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
Datalog: Facts and Rules

**Facts** = tuples in the database

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

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

Table declaration

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

Insert data
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
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
R encodes a graph e.g., connected cities

\[
R = \\
\begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]
R encodes a graph e.g., connected cities

\[
R = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

Example

\[
T(x,y) :\neg R(x,y).
\]

\[
T(x,y) :\neg R(x,z), T(z,y).
\]

Multiple rules for the same IDB means OR

What does it compute?
R encodes a graph e.g., connected cities

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially: T is empty.

What does it compute?

**Example**

\[
R = T(x,y) : - R(x,y), T(z,y) : - R(x,z),
\]

\[
T(x,y) : - R(x,y).
\]

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

Initially: T is empty.

First iteration: T =

First rule generates this

Second rule generates nothing (because T is empty)

What does it compute?

R encodes a graph e.g., connected cities

R =

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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

$T(x,y) :\neg R(x,y)$.  
$T(x,y) :\neg R(x,z), T(z,y)$.

First iteration:

$T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}$

Second iteration:

$T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
2 & 2 \\
1 & 3 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
\end{array}$

What does it compute?

$R$ encodes a graph e.g., connected cities

Initially: $T$ is empty.

New facts

First rule generates this

Second rule generates this
Example

R encodes a graph e.g., connected cities

\[
R = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

Initially:

\[T = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}\]

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

What does it compute?

1. First iteration:

\[T = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}\]

2. Second iteration:

\[T = \begin{array}{c|c|c|c|c}
1 & 2 & 2 & 1 \\
2 & 1 & 2 & 3 \\
1 & 4 & 3 & 4 \\
4 & 5 & 1 & 1 \\
1 & 1 & 2 & 2 \\
2 & 2 & 1 & 3 \\
1 & 3 & 2 & 4 \\
2 & 4 & 1 & 5 \\
3 & 5 & 3 & 5 \\
\end{array}\]

3. Third iteration:

\[T = \begin{array}{c|c|c|c|c|c}
1 & 2 & 2 & 1 \\
2 & 1 & 2 & 3 \\
1 & 4 & 3 & 4 \\
4 & 5 & 1 & 1 \\
1 & 1 & 2 & 2 \\
2 & 2 & 1 & 3 \\
1 & 3 & 2 & 4 \\
2 & 4 & 1 & 5 \\
3 & 5 & 3 & 5 \\
\end{array}\]

Both rules
First rule
Second rule
New fact
**Example**

\[ T(x,y) : R(x,y). \]
\[ T(x,y) : R(x,z), T(z,y). \]

R encodes a graph e.g., connected cities

**First iteration:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially: T is empty.

**Second iteration:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Third iteration:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fourth iteration:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What does it compute?

No new facts. DONE
Datalog Semantics

Fixpoint semantics

• Start:
  \[ \text{IDB}_0 = \text{empty relations} \]
  \[ t = 0 \]

Repeat:
  \[ \text{IDB}_{t+1} = \text{Compute Rules}(\text{EDB}, \text{IDB}_t) \]
  \[ t = t+1 \]

Until \[ \text{IDB}_t = \text{IDB}_{t-1} \]

• Remark: since rules are monotone:
  \[ \emptyset = \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \ldots \]

• It follows that a datalog program w/o functions (+, *, ...) always terminates. (Why?)
Three Equivalent Programs

R encodes a graph e.g., connected cities

\[
\begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

Question: which terminates in fewest iterations?

Right linear

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

Left linear

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

Non-linear

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

R =

CSE 414 - Spring 2018
More Features

- Aggregates
- Grouping
- Negation
Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

**Aggregates**

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

\[
\text{min} \ x : \{ \text{Actor}(x, y, _), y = 'John' \} \
\]

\[
Q(\text{minId}) :- \text{minId} = \text{min} \ x : \{ \text{Actor}(x, y, _), y = 'John' \} \
\]

**Meaning (in SQL)**

```
SELECT min(id) as minId
FROM Actor as a
WHERE a.name = 'John'
```

**Aggregates in Souffle:**

- count
- min
- max
- sum

Assign variable to the value of the aggregate
Aggregates

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

\[
\text{min } x : \{ \text{Actor}(x, y, _), y = 'John' \}
\]

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

What does this even mean???

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

For each person, compute the total number of descendants

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

For each person, compute the total number of descendants

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

\[
\begin{align*}
\text{D}(x,y) & : \text{ParentChild}(x,y). \\
\text{D}(x,z) & : \text{D}(x,y), \text{ParentChild}(y,z).
\end{align*}
\]

\[
\begin{align*}
\text{D}(p_,_), c = \text{count} : \{ \text{D}(p,y) \}.
\end{align*}
\]
Example

For each person, compute the total number of descendants

```prolog
// 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).
Here are *unsafe* datalog rules. What’s “unsafe” about them?

\[
\begin{align*}
U1(x,y) & : \text{ParentChild(“Alice”,x), } y \neq \text{“Bob”} \\
U2(x) & : \text{ParentChild(“Alice”,x), !ParentChild(x,y)}
\end{align*}
\]
Here are **unsafe** datalog rules. What’s “unsafe” about them?

\[
U_1(x,y) : \text{ParentChild}(\text{“Alice”},x), \ y \neq \text{“Bob”}
\]

\[
U_2(x) : \text{ParentChild}(\text{“Alice”},x), \neg \text{ParentChild}(x,y)
\]

Holds for every \(y\) other than “Bob”

\(U_1 = \text{infinite!}\)
Here are \textit{unsafe} datalog rules. What’s “unsafe” about them?

\begin{align*}
U1(x,y) & : \text{ParentChild(“Alice”,x), $y \neq “Bob”}$ \\
U2(x) & : \text{ParentChild(“Alice”,x), !ParentChild(x,y)}
\end{align*}

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

Holds for every $y$ other than “Bob”

$U1 = \text{infinite!}$
Safe Datalog Rules

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

\[ U1(x,y) : \text{ParentChild("Alice",x)}, y \neq "Bob" \]

\[ U2(x) : \text{ParentChild("Alice",x)}, \neg \text{ParentChild(x,y)} \]

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?

\[
\begin{align*}
A() & : - !B(). \\
B() & : - !A().
\end{align*}
\]

• 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

\begin{align*}
D(x,y) & :\text{ ParentChild}(x,y). \\
D(x,z) & : D(x,y), \text{ ParentChild}(y,z). \\
T(p,c) & : D(p,\_), c = \text{ count} : \{ D(p,y) \}. \\
Q(d) & : T(p,d), p = "Alice". \\
\end{align*}

Stratum 1

Stratum 2

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

Stratum 1

\[ D(x,y) : \neg \text{ParentChild}(x,y). \]
\[ D(x,z) : \neg D(x,y), \ \text{ParentChild}(y,z). \]
\[ T(p,c) : \neg D(p,_), \ c = \text{count} : \{ D(p,y) \}. \]
\[ Q(d) : \neg T(p,d), \ p = \text{“Alice”}. \]

Stratum 2

\[ D(x,y) : \neg \text{ParentChild}(x,y). \]
\[ D(x,z) : \neg D(x,y), \ \text{ParentChild}(y,z). \]
\[ Q(x) : \neg D(\text{“Alice”}, x), \ \neg D(\text{“Bob”}, x). \]

Non-stratified

\[ A() : \neg !B(). \]
\[ B() : \neg !A(). \]
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