CSE544
Data Management

Lectures 11

Datalog (Part 1 of 2)

Announcement

- HW3 deadline extended to Tue, May 11
- Review was due today
- Next review: Wed, May 12

Motivation

- SQL designed *relational queries;* Not good at iteration/recursion
- Data processing today require iteration. Common solution: external driver
- Datalog is a language that allows both recursion and relational queries

Datalog

- Designed in the 80's: simple, concise, elegant, very popular in research

- All techniques for recursive relational queries were developed for datalog

- But: no standard, no reference implementation; in HW4 we use Souffle

Outline

- Datalog rules
- Recursion
- Semantics

Next time: extensions, semi-naïve algo.

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

Schema

Datalog: Facts and Rules

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

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

```
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)
```

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

Anatomy of a Rule

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

Anatomy of a Rule

head body

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

Anatomy of a Rule

head body

atom atom atom (aka predicate)

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

Anatomy of a Rule

head | body

atom | atom | atom (aka predicate)

```
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***

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

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’.

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 <-

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

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) ....
```

Outline

- Datalog rules
- Recursion
- Semantics

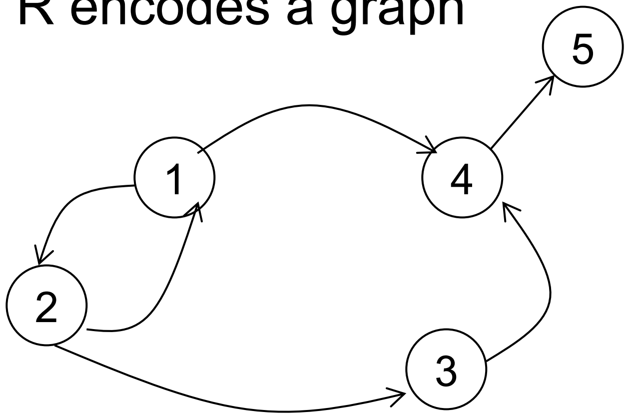
Next time: extensions, semi-naïve algo.

Datalog program

- A datalog program = several rules
- Rules may be recursive
- Set semantics only

Example

R encodes a graph

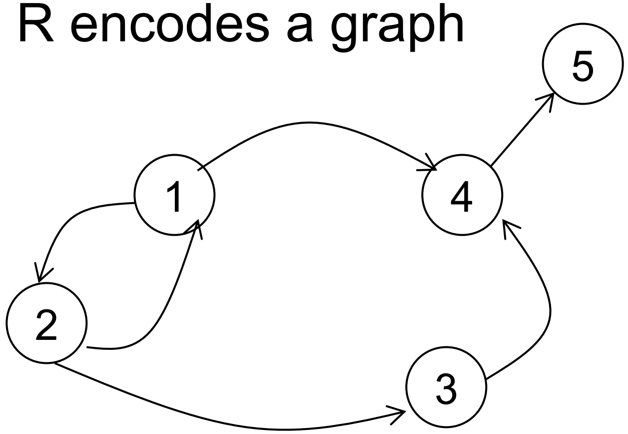


R=

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

Example

R encodes a graph



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

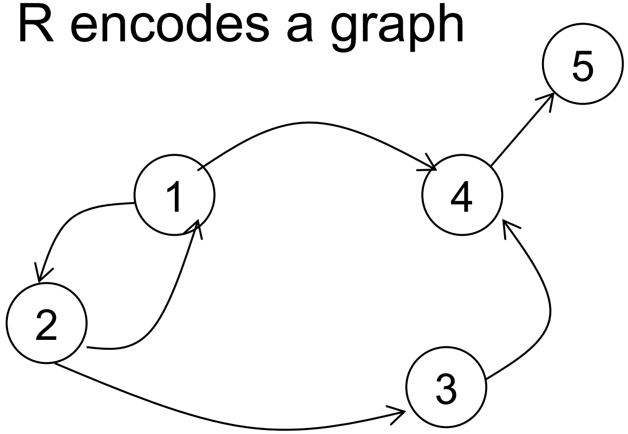
What does it compute?

R=

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

Example

R encodes a graph



R=

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

Initially:

T is empty.

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

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

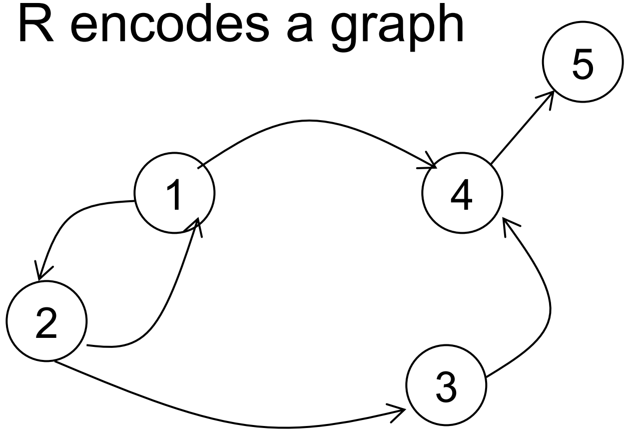
What does it compute?

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 =

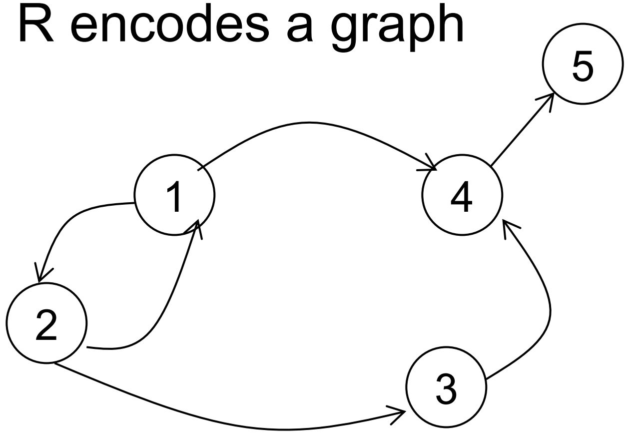
1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule
generates nothing
(because 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

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

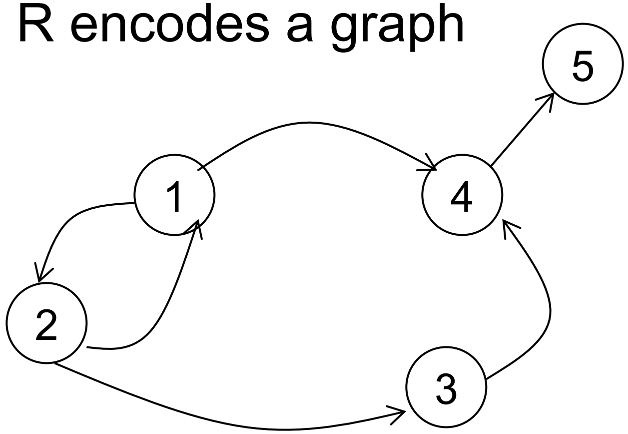
First rule generates this (rows 1–6)

Second rule generates this (rows 7–12)

New facts

Example

R encodes a graph



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

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

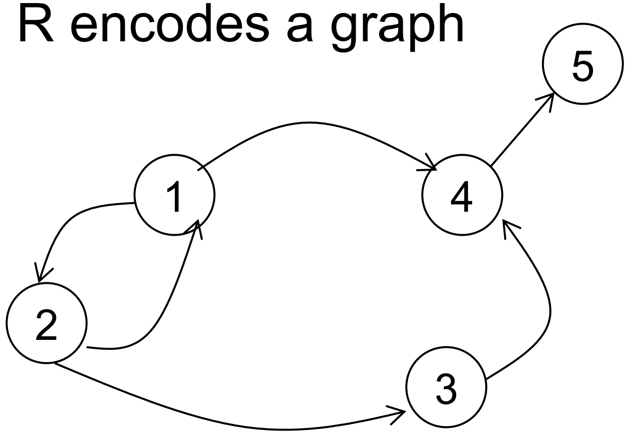
Example

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

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

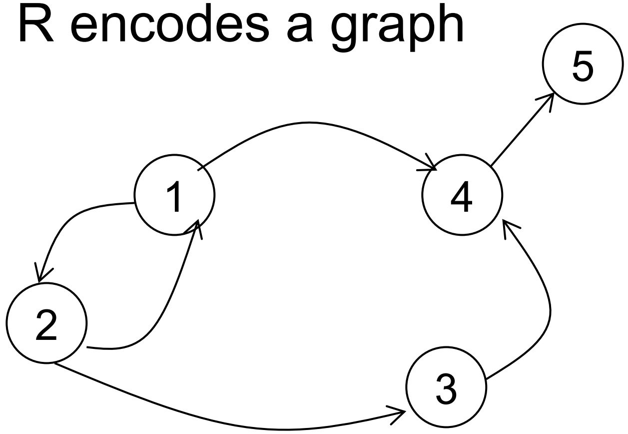
T =

(same)

No new facts. DONE

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

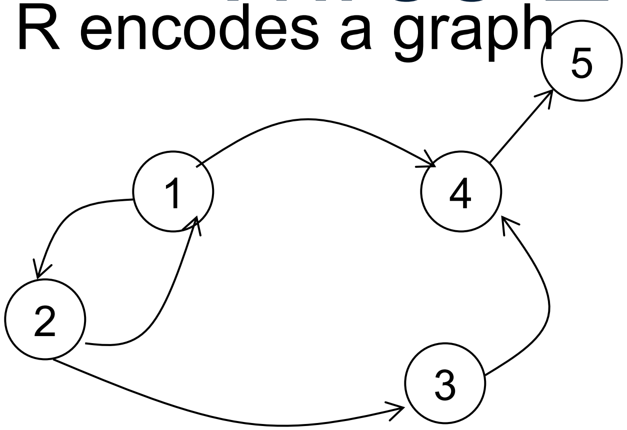
Fourth iteration
T =
(same)

No new facts.
DONE

Iteration k computes pairs (x,y) connected by path of length ≤ k

Three Equivalent Programs

R encodes a graph



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

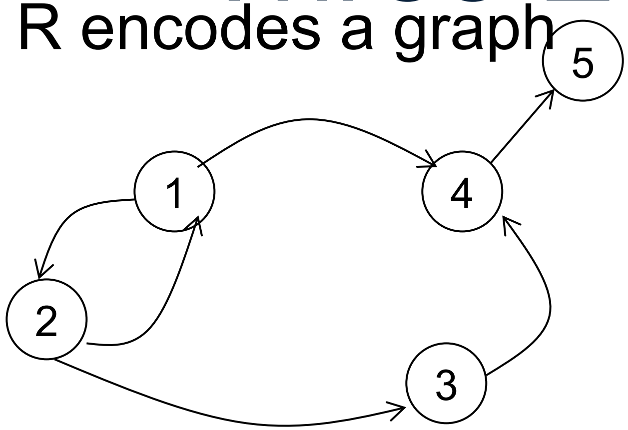
Right linear

R=

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

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)
```

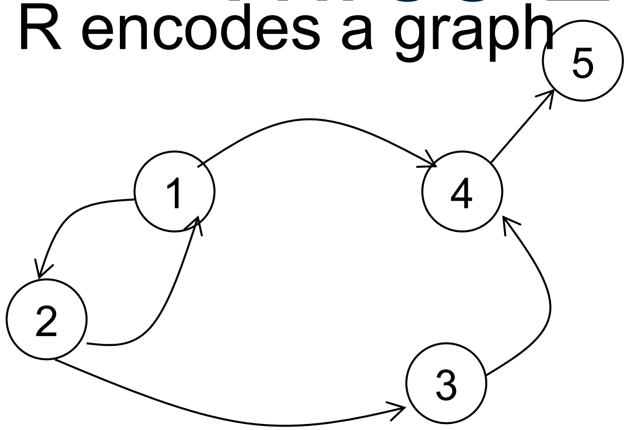
Right linear

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

Left linear

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)
```

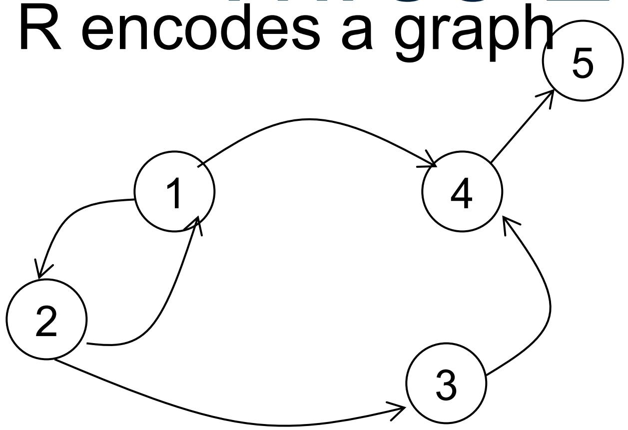
Left linear

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

Non-linear

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

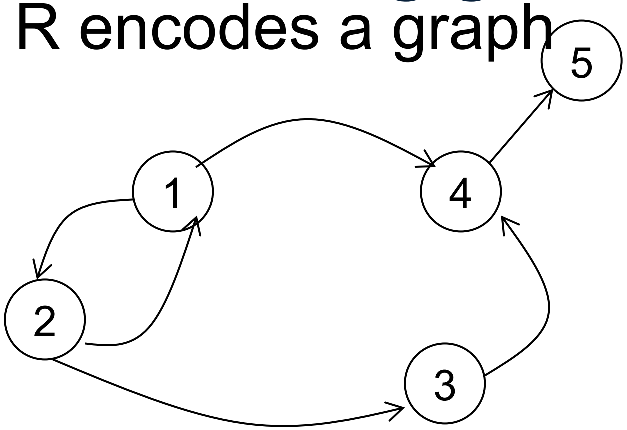
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

Non-linear

Question: how many iterations does each require?

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

#iterations = diameter

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

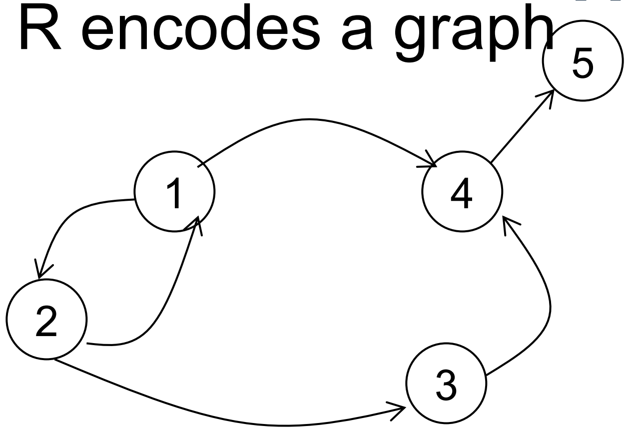
Non-linear

#iterations = log(diameter)

Question: how many iterations does each require?

Multiple IDBs

R encodes a graph



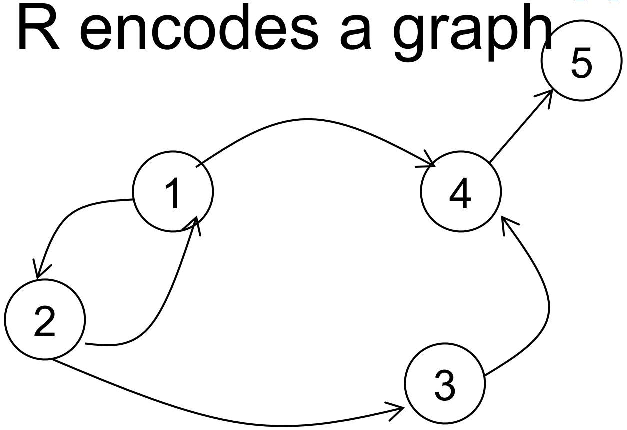
Find pairs of nodes (x,y)
connected by a path of *even* length

R=

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

Multiple IDBs

R encodes a graph



Find pairs of nodes (x,y) connected by a path of *even* length

R=

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

Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z), R(z,y)
Odd(x,y) :- Even(x,z), R(z,y)

Two IDBs: Odd(x,y) and Even(x,y)

Discussion: Recursion in SQL

SQL has everything, including some form of recursion, BUT:

- Single IDB
- Linear query only
- Has bag semantics (why???) which diverges

```
with recursive T as(
    select * from R
    union
    select distinct R.x, T.y
    from R, T
    where R.y=T.x
) select * from T;
```

Outline

- Datalog rules
- Recursion
- Semantics

Next time: extensions, semi-naïve algo.

Naïve Evaluation Algorithm

- Every rule → SPJ* query

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

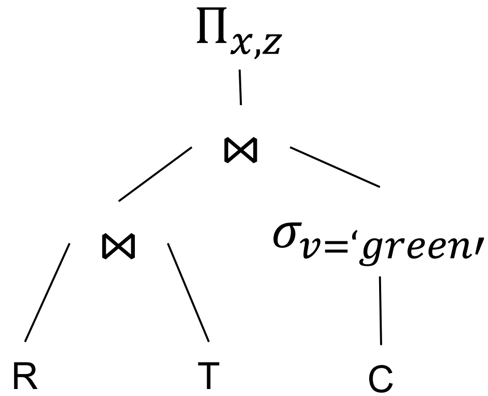
```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

T(x,z) :- R(x,y), T(y,z), C(y,'green')

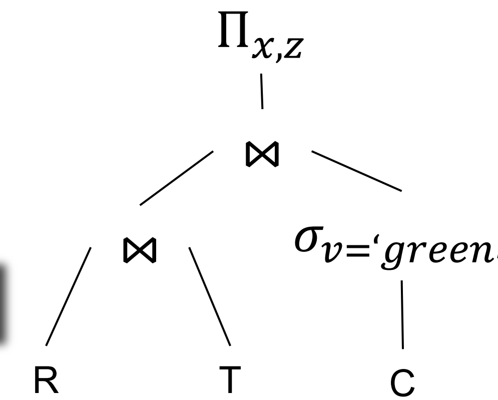


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

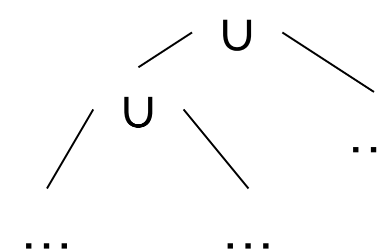
- Every rule → SPJ* query

```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```



- Multiple rules same head → USPJ+

```
T(x,y) :-  …
T(x,y) :-  …
…
```

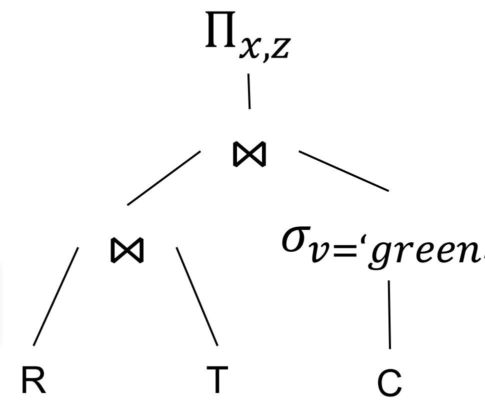


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

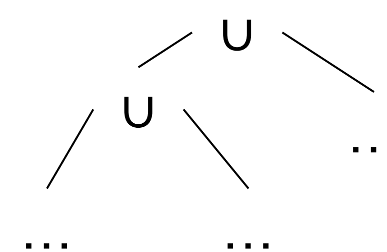
- Every rule → SPJ* query

```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```



- Multiple rules same head → USPJ⁺

```
T(x,y) :-  …
T(x,y) :-  …
      …
```



- Naïve Algorithm:

$IDBs := \emptyset$
repeat $IDBs := USPJs$
until no more change

*SPJ = select-project-join
⁺USPJ = union-select-project-join

Naïve Evaluation Algorithm

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

Naïve Evaluation Algorithm

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

$T := \emptyset$;
repeat
$T := R \cup \Pi_{x,y}(R \bowtie T)$;
until [no more change]

Naïve Evaluation Algorithm

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

Optimization:
Use R only once,
before the loop
(SQL does this)

$T := \emptyset;$

repeat

$T := R \cup \Pi_{x,y}(R \bowtie T);$

until [no more change]

Naïve Evaluation Algorithm

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

Optimization:
Use R only once,
before the loop
(SQL does this)

Will discuss a more
general optimization
called Semi-Naïve
next time

$T := \emptyset$;

repeat

$T := R \cup \Pi_{x,y}(R \bowtie T)$;

until [no more change]

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\text{Odd} \coloneqq \emptyset; \text{Even} \coloneqq \emptyset;$

repeat

$\quad \text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R);$

$\quad \text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R);$

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\mathrm{Odd} \coloneqq \emptyset;\ \mathrm{Even} \coloneqq \emptyset;$

repeat

$\mathrm{Even}_{\mathrm{new}} := \Pi_{x,y}(\mathrm{Odd} \bowtie R);$

$\mathrm{Odd}_{\mathrm{new}} := R \cup \Pi_{x,y}(\mathrm{Even} \bowtie R);$

$\mathrm{Odd} := \mathrm{Odd}_{\mathrm{new}}$

$\mathrm{Even} := \mathrm{Even}_{\mathrm{new}}$

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *in parallel*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\mathrm{Odd} \coloneqq \emptyset;\ \mathrm{Even} \coloneqq \emptyset;$

repeat

$\mathrm{Even}_{\mathrm{new}} := \Pi_{x,y}(\mathrm{Odd} \bowtie R);$

$\mathrm{Odd}_{\mathrm{new}} := R \cup \Pi_{x,y}(\mathrm{Even} \bowtie R);$

if $\mathrm{Odd}{=}\mathrm{Odd}_{\mathrm{new}} \wedge \mathrm{Even}{=}\mathrm{Even}_{\mathrm{new}}$

then break

$\mathrm{Odd}{:=}\mathrm{Odd}_{\mathrm{new}}$

$\mathrm{Even}{:=}\mathrm{Even}_{\mathrm{new}}$

Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Before we show this, a digression: **monotone queries**

Monotone Queries

- A query with input relations R, S, T, … is called *monotone* if, whenever we increase a relation, the query answer also increases (or stays the same)
- *Increase* here means *larger set*

Monotone Queries

- A query with input relations R, S, T, … is called <u>*monotone*</u> if, whenever we increase a relation, the query answer also increases (or stays the same)
- <u>*Increase*</u> here means <u>*larger set*</u>
- Mathematically

If $R \subseteq R', S \subseteq S', \ldots$ **then** $Q(R, S, \ldots) \subseteq Q(R', S', \ldots)$

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

NON-MONOTONE

Which Ops are Monotone?

- Selection: σ_{pred}
- Projection: $\Pi_{A,B,\ldots}$
- Join: $\bowtie$
- Union: $\cup$
- Difference: $-$
- Group-by-sum: $\gamma_{A,B,sum(C)}$

Which Ops are Monotone?

- Selection: σ_{pred} **MONOTONE**
- Projection: $\Pi_{A,B,\ldots}$ **MONOTONE**
- Join: $\bowtie$ **MONOTONE**
- Union: U **MONOTONE**
- Difference: $-$ **NON-MONOTONE**
- Group-by-sum: $\gamma_{A,B,sum(C)}$ **NON-MONOTONE**

Digression

- Understanding monotone v.s. non-monotone queries gives you insights into the complexity of SQL queries
- Rule of thumb: if the English formulation of a query is non-monotone, then you <u>*need*</u> to use a subquery OR aggregate in SQL

Return SUPPLIERS who supply some product with price > $10000

Return SUPPLIERS who supply only products with price > $10000

Back to Datalog

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Will show this next

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone

Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:

$$\text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1})$$

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0(= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:
$IDB_{t+1} = \text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1}) = IDB_{t+2}$

Naïve Evaluation Algorithm

Consequence: The naïve algorithm *terminates*, in $O(n^k)$ steps, where:

- n = number of distinct values in the DB
- k = arity of widest IDB relation

Proof: IDBs increases to ≤ $O(n^k)$ facts

Summary

- Datalog = light-weight syntax, recursion
- Powerful optimizations:
  - Semi-naïve; magic sets; asynchronous exec
- Limitation: monotone queries only

Next time: extensions to non-monotone