Introduction to Data Management

Datalog

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Recap: Facts and Rules

- Some terminology:
  - **Facts**: tuples in the database
  - **Rules**: queries

Find all names of movies made in 2012

```
Q1(n) :- Movie(_, n, 2012).
```

Find all actor names cast in movies made in 2012

```
Q2(f, l) :- Movie(mid, _, 2012),
          Casts(aid, mid),
          Actor(aid, f, l).
```

Find all actor names cast in movies made in 2008 and 2012

```
Q3(f, l) :- Actor(aid, f, l), Casts(aid, mid1), Movie(mid1, _, 2012),
          Casts(aid, mid2), Movie(mid2, _, 2008).
```
Recap: Datalog Terminology

Q2(f, l) :- Movie(mid, _, 2012), Casts(aid, mid), Actor(aid, f, l).

Head variable

Existential variable
Recap: Datalog Terminology

Q4(n) :- Movie(_, n, y), y > 2012.

Relational predicate

Arithmetic predicate

Evaluates to true when the relation contains the tuple described by the arguments
Recap: Datalog Union/OR

Actor(123, “Robert”, “Downey”).
Actor(345, “Gal”, “Gadot”).
Movie(000, “Iron Man”, 2008)
Casts(123, 000).
Casts(123, 999).
Casts(345, 888).

Q(f, l) :- Movie(mid, _, 2012), Casts(aid, mid), Actor(aid, f, l).
Q(f, l) :- Movie(mid, “Justice League”, _), Casts(aid, mid), Actor(aid, f, l).

Datalog syntax is limited so Union/OR is implemented by “building” the results with multiple definitions

“,” in query body encodes conjunction (AND)

Disjunctive Normal Form (DNF)
i.e. OR of ANDs
Recap: Fixed-Point Semantics

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

\[
Q(a, b) :- Edge(a, b).
\]
\[
Q(a, b) :- Q(a, x),
Edge(x, b).
\]

```
1 2
2 1
2 3
1 4
3 4
4 5
```

```
1 2
2 1
2 3
1 4
3 4
4 5
1 1
2 2
1 3
2 4
1 5
3 5
2 5
```

The graph shows the relationships between nodes, with edges indicating the order of execution. The process continues until all nodes are processed, as indicated by the "finished" note.

October 16, 2019
Outline

▪ More Datalog features:
  • Aggregates
  • Grouping
  • Negation
▪ Query safety
▪ Stratification
Find the year the first Spider-Man movie came out

Q(year) :- year = min \( y \) \{ Movie(_, n, y), n = “Spider-Man” \}.

In SQL:

```
SELECT MIN(m.year) AS minyear
FROM Movie m
WHERE m.name = "Spider-Man"
```
Assign variable to the result of the aggregation

Find the year the first Spider-Man movie came out

Q(year) :- year = \( \text{min \ y : \{ Movie(_, n, \text{y}), n = \text{“Spider-Man”}\} } \).

Aggregate function

In SQL:

```
SELECT MIN(m.year) AS year
FROM Movie m
WHERE m.name = "Spider-Man"
```
Aggregation

Find the year the first Spider-Man movie came out

\[ Q(\text{year}) : \text{-} \text{year} = \text{min} \ y \ \{ \text{Movie}(\_ , n, y), n = "\text{Spider-Man}" \} . \]

Aggregates in Souffle:

- **MIN**
- **MAX**
- **COUNT**
- **SUM**
Counting

Find the number of movies named Spider-Man

\[
Q(c) \leftarrow c = \text{count} : \{ \text{Movie}(_, \ n, \ _) \mid n = \text{"Spider-Man"} \}.
\]

No variable here!

.decl Actor(...)
decl Movie(...)decl Casts(...)

Actor(123, “Robert”, “Downey”).
Actor(345, “Gal”, “Gadot”).
Movie(000, “Iron Man”, 2008)
Casts(123, 000).
Casts(123, 999).
Casts(345, 888).
How many movies were made each year?

\[ Q(y, c) := \text{Movie}(_, _, y), \quad c = \text{count} \{ \text{Movie}(_, _, y) \} \].

Datalog:

.decl Actor(…)
.decl Movie(…)
.decl Casts(…)

Actor(123, "Robert", "Downey").
Actor(345, "Gal", "Gadot").
Movie(345, "Spider-Man", 1978)
Movie(000, "Iron Man", 2008)
Casts(123, 000).
Casts(123, 999).
Casts(345, 888).

SQL:

In SQL:

```
SELECT m.year, COUNT(*) AS c
FROM Movie m
GROUP BY m.year
```
Descendents Example

- A genealogy database records parent/child relationship

![Genealogy Diagram]

<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>
Descendents Example

- Count each person’s descendents

```
<table>
<thead>
<tr>
<th>p</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>3</td>
</tr>
<tr>
<td>Dave</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Diagram:
- Alice
  - Carol
  - Eve
  - Fred
- Bob
  - Dave
  - Gil
Descendents Example

- Count each person’s descendents

Alice → Carol → Eve → Fred
Bob → Dave

Answer

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

Eve and Gil don’t appear in results
Descendents Example

- Count each person’s descendants

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

Alice
   └── Bob
      │   └── Dave
      │        └── Eve
      │            └── Fred
      │                        └── Gil
Descendents Example

- Count each person’s descendants

Alice  Dave  Bob
Carol  Eve  Fred  Gil

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

- Count each person’s descendents

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

// name and count
Ans(n,c) :- ParentChild(n,_),
c = count : { D(n,_) }.
Negation

- Denoted by “!”
- Negated existential semantics
Negation Example

- Find all descendents of Bob who are not descendents of Alice

![Family Tree Diagram]

Answer

\( p \)

Dave
Negation Example

- 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).
```
Negation Example

- Find all descendents of Bob who are not descendents of Alice

```datalog
// 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).
```
Negation Example

- 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>
Descendents Example

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

\[
\begin{array}{c}
\text{Alice} \\
\text{Carol} \\
\text{Fred} \\
\text{Bob} \\
\text{Dave} \\
\text{Eve} \\
\text{Gil}
\end{array}
\]

SameGeneration

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>Dave</td>
</tr>
<tr>
<td>Eve</td>
<td>Gil</td>
</tr>
<tr>
<td>Eve</td>
<td>Fred</td>
</tr>
<tr>
<td>Fred</td>
<td>Gil</td>
</tr>
</tbody>
</table>
Two people are in the same generation if they are descendants at the same generation of some common ancestor.

- **Alice**
  - **Carol**
    - **Eve**
      - **Fred**
  - **Bob**
    - **Dave**
      - **Gil**

```
// common parents
SG(x,y) :- ParentChild(a,x), ParentChild(a,y).
// parents in same generation
SG(x,y) :- SG(a,b), ParentChild(a,x), ParentChild(b,y).
```
Two people are in the same generation if they are descendants at the same generation of some common ancestor.

// common parents
SG(x,y) :- ParentChild(a,x), ParentChild(a,y).

// parents in same generation
SG(x,y) :- SG(a,b), ParentChild(a,x), ParentChild(b,y).

But this will include answers such as SG(Carol,Carol) or SG(Eve,Fred) AND SG(Fred,Eve)

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

- Alice
- Carol
- Eve
- Fred
- Bob
- Dave
- Gil

// common parents
\[ SG(x, y) :- ParentChild(a, x), ParentChild(a, y), x < y. \]

// parents in same generation
\[ SG(x, y) :- SG(a, b), ParentChild(a, x), ParentChild(b, y), x < y. \]
Query Safety

- 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

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

\[ U_2(x) \) :- ParentChild("Alice",x), !ParentChild(x,y) \]

\[ U_3(\text{minId}, y) \) :- minId = min x : \{ \text{Actor}(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) :- ParentChild("Alice", x), y != "Bob"

Holds for every y other than "Bob" - infinite!

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

U3(minId, y) :- minId = min x : { Actor(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

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

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

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

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

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

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

Holds for every \( y \) other than "Bob" - infinite!

\[
U_2(x) \; : \; \text{ParentChild}(\text{"Alice"}, x), \; \neg \text{ParentChild}(x, y)
\]

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

\[
U_3(\text{minId}, y) \; : \; \text{minId} = \text{min} \; x \; : \; \{ \text{Actor}(x, y, _) \}
\]

Unclear what \( y \) is - like our SQL aggregate
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
Stratification

- Datalog rules can be declared in any order
- Execution of queries for most Datalog DBMS is divided into strata
  - Why? Recursion doesn’t cope well with aggregates and negation

Not valid in Souffle:

\[
\begin{align*}
A() & : - \neg B(). \\
B() & : - \neg A().
\end{align*}
\]
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).

Stratum 1

Stratum 2

Depends on
Takeaways

- Datalog lets us use recursion with negation and aggregates to answer questions we couldn’t answer with SQL
- We need to be careful to write safe, stratified queries when using these features