Introduction to Database Systems
CSE 414

Lecture 9: Datalog

Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
  - Data models, SQL, Relational Algebra, Datalog
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions

What is Datalog?

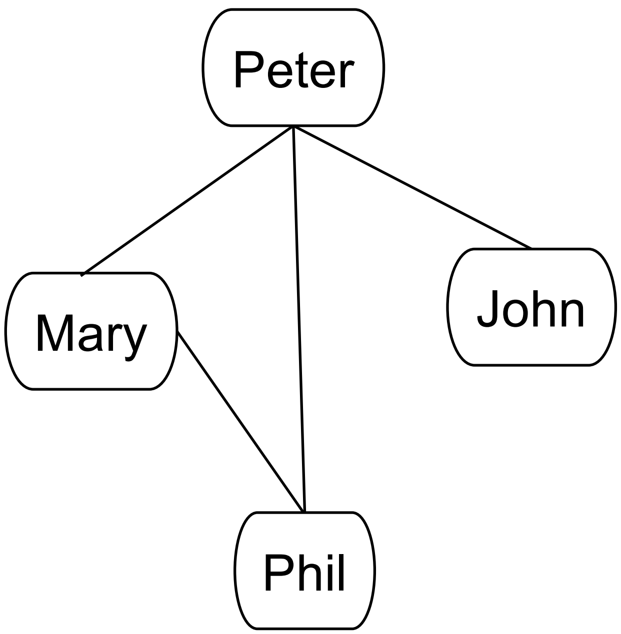
- Another query language for relational model
  - Designed in the 80's
  - Simple, concise, elegant
  - Extends relational queries with *recursion*
- Today is a hot topic:
  - Souffle (we will use in HW4)
  - Eve http://witheve.com/
  - Differential datalog https://github.com/frankmcsherry/differential-dataflow
  - Beyond databases in many research projects: network protocols, static program analysis

Soufflé

- Open-source implementation of Datalog DBMS
- Under active development
- Commercial implementations are available
  - More difficult to set up and use
- "sqlite" of Datalog
  - Set-based rather than bag-based
- Install in your VM
  - Run `sudo yum install souffle` in terminal
  - More details in upcoming HW4

Why bother with *yet* another relational query language?

Example: storing FB friends



As a graph

Or

Person1	Person2	is_friend
Peter	John	1
John	Mary	0
Mary	Phil	1
Phil	Peter	1
…	…	…

As a relation

We will learn the tradeoffs of different data models later this quarter

Compute your friends graph

p1	p2	isFriend
Peter	John	1
John	Mary	0
Mary	Phil	1
Phil	Peter	1
…	…	…

Friends(p1, p2, isFriend)

```
SELECT f.p2
FROM Friends as f
WHERE f.p1 = ‘me’ AND f.isFriend = 1
```

My own friends

```
SELECT f1.p2
FROM Friends as f1,
     (SELECT f.p2
      FROM Friends as f
      WHERE f.p1 = ‘me’ AND
      f.isFriend = 1) as f2
WHERE f1.p1 = f2.p2 AND
      f1.isFriend = 1
```

My FoF

My FoFoF… My FoFoFoF…

Datalog allows us to write *recursive queries* easily

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

Schema

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

```
.decl Actor(id:number, fname:symbol, lname:symbol)
.decl Casts(id:number, mid:number)
.decl Movie(id:number, name:symbol, year:number)

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).
```

Table declaration

Types in Souffle:
`number`
`symbol` (aka varchar)

Insert data

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.
```

SQL

```
SELECT name
FROM Movie
WHERE year = 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

id name year

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

Order of variable matters!

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(iDontCare,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(_,y,z), z=1940.
```

_ = "don't care" variables

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

```
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)

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

Datalog: Terminology

head — body

atom — atom — atom (aka subgoal)

```
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), ...
```

- `Ri(argsi)` called an *atom*, or a *relational predicate*
- `Ri(argsi)` 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`.
- Book uses AND instead of ,

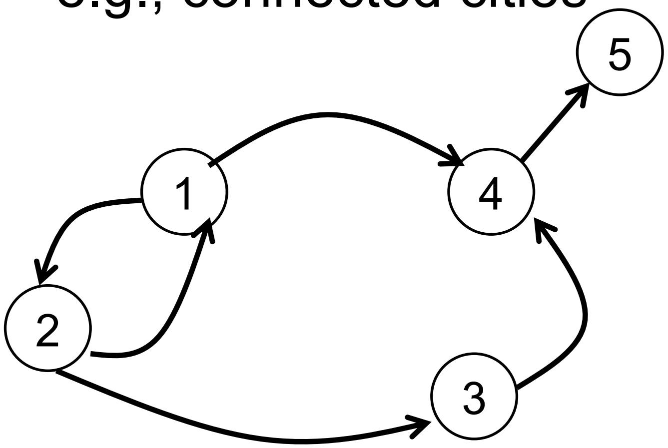
```
Q(args) :- R1(args) AND R2(args) ...
```

Datalog program

- A Datalog program consists of several rules
- Importantly, rules may be recursive!
  – Recall CSE 143!
- Usually there is one distinguished predicate that's the output
- We will show an example first, then give the general semantics.

Example

R encodes a graph
e.g., connected cities

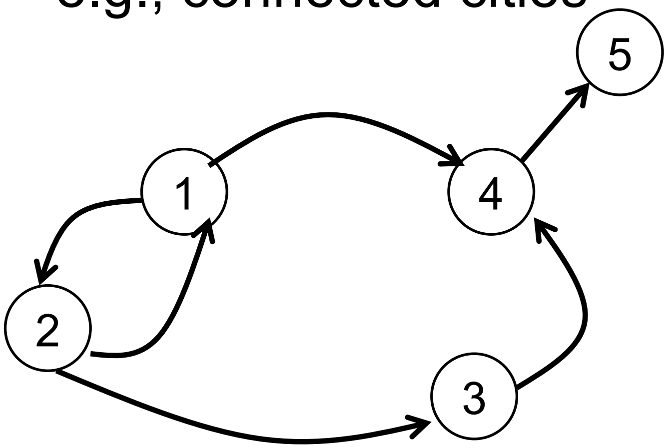


R=

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

Example

R encodes a graph
e.g., connected cities



R=

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

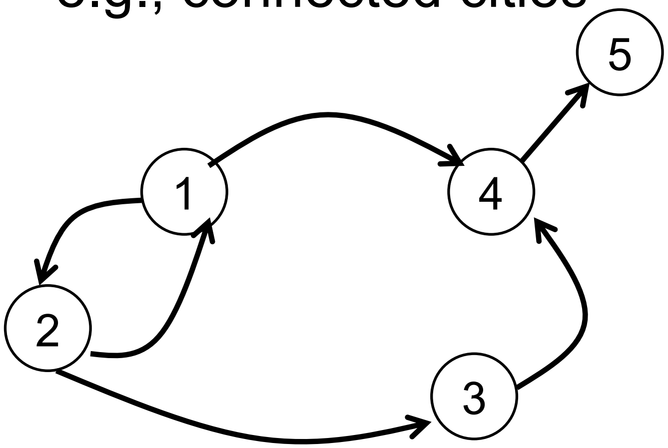
Multiple rules for the same IDB means OR

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

What does it compute?

Example

R encodes a graph
e.g., connected cities



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).
```

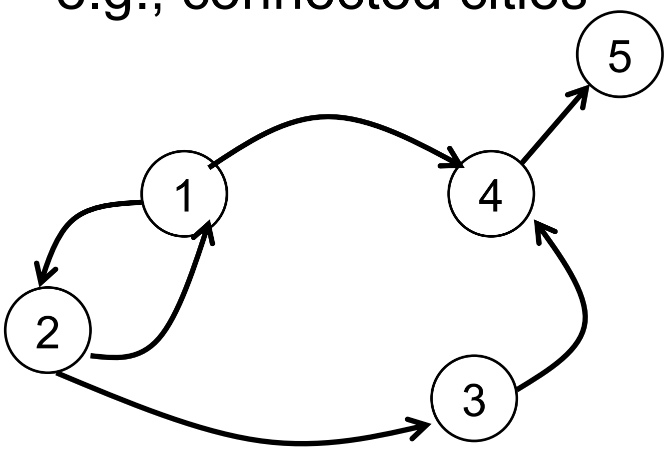
What does it compute?

Example

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

What does it compute?

R encodes a graph
e.g., connected cities



R=

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

Initially:
T is empty.

First iteration:
T =

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

First rule generates this

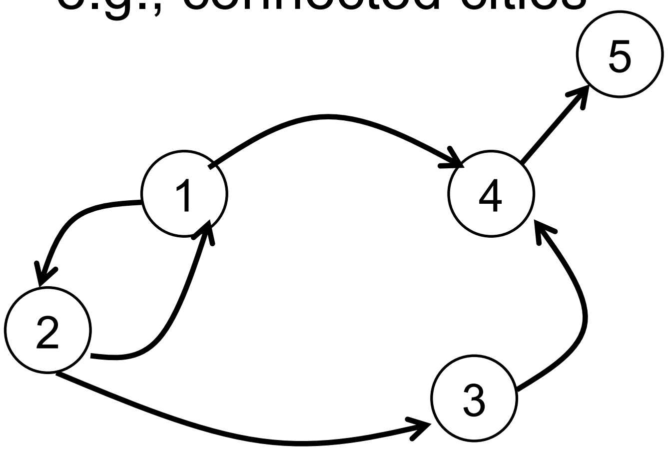
Second rule
generates nothing
(because T is empty)

Example

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

What does it compute?

R encodes a graph
e.g., connected cities



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

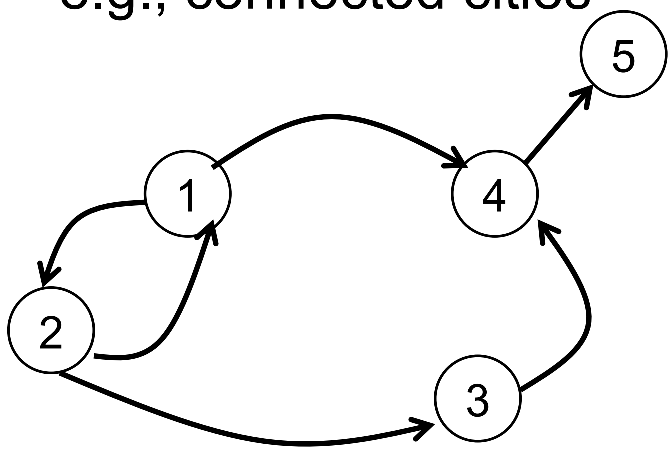
First rule generates this (rows 1–6)

Second rule generates this (rows 7–12)

New facts

Example

R encodes a graph
e.g., connected cities



R=

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

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

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

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 =

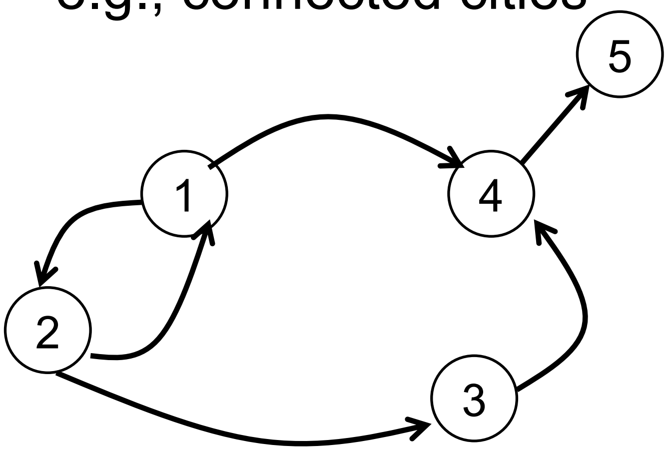
1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

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
e.g., connected cities



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

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

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)

Aggregates

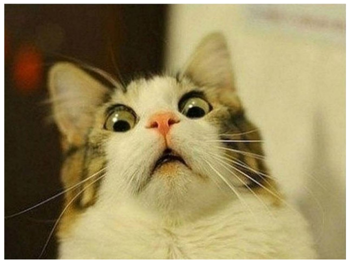
[aggregate name] <var> : { [relation to compute aggregate on] }

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

head

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

What does this even mean???



Can’t use variable that are not aggregated in the outer /head atoms

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
```

Example

For each person, compute the total number of descendants

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

ParentChild(p,c)

Example

For each person, compute the total number of descendants

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

ParentChild(p,c)

Example

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).
```

Example

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
```

Example

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) }.
```

Example

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) }.

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

Example

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) }.

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

Negation: use “!”

Find all descendants of Alice,
who are not descendants of Bob

```
// 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(“Alice”,x), !D(“Bob”,x).
```

ParentChild(p,c)

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)
```

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)
```

ParentChild(p,c)

Safe Datalog Rules

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

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

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

```
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)

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 relational atom

Stratified Datalog

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

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

- A datalog program is <u>*stratified*</u> 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