Introduction to Database Systems
CSE 414

Lecture 9: Datalog
Class Overview

• Unit 1: Intro
• Unit 2: Relational Data Models and Query Languages
  – Data models, SQL, Relational Algebra, Datalog
• Unit 3: Non-relational data
• Unit 4: RDMBS internals and query optimization
• Unit 5: Parallel query processing
• Unit 6: DBMS usability, conceptual design
• Unit 7: Transactions
What is Datalog?

• Another query language for relational model
  – Designed in the 80’s
  – Simple, concise, elegant
  – Extends relational queries with recursion

• Today is a hot topic:
  – Souffle (we will use in HW4)
  – Differential datalog
  – Beyond databases in many research projects: network protocols, static program analysis
• Open-source implementation of Datalog DBMS
• Under active development
• Commercial implementations are available
  – More difficult to set up and use
• “sqlite” of Datalog
  – Set-based rather than bag-based

• Install in your VM
  – Run `sudo yum install souffle` in terminal
  – More details in upcoming HW4
Why bother with *yet* another relational query language?
Example: storing FB friends

As a graph

Or

<table>
<thead>
<tr>
<th>Person1</th>
<th>Person2</th>
<th>is_friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Mary</td>
<td>0</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Phil</td>
<td>Peter</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We will learn the tradeoffs of different data models later this quarter
Compute your friends graph

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>isFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Mary</td>
<td>0</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Phil</td>
<td>Peter</td>
<td>1</td>
</tr>
</tbody>
</table>

Friends(p1, p2, isFriend)

```
SELECT f.p2
FROM Friends as f
WHERE f.p1 = 'me' AND f.isFriend = 1
```

My own friends

```
SELECT f1.p2
FROM Friends as f1,
(SELECT f.p2
 FROM Friends as f
 WHERE f.p1 = 'me' AND f.isFriend = 1) as f2
WHERE f1.p1 = f2.p2 AND f1.isFriend = 1
```

My FoF

```
(SELECT f.p2
 FROM Friends as f
 WHERE f.p1 = 'me' AND f.isFriend = 1) as f2
```

My FoFoF

```
(SELECT f.p2
 FROM Friends as f
 WHERE f.p1 = 'me' AND f.isFriend = 1) as f2
```

My FoFoFoF

Datalog allows us to write recursive queries easily
Datalog: Facts and Rules

**Facts** = tuples in the database  
**Rules** = queries

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

**Facts** = tuples in the database

**Rules** = queries

Table declaration

Types in Souffle:
- number
- symbol (aka varchar)

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).
### Datalog: Facts and Rules

**Facts** = tuples in the database

**Rules** = queries

<table>
<thead>
<tr>
<th>Facts</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor(344759, ‘Douglas’, ‘Fowley’)</td>
<td>Q1(y) :- Movie(x,y,z), z=1940.</td>
</tr>
<tr>
<td>Casts(344759, 29851)</td>
<td></td>
</tr>
<tr>
<td>Casts(355713, 29000)</td>
<td></td>
</tr>
<tr>
<td>Movie(7909, ‘A Night in Armour’, 1910)</td>
<td></td>
</tr>
<tr>
<td>Movie(29000, ‘Arizona’, 1940)</td>
<td></td>
</tr>
<tr>
<td>Movie(29445, ‘Ave Maria’, 1940)</td>
<td></td>
</tr>
</tbody>
</table>
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).
```
Datalog: Facts and Rules

Facts = tuples in the database

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Rules = queries

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

SQL

SELECT name
FROM Movie
WHERE year = 1940

Find Movies made in 1940

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 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.

Order of variable matters!

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(29445, ‘Ave Maria’, 1940).

Rules = queries

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

Find Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

```prolog
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

```prolog
Q1(y) :- Movie(_, y, z), z=1940.
```

_ = “don’t care” variables

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).
Datalog: Facts and Rules

**Facts** = tuples in the database

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Find Actors who acted in Movies made in 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(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).
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).
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
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

- \( R_i(args_i) \) called an **atom**, or a **relational predicate**
- \( R_i(args_i) \) evaluates to true when relation \( R_i \) contains the tuple described by \( args_i \).
  - Example: Actor(344759, ‘Douglas’, ‘Fowley’) is true
- In addition we can also have arithmetic predicates
  - Example: \( z > 1940 \).
- Book uses **AND** instead of ,

\[
Q(args) :- R1(args), R2(args), ...
\]
Datalog program

• A Datalog program consists of several rules
• Importantly, rules may be recursive!
  – Recall CSE 143!
• Usually there is one distinguished predicate that’s the output
• We will show an example first, then give the general semantics.
Example

R encodes a graph e.g., connected cities

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

R encodes a graph e.g., connected cities

\[ R = \]

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

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

What does it compute?

Multiple rules for the same IDB means OR
Example

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

Initially:
T is empty.

<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>
Example

R encodes a graph
e.g., connected cities

Initially:
T is empty.

First iteration:
T =

First rule generates this

Second rule
generates nothing
(because T is empty)
R encodes a graph
e.g., connected cities

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

Initially: T is empty.

First iteration: T =

Second iteration:

\[ T(x,y) \rightarrow R(x,y). \]
\[ T(x,y) \rightarrow R(x,z), T(z,y). \]

What does it compute?

New facts

First rule generates this

Second rule generates this

Example
Example

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>

Initially: T is empty.

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>

Both rules

First rule

Second rule

New fact

What does it compute?

T(x,y) ::= R(x,y).
T(x,y) ::= R(x,z), T(z,y).

CSE 414 - Autumn 2018
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.

First iteration:

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

Second iteration:

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

Third iteration:

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

Fourth iteration:

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

No new facts. DONE

What does it compute?

R encodes a graph e.g., connected cities

CSE 414 - Autumn 2018
More Features

• Aggregates

• Grouping

• Negation
Aggregates

[aggregate name] <var> : { [relation to compute aggregate on] }

\[
\text{min } x : \{ \text{Actor}(x, y, _), y = \text{‘John’} \}
\]

Q(minId) :- minId = min x : \{ \text{Actor}(x, y, _), y = \text{‘John’} \}

Assign variable to the value of the aggregate

Meaning (in SQL)

```
SELECT min(id) as minId
FROM Actor as a
WHERE a.name = ‘John’
```
Aggregates

[aggregate name] <var> : { [relation to compute aggregate on] }

\[
\text{min } x : \{ \text{Actor}(x, y, _), y = 'John' \}
\]

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

What does this even mean???

Can’t use variable that are not aggregated in the outer /head atoms
COUNTING

Q(c) :- c = \text{count} : \{ \text{Actor}(_, y, _), y = 'John' \}

No variable here!

Meaning (in SQL, assuming no NULLs)

```
SELECT count(*) as c
FROM Actor as a
WHERE a.name = 'John'
```
Grouping

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

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```
Example

For each person, compute the total number of descendants

// for each person, compute his/her descendants
Example

For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
For each person, compute the total number of descendants

// 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

// 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

```prolog
// 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
T(p,c) :- D(p,_), c = count : { D(p,y) }.
```
Example

For each person, compute the total number of descendants

// 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
T(p,c) :- D(p,_), c = count : { D(p,y) }.

// Find the number of descendants of Alice
Example

For each person, compute the total number of descendants

// 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
T(p,c) :- D(p,_), c = count : { D(p,y) }.

// Find the number of descendants of Alice
Q(d) :- T(p,d), p = “Alice”.
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?

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

U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
Here are **unsafe** datalog rules. What’s “unsafe” about them?

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

U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)
Here are **unsafe** datalog rules. What’s “unsafe” about them?

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

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

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

Holds for every y other than “Bob”, U1 = infinite!
Here are *unsafe* datalog rules. What’s “unsafe” about them?

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

U2(x) :- 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?

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

• A datalog program is **stratified** if it can be partitioned into **strata**
  – Only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.

• Many Datalog DBMSs (including souffle) accepts only stratified Datalog.
## Stratified Datalog

<table>
<thead>
<tr>
<th>Stratum 1</th>
<th>Stratum 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ D(x,y) :\quad \text{ParentChild}(x,y). ]</td>
<td>[ T(p,c) :\quad \text{D}(p,_), \quad c = \text{count} : { \text{D}(p,y) }. ]</td>
</tr>
<tr>
<td>[ D(x,z) :\quad \text{D}(x,y), \quad \text{ParentChild}(y,z). ]</td>
<td>[ Q(d) :\quad T(p,d), \quad p = \text{“Alice”}. ]</td>
</tr>
</tbody>
</table>

May use D in an agg since it was defined in previous stratum.
Stratified Datalog

Stratum 1

\[ \text{D}(x,y) : - \text{ParentChild}(x,y). \]
\[ \text{D}(x,z) : - \text{D}(x,y), \text{ParentChild}(y,z). \]

Stratum 2

\[ \text{T}(p,c) : - \text{D}(p,_), c = \text{count} : \{ \text{D}(p,y) \}. \]
\[ \text{Q}(d) : - \text{T}(p,d), p = \text{“Alice”}. \]

Non-stratified

\[ \text{A}() : - \text{!B}(). \]
\[ \text{B}() : - \text{!A}(). \]

May use \text{D} in an agg since it was defined in previous stratum

May use \text{!D}

Cannot use \text{!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