CSE 544
Principles of Database Management Systems

Lectures 5: Datalog (1)

Announcement

- Deadline for HW1 has passed…
- Project M2 due on Friday
- HW2 released (datalog / Souffle)

Where We Are

Relational query languages:

- SQL
- Relational Algebra
- Relational Calculus (haven't discussed, but you may look it up)

The can express the same class of queries called ***relational queries***

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number

No higher math in database

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob

Yes! (write it in SQL!)

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob
- Partition all people into three sets P1(X),P2(X),P3(X) s.t. any two friends are in different partitions

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob
- Partition all people into three sets P1(X),P2(X),P3(X) s.t. any two friends are in different partitions

No! NP-complete

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a friend of Bob
- Partition all people into three sets P1(X),P2(X),P3(X) s.t. any two friends are in different partitions
- Find all people who are direct or indirect friends with Alice

Which are Relational Queries? Which are not? And Why?

Friend(X,Y)

- Find all people X whose number of friends is a prime number
- Find all people who are friends with everyone who is not a frien[illegible]

- Partition all people into three sets P1(X),P2(X),P3(X) s.t. any two friends are in different partitions
- Find all people who are direct or indirect friends with Alice

"Recursive query"; PTIME, yet not expressible in RA

Recursive Queries

- *"Find all direct or indirect friends of Alice"*
- Computable in PTIME, yet not expressible in RA
- Datalog: extends RA with recursive queries

Datalog

- Designed in the 80's
- Simple, concise, elegant
- Today is a hot topic, beyond databases: network protocols, static program analysis, DB+ML
- Very few open source implementations, and hard to find
- In HW2 we will use Souffle

```sql
USE AdventureWorks2008R2;
GO
WITH DirectReports (ManagerID, EmployeeID, Title, DeptID, Level)
AS
(
-- Anchor member definition
    SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,
        0 AS Level
    FROM dbo.MyEmployees AS e
    INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL
    WHERE ManagerID IS NULL
    UNION ALL
-- Recursive member definition
    SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,
        Level + 1
    FROM dbo.MyEmployees AS e
    INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
-- Statement that executes the CTE
SELECT ManagerID, EmployeeID, Title, DeptID, Level
FROM DirectReports
INNER JOIN HumanResources.Department AS dp
    ON DirectReports.DeptID = dp.DepartmentID
WHERE dp.GroupName = N'Sales and Marketing' OR Level = 0;
GO
```

```
Manager(eid) :- Manages(_, eid)

DirectReports(eid, 0) :-
        Employee(eid),
        not Manager(eid)

DirectReports(eid, level+1) :-
        DirectReports(mid, level),
        Manages(mid, eid)
```

SQL Query vs Datalog

(which would you rather write?)

(any Java fans out there?)

Outline

- Datalog rules
- Recursion
- Negation, aggregates, stratification
- Semantics
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

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

← Schema

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
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).
```

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

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'.

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

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'.
```

Find Movies made in 1940

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

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').
```

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

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').
```

Find Actors who acted in Movies made in 1940

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
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).
```

```
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)
```

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
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).
```

```
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

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

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

head — body

atom — atom — atom (aka subgoal)

```
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

```
Q(args) :- R1(args), R2(args), ....
```

- $R_i(args_i)$ 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’.
- Some systems use <-
- Some use AND

```
Q(args) <-  R1(args), R2(args), ....
```

```
Q(args) :- R1(args) AND R2(args) ....
```

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)
- $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)
- $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]
- Logically equivalent:
  $\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') ⇒ Q1(y)]

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)
- $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') $\Rightarrow$ Q1(y)]
- Logically equivalent:
  $\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') $\Rightarrow$ Q1(y)]
- Thus, non-head variables are called "existential variables"

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)
- $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]
- Logically equivalent:
  $\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') ⇒ Q1(y)]
- Thus, non-head variables are called "existential variables"
- We want the *smallest* set Q1 with this property (why?)

Outline

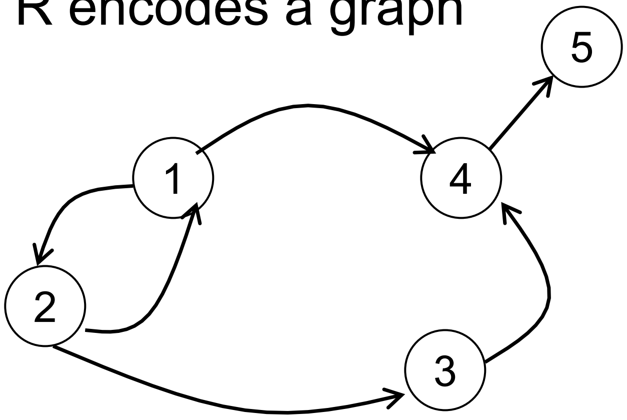
- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

Datalog program

- A datalog program consists of several rules
- Importantly, rules may be recursive!
- 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

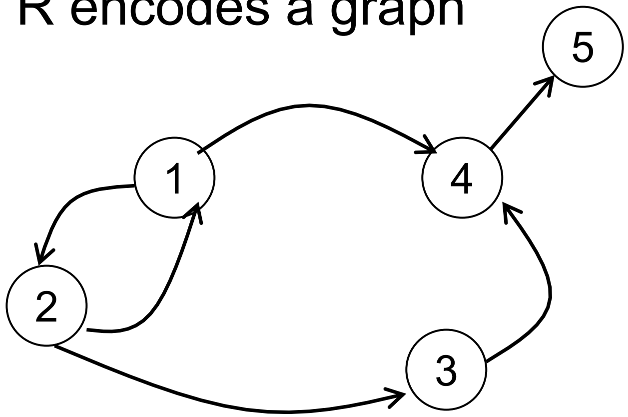


R=

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

Example

R encodes a graph



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

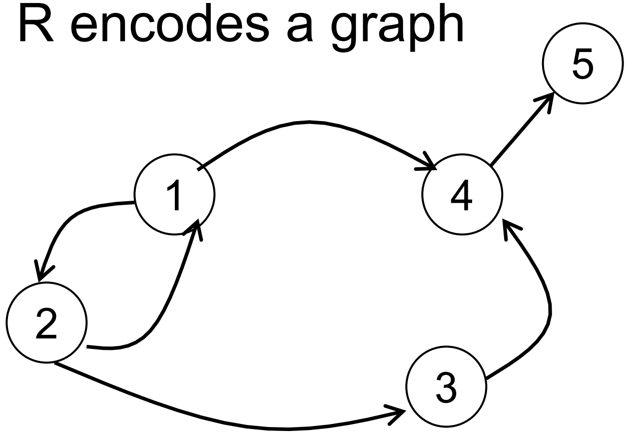
What does it compute?

R=

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

Example

R encodes a graph



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

What does it compute?

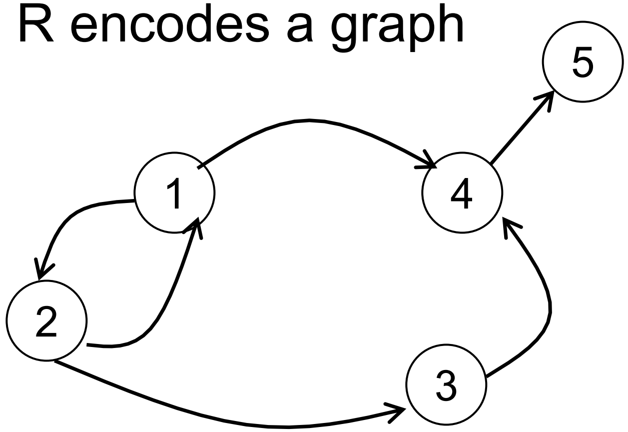
R=

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

Initially:
T is empty.

Example

R encodes a graph



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

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

First rule generates this

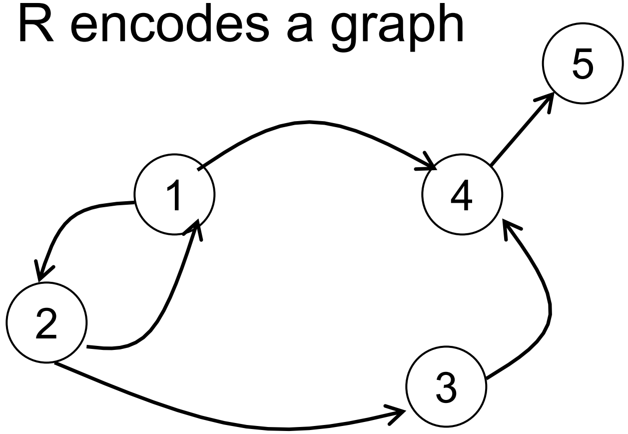
Second rule generates nothing (because T is empty)

Example

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

What does it compute?

R encodes a graph



R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

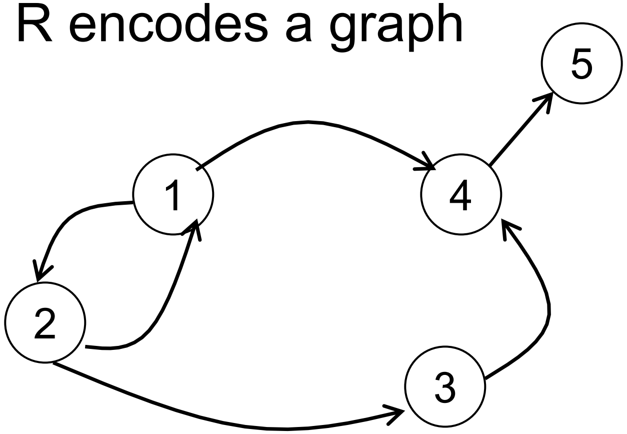
First rule generates this (rows 1–6)

Second rule generates this (rows 7–12)

New facts

Example

R encodes a graph




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

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

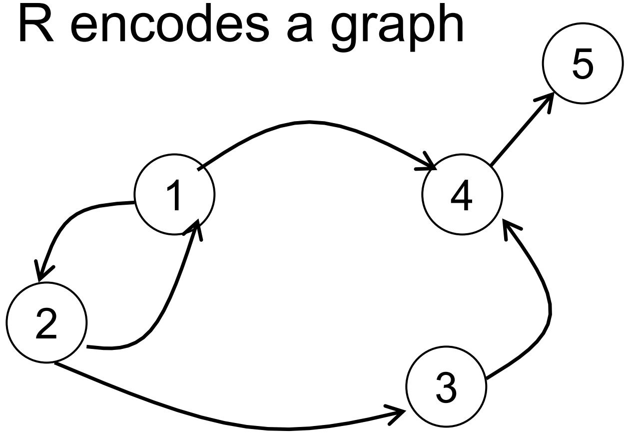
Third iteration:
T =

1	2	Both rules
2	1	Both rules
2	3	First rule
1	4	First rule
3	4	First rule
4	5	First rule
1	1	
2	2	Second rule
1	3	Second rule
2	4	Second rule
1	5	Second rule
3	5	Second rule
2	5	Second rule (New fact)

New fact

Example

R encodes a graph



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

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

What does it compute?

R=

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

Initially:

T is empty.

First iteration:

T =

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

Second iteration:

T =

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

Third iteration:

T =

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

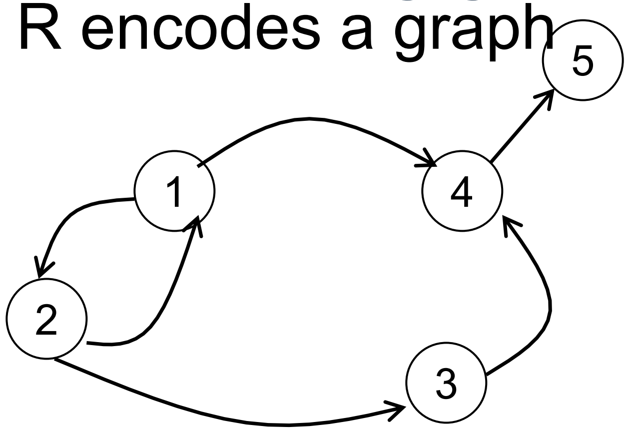
Fourth iteration

T = (same)

No new facts. DONE

Three Equivalent Programs

R encodes a graph



R=

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

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

Right linear

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

Left linear

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

Non-linear

Question: which terminates in fewest iterations?

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

1. Fixpoint Semantics

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

- Remark: since rules are monotone:
  $\emptyset = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \ldots$
- A datalog program w/o functions (+, *, ...) always terminates. (In what time?)

2. Minimal Model Semantics:

- Return the IDB that
  1) For every rule,
     $\forall$vars [(Body(EDB,IDB) ⇒ Head(IDB)]
  2) Is the smallest IDB satisfying (1)

- Theorem: there exists a smallest IDB satisfying (1)

Example

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

1. Fixpoint semantics:

- Start: $T_0 = \emptyset$; $t = 0$
  Repeat:
  $$T_{t+1}(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T_t(z,y))$$
  $$t = t+1$$
  Until $T_t = T_{t-1}$

2. Minimal model semantics: smallest T s.t.

- $\forall x \forall y\ [(R(x,y) \Rightarrow T(x,y)] \wedge$
  $\forall x \forall y \forall z\ [(R(x,z) \wedge T(z,y)) \Rightarrow T(x,y)]$

Datalog Semantics

- The fixpoint semantics tells us how to compute a datalog query
- The minimal model semantics is more declarative: only says what we get
- The two semantics are equivalent meaning: you get the same thing

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

Extensions

- Aggregates, negation
- Stratified datalog

Aggregates

- No commonly agreed syntax
- Each implementation uses it’s own

Aggregates in Souffle

General syntax in Logicblox:

```
Q(x,y,z,v)    :-     Body1(x,y,z), v = sum(w) : { Body2(x,y,z,w) }
```

Meaning (in SQL)

```
select x,y,z, sum(w) as v
from R1, R2, ...
where ...
group by x,y,z
```

Example

For each person, compute the total number of descendants

```
/* We use Souffle syntax (as in the homework) */
/* for each person, compute his/her descendants */
```

Example

For each person, compute the total number of descendants

```
/* We use Souffle syntax (as in the homework) */
/* 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

```
/* We use Souffle syntax (as in the homework) */
/* 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

```
/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
```

ParentChild(p,c)

Example

For each person, compute the total number of descendants

```
/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
/* Find the number of descendants of Alice */
```

Example

For each person, compute the total number of descendants

```
/* We use Souffle syntax (as in the homework) */
/* 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 */
N(x,m) :- D(x,_), m = sum(1) : { D(x,y) }.
/* Find the number of descendants of Alice */
Q(d)  :-  N("Alice",d).
```

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).
```

ParentChild(p,c)

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)
```

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for every
y other than "Bob"
U1 = infinite!

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

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for every y other than "Bob"
U1 = infinite!

```
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)

ParentChild(p,c)

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for every y other than "Bob" U1 = infinite!

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)

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?

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

Stratified Datalog

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

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

- A datalog program is *stratified* if it can be partitioned into strata s.t., for all n, only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.
- Souffle (and others) accepts only stratified datalog.

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
---------------------------------------------
N[x] = m  :-  agg<<m = count()>> D(x,y).
Q(d)  :-  N["Alice"]=d.
```

Stratum 1

Stratum 2

May use D
in an agg because was
defined in previous
stratum

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
----------------------------------------
N(x,m) :-  D(x,_), m = sum(1) : { D(x,y) }.
Q(d)  :-  N("Alice", d).
```

Stratum 1

Stratum 2

May use D in an agg because was defined in previous stratum

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
----------------------------------------
Q(x)  :-  D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use !D

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
-----------------------------------
N(x,m) :-  D(x,_), m = sum(1) : { D(x,y) }.
Q(d)  :-  N("Alice", d).
```

Stratum 1

Stratum 2

May use D in an agg because was defined in previous stratum

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
-----------------------------------
Q(x)  :-  D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use !D

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

Non-stratified

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

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

Datalog Evaluation Algorithms

- Needs to preserve the efficiency of query optimizers, while extending them to recursion
- Two general strategies:
  - Naïve datalog evaluation
  - Semi-naïve datalog evaluation
- Some powerful optimizations:
  - Magic sets (next lecture)

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
....

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

➔ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

➔ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve datalog evaluation algorithm:

```
P1 = P2 = … = Ø
Loop
    NewP1 = SPJU1; NewP2 = SPJU2; …
    if (NewP1 = P1 and NewP2 = P2 and …)
        then exit
    P1 = NewP1; P2 = NewP2; …
Endloop
```

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

➔ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
….

➔ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve datalog evaluation algorithm:

$P_1 = P_2 = \ldots = \emptyset$

Loop

  $NewP_1 = SPJU_1$; $NewP_2 = SPJU_2$; …

  if ($NewP_1 = P_1$ and $NewP_2 = P_2$ and …)

    then exit

  $P_1 = NewP_1$; $P_2 = NewP_2$; …

Endloop

Example:

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

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
….

→ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup$ …
P_2 :- $body_{21} \cup body_{22} \cup$ …
….

→ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
….

Naïve datalog evaluation algorithm:

$P_1 = P_2 = \ldots = \emptyset$

Loop

  $NewP_1 = SPJU_1$; $NewP_2 = SPJU_2$; …

  if ($NewP_1 = P_1$ and $NewP_2 = P_2$ and …)

    then exit

  $P_1 = NewP_1$; $P_2 = NewP_2$; …

Endloop

Example:

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

→ $T(x,y) \text{ :- } R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$

Naïve Datalog Evaluation Algorithm

Datalog program:

P_{i1} :- $body_1$
P_{i2} :- $body_2$
....

➔ Group by IDB predicate

P_1 :- $body_{11} \cup body_{12} \cup \ldots$
P_2 :- $body_{21} \cup body_{22} \cup \ldots$
....

➔ Each rule is a Select-Project-Join-Union query

P_1 :- $SPJU_1$
P_2 :- $SPJU_2$
....

Naïve datalog evaluation algorithm:

$P_1 = P_2 = \ldots = \emptyset$
Loop
    $NewP_1 = SPJU_1$; $NewP_2 = SPJU_2$; ...
    if ($NewP_1 = P_1$ and $NewP_2 = P_2$ and ...)
        then exit
    $P_1 = NewP_1$; $P_2 = NewP_2$; ...
Endloop

Example:

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

➔ $T(x,y) :- R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$

$T = \emptyset$
Loop
    $NewT(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T(z,y))$
    if (NewT = T)
        then exit
    T = NewT
Endloop

Discussion

- A naïve datalog algorithm *always* terminates (why?)
  - Assuming no functions (+, *, ...)

- A datalog program *always* runs in PTIME in the size of the database (why?)

Problem with the Naïve Algorithm

- The same facts are discovered over and over again
- The *semi-naïve* algorithm tries to reduce the number of facts discovered multiple times

Background: Incremental View Maintenace

Let V be a view computed by one datalog rule (no recursion)

V :- body

If (some of) the relations are updated:

$R_1 \leftarrow R_1 \cup \Delta R_1$, $R_1 \leftarrow R_2 \cup \Delta R_2$, …

Then the view is also modified as follows:

$V \leftarrow V \cup \Delta V$

Incremental view maintenance:
Compute ΔV without having to recompute V

Background: Incremental View Maintenace

Example 1:

$V(x,y)$:- $R(x,z), S(z,y)$

If $R \leftarrow R \cup \Delta R$ then what is $\Delta V(x,y)$?

Background: Incremental View Maintenace

Example 1:

V(x,y) :- R(x,z),S(z,y)

If $R \leftarrow R \cup \Delta R$ then what is $\Delta V(x,y)$?

ΔV(x,y) :- ΔR(x,z),S(z,y)

Background: Incremental View Maintenace

Example 2:

V(x,y) :- R(x,z),S(z,y)

If R ← R ∪ΔR and S ← S ∪ΔS then what is ΔV(x,y) ?

Background: Incremental View Maintenace

Example 2:

$V(x,y) \text{ :- } R(x,z),S(z,y)$

If $R \leftarrow R \cup \Delta R$ and $S \leftarrow S \cup \Delta S$ then what is $\Delta V(x,y)$?

$\Delta V(x,y) \text{ :- } \Delta R(x,z),S(z,y)$
$\Delta V(x,y) \text{ :- } R(x,z), \Delta S(z,y)$
$\Delta V(x,y) \text{ :- } \Delta R(x,z), \Delta S(z,y)$

Background: Incremental View Maintenace

Example 3:

$V(x,y)$:- $T(x,z), T(z,y)$

If $T \leftarrow T \cup \Delta T$
then what is $\Delta V(x,y)$?

Background: Incremental View Maintenace

Example 3:

$V(x,y)$:- $T(x,z), T(z,y)$

If $T \leftarrow T \cup \Delta T$
then what is $\Delta V(x,y)$?

$\Delta V(x,y)$:- $\Delta T(x,z), T(z,y)$
$\Delta V(x,y)$:- $T(x,z), \Delta T(z,y)$
$\Delta V(x,y)$:- $\Delta T(x,z), \Delta T(z,y)$

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

$P_1 = \Delta P_1$ = non-recursive-$SPJU_1$, $P_2 = \Delta P_2$ = non-recursive-$SPJU_2$, …

Loop

$\Delta P_1 = \Delta SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$; …

if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and …)

then break

$P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$; …

Endloop

Example:

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

$T = \Delta T$ = **? (non-recursive rule)**

Loop

$\Delta T(x,y)$ = **? (recursive Δ-rule)**

if ($\Delta T = \emptyset$)

then break

$T = T \cup \Delta T$

Endloop

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

$P_1 = \Delta P_1$ = non-recursive-$SPJU_1$, $P_2 = \Delta P_2$ = non-recursive-$SPJU_2$, …

Loop

$\Delta P_1 = \Delta SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$; …

if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and …)

then break

$P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$; …

Endloop

Example:

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

$T(x,y) = R(x,y)$, $\Delta T(x,y) = R(x,y)$

Loop

$\Delta T(x,y) = (R(x,z) \bowtie \Delta T(z,y)) - R(x,y)$

if ($\Delta T = \emptyset$)

then break

$T = T \cup \Delta T$

Endloop

Semi-naïve Evaluation Algorithm

Separate the Datalog program into the non-recursive, and the recursive part.

Each P_i defined by non-recursive-$SPJU_i$ and (recursive-)$SPJU_i$.

$P_1 = \Delta P_1 =$ non-recursive-$SPJU_1$, $P_2 = \Delta P_2 =$ non-recursive-$SPJU_2$, …

Loop

$\Delta P_1 = \Delta\, SPJU_1 - P_1$; $\Delta P_2 = \Delta SPJU_2 - P_2$; …

if ($\Delta P_1 = \emptyset$ and $\Delta P_2 = \emptyset$ and …)

then break

$P_1 = P_1 \cup \Delta P_1$; $P_2 = P_2 \cup \Delta P_2$; …

Endloop

Example:

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

$T(x,y) = R(x,y)$, $\Delta T(x,y) = R(x,y)$

Loop

$\Delta T(x,y) = (R(x,z) \bowtie \Delta T(z,y)) - R(x,y)$

if ($\Delta T = \emptyset$)

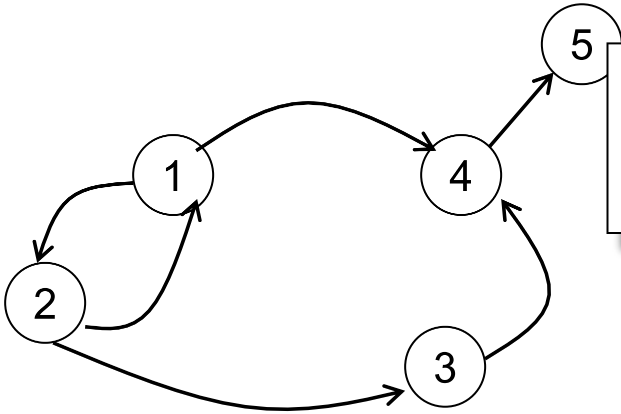
then break

$T = T \cup \Delta T$

Endloop

Note: for any linear datalog programs, the semi-naïve algorithm has only one Δ-rule for each rule!

Example

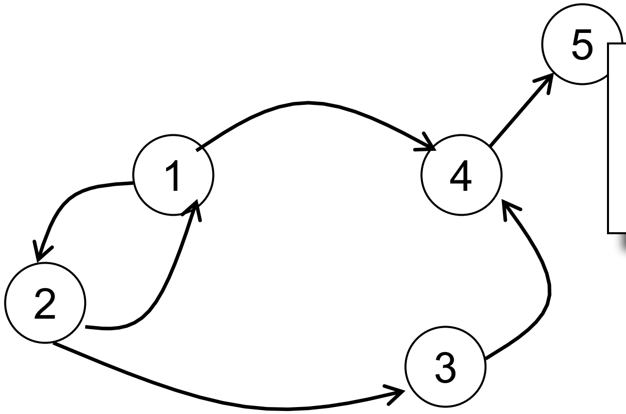


T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

R=

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

Example



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

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

T(x,y) = R(x,y), ΔT(x,y) = R(x,y)

Loop

  ΔT(x,y) =
   (R(x,z) ⋈ ΔT(z,y)) – R(x,y)
   if (ΔT = ∅) break
   T = T∪ΔT

Endloop

R=

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

Initially:

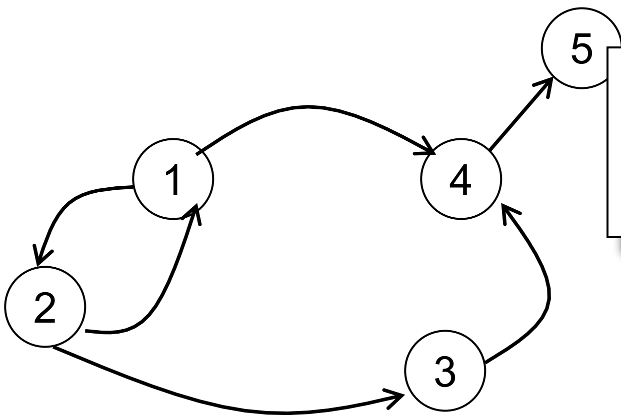
ΔT=

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

T=

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

Example



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

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

T(x,y) = R(x,y), ΔT(x,y) = R(x,y)

Loop

ΔT(x,y) =
(R(x,z) ⋈ ΔT(z,y)) – R(x,y)
if (ΔT = ∅) break
T = T∪ΔT

Endloop

R=

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

Initially:

ΔT=

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

T=

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

First iteration:

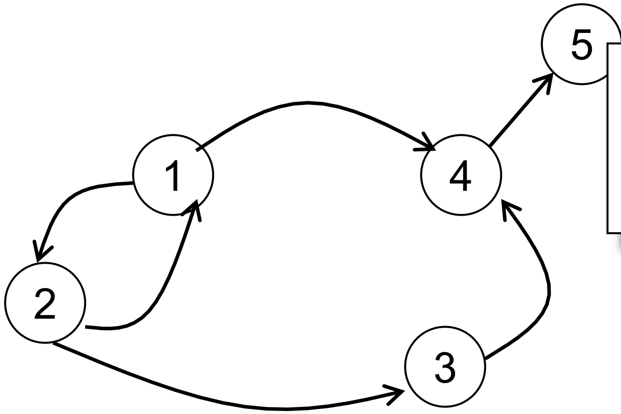
ΔT=
paths of
length 2

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

T=

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

Example



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

T(x,y) = R(x,y), ΔT(x,y) = R(x,y)
Loop
  ΔT(x,y) =
   (R(x,z) ⋈ ΔT(z,y)) – R(x,y)
   if (ΔT = ∅) break
   T = T∪ΔT
Endloop

R=

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

Initially:

ΔT=

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

T=

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

First iteration:

ΔT= paths of length 2

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

T=

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

Second iteration:

ΔT= paths of length 3

1	2
1	4
2	1
2	3
2	5

T=

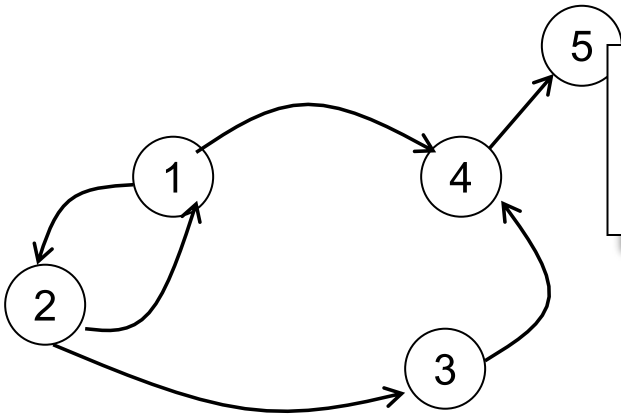
1	2
1	4
2	1
2	3
3	4
4	5
1	1
1	3
1	5
2	2
2	4
3	5
2	5

Example

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

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

T(x,y) = R(x,y), ΔT(x,y) = R(x,y)
Loop
  ΔT(x,y) =
    (R(x,z) ⋈ ΔT(z,y)) – R(x,y)
    if (ΔT = ∅) break
    T = T∪ΔT
Endloop



R=

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

Initially:

ΔT=

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

T=

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

First iteration:

ΔT= paths of length 2

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

T=

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

Second iteration:

ΔT= paths of length 3

1	2
1	4
2	1
2	3
2	5

T=

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

Third iteration:

ΔT= paths of length 4

Discussion of Semi-Naïve Algorithm

- Avoids re-computing some tuples, but not all tuples
- Easy to implement, no disadvantage over naïve

- A rule is called *linear* if its body contains only one recursive IDB predicate:
  - A linear rule always results in a single incremental rule
  - A non-linear rule may result in multiple incremental rules

Outline

- Datalog rules
- Recursion
- Semantics
- Negation, aggregates, stratification
- Naïve and Semi-naïve Evaluation
- Connection to RA – on your own

Datalog v.s. RA (and SQL)

- "Pure" datalog has recursion, but no negation, aggregates: all queries are monotone; impractical
- Datalog ***<u>without recursion</u>***, plus negation and aggregates expresses the same queries as RA: next slides

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Union:
R(A,B,C) ∪ S(D,E,F)

U(x,y,z) :- R(x,y,z)

U(x,y,z) :- S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Intersection:

R(A,B,C) ∩ S(D,E,F)

I(x,y,z) :- R(x,y,z), S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Selection: $\sigma_{x>100 \text{ and } y=\text{'foo'}}$ (R)

L(x,y,z) :- R(x,y,z), x > 100, y='foo'

Selection: $\sigma_{x>100 \textbf{ or } y=\text{'foo'}}$ (R)

L(x,y,z) :- R(x,y,z), x > 100

L(x,y,z) :- R(x,y,z), y='foo'

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Equi-join: $R \bowtie_{R.A=S.D \text{ and } R.B=S.E} S$

J(x,y,z,q) :- R(x,y,z), S(x,y,q)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Projection: $\Pi_A(R)$

P(x) :- R(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

To express difference, we add negation

R – S

D(x,y,z) :- R(x,y,z), NOT S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

Examples

Translate: $\Pi_A(\sigma_{B=3}(R))$

A(a) :- R(a,3,_)

Underscore used to denote an "anonymous variable"
Each such variable is unique

R(A,B,C)
S(D,E,F)
T(G,H)

Examples

Translate: $\Pi_A(\sigma_{B=3}(R) \bowtie_{R.A=S.D} \sigma_{E=5}(S))$

A(a) :- R(a,3,_), S(a,5,_)

These are different "_"s

Friend(name1, name2)
Enemy(name1, name2)

More Examples w/o Recursion

Find Joe's friends, and Joe's friends of friends.

```
A(x) :- Friend('Joe', x)
A(x) :- Friend('Joe', z), Friend(z, x)
```

Friend(name1, name2)

Enemy(name1, name2)

More Examples w/o Recursion

Find all of Joe's friends who do not have any friends except for Joe:

```
JoeFriends(x) :- Friend('Joe',x)
NonAns(x) :- JoeFriends(x), Friend(x,y), y != 'Joe'
A(x) :- JoeFriends(x), NOT NonAns(x)
```

Friend(name1, name2)
Enemy(name1, name2)

More Examples w/o Recursion

Find all people such that all their enemies' enemies are their friends

- Q: if someone doesn't have any enemies nor friends, do we want them in the answer?
- A: Yes!

```
Everyone(x) :- Friend(x,y)
Everyone(x) :- Friend(y,x)
Everyone(x) :- Enemy(x,y)
Everyone(x) :- Enemy(y,x)
NonAns(x) :- Enemy(x,y),Enemy(y,z), NOT Friend(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

Friend(name1, name2)

Enemy(name1, name2)

More Examples w/o Recursion

Find all persons x that have a friend all of whose enemies are x's enemies.

```
Everyone(x) :- Friend(x,y)
NonAns(x) :- Friend(x,y) Enemy(y,z), NOT Enemy(x,z)
A(x) :- Everyone(x), NOT NonAns(x)
```

ParentChild(p,c)

More Examples w/ Recursion

- Two people are in the *same generation* if they are siblings, or if they have parents in the same generation

- Find all persons in the same generation with Alice

More Examples w/ Recursion

- Find all persons in the same generation with Alice
- Let's compute SG(x,y) = "x,y are in the same generation"

```
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
Answer(x) :- SG("Alice", x)
```

Datalog Summary

- EDB (base relations) and IDB (derived relations)
- Datalog program = set of rules
- Datalog is recursive

- Some reminders about semantics:
  – Multiple atoms in a rule mean join (or intersection)
  – Variables with the same name are join variables
  – Multiple rules with same head mean union

Datalog and SQL

- Stratified data (w/ recursion, w/o +,*,...): expresses precisely* queries in PTIME
  - Cannot find a Hamiltonian cycle (why?)
- SQL has also been extended to express recursive queries:
  - Use a recursive “with” clause, also CTE (Common Table Expression)
  - Often with bizarre restrictions...
  - ... Just use datalog

* need to use the < predicate