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
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
Datalog: Facts and Rules

**Facts** = tuples in the database

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

**Rules** = queries
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)
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(29445, ‘Ave Maria’, 1940).

**Rules** = queries

Q1(y) :- Movie(x,y,z), z=’1940’.

Find Movies made in 1940
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, I) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
```
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’).

Find Actors who acted in Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

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

**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)
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.
Datalog: Facts and Rules

**Facts** = tuples in the database

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

**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

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), ....

• $R_i(\text{args}_i)$ 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), ....

- $R_i(\text{args}_i)$ called an **atom**, or a **relational predicate**
- $R_i(\text{args}_i)$ evaluates to true when relation $R_i$ contains the tuple described by $\text{args}_i$.
  - Example: Actor(344759, ‘Douglas’, ‘Fowley’) is true
- In addition we can also have arithmetic predicates
  - Example: $z > ‘1940’$. 
More Datalog Terminology

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

\[ Q(\text{args}) \leftarrow R1(\text{args}), R2(\text{args}), \ldots \]
More Datalog Terminology

• $R_i(\text{args}_i)$ called an *atom*, or a *relational predicate*
• $R_i(\text{args}_i)$ evaluates to true when relation $R_i$ contains the tuple described by $\text{args}_i$.
  – Example: Actor(344759, ‘Douglas’, ‘Fowley’) is true
• In addition we can also have arithmetic predicates
  – Example: $z > ‘1940’$.
• Some systems use $\leftarrow$
• Some use AND
Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement!

\[ Q1(y) : - \text{Movie}(x,y,z), \text{z}='1940'. \]
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

\[ Q1(y) :- \text{Movie}(x,y,z), z='1940'. \]

• If \((x,y,z) \in \text{Movies}\) and \(z = '1940'\) then \(y\) is in answer
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!
  
  \[
  Q1(y) :\text{-} \quad \text{Movie}(x,y,z), \; z='1940'.
  \]

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

\[
\forall x \forall y \forall z \; [(\text{Movie}(x,y,z) \text{ and } z='1940') \Rightarrow Q1(y)]
\]
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

\[ Q1(y) : - \text{Movie}(x,y,z), z='1940'. \]

• If \((x,y,z) \in \text{Movies}\) and \(z = '1940'\) then \(y\) is in answer

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

• We want smallest answer with this property (why?)
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!
  
  \[
  Q1(y) : \neg \text{Movie}(x,y,z), z='1940'.
  \]

• If \((x,y,z) \in \text{Movies} \text{ and } z = '1940'\) then \(y\) is in answer

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

• We want \textit{smallest} answer with this property (why?)

• Logically equivalent:

  \[
  \forall y \ [(\exists x \exists z \text{Movie}(x,y,z) \text{ and } z='1940') \Rightarrow Q1(y)]
  \]
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

\[ Q1(y) : - \text{Movie}(x,y,z), z='1940'. \]

• If \((x,y,z) \in \text{Movies} \) and \(z = '1940'\) then \(y\) is in answer

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

• We want \text{smallest} answer with this property (why?)

• Logically equivalent:

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

• 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

\[
R = \\
\begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]
**Example**

R encodes a graph

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

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

What does it compute?
### Example

R encodes a graph

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

R =

Initially:
T is empty.

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

What does it compute?
Example

What does it compute?

R encodes a graph

Initially: T is empty.

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

First iteration:
T =

First rule generates this

Second rule generates nothing (because T is empty)
Example

R encodes a graph

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>

Initially: T is empty.

First iteration:

T =

<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>

Second iteration:

T =

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

First rule generates this
Second rule generates this

What does it compute?

R encodes a graph

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

R encodes a graph

R =

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<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>

Initially: T is empty.

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

First iteration:
\[ T = \]

Second iteration:
\[ T = \]

Third iteration:
\[ T = \]

What does it compute?

Both rules
First rule
Second rule
New fact
Example

R encodes a graph

Initially:
T is empty.

First iteration:
T =

Second iteration:
T =

Third iteration:
T =

Fourth iteration
T = (same)

No new facts. DONE

What does it compute?

R encodes a graph

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

R=

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<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>

No new facts. DONE
Three Equivalent Programs

R encodes a graph

<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>

R =

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

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

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

Question: which terminates in fewest iterations?

Right linear

Left linear

Non-linear
Outline

• Datalog rules
• Recursion
• Semantics
  • Negation, aggregates, stratification
  • Naïve and Semi-naïve Evaluation
1. 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 $\text{IDB}_t = \text{IDB}_{t-1}$
1. Fixpoint Semantics

- Start: \( \text{IDB}_0 = \) empty relations; \( t = 0 \)
  Repeat:
  \[
  \text{IDB}_{t+1} = \text{Compute Rules(E DB, IDB}_t) \\
  t = t+1
  \]
  Until \( \text{IDB}_t = \text{IDB}_{t-1} \)

- Remark: since rules are monotone:
  \( \emptyset = \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \ldots \)
1. Fixpoint Semantics

- Start: $\text{IDB}_0 =$ empty relations; $t = 0$
  Repeat:
  $\text{IDB}_{t+1} =$ Compute Rules($\text{EDB}$, $\text{IDB}_t$)
  $t = t+1$
  Until $\text{IDB}_t = \text{IDB}_{t-1}$

- Remark: since rules are monotone:
  $\emptyset = \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq ...$

- 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,
     \[ \forall \text{vars} \ [(\text{Body}(\text{EDB}, \text{IDB}) \Rightarrow \text{Head}(\text{IDB})] \]
  2) Is the smallest IDB satisfying (1)
2. Minimal Model Semantics:

• Find some IDB instance that satisfies:
  1) For every rule,
     \[ \forall \text{vars} \ [(\text{Body(EDB,IDB)} \Rightarrow \text{Head(IDB)})] \]
  2) Is the smallest IDB satisfying (1)
2. Minimal Model Semantics:

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

• **Theorem**: there exists a unique such instance
2. Minimal Model Semantics:

- Find some IDB instance that satisfies:
  1) For every rule,
     \[\forall \text{vars} \ [(\text{Body(EDB,IDB)} \Rightarrow \text{Head(IDB)})\]
  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:

- Find some IDB instance that satisfies:
  1) For every rule,
     \[ \forall \text{vars} \ [(\text{Body}(\text{EDB},\text{IDB}) \Rightarrow \text{Head}(\text{IDB})] \]
  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

\[ T(x,y) : R(x,y) \]
\[ T(x,y) : R(x,z), T(z,y) \]
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 $T_t = T_{t-1}$

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

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

2. Minimal model semantics: smallest $T$ s.t.
   • $\forall x \forall y [(R(x,y) \Rightarrow T(x,y)] \land$
     $\forall x \forall y \forall z [(R(x,z) \land T(z,y)) \Rightarrow T(x,y)]$
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
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
Counting

Q(c) :- c = \text{count} : \{ \text{Actor}(\_, \_, \_), \text{y} = \text{‘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) :- Movie(_, _, y), c = count : { Movie(_, _, y) }

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```
A genealogy database (parent/child)

ParentChild

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>David</td>
</tr>
<tr>
<td>Carol</td>
<td>Eve</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

- Alice
- Bob
- Carol
- David
- Eve
- Fred
- George

\[ \text{ParentChild}(p,c) \]
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>3</td>
</tr>
<tr>
<td>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>3</td>
</tr>
<tr>
<td>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>

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) }.
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”.
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

Answer

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
</tr>
</tbody>
</table>
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).
Same Generation

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

ParentChild(p, c)

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>David</td>
</tr>
<tr>
<td>Eve</td>
<td>George</td>
</tr>
<tr>
<td>Fred</td>
<td>George</td>
</tr>
<tr>
<td>Fred</td>
<td>Eve</td>
</tr>
</tbody>
</table>
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)
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?

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

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

\[
U3(\text{minId}, y) :- \text{minId} = \min x : \{ \text{Actor(x, y, _)} \}
\]
Safe Datalog Rules

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

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

\[ \text{U2}(x) : \text{ParentChild}(“Alice”, x), \; !\text{ParentChild}(x, y) \]

\[ \text{U3}(\text{minId}, y) : \text{minId} = \min x : \{ \text{Actor}(x, y, _) \} \]
Here are *unsafe* datalog rules. What’s “unsafe” about them?

\[
\begin{align*}
U_1(x,y) & : \neg \text{ParentChild}(\text{“Alice”}, x), \ y \neq \text{“Bob”} \\
U_2(x) & : \neg \text{ParentChild}(\text{“Alice”}, x), \ \neg \text{ParentChild}(x, y) \\
U_3(\text{minId}, y) & : \text{minId} = \min x : \{ \text{Actor}(x, y, _) \}
\end{align*}
\]
Safe Datalog Rules

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

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

\[ \text{U2}(x) : - \text{ParentChild}(“Alice”,x), \ !\text{ParentChild}(x,y) \]

\[ \text{U3}(\text{minId}, y) : - \text{minId} = \min x : \{ \text{Actor}(x, y, _) \} \]

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

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

Unclear what y is
Here are *unsafe* datalog rules. What’s “unsafe” about them?

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

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

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

\[ U3(\text{minId}, y) : \text{minId} = \min x : \{ \text{Actor}(x, y, _) \} \]
Making Rules Safe

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

\[
U1(x,y) :\text{ ParentChild(“Alice”,x), } y \neq \text{ “Bob”}
\]
Return pairs \((x,y)\) where \(x\) is a child of Alice, and \(y\) is anybody

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

\[
U1(x,y) :\text{ ParentChild(“Alice”,} x\text{), Person}(y)\text{, } y \neq \text{“Bob”}
\]
Making Rules Safe

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

\[
U2(x) :- \text{ParentChild}("Alice",x), \neg \text{ParentChild}(x,y)
\]
Find Alice’s children who don’t have children.

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

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

Find the smallest Actor ID and his/her first name

\[ U3(\text{minId}, y) :\ - \text{minId} = \min x : \{ \text{Actor}(x, y, _) \} \]
Making Rules Safe

Find the smallest Actor ID and his/her first name

\[ \text{U3(minId, y)} ::= \text{minId} = \text{min x : \{ Actor(x, y, _) \}} \]

\[ \text{U3(minId, y)} ::= \text{minId} = \text{min x : \{ Actor(x, _, _) \}}, \text{Actor(minId, y, _)} \]
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

<table>
<thead>
<tr>
<th>D(x,y) :- ParentChild(x,y).</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(x,z) :- D(x,y), ParentChild(y,z).</td>
</tr>
<tr>
<td>T(p,c) :- D(p,_), c = count : { D(p,y) }.</td>
</tr>
<tr>
<td>Q(d) :- T(p,d), p = “Alice”.</td>
</tr>
</tbody>
</table>

Stratum 1

Stratum 2

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

**Stratum 1**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(x, y) :- ParentChild(x, y).</td>
<td></td>
</tr>
<tr>
<td>D(x, z) :- D(x, y), ParentChild(y, z).</td>
<td></td>
</tr>
</tbody>
</table>

**Stratum 2**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(p, c) :- D(p, _), c = \text{count} : { D(p, y) }.</td>
<td>May use ( D ) in an agg since it was defined in previous stratum.</td>
</tr>
<tr>
<td>Q(d) :- T(p, d), p = “Alice”.</td>
<td></td>
</tr>
</tbody>
</table>

**Non-stratified**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A() :- !B().</td>
<td>May use !D</td>
</tr>
<tr>
<td>B() :- !A().</td>
<td>Cannot use !A</td>
</tr>
</tbody>
</table>

D(\ldots) : Primary predicate
ParentChild(\ldots) : Secondary predicate
T(\ldots) : Aggregation predicate
Q(\ldots) : Query predicate
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 Maintenance

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

\[ V \leftarrow \text{body} \]

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

• Then the view is also modified as follows: \( V \leftarrow V \cup \Delta V \)

**Incremental view maintenance:** Compute \( \Delta V \) without having to recompute \( V \)
Background: Incremental View Maintenance

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 Maintenance

Example 1:

\[ V(x,y) : \leftarrow R(x,z), S(z,y) \]

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

\[ \Delta V(x,y) : \leftarrow \Delta R(x,z), S(z,y) \]
Background: Incremental View Maintenance

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 \( \Delta V(x,y) \)?
Background: Incremental View Maintenance

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 Maintenance

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 Maintenance

Example 3:

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

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

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

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

$\Delta V(x,y) :- \Delta T(x,z), \Delta T(z,y)$
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$_i$.

\[
P_1 = \Delta P_1 = \text{non-recursive-}SPJU_1, \\
P_2 = \Delta P_2 = \text{non-recursive-}SPJU_2, \\
\ldots
\]

Loop

\[
\Delta P_1 = \Delta SPJU_1 - P_1; \quad \Delta P_2 = \Delta SPJU_2 - P_2; \quad \ldots
\]

if \((\Delta P_1 = \emptyset \text{ and } \Delta P_2 = \emptyset \text{ and } \ldots)\) then break

\[
P_1 = P_1 \cup \Delta P_1; \quad P_2 = P_2 \cup \Delta P_2; \quad \ldots
\]

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$_i$.

$P_1 = \Delta P_1 = \text{non-recursive-SPJU}_1$, $P_2 = \Delta P_2 = \text{non-recursive-SPJU}_2$, …

Loop

$\Delta P_1 = \Delta \text{SPJU}_1 - P_1$; $\Delta P_2 = \Delta \text{SPJU}_2 - P_2$; …

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 $\Delta$-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$_i$.

$P_1 = \Delta P_1 = \text{non-recursive-}SPJU_1$, $P_2 = \Delta P_2 = \text{non-recursive-}SPJU_2$, ...

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:

$T(x,y) \leftarrow R(x,y)$
$T(x,y) \leftarrow R(x,z), T(z,y)$

$T(x,y) = R(x,y)$, $\Delta T(x,y) = R(x,y)$

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-)$SPJU_i$.

$P_1 = \Delta P_1 = \text{non-recursive-}SPJU_1$, $P_2 = \Delta P_2 = \text{non-recursive-}SPJU_2$, …

**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:**

$T(x,y) \leftarrow R(x,y)$

$T(x,y) \leftarrow R(x,z), T(z,y)$

$T(x,y) = R(x,y), \Delta T(x,y) = R(x,y)$

**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 $\Delta$-rule for each rule!
R encodes a graph

Initially:

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

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

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

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

---

Example

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

\[
\begin{array}{c|c}
T= R, \Delta T = R \\
\text{Loop} \\
\Delta T(x,y) = R(x,z), \Delta T(z,y) \rightarrow R(x,y) \\
\text{if} (\Delta T = \emptyset) \text{ then break} \\
T = T \cup \Delta T \\
\text{Endloop}
\end{array}
\]
Example

R encodes a graph

Initially:

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

First iteration:

\[ T = R, \Delta T = R \]

Loop

\[ \Delta T(x,y) = R(x,z), \Delta T(z,y) \rightarrow R(x,y) \]

if \( \Delta T = \emptyset \)

then break

\[ T = T \cup \Delta T \]

Endloop
Example

R encodes a graph

Initially:

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

First iteration:

Second iteration:

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
Example

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

Initially:

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

\[ \Delta T = \text{paths of length } 2 \]

First iteration:

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

Second iteration:

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

Third iteration:

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

\[ \Delta T = \text{paths of length } 3 \]

\[ \Delta T = \text{paths of length } 4 \]

R encodes a graph

Loop

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

if \( \Delta T = \emptyset \) then break

\[ T = T \cup \Delta T \]

Endloop
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