Introduction to Data Management

Datalog (continued)

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

HW 3 due tonight!
HW 4 out tomorrow
• ‘written’ RA portion and Datalog programming
Outline

Review of recursive example

Negation

Query safety

Stratification
Recursive Datalog

Graph reachability has recursive semantics
- Flights: Can I reach city A from city B?
- Family lineage: Who are my ancestors?
- ...

.decl Edge(...)  
Edge(1, 2).  
Edge(2, 1).  
Edge(2, 3).  
Edge(1, 4).  
Edge(3, 4).  
Edge(4, 5).
Recursive Datalog

Graph reachability has recursive semantics
  • Flights: Can I reach city A from city B?
  • Family lineage: Who are my ancestors?
  • ...

```datalog
.decl Edge(...)

Edge (1, 2).
Edge (2, 1).
Edge (2, 3).
Edge (1, 4).
Edge (3, 4).
Edge (4, 5).
```
Recursive Datalog

Graph reachability has recursive semantics

• Flights: Can I reach city A from city B?
• Family lineage: Who are my ancestors?
• ...

Find all pairs of nodes where one is reachable from another

\[ Q(a, b) : \neg \text{Edge}(a, b) . \]
\[ Q(a, b) : \neg Q(a, x), \]
\[ \quad \text{Edge}(x, b) . \]
Sets, not bags

- Datalog is defined purely over sets
- No duplicates allowed: automatic de-duplication
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

\[
\text{Q: } \begin{bmatrix} a & b \end{bmatrix}
\]

Edge(1, 2).
Edge(2, 1).
Edge(2, 3).
Edge(1, 4).
Edge(3, 4).
Edge(4, 5).

\[
Q(a, b) :- \text{Edge}(a, b).
Q(a, b) :- Q(a, x),
\text{Edge}(x, b).
\]
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

Q(a, b) :- Edge(a, b).
Q(a, b) :- Q(a, x),
          Edge(x, b).

Edge(1, 2).
Edge(2, 1).
Edge(2, 3).
Edge(1, 4).
Edge(3, 4).
Edge(4, 5).
Fixed-Point Semantics

Main Idea: Keep executing until no new results are added (until a fixed point is reached).

Edge(1, 2).
Edge(2, 1).
Edge(2, 3).
Edge(1, 4).
Edge(3, 4).
Edge(4, 5).

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

Q(a, b) \ :- \ Edge(a, b).
Q(a, b) \ :- \ Q(a, x),
\hspace{1cm} Edge(x, b).
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

\[ Q(a, b) : - \text{Edge}(a, b). \]
\[ Q(a, b) : - Q(a, x), \]
\[ \quad \text{Edge}(x, b). \]
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

Q(a, b) :- Edge(a, b).
Q(a, b) :- Q(a, x),
            Edge(x, b).
Fixed-Point Semantics

Main Idea: Keep executing until no new results are added (until a fixed point is reached).

Q(a, b) :- Edge(a, b).
Q(a, b) :- Q(a, x),
              Edge(x, b).

<table>
<thead>
<tr>
<th>a</th>
<th>b</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>

Datalog

Edge(1, 2).
Edge(2, 1).
Edge(2, 3).
Edge(1, 4).
Edge(3, 4).
Edge(4, 5).
Fixed-Point Semantics

Main Idea: Keep executing until no new results are added (until a fixed point is reached).

\[
\begin{align*}
\text{Q(a, b)} & : \text{Edge(a, b).} \\
\text{Q(a, b)} & : \text{Q(a, x), Edge(x, b).}
\end{align*}
\]
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

\[
\begin{align*}
Q(a, b) & :- \text{Edge}(a, b). \\
Q(a, b) & :- Q(a, x), \\
& \quad \text{Edge}(x, b).
\end{align*}
\]
Functions and Termination

Including functions (+, *, ...) might cause non-termination in recursive queries

// distance between two nodes
Q(a, b, 1) :- Edge(a, b).
Q(a, b, n+1) :- Q(a, x, n),
    Edge(x, b).

1

\[ a \quad b \quad n \]

2
Including functions (+, *, ...) might cause non-termination in recursive queries

// distance between two nodes
Q(a, b, 1) :- Edge(a, b).
Q(a, b, n+1) :- Q(a, x, n),
                Edge(x, b).

Datalog

\[
\begin{array}{ccc}
  a & b & n \\
  \hline \\
  1 & 2 & 1 \\
  2 & 1 & 1 \\
\end{array}
\]
Including functions (+, *, ...) might cause non-termination in recursive queries

// distance between two nodes
Q(a, b, 1) :- Edge(a, b).
Q(a, b, n+1) :- Q(a, x, n),
              Edge(x, b).
Including functions (+, *, ...) might cause non-termination in recursive queries

// distance between two nodes
Q(a, b, 1) :- Edge(a, b).
Q(a, b, n+1) :- Q(a, x, n),
              Edge(x, b).

1

a b n
1 2 1
2 1 1

2

a b n
1 2 1
2 1 1
1 1 2
2 2 2

n

1 2 1
2 1 1
1 1 2
2 2 2
1 2 3
2 1 3
Including functions (+, *, ...) might cause non-termination in recursive queries

// distance between two nodes
Q(a, b, 1) :- Edge(a, b).
Q(a, b, n+1) :- Q(a, x, n),
                Edge(x, b).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

... Always more results because n keeps increasing!
Descendants Example

A genealogy database records parent/child relationship

ParentChild

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>Dave</td>
</tr>
<tr>
<td>Carol</td>
<td>Fred</td>
</tr>
<tr>
<td>Carol</td>
<td>Eve</td>
</tr>
<tr>
<td>Dave</td>
<td>Eve</td>
</tr>
<tr>
<td>Dave</td>
<td>Gil</td>
</tr>
<tr>
<td>Fred</td>
<td>Gil</td>
</tr>
</tbody>
</table>
Find each person’s descendents
Descendents Example

Find each person’s descendents

// compute descendents of x
D(x, y) :- ParentChild(x, y).
D(x, y) :- D(x, z), ParentChild(z, y).

Alice
  ↓
  ↓
Carol
  ↓
  ↓
Fred
  ↓
  ↓
Eve
  ↓
  ↓
Dave
  ↓
  ↓
Bob
  ↓
  ↓
Gil
Negation

Denoted by “!”
- Souffle uses “!”
- More generally the word “NOT” is used, but souffle will not accept that syntax

Meaning is negated existential semantics
Find all descendents of Bob who are not descendents of Alice

Answer

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dave</td>
</tr>
</tbody>
</table>
Find all descendents of Bob who are not descendents of Alice

// compute descendents of x
D(x, y) :- ParentChild(x, y).
D(x, y) :- D(x, z), ParentChild(z, y).

D(x, y) :- ParentChild(x, y).
Find all descendents of Bob who are not descendents of Alice

// compute descendents of x
D(x,y) :- ParentChild(x,y).
D(x,y) :- D(x,z), ParentChild(z,y).

// name and count
Ans(x) :- D('Bob',x), !D('Alice',x).
Find all descendents of Bob who are not descendents of Alice

// compute descendents of x
D(x,y) :- ParentChild(x,y).
D(x,y) :- D(x,z), ParentChild(z,y).

// name and count
Ans(x) :- D('Bob',x), !D('Alice',x).

All possible answers

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
</tr>
<tr>
<td>Dave</td>
</tr>
<tr>
<td>Eve</td>
</tr>
<tr>
<td>Fred</td>
</tr>
<tr>
<td>Gil</td>
</tr>
</tbody>
</table>

All non-answers

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
</tr>
<tr>
<td>Eve</td>
</tr>
<tr>
<td>Fred</td>
</tr>
<tr>
<td>Gil</td>
</tr>
</tbody>
</table>
We want our query results bound to existing values only.

Unsafe queries are those queries that do not always have the same finite results.

- Domain-dependent answers or
- Infinite solutions
Unsafe Rules

We want our query results bound to existing values only.

Unsafe queries are those queries that do not always have the same finite results.

• Domain-dependent answers or
• Infinite solutions

U1(x,y) :- ParentChild(“Alice”,x), y ≠ “Bob”

U2(x) :- ParentChild(“Alice”,x), !ParentChild(x,y)
Unsafe Rules

We want our query results bound to existing values only.

Unsafe queries are those queries that do not always have the same finite results.

• Domain-dependent answers or
• Infinite solutions

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

Unsafe: Holds for every \( y \) other than “Bob”. Since \( y \) does not have to be in a related, output is infinite!

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

We want our query results bound to existing values only

Unsafe queries are those queries that do not always have the same finite results.

- Domain-dependent answers or
- Infinite solutions

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

Unsafe: Holds for every \( y \) other than “Bob”. Since \( y \) does not have to be in a related, output is infinite!

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

Unsafe: We get all \( x \) still - \( y \) is unbound, so there always exists some \( y \) that \( x \) is not parent of
Safe Rules

- In Datalog we always want to write “safe” rules
- Let’s define what a safe rule is
Safe Rules

Datalog rules are safe if every variable appears in some positive relational atom

- Positive atom implicitly restricts/defines domain of its variables

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

Now the domain of Bob is restricted, so query is safe
Datalog rules are safe if every variable appears in some positive relational atom

- Positive atom implicitly restricts/defines domain of its variables

\[ U_1(x,y) :- \text{ParentChild}("Alice", x), \text{ParentChild}(y,_), y \neq "Bob" \]

Now the domain of Bob is restricted, so query is safe

\[ \text{Parents}(x) :- \text{ParentChild}(x, _) \]
\[ U_2(x) :- \text{ParentChild}("Alice", x), \neg \text{Parents}(x) \]

Now we don’t have the y domain to worry about, so query is safe
Safe Rules

Datalog rules are safe if **every variable appears in some positive relational atom**

- Positive atom implicitly restricts/defines domain of its variables

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

Now the domain of Bob is restricted, so query is safe

Parents(x) :- ParentChild(x, _)
U2(x) :- ParentChild("Alice", x), ! Parents(x)

Now we don’t have the y domain to worry about, so query is safe
```
We have one more desired property for Datalog programs:

Stratification
Stratification

In theory, Datalog rules can be declared in any order.

In practice, Execution of queries for most Datalog DBMS must be divided into strata.

- Why? Recursion doesn’t cope well with aggregates and negation.

Not valid in Souffle:

```plaintext
A () :- !B () .
B () :- !A () .
```
In theory, Datalog rules can be declared in any order.

In practice, Execution of queries for most Datalog DBMS must be divided into strata.

- Why? Recursion doesn’t cope well with aggregates and negation.

```
Not valid in Souffle:
A() :- !B().
B() :- !A().
```

- Problem: B not defined before being used in the first rule.
A Datalog program is stratified if it can be partitioned in strata
Rules in strata 1….n can be used in aggregates or negation in strata n+1

// compute descendents of x
D(x,y) :- ParentChild(x,y).
D(x,y) :- D(x,z), ParentChild(z,y).

// name and count
Ans(x) :- D('Bob',x), !D('Alice',x).
Takeaways

Datalog lets us use recursion with negation to answer questions we couldn’t answer with SQL.

We need to be careful to write safe, stratified queries when using these features.