Advanced Topics in Data Management

Lecture 5
Datalog
Motivation

- SQL can express *relational queries*; Iteration/recursion need CTE construct

- 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
Agenda

- Definition
- Naïve Algorithm
- Termination
- Semi-naïve Algorithm
Datalog program

• A datalog program = several rules

• Rules may be recursive

• Set semantics only
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

```plaintext
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

```plaintext
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’.

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

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

Find Movies made in 1940
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.
Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

Facts = tuples in the database

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

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

**Facts** = tuples in the database

- `Actor(344759, 'Douglas', 'Fowley').`
- `Casts(344759, 29851).`
- `Casts(355713, 29000).`
- `Movie(7909, 'A Night in Armour', 1910).`
- `Movie(29000, 'Arizona', 1940).`
- `Movie(29445, 'Ave Maria', 1940).`

**Rules** = queries

- `Q1(y) :- Movie(x,y,z), z='1940'.`
- `Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').`
- `Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910), Casts(z,x2), Movie(x2,y2,1940).`

**Extensional Database Predicates** = EDB = `Actor`, `Casts`, `Movie`

**Intensional Database Predicates** = IDB = `Q1`, `Q2`, `Q3`
Anatomy of a Rule

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

\[ f, l = \text{head variables} \]
\[ x,y,z = \text{existential variables} \]
Anatomy of a Rule

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

atom

f, l = head variables
x, y, z = existential variables
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
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
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
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
Discussion

- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z)  -- join
  - Q2(x,y) :- R(x,y), y > 5       -- selection
  - Q3(x)  :- R(x,y)              -- projection
  - Q4(x,z) :- R(x,y),S(y,5),T(y,z)  -- SPJ
Discussion

- Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  - Q1(x,y,z) :- R(x,y), S(y,z) -- join
  - Q2(x,y) :- R(x,y), y > 5 -- selection
  - Q3(x) :- R(x,y) -- projection
  - Q4(x,z) :- R(x,y), S(y,5), T(y,z) -- SPJ

- Unions can be obtained by writing two rules:
  - Q5(x,y) :- R(x,y)
    Q5(x,y) :- S(x,y)
Discussion

• Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  – Q1(x,y,z) :- R(x,y), S(y,z)  -- join
  – Q2(x,y) :- R(x,y), y > 5  -- selection
  – Q3(x) :- R(x,y)  -- projection
  – Q4(x,z) :- R(x,y),S(y,5),T(y,z)  -- SPJ

• Unions can be obtained by writing two rules:
  – Q5(x,y) :- R(x,y)
    Q5(x,y) :- S(x,y)

• Difference: it’s getting complicated; more on this later
Discussion

• Datalog rules make it very easy to express Selections-Projection-Join (SPJ) queries
  – Q1(x,y,z) :- R(x,y), S(y,z)  
  – Q2(x,y) :- R(x,y), y > 5
  – Q3(x) :- R(x,y)
  – Q4(x,z) :- R(x,y), S(y,5), T(y,z)
  
• Unions can be obtained by writing two rules:
  – Q5(x,y) :- R(x,y)
  – Q5(x,y) :- S(x,y)

• Difference: it’s getting complicated; more on this later
• New! In datalog we can write recursive rules!
Processing Graphs in Datalog

Graph

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>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

<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>

Pattern Matching

\[
\text{Answer}(x,y,z) :\ - \ R(x,y), R(x,z), R(y,z)
\]
Discussion

• SQL uses the *named* perspective:
  – Each attribute has a name
  – The order of the attributes is irrelevant
  – `Person.address` or `Person.ssn` or `Person.age`
  – Need to know the names of an attribute
Discussion

• SQL uses the *named* perspective:
  – Each attribute has a name
  – The order of the attributes is irrelevant
  – `Person.address` or `Person.ssn` or `Person.age`
  – Need to know the names of an attribute

• Datalog uses the *unnamed* perspective:
  – The order of the attributes is important
  – Attributes do not have names
  – `Person(x,y,z,u,v)`: here x is ssn, y is address, etc
  – Need to know the position of an attribute
Recursion

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>
Recursion

Descendants of node 2

\[
\begin{align*}
D(x) & \text{ :- } R(2, x) \\
D(y) & \text{ :- } D(x), R(x, y)
\end{align*}
\]

\[R= \]

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

Descendants of node 2

Recursive rule

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

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

How recursion works in datalog:
Initially $D = \emptyset$

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

Descendants of node 2

<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>
Recursion

Descendants of node 2

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

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}

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>
Recursion

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:
Recursion

Descendents 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}

<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>
Recursion

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:

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

Descendants of node 2

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

Descendants of node 2

R(x) :- R(2, x)
R(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}
Recursion

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

<table>
<thead>
<tr>
<th>R=</th>
<th>1</th>
<th>2</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>
Discussion

• Datalog is designed for recursion

• Will prove that it always terminates

• To prove that, first need to define the semantics: **Naïve Algorithm**
Agenda

• Definition

• Naïve Algorithm

• Termination

• Semi-naïve Algorithm
Naïve Evaluation Algorithm

• Convert all datalog rules into a set of Union-Select-Project-Join (USPJ)

• Called: *Immediate Consequence Operator*

• Naïve Evaluation Algorithm: Repeatedly apply the ICO until fixpoint
Immediate Consequence Operator

Every rule $\rightarrow$ SPJ query

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

Multiple rules same IDB head $\rightarrow$ USPJ

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

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

$\ldots$

The **ICO** consists of all USPJ’s for all IDBs
Naïve Evaluation Algorithm

\[
\begin{align*}
IDB_0 & := \emptyset \\
\textbf{for } t = 1, \ldots, \infty \textbf{ do} \\
& \quad IDB_t := ICO(IDB_{t-1}) \\
& \quad \textbf{if } IDB_t = IDB_{t-1} \textbf{ then break}
\end{align*}
\]
Naïve Evaluation Algorithm

\[
\begin{align*}
D(x) & : - \ R(2,x) \\
D(y) & : - \ D(x),R(x,y)
\end{align*}
\]
Naïve Evaluation Algorithm

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

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

\begin{align*}
D(x) &\colon\text{ R}(2,x) \\
D(y) &\colon\text{ D}(x),\text{R}(x,y)
\end{align*}

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

\[
\begin{align*}
D(x) & : - R(2,x) \\
D(y) & : - D(x), R(x,y)
\end{align*}
\]

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

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

\[
D := \emptyset;\\
\text{repeat}\\
\quad D := \Pi_{R.dst} (\sigma_{R.src=2} (R)) \cup \Pi_{R.dst} (D \bowtie_{D.node=R.src} R);\\
\text{until} \ [\text{no more change}]
\]
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

\[ R = \{ (x, y), (x, z), (z, y) \} \]

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.

Initially:

T is empty.

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)

What does it compute?
Example

Initially:
T is empty.

First iteration:
T =

First rule generates this

Second rule generates nothing (because T is empty)

What does it compute?

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

Let $R(x,y)$ and $T(x,y)$ be defined as follows:

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

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

Initially:

$T$ is empty.

First iteration:

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

New facts

Second iteration:

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

First rule generates this

Second rule generates this

What does it compute?
Example

Initially:

\[ T = \]

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

T is empty.

First rule:

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

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

First iteration:

Second iteration:

Third iteration:

What does it compute?
Example

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 =

<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>

Second iteration: T =

<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>

Third iteration: T =

<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>

Fourth iteration: T = (same)

No new facts. DONE
Example

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

Initially:
\[
\begin{array}{ccc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

\[ T \] is empty.

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

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

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

Fourth iteration:
\[
\begin{array}{ccc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
3 & 5 \\
\end{array}
\]

\[ \text{Iteration } k \text{ computes pairs } (x,y) \text{ connected by path of length } \leq k \]

What does it compute?

No new facts.

DONE
Three Equivalent Programs

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

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

Right linear
Three Equivalent Programs

\[ R = \]

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

Right linear

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

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

Left linear

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

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

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=

<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>
Three Equivalent Programs

<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>

Question: how many iterations does each require?

Right linear

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

Left linear

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

Non-linear

$T(x,y) :\neg R(x,y)$
$T(x,y) :\neg T(x,z), T(z,y)$
Three Equivalent Programs

<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=

Question: how many iterations does each require?
Multiple IDBs

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

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>
Multiple IDBs

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

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

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

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

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\((z,y)\)

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

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]
We have two IDBs: Odd(x,y) and Even(x,y)

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

\[
\begin{align*}
\text{Odd}(x,y): & \text{ } \text{- } R(x,y) \\
\text{Even}(x,y): & \text{ } \text{- } \text{Odd}(x,z), \text{ } R(z,y) \\
\text{Odd}(x,y): & \text{ } \text{- } \text{Even}(x,z), \text{ } R(z,y)
\end{align*}
\]
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_{new} := \Pi_{x,y} (Odd \bowtie R);
Odd_{new} := R \cup \Pi_{x,y} (Even \bowtie R);
\}
if Odd=Odd_{new} \land Even=Even_{new}
then break
Odd:=Odd_{new}
Even:=Even_{new}
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)

<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>
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>
Labeled Graphs

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>

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

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>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>red</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>green</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>green</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>blue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>red</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

R encodes a graph

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

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

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

\begin{tabular}{|c|c|c|}
\hline
1 & 2 & red \\
2 & 1 & green \\
2 & 3 & green \\
1 & 4 & blue \\
3 & 4 & green \\
4 & 5 & red \\
\hline
\end{tabular}
Labeled Graphs

R encodes a graph

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

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

\[P(x,y,c) :- R(x,y,c)\]
\[P(x,y,c) :- R(x,z,c), P(z,y,c)\]
\[\text{Answer}(x,y) :- P(x,y,c)\]

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.

\[ S_0(2) : S_0(x), R(x, y, c), c \neq \text{red} \]

\[ S_1(y) : S_0(x), R(x, y, \text{red}) \]

\[ S_1(y) : S_1(x), R(x, y, c), c \neq \text{red} \]

R encodes a graph

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

$S_0(2) :- \cdot$
Labeled Graphs

R encodes a graph

\[ S_0(y) : - S_0(x), R(x, y, c), c \neq \text{\textquote{red}}. \]

\[ S_1(y) : - S_1(x), R(x, y, c), c \neq \text{\textquote{red}}. \]

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

Automaton:

\[ S_0 \rightarrow \text{\textquote{red}} \rightarrow S_1 \]

\[ \neq \text{\textquote{red}} \]

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 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>1</td>
<td>2</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>

S0(2) :- .
S0(y) :- S0(x), R(x,y,c), c!=‘red’.
S1(y) :- S0(x), R(x,y,’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>

S0(2) :- .
S0(y) :- S0(x), R(x,y,c), c!=‘red’.
S1(y) :- S0(x), R(x,y,’red’).
S1(y) :- S1(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 $T(x,y) :- T(x,z), T(z,y)$
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

\[
\begin{align*}
T(x,y) &\colon= R(x,y) \\
T(x,y) &\colon= R(x,z), T(z,y)
\end{align*}
\]
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;
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
Agenda

• Definition

• Naïve Algorithm

• Termination

• Semi-naïve Algorithm
Runtime of a Datalog Program

**Theorem** The Naïve Algorithm:
- Always terminates
- Terminates in a number of steps that is polynomial in the size of the database

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

This is cool! Compare with java, python, etc
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 \( R, S, T, \ldots \) 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}
Monotone Queries

• A query with input relations $R, S, T, \ldots$ 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

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

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

- Selection: $\sigma_{\text{pred}}$ MONOTONE
- Projection: $\Pi_{A,B,...}$ MONOTONE
- Join: $\bowtie$ MONOTONE
- Union: $U$ MONOTONE
- Difference: $-$ NON-MONOTONE
- Group-by-sum: $\gamma_{A,B,sum(C)}$ NON-MONOTONE
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”
Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

\[
\begin{align*}
\text{IDB}_0 & := \emptyset; \quad t := 0 \\
\text{repeat} & \quad \text{IDB}_{t+1} := \text{USPJ(IDB}_t); \quad t := t + 1 \\
\text{until} & \quad \text{no more change}
\end{align*}
\]
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

\begin{verbatim}
IDB_0 := \emptyset; t := 0
repeat IDB_{t+1} := USPJ(IDB_t); t := t + 1
until no more change
\end{verbatim}
Proof

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 $IDB_0 (= \emptyset) \subseteq IDB_1$

\[\begin{align*}
\text{IDB}_0 & := \emptyset; \quad t := 0 \\
\text{repeat} \ \ \text{IDB}_{t+1} & := \text{USPJ}(\text{IDB}_t); \quad t := t + 1 \\
\text{until} \no \text{more change}
\end{align*}\]
Proof

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: $USPJ(IDB_t) \subseteq USPJ(IDB_{t+1})$
Proof

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

```plaintext
IDB_0 := \emptyset; \ t := 0
repeat \ IDB_{t+1} := 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
Same Generation

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

```
SG

<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)

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
Agenda

• Definition

• Naïve Algorithm

• Termination

• Semi-naïve Algorithm
Main Idea

• The naïve algorithm re-re-computes the same facts over and over again

• Main idea: compute only the new facts of the IDBs, from the previous new facts

• Need: Incremental View Maintenance
Incremental View Maintenance

• A **view** is a relation defined by a query

```
CREATE VIEW PathsLengthTwo AS
    SELECT e1.src as src, e2.dst as dst
    FROM Edge e1, Edge e2
    WHERE e1.dst = e2.src
```

• In datalog: every IDB is a **view**

```
PathsLengthTwo(x,y) :- Edge(x,z), Edge(z,y)
```
Incremental View Maintenance

• Suppose we have computed the view:
  \[ V := \text{[monotone query here]} \]
Incremental View Maintenance

• Suppose we have computed the view:
  \[ V := \text{[monotone query here]} \]

• One, or several of the base relations is updated: e.g. we insert new tuples:
  \[ R := R \cup \Delta R \]
Incremental View Maintenance

• Suppose we have computed the view:
  \[ V := [\text{monotone query here}] \]

• One, or several of the base relations is updated: e.g. we insert new tuples:
  \[ R := R \cup \Delta R \]

• Then the view needs to be incremented with some new tuples:
  \[ V := V \cup \Delta V \]
Incremental View Maintenance

• Suppose we have computed the view:
  \[ V := \text{[monotone query here]} \]

• One, or several of the base relations is updated: e.g. we insert new tuples:
  \[ R := R \cup \Delta R \]

• Then the view needs to be incremented with some new tuples:
  \[ V := V \cup \Delta V \]

• Problem: compute \( \Delta V \) without recomputing \( V \)
Incremental View Maintenance

Fix a USPJ query $Q$ with inputs $R_1, R_2, \ldots$:

$$Q(R_1, R_2, \ldots)$$

The delta query $\Delta Q$ is any query that has the property:

$$Q(R_1 \cup \Delta R_1, R_2 \cup \Delta R_2, \ldots) = Q(R_1, R_2, \ldots) \cup \Delta Q(R_1, \Delta R_1, R_2, \Delta R_2, \ldots)$$
Incremental View Maintenance

Example 1:

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

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

Example 1:

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

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

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

Example 2:

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

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

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

Example 3:

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

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

Example 3:

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

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

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

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

\[ \Delta V(x,y) : - \Delta T(x,z), \Delta T(z,y) \]
Incremental View Maintenance

Fix a USPJ query $Q$ with inputs $R_1, R_2, \ldots$:

\[
Q(R_1, R_2, \ldots)
\]

The delta query $\Delta Q$ is any query that has the property:

\[
Q(R_1 \cup \Delta R_1, R_2 \cup \Delta R_2, \ldots) = Q(R_1, R_2, \ldots) \cup \Delta Q(R_1, \Delta R_1, R_2, \Delta R_2, \ldots)
\]
Semi-Naïve Evaluation Algorithm

\[ \text{IDB} = \text{SPJU}(\emptyset) \]
\[ \Delta\text{IDB} = \text{IDB} \]

Loop

\[ \Delta\text{IDB} := \Delta\text{SPJU}(\text{IDB}, \Delta\text{IDB}) \quad -- \quad \text{IDB} \]
\[ \text{IDB} := \text{IDB} \cup \Delta\text{IDB} \]

if \((\Delta\text{IDB} = \emptyset)\) then break

End Loop

Apply only the non-recursive rules
From Naïve to Semi-Naive

Naïve Algorithm:

\[ \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \cdots \]

\[
\begin{align*}
\text{IDB}_0 & := \emptyset \\
t & := 0 \\
\text{Loop} \\
\quad \text{IDB}_{t+1} & := \text{SPJU}(\text{IDB}_t) \\
\quad \text{if} \ (\text{IDB}_{t+1} = \text{IDB}_t) \ \text{then} \ \text{break} \\
\quad t & := t + 1 \\
\text{End Loop}
\end{align*}
\]
From Naïve to Semi-Naïve

Naïve Algorithm:

\[
\Delta IDB_t = IDB_t - IDB_{t-1}
\]

\[
\begin{align*}
IDB_0 &\subseteq IDB_1 \subseteq IDB_2 \subseteq \cdots \\
IDB_0 &:= \emptyset \\
t &:= 0 \\
\text{Loop} \\
\quad IDB_{t+1} &:= \text{SPJU}(IDB_t) \\
\quad \text{if } (IDB_{t+1} = IDB_t) \text{ then break} \\
\quad t &:= t+1 \\
\text{End Loop}
\end{align*}
\]
From Naïve to Semi-Naive

Naïve Algorithm:

$$\Delta IDB_t = IDB_t - IDB_{t-1}$$

$$IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \cdots$$

$$\begin{align*}
IDB_0 &:= \emptyset; \\
t &:= 0 \\
\text{Loop} \\
\quad IDB_{t+1} &:= \text{SPJU}(IDB_t) \\
\quad \text{if } (IDB_{t+1} = IDB_t) \text{ then break} \\
\quad t &:= t+1 \\
\text{End Loop}
\end{align*}$$

$$\begin{align*}
IDB_0 &:= \emptyset; \\
\Delta IDB_0 &:= \emptyset; \\
t &:= 0 \\
\text{Loop} \\
\quad \Delta IDB_{t+1} &:= \text{SPJU}(IDB_t) -- IDB_t \\
\quad IDB_{t+1} &:= IDB_t \cup \Delta IDB_{t+1} \\
\quad \text{if } (\Delta IDB_{t+1} = \emptyset) \text{ then break} \\
\quad t &:= t+1 \\
\text{End Loop}
\end{align*}$$
From Naïve to Semi-Naïve

Naïve Algorithm:

\[ \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \cdots \]

\[ \begin{align*} 
\text{IDB}_0 &:= \emptyset \\
t &:= 0 \\
\text{Loop} \\
\text{IDB}_{t+1} &:= \text{SPJU} (\text{IDB}_t) \\
\text{if } (\text{IDB}_{t+1} = \text{IDB}_t) \text{ then break} \\
t &:= t+1 \\
\text{End Loop} 
\end{align*} \]

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

\[ \begin{align*} 
\text{IDB}_0 &:= \emptyset; \quad \Delta \text{IDB}_0 := \emptyset; \\
t &:= 0 \\
\text{Loop} \\
\Delta \text{IDB}_{t+1} &:= \text{SPJU} (\text{IDB}_t) -- \text{IDB}_t \\
\text{IDB}_{t+1} &:= \text{IDB}_t \cup \Delta \text{IDB}_{t+1} \\
\text{if } (\Delta \text{IDB}_{t+1} = \emptyset) \text{ then break} \\
t &:= t+1 \\
\text{End Loop} 
\end{align*} \]
From Naïve to Semi-Naive

Naïve Algorithm:

\[ \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \cdots \]

\[
\begin{align*}
\text{IDB}_0 &:= \emptyset; \\
\Delta \text{IDB}_0 &:= \emptyset; \\
t &:= 0 \\
\text{Loop} \\
\text{IDB}_{t+1} &:= \text{SPJU}(\text{IDB}_t) \\
\text{if} \ (\text{IDB}_{t+1} = \text{IDB}_t) \text{ then break} \\
t &:= t+1 \\
\text{End Loop}
\end{align*}
\]

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

\[
\begin{align*}
\text{IDB}_0 &:= \emptyset; \\
\text{IDB}_0 &:= \emptyset; \\
t &:= 0 \\
\text{Loop} \\
\Delta \text{IDB}_{t+1} &:= \text{SPJU}(\text{IDB}_t) -- \text{IDB}_t \\
\text{IDB}_{t+1} &:= \text{IDB}_t \cup \Delta \text{IDB}_{t+1} \\
\text{if} \ (\Delta \text{IDB}_{t+1} = \emptyset) \text{ then break} \\
t &:= t+1 \\
\text{End Loop}
\end{align*}
\]
From Naïve to Semi-Naive

Naïve Algorithm:

\[ \Delta IDB_t = IDB_t - IDB_{t-1} \]

\[
\begin{align*}
IDB_0 &:= \emptyset; \quad \Delta IDB_0 := \emptyset; \\
t &:= 0 \\
\text{Loop} &
\end{align*}
\]

\[
\begin{align*}
\Delta IDB_{t+1} &:= \text{SPJU}(IDB_t) \cup \Delta IDB_{t+1} \\
IDB_{t+1} &:= IDB_t \cup \Delta IDB_{t+1} \\
\text{if } (\Delta IDB_{t+1} = \emptyset) \text{ then break} \\
t &:= t+1 \\
\text{End Loop}
\end{align*}
\]

\[ \Delta SPJU(IDB_{t-1}, \Delta IDB_t) \]
From Naïve to Semi-Naive

Naïve Algorithm:

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

\[ \text{IDB}_0 := \emptyset; \quad \Delta \text{IDB}_0 := \emptyset; \]

\[ t := 0 \]

Loop

\[ \Delta \text{IDB}_{t+1} := \text{SPJU}(\text{IDB}_t) - \text{IDB}_t \]

\[ \text{IDB}_{t+1} := \text{IDB}_t \cup \Delta \text{IDB}_{t+1} \]

if \((\Delta \text{IDB}_{t+1} = \emptyset)\) then break

\[ t := t + 1 \]

End Loop

But need to start from \(t=1\), since \(\text{IDB}_{-1}\) undefined
133

Semi-Naïve Algorithm

\[ \text{IDB}_1 := \text{SPJU}(\emptyset); \quad \Delta\text{IDB}_1 := \text{IDB}_1; \]
\[ t := 1 \]
Loop
\[ \Delta\text{IDB}_{t+1} := \Delta\text{SPJU}(\text{IDB}_{t-1}, \Delta\text{IDB}_t) \quad \text{--} \quad \text{IDB}_t \]
\[ \text{IDB}_{t+1} := \text{IDB}_t \cup \Delta\text{IDB}_{t+1} \]
if \((\Delta\text{IDB}_{t+1} = \emptyset)\) then break
\[ t := t+1 \]
End Loop
Semi-Naïve Evaluation Algorithm

\[ \text{IDB} = \text{SPJU}(\emptyset) \]
\[ \Delta\text{IDB} = \text{IDB} \]

Loop

\[ \Delta\text{IDB} := \Delta\text{SPJU}(\text{IDB}, \Delta\text{IDB}) \] -- IDB
\[ \text{IDB} := \text{IDB} \cup \Delta\text{IDB} \]

if (\( \Delta\text{IDB} = \emptyset \)) then break

End Loop

Apply only the non-recursive rules
Example: Linear Recursion

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

\[ R = \]

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

\[ T(x,y) \iff R(x,y) \]
\[ T(x,y) \iff R(x,z), T(z,y) \]

\[ R = \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

\[
T := R \quad \Delta T := T
\]

Loop

\[
\Delta T(x, y) := \Pi_{x,y} (R(x, z) \bowtie \Delta T(z, y)) - T(x, y)
\]

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

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

End Loop
Example

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

\[ R = \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

Step 0:

\[ \Delta T \]

\[ T \]

\[ T := R \quad \Delta T := T \]

Loop

\[ \Delta T(x,y) := \Pi_{x,y}(R(x,z) \bowtie \Delta T(z,y)) - T(x,y) \]

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

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

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

**Example**

\[
R =
\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

\[
\Delta T\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

\[
T\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

\[
\Delta T\]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

\[
T \left( x, y \right) : = R \left( x, y \right) \\
\Delta T := T \\
\]

Loop

\[
\Delta T(x,y) := \Pi_{x,y}(R(x,z) \bowtie \Delta T(z,y)) - T(x,y) \\
T := T \cup \Delta T \\
\]

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

End Loop
Example:

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

\[ R = \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

**Step 0:**

\[ \Delta T \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

**Step 1:**

\[ \Delta T \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>e</td>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>c</td>
<td>e</td>
</tr>
</tbody>
</table>

**Step 2:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>e</td>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>c</td>
<td>e</td>
</tr>
</tbody>
</table>

**Updates:**

\[ T \leftarrow T \cup \Delta T \]

\[ \text{if } (\Delta T = \emptyset) \text{ then break} \]

\[ \Delta T := \Pi_{x,y}(R(x,z) \bowtie \Delta T(z,y)) - T(x,y) \]

\[ T := R \]

**Loop**
Example: Non-linear Recursion

\[
T(x, y) :\quad R(x, y) \\
T(x, y) :\quad T(x, z), T(z, y)
\]
Example: Non-linear Recursion

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

Semi-naïve algorithm:

\[
\begin{align*}
T & := R \quad \Delta T := T \\
\text{Loop} & \\
\Delta T(x, y) & := (\Pi_{x,y}(T(x, z) \bowtie \Delta T(z, y))) \cup \Pi_{x,y}(\Delta T(x, z) \bowtie T(z, y)) \cup \Pi_{x,y}(\Delta T(x, z) \bowtie \Delta T(z, y))) - T(x, y) \\
T & := T \cup \Delta T \\
\text{if } (\Delta T = \emptyset) \text{ then break} \\
\text{End Loop}
\end{align*}
\]
Example: Same Generation

\[
SG(x,y) :\ ParentChild(p,x),\ ParentChild(p,y) \\
SG(x,y) :\ ParentChild(p,x),\ SG(p,q),\ ParentChild(q,y)
\]
Example: Same Generation

\[
SG(x,y) : \text{ParentChild}(p,x), \text{ParentChild}(p,y)
\]

\[
SG(x,y) : \text{ParentChild}(p,x), SG(p,q), \text{ParentChild}(q,y)
\]

\[
SG := \text{ParentChild} \quad \Delta SG := SG
\]

Loop

\[
\Delta SG(x, y) := \Pi_{x,y}(\text{ParentChild}(p, x) \bowtie \Delta SG(p, q) \bowtie \text{ParentChild}(q, y)) - SG(x, y)
\]

SG := SG \cup \Delta SG

if (\Delta SG = \emptyset) then break

End Loop
Discussion

• Most datalog engines implement the semi-naïve algorithm

• Notice: it only works when the recursion has a monotone body
Summary

• Datalog = light-weight syntax, recursion
• Several optimizations
• Limitations:
  – Monotone queries work great
  – Non-monotone queries: stratification
• SQL: supports limited recursion only