Introduction to Database Systems
CSE 414

Lecture 9: More Datalog

Announcements

- Midterm in class on Friday 5/4
  - You can bring one letter-size sheet of notes (can write on both sides)
  - Practice exams available on website
- Game plan:
  - HW3/WQ3: due next Tues 4/17
  - HW4/WQ4: due on 4/24
  - HW5/WQ5: due on 5/1
  - HW6: released on 5/4

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   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).
```

EDB

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

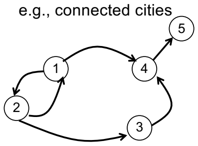
```
Q2(f,l) :- Actor(z,f,l), Casts(z,x),
           Movie(x,y,1940).
```

```
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

Example

R encodes a graph
e.g., connected cities

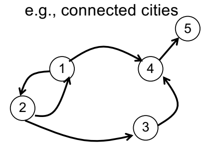


R=

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

Example

R encodes a graph
e.g., connected cities



Multiple rules for the same IDB means OR

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

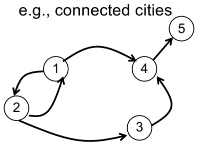
What does it compute?

R=

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

Example

R encodes a graph
e.g., connected cities



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

What does it compute?

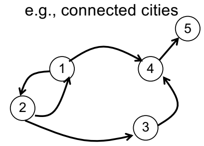
R=

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

Initially:
T is empty.

Example

R encodes a graph
e.g., connected cities



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

2 2 2 1 1 2

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

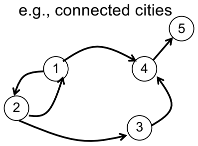
1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule generates nothing (because T is empty)

Example

R encodes a graph
e.g., connected cities



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

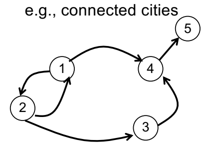
First rule generates this

Second rule generates this

New facts

Example

R encodes a graph
e.g., connected cities



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

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

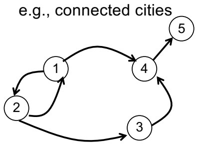
Both rules

First rule

Second rule

Example

R encodes a graph
e.g., connected cities



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

Datalog Semantics

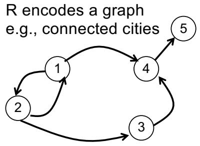
Fixpoint semantics

- 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:
  $\emptyset = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \ldots$
- It follows that a datalog program w/o functions (+, *, ...) always terminates. (Why?)

Three Equivalent Programs

R encodes a graph
e.g., connected cities



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?

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

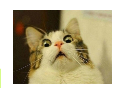
[aggregate name] <var> : { [relation to compute aggregate on] }


```
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
```

ParentChild(p,c)

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).
```


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).

// For each person, count the number of 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).
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) }.
```


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).

// 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
```

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).

// 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 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

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)

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)
```

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 ***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