CSE 544
Principles of Database Management Systems
Lectures 5: Datalog (1)
Announcement

• Deadline for HW1 has passed…

• Project M2 due on Friday

• HW2 released (datalog / Souffle)
Where We Are

Relational query languages:
• SQL
• Relational Algebra
• Relational Calculus (haven’t discussed, but you may look it up)

The can express the same class of queries called relational queries
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

• Find all people X whose number of friends is a prime number

• Find all people who are friends with everyone who is not a friend of Bob

• Partition all people into three sets $P_1(X), P_2(X), P_3(X)$ s.t. any two friends are in different partitions

• Find all people who are direct or indirect friends with Alice
Which are Relational Queries? Which are not? And Why?

\text{Friend}(X,Y)

- Find all people X whose number of friends is a prime number

\begin{itemize}
  \item Find all people who are friends with everyone who is not a friend of Bob
  \item Partition all people into three sets \text{P}_1(X), \text{P}_2(X), \text{P}_3(X) s.t. any two friends are in different partitions
  \item Find all people who are direct or indirect friends with Alice
\end{itemize}

\begin{itemize}
  \item \textbf{No higher math in database}
\end{itemize}
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

• Find all people X whose number of friends is a prime number
• Find all people who are friends with everyone who is not a friend of Bob
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

• Find all people X whose number of friends is a prime number
• Find all people who are friends with everyone who is not a friend of Bob

Yes! (write it in SQL!)
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)
• Find all people X whose number of friends is a prime number
• Find all people who are friends with everyone who is not a friend of Bob

• Partition all people into three sets P1(X), P2(X), P3(X) s.t. any two friends are in different partitions
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob

- Partition all people into three sets P1(X), P2(X), P3(X) s.t. any two friends are in different partitions

No! NP-complete
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)
- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob

• Partition all people into three sets P1(X), P2(X), P3(X) s.t. any two friends are in different partitions
• Find all people who are direct or indirect friends with Alice
Which are Relational Queries? Which are not? And Why?

Friend(X,Y)
- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob

- Partition all people into three sets P1(X), P2(X), P3(X) s.t. any two friends are in different partitions
- Find all people who are direct or indirect friends with Alice

“Recursive query”; PTIME, yet not expressible in RA
Recursive Queries

• “Find all direct or indirect friends of Alice”
• Computable in PTIME, yet not expressible in RA
• Datalog: extends RA with recursive queries
Datalog

• Designed in the 80’s
• Simple, concise, elegant
• Today is a hot topic, beyond databases: network protocols, static program analysis, DB+ML
• Very few open source implementations, and hard to find
• In HW2 we will use Souffle
SQL Query vs Datalog
(which would you rather write?)
(any Java fans out there?)
Outline

• Datalog rules
• Recursion
• Negation, aggregates, stratification
• Semantics
• Naïve and Semi-naïve Evaluation
• Connection to RA – on your own
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).
Movie(29445, ‘Ave Maria’, 1940).

Rules = queries
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’. 
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’.

Find 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’).
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).
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).
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).

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).
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
Datalog: Terminology

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

- $R_i(\text{args}_i)$ 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: $z > `1940'$.
- Some systems use $\leftarrow$
- Some use AND

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

\[
\text{Q(args)} :\text{- R1(args) AND R2(args)}, \ldots
\]
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

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

- Meaning of a datalog rule = a logical statement!

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

- For all \(x, y, z\): if \((x,y,z) \in \text{Movies}\) and \(z = '1940'\) then \(y\) is in \(Q1\) (i.e. is part of the answer)

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

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

• For all \( x, y, z \): if \((x,y,z) \in \text{Movies}\) and \( z = ‘1940’ \) then \( y \) is in \( Q1 \) (i.e. is part of the 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!

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

• For all x, y, z: if \((x,y,z) \in \text{Movies}\) and \(z = '1940'\) then y is in Q1 (i.e. is part of the answer)

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

• 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!

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

• For all \(x, y, z\): if \((x,y,z) \in \text{Movies}\) and \(z = '1940'\) then \(y\) is in \(Q1\) (i.e. is part of the answer)

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

• Logically equivalent:

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

• Thus, non-head variables are called "existential variables"
Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement!

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

- For all \( x, y, z \): if \((x,y,z) \in \text{Movies}\) and \( z = '1940' \) then \( y \) is in \( Q1 \) (i.e. is part of the answer)

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

- Logically equivalent:

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

- Thus, non-head variables are called "existential variables"

- We want the \textit{smallest} set \( Q1 \) with this property (why?)
Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own
Datalog program

• A datalog program consists of several rules
• Importantly, rules may be recursive!
• Usually there is one distinguished predicate that’s the output
• We will show an example first, then give the general semantics.
R encodes a graph

R =

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

R encodes a graph

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

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

What does it compute?
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.

What does it compute?

Example

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

R encodes a graph

1
2
3
4
5

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)

What does it compute?
Example

R encodes a graph

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

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

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

What does it compute?

First rule generates this

Second rule generates this

New facts
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>

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>

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

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

New fact

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>

Both rules

First rule

Second rule

What does it compute?
Example

R encodes a graph

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

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

Initially:

T is empty.

First iteration:

T =

Second iteration:

T =

Third iteration:

T =

Fourth iteration:

T = (same)

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

Question: which terminates in fewest iterations?
Outline

- Datalog rules
- Recursion
- **Semantics**
  - Negation, aggregates, stratification
  - Naïve and Semi-naïve Evaluation
  - Connection to RA – on your own
1. Fixpoint Semantics

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

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

• A datalog program w/o functions (+, *, ...) always terminates. (In what time?)
2. Minimal Model Semantics:

- Return the IDB that
  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 smallest IDB satisfying (1)
Example

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 $T_t = T_{t-1}$

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
- Connection to RA – on your own
Extensions

• Aggregates, negation

• Stratified datalog
Aggregates

• No commonly agreed syntax

• Each implementation uses it’s own
Aggregates in Souffle

General syntax in Logicblox:

\[ Q(x,y,z,v) \; : - \; \text{Body1}(x,y,z), \; v = \text{sum}(w) : \{ \text{Body2}(x,y,z,w) \} \]

Meaning (in SQL)

```sql
select x,y,z, sum(w) as v
from R1, R2, ... 
where ...
group by x,y,z
```
Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* for each person, compute his/her descendants */

ParentChild(p,c)
Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
/* For each person, count the number of descendants */
Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
/* Find the number of descendants of Alice */
Example

For each person, compute the total number of descendants

/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
/* Find the number of descendants of Alice */
Q(d) :- N(“Alice”,d).
Negation: use “!”

Find all descendants of Alice, who are not descendants of Bob

/* for each person, compute his/her descendants */
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
/* Compute the answer: notice the negation */
Q(x) :- D("Alice",x), !D("Bob",x).
Safe Datalog Rules

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

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

\[
\text{U2}(x) : - \text{ParentChild}("Alice",x), !\text{ParentChild}(x,y)
\]
Safe Datalog Rules

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

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

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

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

\( U1 = \text{infinite!} \)
Safe Datalog Rules

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

U1(x,y) :- ParentChild("Alice",x), y != "Bob"

Holds for every y other than "Bob"
U1 = infinite!

U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)

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

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

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

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

A datalog rule is *safe* if every variable appears in some positive relational atom.
Stratified Datalog

• Recursion does not cope well with aggregates or negation
• Example: what does this mean?

A() :- !B().
B() :- !A().
Stratified Datalog

• Recursion does not cope well with aggregates or negation
• Example: what does this mean?
  
  \[
  \begin{align*}
  A() & : - !B(). \\
  B() & : - !A().
  \end{align*}
  \]

• A datalog program is **stratified** if it can be partitioned into strata s.t., for all n, only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.
• Souffle (and others) accepts only stratified datalog.
Stratified Datalog

Stratum 1

D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

Stratum 2

N[x] = m :- agg<<m = count()>> D(x,y).
Q(d) :- N["Alice"] = d.

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

<table>
<thead>
<tr>
<th>Stratum 1</th>
<th>Stratum 2</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>
<tr>
<td>N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.</td>
<td></td>
</tr>
<tr>
<td>Q(d) :- N(&quot;Alice&quot;, d).</td>
<td></td>
</tr>
</tbody>
</table>

May use D in an agg because was defined in previous stratum

May use !D
Stratified Datalog

Stratum 1

D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
Q(x) :- D("Alice", x), !D("Bob", x).

Stratum 2

N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
Q(d) :- N("Alice", d).

Non-stratified

A() :- !B().
B() :- !A().

May use D in an agg because was defined in previous stratum

May use !D

Cannot use !A
Stratified Datalog

• If we don’t use aggregates or negation, then the datalog program is already stratified
• If we do use aggregates or negation, it is usually quite natural to write the program in a stratified way
Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own
Datalog Evaluation Algorithms

• Needs to preserve the efficiency of query optimizers, while extending them to recursion
• Two general strategies:
  – Naïve datalog evaluation
  – Semi-naïve datalog evaluation
• Some powerful optimizations:
  – Magic sets (next lecture)
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} :- \text{body}_1 \]
\[ P_{i2} :- \text{body}_2 \]
....
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} ::= \text{body}_1 \]
\[ P_{i2} ::= \text{body}_2 \]
\[ \ldots \]

\[ P_1 ::= \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_2 ::= \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Group by IDB predicate
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} : \text{body}_1 \]
\[ P_{i2} : \text{body}_2 \]
\[ \ldots \]

Group by IDB predicate

\[ P_1 : \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_2 : \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Each rule is a Select-Project-Join-Union query

\[ P_1 : \text{SPJU}_1 \]
\[ P_2 : \text{SPJU}_2 \]
\[ \ldots \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} \leftarrow \text{body}_1 \]
\[ P_{i2} \leftarrow \text{body}_2 \]
\[ \ldots \]

Group by IDB predicate

\[ P_1 \leftarrow \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_2 \leftarrow \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Each rule is a Select-Project-Join-Union query

Naïve datalog evaluation algorithm:

\[ P_1 = P_2 = \ldots = \emptyset \]

Loop

NewP_1 = SPJU_1; NewP_2 = SPJU_2; \ldots

if (NewP_1 = P_1 and NewP_2 = P_2 and \ldots)
then exit

P_1 = NewP_1; P_2 = NewP_2; \ldots

Endloop
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} :- \text{body}_1 \]
\[ P_{i2} :- \text{body}_2 \]
\[ \text{...} \]

\[ P_1 :- \text{body}_{11} \cup \text{body}_{12} \cup \text{...} \]
\[ P_2 :- \text{body}_{21} \cup \text{body}_{22} \cup \text{...} \]
\[ \text{...} \]

Group by IDB predicate

Each rule is a \text{Select-Project-Join-Union} query

Naïve datalog evaluation algorithm:

\[ P_1 = P_2 = \ldots = \emptyset \]

Loop

\[ \text{New}P_1 = \text{SPJU}_1; \text{New}P_2 = \text{SPJU}_2; \ldots \]
\[ \text{if} (\text{New}P_1 = P_1 \text{ and } \text{New}P_2 = P_2 \text{ and } \ldots) \]
\[ \text{then exit} \]
\[ P_1 = \text{New}P_1; P_2 = \text{New}P_2; \ldots \]

Endloop

Example:

\[ T(x,y) :- R(x,y) \]
\[ T(x,y) :- R(x,z), T(z,y) \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i1} \leftarrow \text{body}_{1} \]
\[ P_{i2} \leftarrow \text{body}_{2} \]

Group by IDB predicate

\[ P_{1} \leftarrow \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_{2} \leftarrow \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]

Each rule is a Select-Project-Join-Union query

Naïve datalog evaluation algorithm:

\[ P_1 = P_2 = \ldots = \emptyset \]

Loop

\[ \text{NewP}_1 = \text{SPJU}_1; \text{NewP}_2 = \text{SPJU}_2; \ldots \]
\[ \text{if } (\text{NewP}_1 = P_1 \text{ and NewP}_2 = P_2 \text{ and } \ldots ) \]
\[ \text{then exit} \]

\[ P_1 = \text{NewP}_1; P_2 = \text{NewP}_2; \ldots \]

Endloop

Example:

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

\[ T(x,y) \leftarrow R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y)) \]
Naïve Datalog Evaluation Algorithm

Datalog program:

\[ P_{i_1} : \text{body}_1 \]
\[ P_{i_2} : \text{body}_2 \]
\[ \ldots \]

\[ P_1 : \text{body}_{11} \cup \text{body}_{12} \cup \ldots \]
\[ P_2 : \text{body}_{21} \cup \text{body}_{22} \cup \ldots \]
\[ \ldots \]

Each rule is a Select-Project-Join-Union query

Group by IDB predicate

Naïve datalog evaluation algorithm:

\[ P_1 = P_2 = \ldots = \emptyset \]

Loop

New\(P_1 = \text{SPJU}_1; \) New\(P_2 = \text{SPJU}_2; \) \ldots

if (New\(P_1 = P_1\) and New\(P_2 = P_2\) and \ldots) then exit

\[ P_1 = \text{New}P_1; \; P_2 = \text{New}P_2; \ldots \]

Endloop

Example:

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

\[ \Rightarrow T(x, y) : \text{R}(x, y) \cup \Pi_{xy}(\text{R}(x, z) \bowtie T(z, y)) \]

\[ T= \emptyset \]

Loop

New\(T(x, y) = \text{R}(x, y) \cup \Pi_{xy}(\text{R}(x, z) \bowtie T(z, y)) \)

if (New\(T = T\)) then exit

\[ T = \text{New}T \]

Endloop
Discussion

• A naïve datalog algorithm always terminates (why?)
  – Assuming no functions (+, *, ...)

• A datalog program always runs in PTIME in the size of the database (why?)
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)

$$V : - \text{body}$$

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:

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

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

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

Maintenace

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 $P_i$ 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**

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

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

then break

$$P_1 = P_1 \cup \Delta P_1; \ P_2 = P_2 \cup \Delta P_2; \ \ldots$$

**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 Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each $P_i$ defined by non-recursive-$\text{SPJU}_i$ and (recursive-)$\text{SPJU}_i$.

Example:

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

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

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

Loop

$$\Delta T(x,y) = (R(x,z) \bowtie \Delta T(z,y)) - R(x,y)$$

if ($\Delta T = \emptyset$)

then break

$$T = T \cup \Delta T$$

Endloop
**Semi-naïve Evaluation Algorithm**

Separate the Datalog program into the non-recursive, and the recursive part.

Each $P_i$ defined by non-recursive-$\text{SPJU}_i$ and (recursive-)$\text{SPJU}_i$.

<table>
<thead>
<tr>
<th>$P_1 = \Delta P_1 = \text{non-recursive-SPJU}_1$, $P_2 = \Delta P_2 = \text{non-recursive-SPJU}_2$, …</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop</strong></td>
</tr>
<tr>
<td>$\Delta P_1 = \Delta \text{SPJU}_1 - P_1$; $\Delta P_2 = \Delta \text{SPJU}_2 - P_2$; …</td>
</tr>
<tr>
<td>if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and …)</td>
</tr>
<tr>
<td>then break</td>
</tr>
<tr>
<td>$P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$; …</td>
</tr>
<tr>
<td><strong>Endloop</strong></td>
</tr>
</tbody>
</table>

**Example:**

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

Note: for any linear datalog programs, the semi-naïve algorithm has only one $\Delta$-rule for each rule!
Example

\[ R = \]

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

\[ \]

1 | 2  
1 | 4  
2 | 1  
2 | 3  
3 | 4  
4 | 5
Example

Initially:

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

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

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

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

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

Loop

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

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

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

Endloop
Example

Initially:

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

\[
\begin{align*}
\Delta T &= \\
\begin{array}{cc}
1 & 2 \\
1 & 4 \\
2 & 1 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
\end{array}
\end{align*}
\]

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

First iteration:

\[
\begin{align*}
\Delta T &= \\
\begin{array}{cc}
1 & 1 \\
1 & 3 \\
1 & 5 \\
2 & 2 \\
2 & 4 \\
3 & 5 \\
\end{array}
\end{align*}
\]

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

\[
\begin{align*}
T(x,y) &= R(x,y), \Delta T(x,y) = R(x,y) \\
\text{Loop} & \\
\Delta T(x,y) &= (R(x,z) \bowtie \Delta T(z,y)) - R(x,y) \\
\text{if } (\Delta T = \emptyset) \text{ break} & \\
T &= T \cup \Delta T \\
\text{Endloop}
\end{align*}
\]
Example

Initially:

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

First iteration:

\[ \Delta T(x,y) \]
\[ \Delta T(x,y) = (R(x,z) \bowtie \Delta T(z,y)) - R(x,y) \]
\[ \text{if } (\Delta T = \emptyset) \text{ break} \]
\[ T = T \cup \Delta T \]

Endloop

Second iteration:

\[ \Delta T(x,y) \]
\[ \Delta T(x,y) = (R(x,z) \bowtie \Delta T(z,y)) - R(x,y) \]
\[ \text{if } (\Delta T = \emptyset) \text{ break} \]
\[ T = T \cup \Delta T \]

Endloop
**Example**

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

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

Initially:

\[ R = \]

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

\[ \Delta T = \]

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

\[ T(\cdot,\cdot) : R(\cdot,\cdot) \]

First iteration:

\[ T(\cdot,\cdot) : \]

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

\[ \Delta T = \]

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

\[ \text{paths of length 2} \]

Second iteration:

\[ T(\cdot,\cdot) : \]

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

\[ \Delta T = \]

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

\[ \text{paths of length 3} \]

Third iteration:

\[ T(\cdot,\cdot) : \]

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

\[ \Delta T = \]

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

\[ \text{paths of length 4} \]

\[ \text{Loop} \]

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

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

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

\[ \text{Endloop} \]
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
Outline

• Datalog rules
• Recursion
• Semantics
• Negation, aggregates, stratification
• Naïve and Semi-naïve Evaluation
• Connection to RA – on your own
Datalog v.s. RA (and SQL)

• “Pure” datalog has recursion, but no negation, aggregates: all queries are monotone; impractical

• Datalog **without recursion**, plus negation and aggregates expresses the same queries as RA: next slides
RA to Datalog by Examples

Union:
\[ R(A,B,C) \cup S(D,E,F) \]

\[ U(x,y,z) :- R(x,y,z) \]
\[ U(x,y,z) :- S(x,y,z) \]
RA to Datalog by Examples

Intersection:
R(A,B,C) ∩ S(D,E,F)

I(x,y,z) :- R(x,y,z), S(x,y,z)
RA to Datalog by Examples

Selection: \( \sigma_{x>100 \text{ and } y='foo'}(R) \)
\[ L(x,y,z) : R(x,y,z), \ x > 100, \ y='foo' \]

Selection: \( \sigma_{x>100 \text{ or } y='foo'}(R) \)
\[ L(x,y,z) : R(x,y,z), \ x > 100 \]
\[ L(x,y,z) : R(x,y,z), \ y='foo' \]
RA to Datalog by Examples

Equi-join: \( R \bowtie_{R.A=S.D \text{ and } R.B=S.E} S \)

\( J(x,y,z,q) :- R(x,y,z), S(x,y,q) \)
RA to Datalog by Examples

Projection: $\Pi_A(R)$

$P(x) :- R(x,y,z)$
RA to Datalog by Examples

To express difference, we add negation

\[ R - S \]

\[ D(x,y,z) \leftarrow R(x,y,z), \text{NOT } S(x,y,z) \]
Examples

Translate: $\Pi_A(\sigma_{B=3} (R) )$

A(a) :- R(a,3,_) 

Underscore used to denote an "anonymous variable"
Each such variable is unique
Examples

Translate: $\prod_A (\sigma_{B=3} (R) \bowtie_{R.A=S.D} \sigma_{E=5} (S))$

$A(a) :- R(a,3,\_), S(a,5,\_)$

These are different "\_"s
More Examples w/o Recursion

Find Joe's friends, and Joe's friends of friends.

A(x) :- Friend('Joe', x)
A(x) :- Friend('Joe', z), Friend(z, x)
More Examples w/o Recursion

Find all of Joe's friends who do not have any friends except for Joe:

```
JoeFriends(x) :- Friend('Joe',x)
NonAns(x) :- JoeFriends(x), Friend(x,y), y != 'Joe'
A(x) :- JoeFriends(x), NOT NonAns(x)
```
More Examples w/o Recursion

Find all people such that all their enemies' enemies are their friends

• Q: if someone doesn't have any enemies nor friends, do we want them in the answer?
• A: Yes!

\begin{align*}
\text{Everyone}(x) & : \text{Friend}(x,y) \\
\text{Everyone}(x) & : \text{Friend}(y,x) \\
\text{Everyone}(x) & : \text{Enemy}(x,y) \\
\text{Everyone}(x) & : \text{Enemy}(y,x) \\
\text{NonAns}(x) & : \text{Enemy}(x,y), \text{Enemy}(y,z), \text{NOT} \text{Friend}(x,z) \\
\text{A}(x) & : \text{Everyone}(x), \text{NOT} \text{NonAns}(x)
\end{align*}
More Examples w/o Recursion

Find all persons $x$ that have a friend all of whose enemies are $x$'s enemies.

\begin{align*}
\text{Everyone}(x) & : \text{Friend}(x,y) \\
\text{NonAns}(x) & : \text{Friend}(x,y) \ \text{Enemy}(y,z), \ \text{NOT} \ \text{Enemy}(x,z) \\
A(x) & : \text{Everyone}(x), \ \text{NOT} \ \text{NonAns}(x)
\end{align*}
More Examples w/ Recursion

• Two people are in the same generation if they are siblings, or if they have parents in the same generation

• Find all persons in the same generation with Alice
More Examples w/ Recursion

• Find all persons in the same generation with Alice
• Let’s compute \( SG(x,y) = "x, y are in the same generation" \)

\[
\begin{align*}
SG(x,y) & : \text{ParentChild}(p,x), \text{ParentChild}(p,y) \\
SG(x,y) & : \text{ParentChild}(p,x), \text{ParentChild}(q,y), \text{SG}(p,q) \\
\text{Answer}(x) & : \text{SG}("Alice", x)
\end{align*}
\]
Datalog Summary

- EDB (base relations) and IDB (derived relations)
- Datalog program = set of rules
- Datalog is recursive

Some reminders about semantics:
- Multiple atoms in a rule mean join (or intersection)
- Variables with the same name are join variables
- Multiple rules with same head mean union
Datalog and SQL

- Stratified data (w/ recursion, w/o +, *, ...): expresses precisely* queries in PTIME
  - Cannot find a Hamiltonian cycle (why?)

- SQL has also been extended to express recursive queries:
  - Use a recursive “with” clause, also CTE (Common Table Expression)
  - Often with bizarre restrictions...
  - ... Just use datalog

* need to use the < predicate