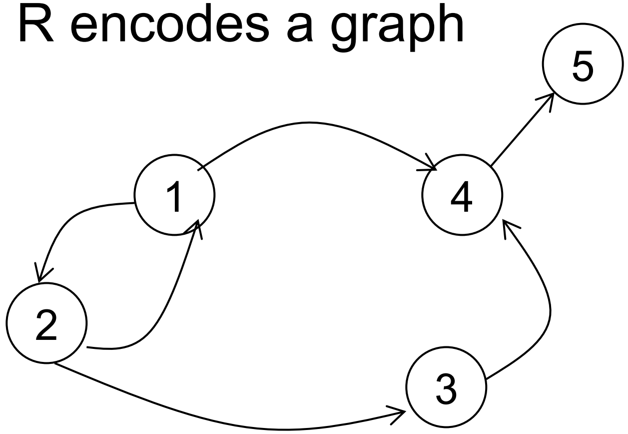
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

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

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 =

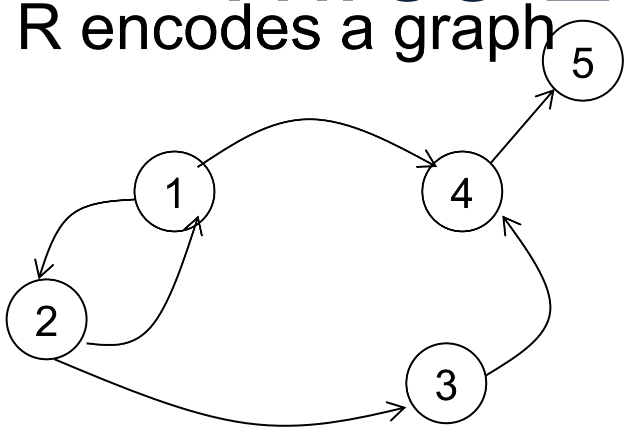
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

Fourth iteration
T =
(same)

No new facts.
DONE

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: which terminates in fewest iterations?

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation

1. Fixpoint Semantics

- Start: IDB_0 = empty relations; t = 0
  Repeat:
  IDB_{t+1} = Compute Rules(EDB, IDB_t)
  t = t+1
  Until $IDB_t = IDB_{t-1}$

1. Fixpoint Semantics

- Start: IDB_0 = empty relations; t = 0
  Repeat:
  IDB_{t+1} = Compute Rules(EDB, IDB_t)
  t = t+1
  Until $IDB_t = IDB_{t-1}$

- Remark: since rules are monotone:
  $\emptyset = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \ldots$

1. Fixpoint Semantics

- Start: IDB_0 = empty relations; t = 0
  Repeat:
  IDB_{t+1} = Compute Rules(EDB, IDB_t)
  t = t+1
  Until $IDB_t = IDB_{t-1}$

- Remark: since rules are monotone:
  $\emptyset = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \ldots$
- A datalog program w/o functions (+, *, ...) always terminates. (In what time?)

2. Minimal Model Semantics:

- Find some IDB instance that satisfies:
  1) For every rule,
     $\forall$vars [(Body(EDB,IDB) $\Rightarrow$ Head(IDB)]
  2) Is the smallest IDB satisfying (1)

2. Minimal Model Semantics:

How?

- Find some IDB instance that satisfies:
  1) For every rule,
     ∀vars [(Body(EDB,IDB) ⇒ Head(IDB)]
  2) Is the smallest IDB satisfying (1)

2. Minimal Model Semantics:

How?

- Find some IDB instance that satisfies:
  1) For every rule,
     ∀vars [(Body(EDB,IDB) ⇒ Head(IDB)]
  2) Is the smallest IDB satisfying (1)
- **Theorem**: there exists a unique such instance

2. Minimal Model Semantics:

How?

- Find some IDB instance that satisfies:
  1) For every rule,
     $\forall$vars [(Body(EDB,IDB) $\Rightarrow$ Head(IDB)]
  2) Is the smallest IDB satisfying (1)
- **Theorem**: there exists a unique such instance
- It doesn't tell us how to find it…

2. Minimal Model Semantics:

How?

- Find some IDB instance that satisfies:
  1) For every rule,
     $\forall$vars [(Body(EDB,IDB) ⇒ Head(IDB)]
  2) Is the smallest IDB satisfying (1)
- **Theorem**: there exists a unique such instance
- It doesn't tell us how to find it…
- …but we know how: compute fixpoint!

Example

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

Example

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

1. Fixpoint semantics:

- Start: $T_0 = \emptyset$; $t = 0$
  Repeat:
  $T_{t+1}(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T_t(z,y))$
  $t = t+1$
  Until $T_t = T_{t-1}$

Example

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

1. Fixpoint semantics:

- Start: $T_0 = \emptyset$; $t = 0$
  Repeat:
  $T_{t+1}(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T_t(z,y))$
  $t = t+1$
  Until $T_t = T_{t-1}$

2. Minimal model semantics: smallest T s.t.

- $\forall x \forall y\ [(R(x,y) \Rightarrow T(x,y)] \wedge$
  $\forall x \forall y \forall z\ [(R(x,z) \wedge T(z,y)) \Rightarrow T(x,y)]$

Datalog Semantics

- The fixpoint semantics tells us how to compute a datalog query
- The minimal model semantics is more declarative: only says what we get
- The two semantics are equivalent meaning: you get the same thing

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation

More Features

- Aggregates
- Grouping
- Negation

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

Aggregates

[aggregate name] <var> : { [relation to compute aggregate on] }

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

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

Assign variable to the value of the aggregate

Meaning (in SQL)

```
SELECT min(id) as minId
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)

Counting

```
Q(c) :- c = count : { Actor(_, y, _), y = 'John' }
```

No variable here!

Meaning (in SQL, assuming no NULLs)

```
SELECT count(*) as c
FROM Actor as a
WHERE a.name = 'John'
```

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

Grouping

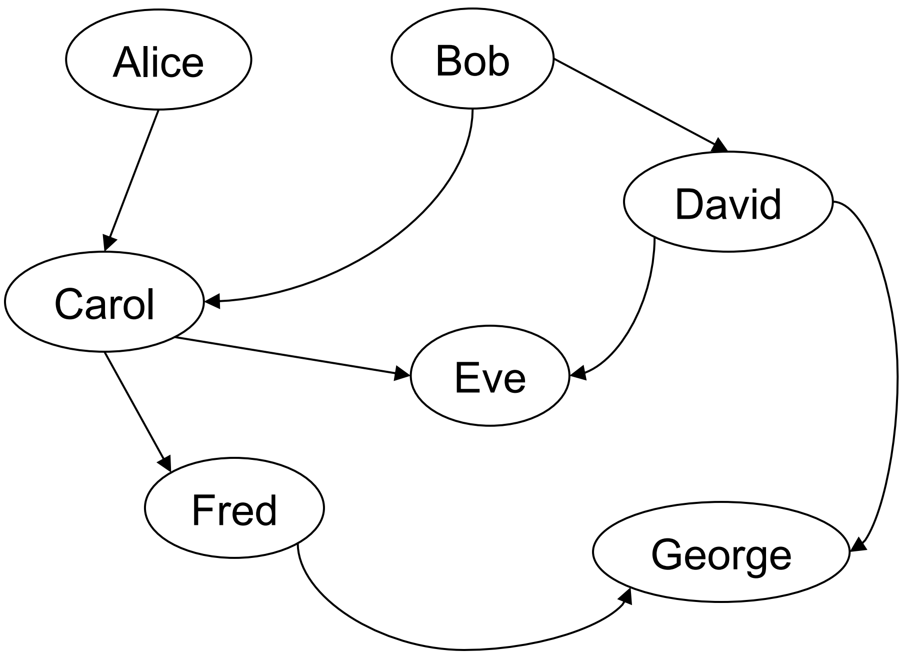
```
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }
```

Meaning (in SQL)

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

Examples

A genealogy database (parent/child)



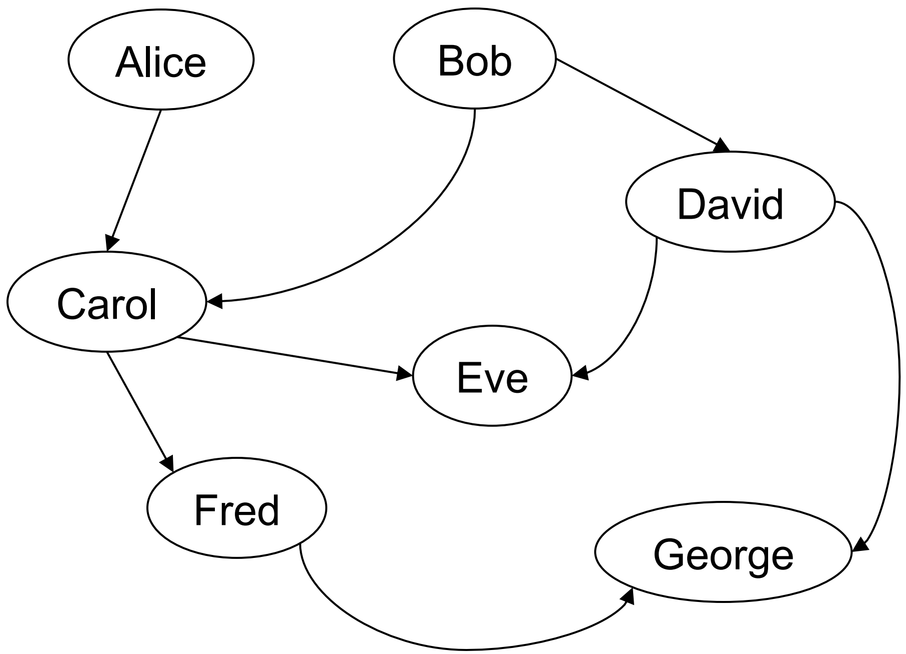
ParentChild

p	c
Alice	Carol
Bob	Carol
Bob	David
Carol	Eve
…	

ParentChild(p,c)

Count Descendants

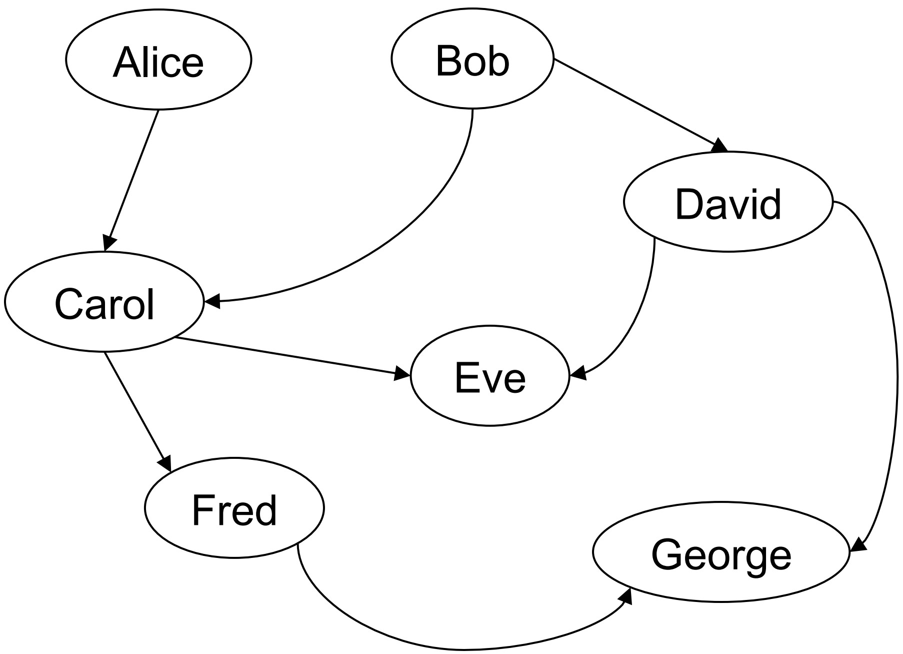
For each person, count his/her descendants



ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



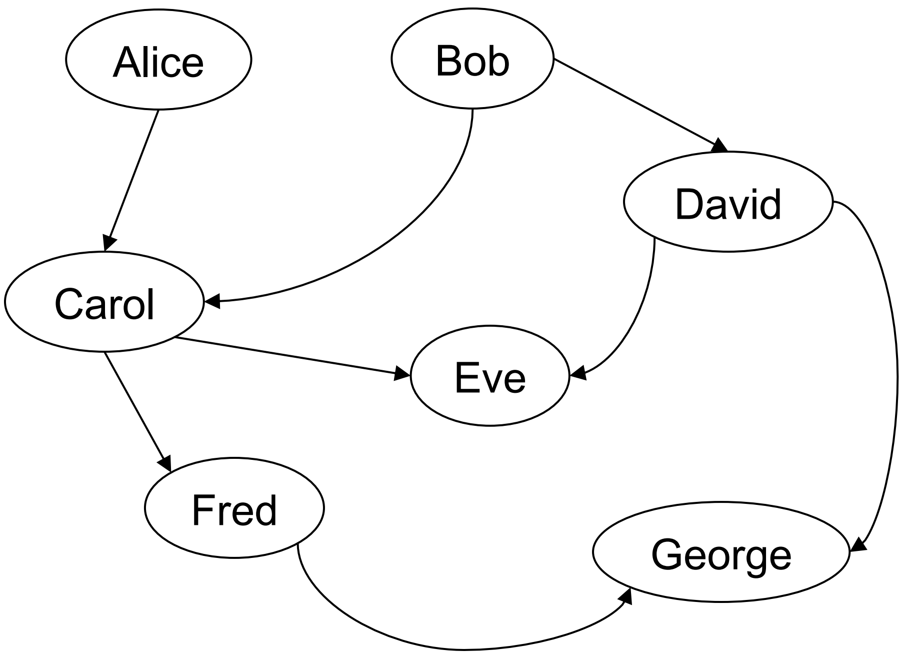
Answer

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?)

Count Descendants

For each person, compute the total number of descendants

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

Count Descendants

For each person, compute the total number of descendants

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

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
```

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,y) }.
```

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,y) }.
```

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,y) }.

// Find the number of descendants of Alice
```

Count Descendants

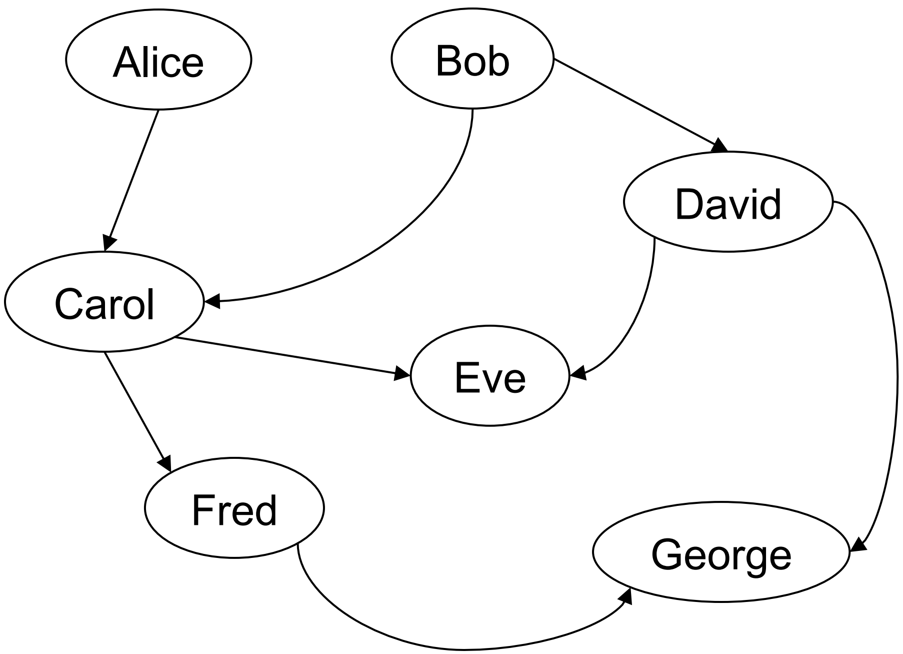
How many descendants does Alice have?

```
// 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,y) }.

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

ParentChild(p,c)

Negation: use “!”

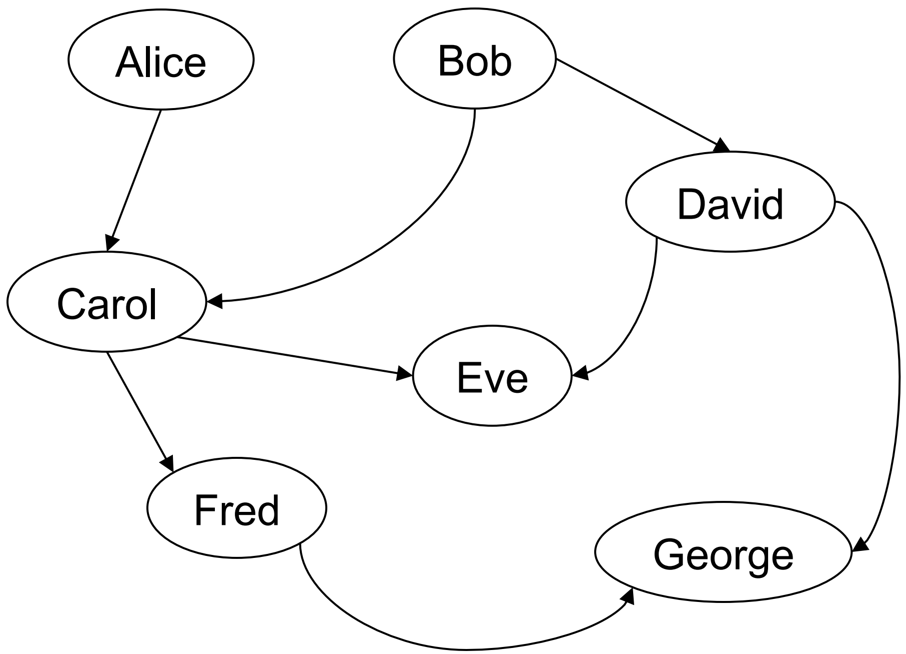
Find all descendants of Bob that are not descendants of Alice



ParentChild(p,c)

Negation: use “!”

Find all descendants of Bob that are not descendants of Alice



Answer

x
David

Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

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

Negation: use “!”

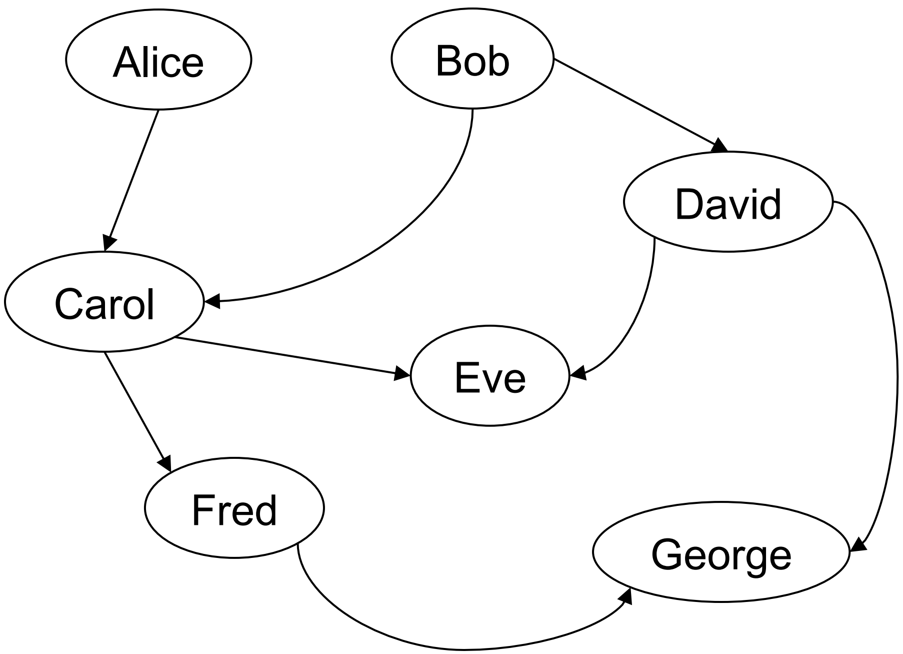
Find all descendants of Bob that are not descendants of Alice

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

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

ParentChild(p,c)

Same Generation

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



SG

p1	p2
Carol	David
Eve	George
Fred	George
Fred	Eve

Same Generation

Compute pairs of people at the same generation

```
// common parent
```

Same Generation

Compute pairs of people at the same generation

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

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?

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
```

Safe Datalog Rules

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

```
U1(x,y) :- ParentChild(“Alice”,x), y != “Bob”
```

```
U2(x) :- ParentChild(“Alice”,x), !ParentChild(x,y)
```

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

ParentChild(p,c)

Safe Datalog Rules

Holds for
every y other than “Bob”
U1 = infinite!

Here are *unsafe* datalog rules. What’s “unsafe” about them ?

```
U1(x,y) :- ParentChild(“Alice”,x), y != “Bob”
```

```
U2(x) :- ParentChild(“Alice”,x), !ParentChild(x,y)
```

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

Safe Datalog Rules

Holds for
every y other than "Bob"
U1 = infinite!

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

```
U1(x,y) :- ParentChild("Alice",x), y != "Bob"
```

```
U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
```

Want Alice's childless children,
but we get all children x (because
there exists some y that x is not parent of y)

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

ParentChild(p,c)

Safe Datalog Rules

Holds for
every y other than “Bob”
U1 = infinite!

Here are *unsafe* datalog rules. What’s “unsafe” about them ?

```
U1(x,y) :- ParentChild(“Alice”,x), y != “Bob”
```

```
U2(x) :- ParentChild(“Alice”,x), !ParentChild(x,y)
```

Want Alice’s childless children,
but we get all children x (because
there exists some y that x is not parent of y)

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

Unclear what y is

Safe Datalog Rules

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

```
U1(x,y) :- ParentChild("Alice",x), y != "Bob"
```

```
U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
```

A datalog rule is *safe* if every variable appears in some positive, non-aggregated relational atom

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

Making Rules Safe

Return pairs (x,y) where x is a child of Alice, and y is anybody

```
U1(x,y) :- ParentChild("Alice",x), y != "Bob"
```

Making Rules Safe

Return pairs (x,y) where x is a child of Alice, and y is anybody

```
U1(x,y) :- ParentChild(“Alice”,x), y != “Bob”
```

```
U1(x,y) :- ParentChild(“Alice”,x), Person(y), y != “Bob”
```

Making Rules Safe

Find Alice's children who don't have children.

```
U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
```

Making Rules Safe

Find Alice's children who don't have children.

```
U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
```

```
HasChildren(x) :- ParentChild(x,y)
U2(x) :- ParentChild("Alice",x), !HasChildren(x)
```

Making Rules Safe

Find the smallest Actor ID and his/her first name

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

Making Rules Safe

Find the smallest Actor ID and his/her first name

```
U3(minId, y) :- minId = min x : { Actor(x, y, _) }
```

```
U3(minId, y) :- minId = min x : { Actor(x, _, _) }, Actor(minId, y, _)
```

Stratified Datalog

- Recursion does not cope well with aggregates or negation
- Example: what does this mean?

```
A() :- !B().
B() :- !A().
```

- A datalog program is ***stratified*** if it can be partitioned into *strata*
  - Only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.

- Many Datalog DBMSs (including souffle) accepts only stratified Datalog.

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

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

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).
```

Stratum 1

```
T(p,c) :- D(p,_), c = count : { D(p,y) }.
Q(d) :- T(p,d), p = “Alice”.
```

Stratum 2

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

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

```
Q(x) :- D(“Alice”,x), !D(“Bob”,x).
```

Stratum 2

May use !D

```
A() :- !B().
B() :- !A().
```

Non-stratified

Cannot use !A

Stratified Datalog

- If we don't use aggregates or negation, then the Datalog program is already stratified
- If we do use aggregates or negation, it is usually quite natural to write the program in a stratified way

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation

Evaluation

Naïve evaluation: fixpoint semantics:

- At each iteration, compute a relational query
- Repeat until no more change

Semi-naïve evaluation

- Compute only *delta*'s at each iteration

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:

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

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

Background: Incremental View Maintenace

Example 1:

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

If R ← R ∪ΔR then what is ΔV(x,y) ?

ΔV(x,y) :- ΔR(x,z),S(z,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) ?

Background: Incremental View Maintenace

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 $\Delta V(x,y)$?

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

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 IDB 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 Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each IDB 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

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 ($\Delta T = \emptyset$)

then break

$T = T \cup \Delta T$

Endloop

Semi-naïve Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each IDB 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

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 Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each IDB 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

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

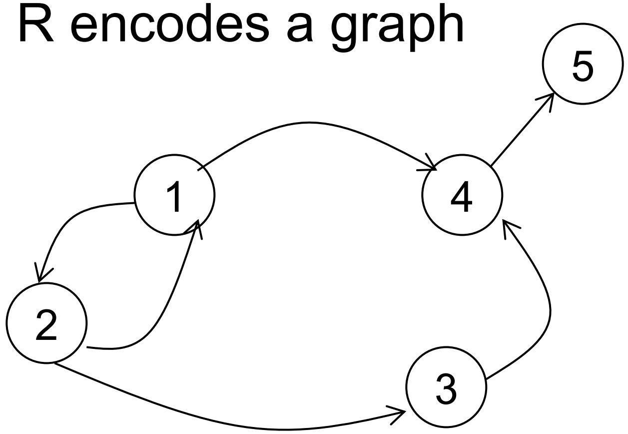
$T = T \cup \Delta T$

Endloop

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

Example

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=

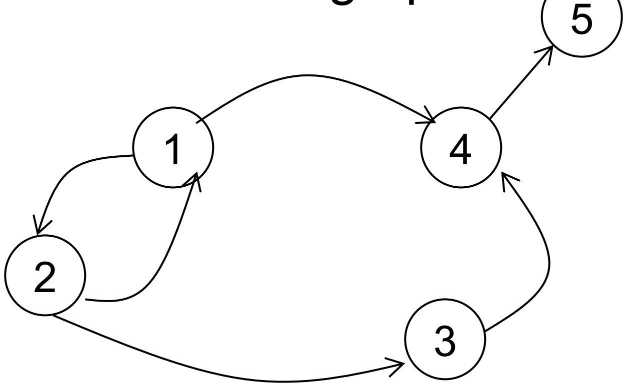
1	2
1	4
2	1
2	3
3	4
4	5

T=

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

Example

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

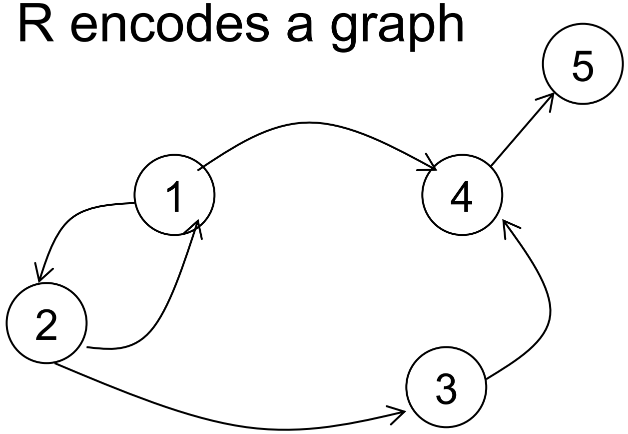
1	1
1	3
1	5
2	2
2	4
3	5

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

Example

R encodes a graph



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

```
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

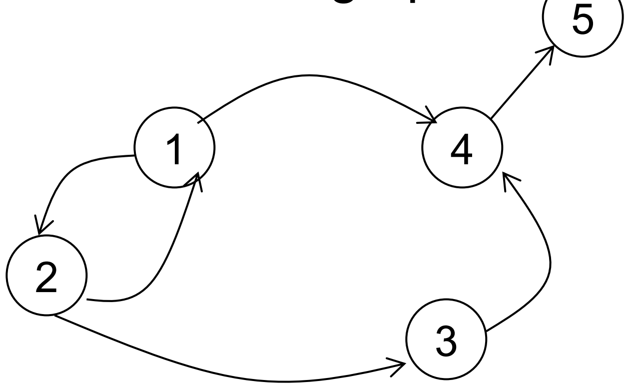
2	5

T=

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

Example

R encodes a graph



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

```
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

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