Introduction to Database Systems
CSE 414

Lecture 10: More Datalog
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
• Gradiance 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/
  – 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

Peter
  /   
Mary  John
     /   
    Phil

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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Friends(p1, p2, isFriend)

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

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

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

Table declaration
Types in Souffle:
number
symbol (aka varchar)
Insert data
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(x,y,z), z=1940.
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

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

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.

**SQL**

- SELECT name
- FROM Movie
- WHERE year = 1940

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(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

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

Rules = queries

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

_ = “don’t care” variables

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).

**Rules** = queries

- Q1(y) :- Movie(x, y, z), z=1940.
- Q2(f, l) :- Actor(z, f, l), Casts(z, x), Movie(x, y, 1940).

Find Actors who acted in 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(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

- \( R_i \)(\( \text{args}_i \)) 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: \( \text{Actor}(344759, \text{‘Douglas’, ‘Fowley’}) \) is true
- In addition we can also have arithmetic predicates
  - Example: \( z > 1940 \).
- Book uses AND instead of ,

\[
Q(\text{args}) : - R1(\text{args}), R2(\text{args}), \ldots
\]

\[
Q(\text{args}) : - R1(\text{args}) \text{ AND } R2(\text{args}) \ldots
\]
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.
R encodes a graph
  e.g., connected cities

\[
R = \begin{pmatrix}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5
\end{pmatrix}
\]
Example

R encodes a graph e.g., connected cities

R =

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

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

Multiple rules for the same IDB means OR

What does it compute?
R encodes a graph e.g., connected cities

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

Initially:
T is empty.

What does it compute?

Example

Initially:
T is empty.

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]
R encodes a graph e.g., connected cities

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

Initially: T is empty.

---

Example

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

First iteration:

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

First rule generates this

Second rule generates nothing (because T is empty)

What does it compute?

R encodes a graph e.g., connected cities
Example

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

First iteration:
T =

Second iteration:
T =

What does it compute?

R encodes a graph
e.g., connected cities

New facts

First rule generates this

Second rule generates this

Initially:
T is empty.

R =

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

CSE 414 - Autumn 2018
R encodes a graph e.g., connected cities

Example

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

First iteration:
T =

Second iteration:
T =

Third iteration:
T =

CSE 414 - Autumn 2018

What does it compute?

New fact
Example

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 =

Third iteration: T =

Fourth iteration T = (same)

No new facts.

DONE

What does it compute?

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

CSE 414 - Autumn 2018
Datalog Semantics

Fixpoint semantics
• Start:
  \( \text{IDB}_0 = \text{empty relations} \)
  \( t = 0 \)
Repeat:
  \( \text{IDB}_{t+1} = \text{Compute Rules}(\text{EDB}, \text{IDB}_t) \)
  \( t = t+1 \)
Until \( \text{IDB}_t = \text{IDB}_{t-1} \)
More Features

• Aggregates

• Grouping

• Negation
Aggregates

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

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

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

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

Meaning (in SQL)

```
SELECT min(id) as minId
FROM Actor as a
WHERE a.name = ‘John’
```
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) :- 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(p,c)

<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>David</td>
</tr>
<tr>
<td>Carol</td>
<td>Eve</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

ParentChild(p,c)
Count Descendants

For each person, count his/her descendants

Paths:

<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:

<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>

Descendants:

<table>
<thead>
<tr>
<th>x</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</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>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</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>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>

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
For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
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).
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
```
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) }.
Count Descendants

How many descendants does Alice have?

// 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) }.
Count Descendants

How many descendants does Alice have?

// 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
Count Descendants

How many descendants does Alice have?

// 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”.

ParentChild(p,c)
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

Alice

Bob

Carol

David

Eve

Fred

George

ParentChild(p,c)
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

ParentChild(p,c)

Answer

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>David</td>
</tr>
</tbody>
</table>
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

// 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(“Bob”,x), !D(“Alice”,x).
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
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
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(p,q)
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
   :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
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)
Same Generation

Compute pairs of people at the same generation

```prolog
// 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)
```

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

How to fix?
Same Generation

Compute pairs of people at the same generation

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

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y),
           SG(p,q), x < y
Here are _unsafe_ datalog rules. What’s “unsafe” about them?

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

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

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

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

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

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

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

Unclear what y is
Here are *unsafe* datalog rules. What’s “unsafe” about them?

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

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

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

\[
U3(\text{minId,}y) :- \text{minId = min } x : \{ \text{Actor(x, y, _)} \}
\]
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) accept only stratified Datalog.
Stratified Datalog

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

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

Stratum 1

D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
T(p,c) :- D(p,_), c = count : { D(p,y) }.
Q(d) :- T(p,d), p = “Alice”.

Stratum 2

D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
Q(x) :- D(“Alice”,x), !D(“Bob”,x).

Non-stratified

A() :- !B().
B() :- !A.

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

May use !D

Cannot use !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.