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
Outline

- Why Datalog?
- Terminology and semantics
- Examples and demo
• We have SQL and RA. Why would we want to know about yet another query language?
  • Recursive queries!
  • Arguably simpler to use
  • Used frequently in research and data analytics

• Declarative logic programming language
  • Derived from another programming language: Prolog
  • Similar structure to first-order logic
Example: Facebook friends

As a graph

- Peter
- Mary
- Phil
- John

As a relation

<table>
<thead>
<tr>
<th>Person1</th>
<th>Person2</th>
<th>IsFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Mary</td>
<td>1</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Peter</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Phil</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Compute your friends graph

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>IsFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Mary</td>
<td>1</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Peter</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Phil</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**My friends**

```sql
SELECT F.P2
FROM Friends F
WHERE F.P1 = "me" AND F.IsFriend = 1
```

**My friends of friends**

```sql
SELECT F.P2
FROM Friends F1, Friends F2
WHERE F1.P1 = "me" AND F1.IsFriend = 1 AND F1.P2 = F2.P1 AND F2.IsFriend
```

**My fofofs ... my fofofofs ...**

**Where does it end??**
Example: Facebook friends

Compute your friends graph

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>IsFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Mary</td>
<td>1</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Peter</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Phil</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

My friends

```sql
SELECT F.P2
FROM Friends F
WHERE F.P1 = "me" AND F.IsFriend = 1
```

My friends of friends

```sql
SELECT F.P2
FROM Friends F1, Friends F2
WHERE F1.P1 = "me" AND F1.IsFriend = 1
AND F1.P2 = F2.P1 AND F2.IsFriend
```

My fofofs ... my fofofofs ...

Where does it end??

Datalog will allow us to write recursive queries
Souffle

- DBMS that uses Datalog as its query language
  - Data Model:
    - Data instance: still relational
    - Schema: still relational
    - Query language: Datalog rather than SQL
- Open-source project from Oracle
- **Pure set semantics** (finally!)
Souffle

- DBMS that uses Datalog as its query language
  - Data Model:
    - Data instance: still relational
    - Schema: still relational
    - Query language: Datalog rather than SQL
- Open-source project from Oracle
- Pure set semantics (finally!)

To play around with it now:
sudo yum install souffle
Facts and Rules

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

Only two types in Souffle:
- symbol (strings)
- number (integers)

**Schema declaration**

```datalog
.decl Actor(id:number, fname:symbol, lname:symbol)
.decl Movie(id:number, name:symbol, year:number)
.decl Casts(aid:number, mid:number)
```

```
Actor(123, "Robert", "Downey").
Actor(345, "Gal", "Gadot").
Movie(000, "Iron Man", 2008)
Casts(123, 000).
Casts(123, 999).
Casts(345, 888).
```
Some terminology:

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

---

**Schema declaration**

```datalog
.decl Actor(id:number, fname:symbol, lname:symbol)
.decl Movie(id:number, name:symbol, year:number)
.decl Casts(aid:number, mid:number)
```

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

---

Only two types in Souffle:
- symbol (strings)
- number (integers)
Facts and Rules

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

```
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).
```
Some terminology:

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

Find all names of movies made in 2012

```datalog
.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).
```
Facts and Rules

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

```datalog
.decl Actor(...) 
.decl Movie(...) 
.decl Casts(...) 

Actor(123, "Robert", "Downey"). 
Actor(345, "Gal", "Gadot"). 
Movie(000, "Iron Man", 2008) 
Movie(999, "Avengers", 2012) 
Casts(123, 000). 
Casts(123, 999). 
Casts(345, 888).
```

Find all names of movies made in 2012

```datalog
.decl Q1(n:symbol) 
Q1(n) :- Movie(i, n, y), y = 2012. 
```
Facts and Rules

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

```
.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).  
```

Find all names of movies made in 2012

```
Q1(n) :- Movie(i, n, y), y = 2012.
```

Variables correspond to declared attributes
Some terminology:

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

Find all names of movies made in 2012

Q1(n) :- Movie(_, n, y), y = 2012.

“_” means we don’t care about the value
Facts and Rules

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

Find all names of movies made in 2012

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

Actor(123, "Robert", "Downey").
Actor(345, "Gal", "Gadot").
Movie(000, "Iron Man", 2008)
Movie(999, "Avengers", 2012)
Movie(888, "Justice League", 2017)
Casts(123, 000).
Casts(123, 999).
Casts(345, 888).
```

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

Implied equality
Facts and Rules

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

Find all names of movies made in 2012

```datalog
.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). 

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

In SQL:

```
SELECT m.name 
FROM Movie m 
WHERE m.year = 2012 
```
Facts and Rules

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

Find all names of movies made in 2012

\[ Q1(n) :- \text{Movie}(_, n, 2012). \]

Find all actor names cast in movies made in 2012

.\text{decl} \text{Actor}(\ldots)
.\text{decl} \text{Movie}(\ldots)
.\text{decl} \text{Casts}(\ldots)

\text{Actor}(123, \text{“Robert”, “Downey”}).
\text{Actor}(345, \text{“Gal”, “Gadot”}).
\text{Movie}(000, \text{“Iron Man”, 2008})
\text{Movie}(999, \text{“Avengers”, 2012}).
\text{Movie}(888, \text{“Justice League”, 2017}).
\text{Casts}(123, 000).
\text{Casts}(123, 999).
\text{Casts}(345, 888).
Facts and Rules

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

### Examples

- **Find all names of movies made in 2012**
  
  ```
  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).
  
  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).
  ```

  *Same variable, different predicate is an equijoin*
Facts and Rules

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

```datalog
.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).
```

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

Find all names of movies made in 2012

Find all actor names cast in movies made in 2012

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

In SQL:

```sql
SELECT a.fname, a.lname
FROM Movie m, Casts c, Actor a
WHERE m.id = c.mid AND c.aid = a.id
    AND m.year = 2012
```
Facts and Rules

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

---

### Facts

- `Actor(123, "Robert", "Downey")`
- `Actor(345, "Gal", "Gadot")`
- `Movie(000, "Iron Man", 2008)`
- `Movie(999, "Avengers", 2012)`
- `Movie(888, "Justice League", 2017)`
- `Casts(123, 000)`
- `Casts(123, 999)`
- `Casts(345, 888)`

---

### Rules

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

Find all names of 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 2012

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

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

Find all names of movies made in 2012

```
.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). 
```

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

Find all actor names cast in movies made in 2012

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

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

Q3(f, l) :- 甲方(aid, f, l), Casts(aid, mid1), Movie(mid1, _, 2012), 
            Casts(aid, mid2), Movie(mid2, _, 2008).
Facts and Rules

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

### Extensional Database Predicates (EDBs)

```datalog
.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).
```

### Intensional Database Predicates (IDBs)

```datalog
.decl Q1(...) 
.decl Q2(...) 
.decl Q3(...) 

Q1(n) :- Movie(_, n, 2012).
Q2(f, l) :- Movie(mid, _, 2012), Casts(aid, mid), Actor(aid, f, l).
Q3(f, l) :- Actor(aid, f, l), Casts(aid, mid1), Movie(mid1, _, 2012), Casts(aid, mid2), Movie(mid2, _, 2008).
```
Datalog Terminology

Head

\( Q2(f, l) :- \)

Body

\( \text{Movie}(\text{mid}, _, 2012), \text{Casts}(\text{aid}, \text{mid}), \text{Actor}(\text{aid}, f, l). \)
Q2(f, l) :- Movie(mid, _, 2012), Casts(aid, mid), Actor(aid, f, l).
Datalog Terminology

Find all names of movies made since 2012

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

.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).
Datalog Terminology

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

Relational predicate

Arithmetic predicate
Datalog Terminology

\[ Q_4(n) \text{ :- Movie}(_{\text{rel}}, n, y), \quad y > 2012. \]

Relational predicate

Arithmetic predicate

Evaluates to true when the relation contains the tuple described by the arguments.
Existential Semantics

In general, we return all values of head variables such that there exists existential variable values described by the query.

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

Q2(f, l) = ∃mid. ∃aid. ∃x. (Movie(mid, x, 2012) ∧ Casts(aid, mid) ∧ Actor(aid, f, l))
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).

"." in query body encodes conjunction (AND)
Datalog syntax is limited so Union/OR is implemented by “building” the results with multiple definitions.

```
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).
```

"," in query body encodes conjunction (AND).

Disjunctive Normal Form (DNF) i.e. OR of ANDs.
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?
  • ...

```prolog
.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

\[
Q(a, b) : \neg \text{Edge}(a, b).
\]
\[
Q(a, b) : \neg Q(a, x),
\quad \text{Edge}(x, b).
\]
Fixed-Point Semantics

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

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

\[
\begin{align*}
Q(a, b) & : \text{ Edge}(a, b). \\
Q(a, b) & : Q(a, x), \\
& \quad \text{ Edge}(x, b).
\end{align*}
\]
Fixed-Point Semantics

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

\[ Q(a, b) \leftarrow \text{Edge}(a, b). \]
\[ Q(a, b) \leftarrow Q(a, x), \text{Edge}(x, b). \]

\[ \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]
Main Idea: Keep executing until no new results are added (until a fixed point is reached).

\[
Q(a, b) \leftarrow \text{Edge}(a, b).
\]

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

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

\[
\begin{align*}
Q(a, b) & \leftarrow \text{Edge}(a, b). \\
Q(a, b) & \leftarrow Q(a, x), \text{Edge}(x, b).
\end{align*}
\]
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

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

```
a | b | n
---|---|---
1  | 2 | 1
2  | 1 | 1
```
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).
```
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).

Always more results because n keeps increasing!
Takeaways

- Datalog is another relational query language
- Logical programming languages let us write simple statements that pack a lot of power