Announcements

- HW 3 due Friday
  - Upload data with DataGrip editor – see message board
  - Azure timeout for question 5:
    - Try DataGrip or SQLite
    - Remember 2 late-day policy
- Gradience web quizzes were offline: WQ3 due date extended one day

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)
  - Eve http://witheve.com/
  - Differential datalog https://github.com/frankmcsherry/differential-dataflow
  - Beyond databases in many research projects: network protocols, static program analysis

Why bother with yet another relational query language?

Example: storing FB friends

```
Peter
  Mary
  John
  Phil

Or
Person1  Person2  is_friend
Peter    John     1
John     Mary     0
Mary     Phil     1
Phil     Peter    1
```

As a graph

As a relation

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)

Datalog allows us to write recursive queries easily.

My own friends

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

My FoF

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

Datalog: Facts and Rules

<table>
<thead>
<tr>
<th>Facts = tuples in the database</th>
<th>Rules = queries</th>
</tr>
</thead>
</table>

- dec Actor(id: number, fname: symbol, lname: symbol)
- dec Casts(pid: number, mid: number)
- dec Movie(id: number, name: symbol, year: number)

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).

Find Movies made in 1940

SQL

SELECT name
FROM Movie
WHERE year = 1940
Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Find Movies made in 1940

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Find Actors who acted in Movies made in 1940

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Find Actors who acted in a Movie in 1940 and in one in 1910

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3
Datalog: Terminology

- **Head**
  - Variables: \( f, l \)
  - \( f, l \) are head variables

- **Body**
  - Variables: \( x, y, z \)
  - \( x, y, z \) are existential variables

- **Atom** (aka subgoal)

- **Example**
  - \( Q(f, l) : - \text{Actor}(z,f,l), \text{Casts}(z,x), \text{Movie}(x,y,1948) \)

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: \( z > 1940 \).
- Book uses \& instead of \&

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

\[
Q(\text{args}_i) : - R_1(\text{args}_i), R_2(\text{args}_i), ...
\]

- \( R_1(\text{args}_i) \) called an atom, or a relational predicate
- \( R_2(\text{args}_i) \) evaluates to true when relation \( R_2 \) 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: \( z > 1940 \).
- Book uses AND instead of \&
Datalog Semantics

- Fixpoint semantics
  - Start:
    - $IDB_0 = \text{empty relations}$
    - $t = 0$
  - Repeat:
    - $IDB_{t+1} = \text{Compute Rules}(EDB, IDB_t)$
    - $t = t + 1$
  - Until $IDB_t = IDB_{t+1}$

More Features

- Aggregates
- Grouping
- Negation
Aggregates

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

- min x : { Actor(x, y, _), y = 'John' }
- minId := minId = min x : { Actor(x, y, _), y = 'John' }

Aggregates in Souffle:
- count
- min
- max
- sum

Assign variable to the value of the aggregate

Meaning (in SQL)

```sql
SELECT min(id) as minId FROM Actor as a WHERE a.name = 'John'
```

Counting

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

Meaning (in SQL, assuming no NULLs)

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

Grouping

Q(y,c) := Movie(_, y), c = count : { Movie(_, y) }

Meaning (in SQL)

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

Examples

A genealogy database (parent/child)

ParentChild

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Alice</td>
<td>Eve</td>
</tr>
<tr>
<td>Alice</td>
<td>Fred</td>
</tr>
<tr>
<td>Alice</td>
<td>George</td>
</tr>
</tbody>
</table>

Paths:

Count Descendants

For each person, count his/her descendants

ParentChild

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Alice</td>
<td>Eve</td>
</tr>
<tr>
<td>Alice</td>
<td>Fred</td>
</tr>
<tr>
<td>Alice</td>
<td>George</td>
</tr>
</tbody>
</table>

Count Descendants

For each person, count his/her descendants

ParentChild

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Alice</td>
<td>Eve</td>
</tr>
<tr>
<td>Alice</td>
<td>Fred</td>
</tr>
<tr>
<td>Alice</td>
<td>George</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

Paths:

Alice → Bob → Carol
Alice → Eve
Alice → Fred
Alice → George

Descendants:

x  count
Alice  4
Bob    5
Carol  3
Eve    4
Fred   1
George 0

Count Descendants

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 his/her descendants
D(x, z) :- D(x, y), ParentChild(y, z).

Note: Eve and George do not appear in the answer (why?)
Count Descendants

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,c), p = "Alice".

Negation: use "!"

Find all descendants of Bob that are not descendants of Alice
Negation: use “!”
Find all descendants of Bob that are not descendants of Alice

Answer

Same Generation
Two people are in the same generation if they are descendants at the same generation of some common ancestor

Same Generation
Compute pairs of people at the same generation

// common parent
SG(x,y) := ParentChild(p,x), ParentChild(p,y)
Same Generation

Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q), x < y

Problem: this includes answers like SG(Carol, Carol)
And also SG(Eve, George), SG(George, Eve)

How to fix?
Here are unsafe datalog rules. What’s “unsafe” about them?

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

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

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

A datalog rule is safe if every variable appears in some positive, non-aggregated relational atom.

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

Safe Datalog Rules

Stratified Datalog

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

\[ A() : \nequiv !B(). \]

\[ B() : \nequiv !A(). \]

- 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) accept only stratified Datalog.
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