DATA516/CSED516
Scalable Data Systems and Algorithms
Lecture 7
Datalog
Announcements

• HW4 is posted: 3 mini-homeworks

• Project Milestone due on Nov. 26

• Last three lectures:
  – Nov. 23 – last regular lecture
  – Nov. 30 – meetings to discuss your project
  – Dec. 07 – project presentations
Outline

• Recap: Datalog basics
• Naïve Evaluation Algorithm
• Monotone Queries
• Non-monotone Extensions
Datalog program

• A datalog program = several rules

• Rules may be recursive

• Set semantics only
Processing Graphs in Datalog

Graph

```
1 -- 2
   |  |
   v  v
3 -- 4
   |  |
   v  v
2 -- 5
```

Pattern Matching

```
R=
```

<table>
<thead>
<tr>
<th>src</th>
<th>dst</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>3</td>
</tr>
</tbody>
</table>
Processing Graphs in Datalog

Graph

Pattern Matching

Answer(x,y,z) :- R(x,y), R(x,z), R(y,z)
R encodes a graph

Descendants of node 2

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>
Example

R encodes a graph

Descendants of node 2

\[
D(x) :- R(2, x)
\]

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

R encodes a graph

Descendants of node 2

Recursive rule

\[
D(x) : \neg R(2, x) \\
D(y) : \neg D(x), R(x, y)
\]

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>
Example

R encodes a graph

Descendants of node 2

\[
\text{D(x)} : \text{R}(2, x) \\
\text{D(y)} : \text{D(x)}, \text{R}(x, y)
\]

How recursion works in datalog:
Initially D = empty

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

R encodes a graph

Descendants of node 2

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

How recursion works in datalog:
Initially D = empty
• Compute both rules:

<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>
Example

R encodes a graph

Descendants of node 2

\[ D(x) :- R(2, x) \]
\[ D(y) :- D(x), R(x, y) \]

Recursive rule

How recursion works in datalog:
Initially \( D = \text{empty} \)
- Compute both rules:
  \( \text{...now } D = \{1,3\} \)
Example

R encodes a graph

Descendants of node 2

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>

How recursion works in datalog:
Initially D = empty
- Compute both rules:
  - \( D(x) :- R(2, x) \)
  - \( D(y) :- D(x), R(x, y) \)
  
  ...now D = \{1,3\}
- Compute both rules:
Example

R encodes a graph

Descendants of node 2

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

How recursion works in datalog:
Initially D = empty
• Compute both rules:
  ...now D = \{1,3\}
• Compute both rules:
  ...now D = \{1,3,2,4\}
Example

R encodes a graph

Descendants of node 2

\[ D(x) :- R(2, x) \]
\[ D(y) :- D(x), R(x,y) \]

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  ...now D = \{1,3\}
- Compute both rules:
  ...now D = \{1,3,2,4\}
- Compute both rules:

<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>
Example

R encodes a graph

Descendants of node 2

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

How recursion works in datalog:
Initially D = empty

• Compute both rules:
  ...now D = \{1,3\}

• Compute both rules:
  ...now D = \{1,3,2,4\}

• Compute both rules:
  ...now D = \{1,3,2,4,5\}
Example

R encodes a graph

Descendants of node 2

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

How recursion works in datalog:
Initially D = empty
- Compute both rules:
  ...now D = {1,3}
- Compute both rules:
  ...now D = {1,3,2,4}
- Compute both rules:
  ...now D = {1,3,2,4,5}
- Compute both rules:
  ...nothing new. STOP

R =

<table>
<thead>
<tr>
<th></th>
<th>2</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>
Outline

• Recap: Datalog basics
  • Naïve Evaluation Algorithm
• Monotone Queries
• Non-monotone Extensions
Naïve Evaluation Algorithm

• Every rule $\rightarrow$ SPJ* query

*SPJ = select-project-join
*USPJ = union-select-project-join
Naïve Evaluation Algorithm

• Every rule \( \rightarrow \) SPJ* query

\[
T(x,z) : \neg R(x,y), T(y,z), \neg C(y,'green')
\]

*SPJ = select-project-join
*USPJ = union-select-project-join
Naïve Evaluation Algorithm

• Every rule $\rightarrow$ SPJ* query

$T(x,z) : R(x,y), T(y,z), C(y, 'green')$
Naïve Evaluation Algorithm

- Every rule $\rightarrow$ SPJ* query

  $T(x,z) :- R(x,y), T(y,z), C(y,'green')$

- Multiple rules same head $\rightarrow$ USPJ+

  $T(x,y) :- \ldots$

  $T(x,y) :- \ldots$

  $\ldots$

*SPJ = select-project-join
*USPJ = union-select-project-join
Naïve Evaluation Algorithm

- Every rule $\rightarrow$ SPJ* query

$$T(x,z) :\ R(x,y), \ T(y,z), \ C(y,’green’)$$

- Multiple rules same head $\rightarrow$ USPJ+

$$T(x,y) :\ ...$$
$$T(x,y) :\ ...$$
$$...$$

- Naïve Algorithm:

$$\textit{IDBs} := \emptyset$$
$$\textbf{repeat} \quad \textit{IDBs} := \textit{USPJs}$$
$$\textbf{until} \quad \text{no more change}$$

*SPJ = select-project-join
*USPJ = union-select-project-join
Naïve Evaluation Algorithm

\[ D(x) :- R(2,x) \]
\[ D(y) :- D(x), R(x,y) \]
Naïve Evaluation Algorithm

\[ D(x) : \text{:- } R(2,x) \]
\[ D(y) : \text{:- } D(x), R(x,y) \]

\[ \Pi_{R.\text{dst}}(\sigma_{R.\text{src}=2}(R)) \]
Naïve Evaluation Algorithm

\[ D(x) : - R(2,x) \]
\[ D(y) : - D(x), R(x,y) \]

\[ \Pi_{R.dst} (\sigma_{R.src=2}(R)) \cup \Pi_{R.dst} (D \bowtie_{D.node=R.src} R); \]
Naïve Evaluation Algorithm

\[ \text{D}(x) : \text{R}(2, x) \]
\[ \text{D}(y) : \text{D}(x), \text{R}(x, y) \]

\[ \Pi_{R \text{.} \text{dst}}(\sigma_{R \text{.} \text{src}=2}(R)) \cup \Pi_{R \text{.} \text{dst}}(D \bowtie_{D \text{.} \text{node}=R \text{.} \text{src}} R); \]
Naïve Evaluation Algorithm

\[ D(x) :\text{ } R(2,x) \]
\[ D(y) :\text{ } D(x),R(x,y) \]

\[ D := \emptyset; \]

\textbf{repeat} \hspace{1cm} \textbf{until} [no more change]

\[ D := \Pi_{R.\text{dst}}(\sigma_{R.\text{src}=2}(R)) \cup \Pi_{R.\text{dst}}(D \bowtie_{D.\text{node}=R.\text{src}} R); \]
Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database
Example

R encodes a graph

R =

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

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

What does it compute?
Example

R encodes a graph

Initially:
T is empty.

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

What does it compute?

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

Initially:
T is empty.

R encodes a graph

First iteration:
T =

First rule generates this

Second rule generates nothing (because T is empty)

What does it compute?

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

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

R encodes a graph

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

Initially: T is empty.

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

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

First rule generates this
Second rule generates this

New facts
Example

R encodes a graph

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

Initially: \( T \) is empty.

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

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

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

What does it compute?

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

R encodes a graph

First rule

Second rule

Both rules

New fact
Example

R encodes a graph

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

Initially:
T is empty.

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

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

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

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

What does it compute?

No new facts.

DONE
**Example**

R encodes a graph

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

Initially: T is empty.

\[ T = \]

**First iteration:**

\[ T = \]

**Second iteration:**

\[ T = \]

**Third iteration:**

\[ T = \]

**Fourth iteration:**

\[ T = \]

Iteration k computes pairs (x,y) connected by path of length ≤ k

What does it compute?

No new facts.

DONE
Three Equivalent Programs

R encodes a graph

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

Right linear

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

R encodes a graph

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

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)

Right linear
Left linear
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>

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

R encodes a graph

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

Question: how many iterations does each require?

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)
Three Equivalent Programs

R encodes a graph

R =

<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>

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: how many iterations does each require?

Right linear

Left linear

Non-linear

#iterations = diameter

#iterations = log(diameter)
Multiple IDBs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a path of even length

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

Find pairs of nodes \((x,y)\) connected by a path of even length

**R encodes a graph**

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

\(\text{Odd}(x,y) \iff \text{R}(x,y)\)
Multiple IDBs

R encodes a graph

Find pairs of nodes (x,y) connected by a path of even length

Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z), R(z,y)

<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>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></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>
Multiple IDBs

R encodes a graph

Find pairs of nodes \((x, y)\) connected by a path of \textit{even} length

Odd\((x, y)\) :- R\((x, y)\)

Even\((x, y)\) :- Odd\((x, z)\), R\((z, y)\)

Odd\((x, y)\) :- Even\((x, z)\), R\((z, y)\)

R =

<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></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Multiple IDBs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a path of *even* length

\[
\begin{align*}
Odd(x,y) & : - R(x,y) \\
Even(x,y) & : - Odd(x,z), R(z,y) \\
Odd(x,y) & : - Even(x,z), R(z,y)
\end{align*}
\]

Two IDBs: Odd(x,y) and Even(x,y)
Labeled Graphs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a green path

GreenP\((x,y)\) :-

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
Labeled Graphs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a **green** path

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

GreenP(x,y) :- R(x,y,’green’)
Labeled Graphs

R encodes a graph

Find pairs of nodes (x,y) connected by a green path

GreenP(x,y) :- R(x,y,’green’)
GreenP(x,y) :- R(x,z,’green’),GreenP(z,y)
Labeled Graphs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a **monochromatic** path

\[
P(x,y,c) : R(x,y,c)\]

\[
P(x,y,c) : R(x,z,c), P(z,y,c)\]

\[
Answer(x,y) : P(x,y,c) \quad \text{why needed?}\]

\[
\text{R=}
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
Labeled Graphs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a *monochromatic* path

\[
P(x,y,c) :- R(x,y,c)
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
Labeled Graphs

R encodes a graph

Find pairs of nodes \((x,y)\) connected by a **monochromatic** path

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

\[ P(x,y,c) :- R(x,y,c) \]
\[ P(x,y,c) :- R(x,z,c), P(z,y,c) \]
Labeled Graphs

Find pairs of nodes \((x,y)\) connected by a *monochromatic* path

We join on both the node \(z\), and the color \(c\)

\[
P(x,y,c) \leftarrow R(x,y,c)
\]

\[
P(x,y,c) \leftarrow R(x,z,c), P(z,y,c)
\]
### Labeled Graphs

**R encodes a graph**

![Graph Diagram]

Find pairs of nodes \((x, y)\) connected by a *monochromatic* path

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

**R =**

- \(P(x, y, c) \leftarrow R(x, y, c)\)
- \(P(x, y, c) \leftarrow R(x, z, c), P(z, y, c)\)

**Answer\((x, y) \leftarrow P(x, y, c)\) – why needed?**

We join on both the node \(z\), and the color \(c\).
Labeled Graphs

Find all nodes reachable from node 2 by a path containing exactly one red edge.

R encodes a graph

R =

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
Labeled Graphs

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:

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>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
```
Labeled Graphs

R encodes a graph

 Automaton:

Find all nodes reachable from node 2 by a path containing exactly one red edge.

NoRed(2). :-.
Labeled Graphs

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>red</td>
<td>green</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>red</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{NoRed(2).} \quad \text{:- .}
\]
\[
\text{NoRed(y) \ :- NoRed(x), } R(x,y,c), c\neq \text{‘red’}.
\]
Labeled Graphs

R encodes a graph

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>red</td>
</tr>
</tbody>
</table>
Labeled Graphs

R encodes a graph

Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:

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

NoRed(2). :- .
NoRed(y) :- NoRed(x), R(x,y,c), c!=‘red’.
OneRed(y) :- NoRed(x), R(x,y,’red’).
OneRed(y) :- OneRed(x), R(x,y,c), c!=‘red’.
Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:
Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

• Single IDB
  – Called: Common Table Expression, CTE
  – Cannot write Odd/Even, Red/NoRed, etc
Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

• Single IDB
  – Called: Common Table Expression, CTE
  – Cannot write Odd/Even, Red/NoRed, etc

• Linear query only
  – Cannot write $T(x,y) \iff T(x,z), T(z,y)$
Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

• Single IDB
  – Called: Common Table Expression, CTE
  – Cannot write Odd/Even, Red/NoRed, etc

• Linear query only
  – Cannot write $T(x,y) :- T(x,z), T(z,y)$

• Has bag semantics (really???)
  – May not terminate!
Discussion: Recursion in SQL

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

with recursive T as(
  select * from R
  union
  select distinct R.x, T.y
  from R, T
  where R.y=T.x
)
select * from T;

Relation T is called a Common Table Expression CTE
Naïve Evaluation Algorithm

• When multiple IDBs: need to compute their new values together:

\[
\begin{align*}
\text{Odd}(x,y) & : - R(x,y) \\
\text{Even}(x,y) & : - \text{Odd}(x,z), R(z,y) \\
\text{Odd}(x,y) & : - \text{Even}(x,z), R(z,y)
\end{align*}
\]
Naïve Evaluation Algorithm

• When multiple IDBs: need to compute their new values \textit{together}:

\[
\begin{align*}
\text{Odd}(x,y) & : R(x,y) \\
\text{Even}(x,y) & : \text{Odd}(x,z), R(z,y) \\
\text{Odd}(x,y) & : \text{Even}(x,z), R(z,y)
\end{align*}
\]

\[
\begin{align*}
\text{Odd} & := \emptyset; \ \text{Even} := \emptyset; \\
\text{repeat} \\
\text{Even}_{\text{new}} & := \Pi_{x,y} (\text{Odd} \bowtie R); \\
\text{Odd}_{\text{new}} & := R \cup \Pi_{x,y} (\text{Even} \bowtie R);
\end{align*}
\]
Naïve Evaluation Algorithm

• When multiple IDBs: need to compute their new values together:

Odd(x,y) := R(x,y)
Even(x,y) := Odd(x,z), R(z,y)
Odd(x,y) := Even(x,z), R(z,y)

Odd := ∅; Even := ∅;

repeat

Even\text{new} := \Pi_{x,y} (Odd \bowtie R);
Odd\text{new} := R \cup \Pi_{x,y} (Even \bowtie R);

Odd:=Odd\text{new}
Even:=Even\text{new}
Naïve Evaluation Algorithm

• When multiple IDBs: need to compute their new values *together*:

Odd(x,y) := R(x,y)
Even(x,y) := Odd(x,z), R(z,y)
Odd(x,y) := Even(x,z), R(z,y)

Odd := ∅; Even := ∅;
repeat
Even\text{new} := \Pi_{x,y} (Odd \bowtie R);
Odd\text{new} := R \cup \Pi_{x,y} (Even \bowtie R);
if Odd = Odd\text{new} \land Even = Even\text{new} then break
Odd := Odd\text{new}
Even := Even\text{new}
Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Before we show this, a digression: monotone queries
Outline

• Recap: Datalog basics
• Naïve Evaluation Algorithm
• Monotone Queries
• Non-monotone Extensions
• Semi-naïve Evaluation Algorithm
Review: Montone Functions

• A function $f(x)$ is called *monotonically increasing*, or just *monotone* if:

$$\text{If } x \leq y \text{ then } f(x) \leq f(y)$$
Monotone Queries

• A query with input relations R, S, T, ... is called **monotone** if, whenever we increase a relation, the query answer also increases (or stays the same)

• *Increase* here means *larger set*
Monotone Queries

- A query with input relations R, S, T, ... is called \textit{monotone} if, whenever we increase a relation, the query answer also increases (or stays the same).
- \textit{Increase} here means \textit{larger set}.
- Mathematically:

\[
\text{If } R \subseteq R', S \subseteq S', ... \text{ then } Q(R, S, ...) \subseteq Q(R', S', ...) \]
Which Queries are Monotone?

```sql
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```
Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```
Which Queries are Monotone?

**MONOTONE**

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```
Which Queries are Monotone?

Monotone

SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2

SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
Which Queries are Monotone?

**MONOTONE**

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

**MONOTONE**

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```
Which Queries are Monotone?

**MONOTONE**

- SELECT DISTINCT x.sno, x.name
  FROM Supplier x, Supply y
  WHERE x.sno = y.sno and y.pno = 2

- SELECT DISTINCT x.sno, x.name
  FROM Supplier x, Supply y
  WHERE x.sno = y.sno and y.pno != 2

**MONOTONE**

- SELECT x.city, count(*)
  FROM Supplier x
  GROUP BY x.city

**NON-MONOTONE**

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Which Queries are Monotone?

**MONOTONE**

```sql
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

**MONOTONE**

```sql
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

**MONOTONE**

```sql
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
    FROM Supply y
    WHERE y.pno = 2 )
```

**NON-MONOTONE**

```sql
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```
Which Queries are Monotone?

**Monotone**

1. `SELECT DISTINCT x.sno, x.name FROM Supplier x, Supply y WHERE x.sno = y.sno and y.pno = 2`
2. `SELECT DISTINCT x.sno, x.name FROM Supplier x, Supply y WHERE x.sno = y.sno and y.pno != 2`

**Non-Monotone**

3. `SELECT x.city, count(*) FROM Supplier x GROUP BY x.city`

**Monotone**

4. `SELECT x.sno, x.sname FROM Supplier x WHERE x.sno IN (SELECT y.sno FROM Supply y WHERE y.pno = 2)`

---

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Which Queries are Monotone?

**MONOTONE**

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
    FROM Supply y
    WHERE y.pno = 2 )
```

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
    FROM Supply y
    WHERE y.pno != 2 )
```

**NON-MONOTONE**

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2

SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2

SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
  FROM Supply y
  WHERE y.pno = 2 )

SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
  FROM Supply y
  WHERE y.pno != 2 )

SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city

MONOTONE

MONOTONE

MONOTONE

NON-MONOTONE

NON-MONOTONE
Which Ops are Monotone?

- Selection: $\sigma_{\text{pred}}$
- Projection: $\Pi_{A,B,...}$
- Join: $\bowtie$
- Union: $\cup$
- Difference: $-$
- Group-by-sum: $\gamma_{A,B,sum(C)}$
Which Ops are Monotone?

- Selection: $\sigma_{pred}$ MONOTONE
- Projection: $\Pi_{A,B,...}$ MONOTONE
- Join: $\bowtie$ MONOTONE
- Union: $\cup$ MONOTONE
- Difference: $-$ NON-MONOTONE
- Group-by-sum: $\gamma_{A,B,sum(C)}$ NON-MONOTONE
Fun Fact

- A SELECT-FROM-WHERE query (without aggregates or subqueries) is monotone

```
SELECT [DISTINCT] ...
FROM R1 x1, R2 x2, ...
WHERE ...
```
Fun Fact

• A SELECT-FROM-WHERE query (without aggregates or subqueries) is monotone

SELECT [DISTINCT] ... FROM R1 x1, R2 x2, ... WHERE ...

• Proof: the nested loop semantics! When we add tuples to one relation, we cannot lose answers:

for x1 in R1 do:
   for x2 in R2 do:
      ...

Tips for Writing SQL Queries

• If the English formulation of a query is non-monotone, then you **need** to use a subquery OR aggregate in SQL.

Return SUPPLIERS who supply some product with price > $10000

Return SUPPLIERS who supply only products with price > $10000
Back to Datalog

Naïve Algorithm:
• Always terminates
• Terminates in a number of steps that is polynomial in the size of the database
• This is cool! Compare with java, python, etc

Assumptions:
• Set semantics only
• Monotone rules only
• No “value invention”
Naïve Evaluation Algorithm

**Fact:** every USPJ query is monotone

**Proof:** uses only $\sigma, \Pi, \bowtie, U$

\[
IDB_0 := \emptyset; \quad t := 0
\]

repeat
\[
IDB_{t+1} := USPJ(IDB_t); \quad t := t + 1
\]
until no more change
Naïve Evaluation Algorithm

**Fact:** every USPJ query is monotone

**Proof:** uses only $\sigma, \Pi, \bowtie, \cup$

**Fact:** the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

**Proof:** by induction
Naïve Evaluation Algorithm

**Fact:** every USPJ query is monotone
**Proof:** uses only $\sigma, \Pi, \bowtie, \cup$

**Fact:** the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

**Proof:** by induction $IDB_0(= \emptyset) \subseteq IDB_1$

\[
IDB_0 := \emptyset; \ t := 0 \\
\text{repeat } IDB_{t+1} := \text{USPJ}(IDB_t); \ t := t + 1 \\
\text{until no more change}
\]
Naïve Evaluation Algorithm

**Fact:** every USPJ query is monotone

**Proof:** uses only $\sigma, \Pi, \bowtie, U$

**Fact:** the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

**Proof:** by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:

$USPJ(IDB_t) \subseteq USPJ(IDB_{t+1})$
Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:

$IDB_{t+1} = \text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1}) = IDB_{t+2}$

$IDB_0 := \emptyset; \ t := 0$
repeat $IDB_{t+1} := \text{USPJ}(IDB_t); \ t := t + 1$
until no more change
Naïve Evaluation Algorithm

**Consequence**: The naïve algorithm terminates, in $O(n^k)$ steps, where:

- $n =$ number of distinct values in the DB
- $k =$ arity of widest IDB relation

**Proof**: IDBs increases to $\leq O(n^k)$ facts
Recap

Naïve Algorithm:
• Always terminates
• Terminates in a number of steps that is polynomial in the size of the database
• This is cool! Compare with java, python, etc

Assumptions:
• Set semantics only
• Monotone rules only
• No “value invention”

Will show this next
Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions
Non-monotone Extensions

- Aggregates
  - No standard syntax
  - We will follow Souffle

- Grouping

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

Aggregates

Q(m) : m = \textbf{min} \ x : \{ \text{Actor}(x, y, _), y = 'John' \}
Aggregates

\[ Q(m) : \text{\texttt{m = min}} \ x : \{ \text{Actor}(x, y, _), y = \text{\textquote{John}} \} \]

Meaning (in SQL)

```sql
SELECT min(id) as m
FROM Actor as a
WHERE a.name = \text{\textquote{John}}
```
Aggregates

\[ Q(m) :- m = \min x : \{ \text{Actor}(x, y, _), y = \text{‘John’} \} \]

Meaning (in SQL)

```
SELECT min(id) as m
FROM Actor as a
WHERE a.name = ‘John’
```

Aggregates in Souffle:
- count
- min
- max
- sum
Grouping

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

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

Meaning (in SQL)

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

A genealogy database (parent/child)

ParentChild $(p,c)$

```
<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

ParentChild(p,c)
Count Descendants

For each person, count his/her descendants

<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

Compute transitive closure of ParentChild

// for each person, compute his/her descendants
Count Descendants

Compute transitive closure of ParentChild

// 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).
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,_) }.
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,_) }.
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,_) }.

// 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

<table>
<thead>
<tr>
<th>SG</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
<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)

How to fix?
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
Stratified Datalog

Recursion conflicts with non-monotone queries

• Example: what does this mean?

```
Happy(Bob):- !Happy(Alice).
Happy(Alice) :- !Happy(Bob).
```

• A program is **stratified** if it can be partitioned into **strata**, such that every IDB predicate in a non-monotone position has been defined in an earlier **stratum**
Stratified Datalog

Stratum 1

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

Stratum 2

\[
\begin{align*}
Q(d) & : \text{T}(p,d), p = \text{“Alice”}.
\end{align*}
\]

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

Stratum 1

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

Stratum 2

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

Stratum 1

\[ \text{Happy}(\text{Bob}) : !\text{Happy}(\text{Alice}). \]
\[ \text{Happy}(\text{Alice}) : !\text{Happy}(\text{Bob}). \]

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

May use \( !D \)
Some Datalog Optimizations

• Every USPJ optimized traditionally

• Semi-naïve evaluation

• Magic sets

• Asynchronous execution
Summary

• Datalog = light-weight syntax, recursion

• Data independence, optimizations

• Limitations:
  – Monotone queries work great
  – Non-monotone queries: various restrictions