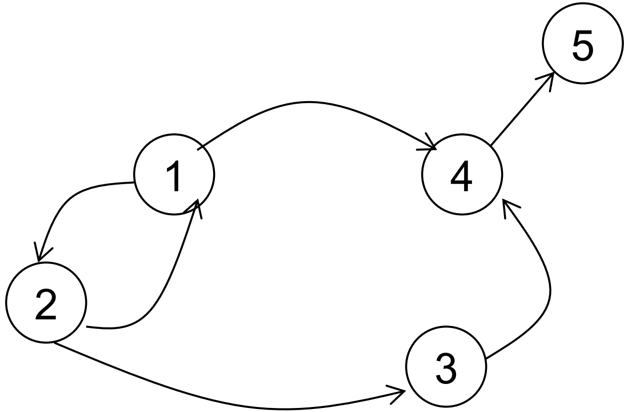
CSE544
Data Management

Lecture 12

Datalog (Part 2 of 2)

Review: Datalog




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

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

Second rule generates this

New facts

Comments

Two conventions for arguments:

- Named perspective (SQL):
  Part.price, Part.weight, Part.name
- Unnamed perspective (datalog, Java, math):
  f(x,y); Part(x,y,z), x/y

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

- Intially T = R
- Then T = R⋈R, R⋈R⋈R, R⋈R⋈R⋈R,…

Outline

- Naïve Evaluation Algorithm
- Extensions
- Semi-naïve Algorithm

Naïve Evaluation Algorithm

- Every rule → SPJ* query

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

Naïve Evaluation Algorithm

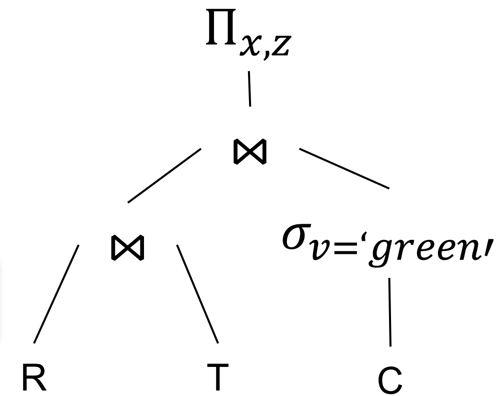
- Every rule → SPJ* query

```
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



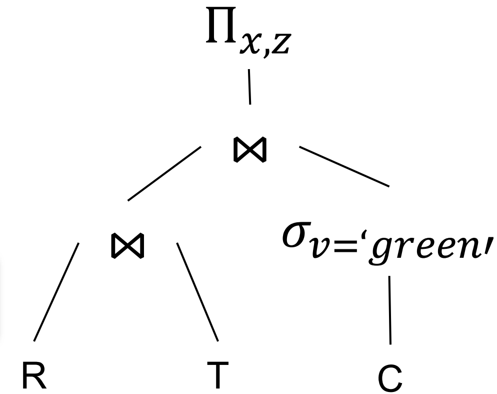
T(x,z) :- R(x,y), T(y,z), C(y,'green')

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

Naïve Evaluation Algorithm

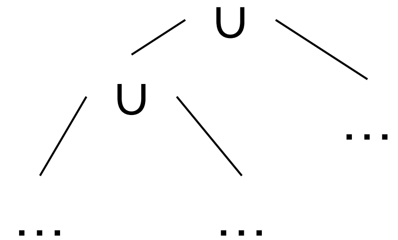
- Every rule → SPJ* query

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



- Multiple rules same head → USPJ+

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

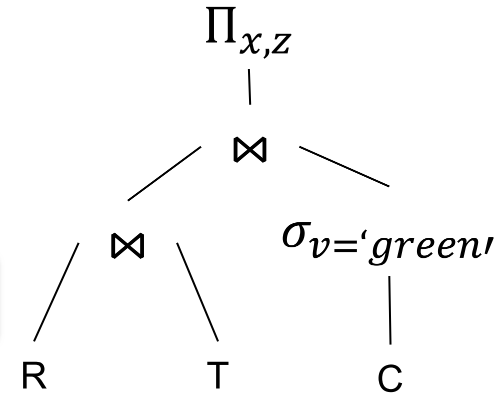


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

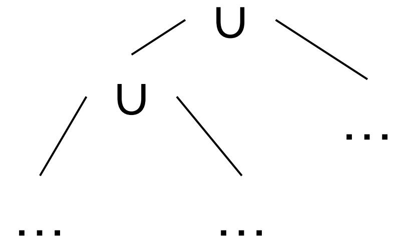
- Every rule → SPJ* query

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

- 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)
```

$\text{Odd} := \emptyset; \text{Even} := \emptyset;$

repeat

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

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

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)
```

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

repeat

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

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

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

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

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)
```

$\mathrm{Odd} := \emptyset; \mathrm{Even} := \emptyset;$

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 *monotone* if, whenever we *increase* a relation, the query answer also *increases* (or stays the same)
- *Increase* here means *larger set*
- 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 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

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.price >= 10000 )
```

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.price >= 10000 )
```

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.price >= 10000 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno < 10000 )
```

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.price >= 10000 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno < 10000 )
```

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

MONOTONE (Selection, Projection, Join, Union)

NON-MONOTONE (Difference, Group-by-sum)

Digression

- If the English formulation of a query is non-monotone, then you *need* 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

Digression

- If the English formulation of a query is non-monotone, then you *need* to use a subquery OR aggregate in SQL

Return SUPPLIERS who supply some product with price ≥ $10000

MONOTONE

NON-MONOTONE

Return SUPPLIERS who supply only products with price ≥ $10000

Back to Datalog

Naïve Evaluation Algorithm:

$IDB := \emptyset$
repeat
  $IDB := USPJ(IDB)$
until no more change

- 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 $\sigma, \Pi, \bowtie, \cup$

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

IDBs grow: $IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \cdots$

Q: for how many steps can they grow?

Naïve Evaluation Algorithm

IDBs grow: $IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \cdots$

Q: for how many steps can they grow?

A: they can hold at most $O(n^k)$ tuples

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

Fact: naïve algo *terminates* in $O(n^k)$ steps

Summary

- Datalog = light-weight syntax, recursion
- Always terminates, always in PTIME
- **Limitation**: monotone queries only
- Next: extensions to non-monotone queries

Outline

- Naïve Evaluation Algorithm
- Extensions
- Semi-naïve Algorithm

Non-monotone Extensions

- Aggregates
- Grouping
- Negation

No standard syntax
We will follow Souffle

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

Aggregates

Syntax: min x : { Actor(x, y, _), y = ‘John’ }

```
Q(m) :- m = min x : { Actor(x, y, _), y = ‘John’ }
```

Meaning (in SQL)

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

Aggregates in Souffle:

- `count`
- `min`
- `max`
- `sum`

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

Grouping

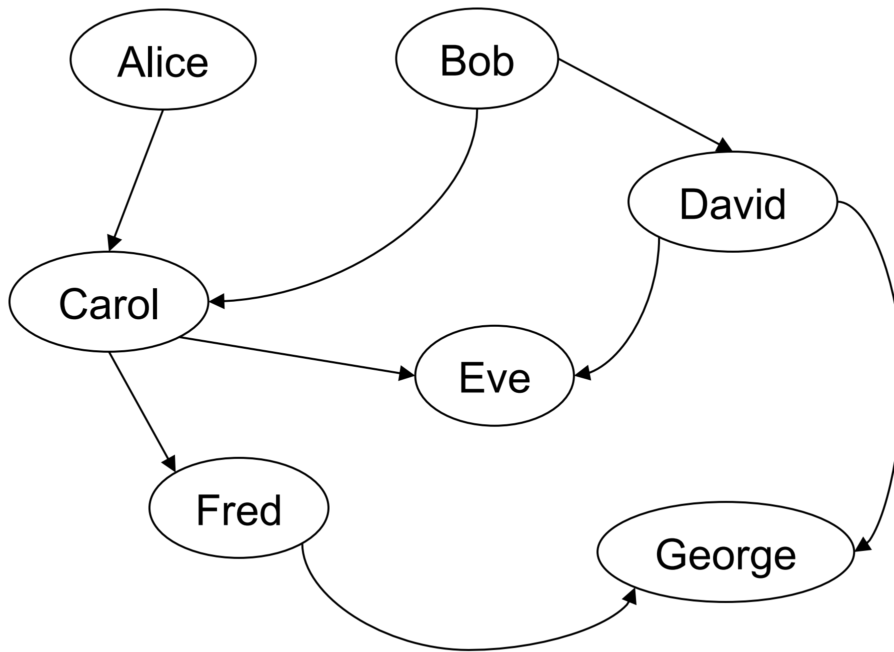
```
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }
```

Meaning (in SQL)

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

Examples

A genealogy database (parent/child)



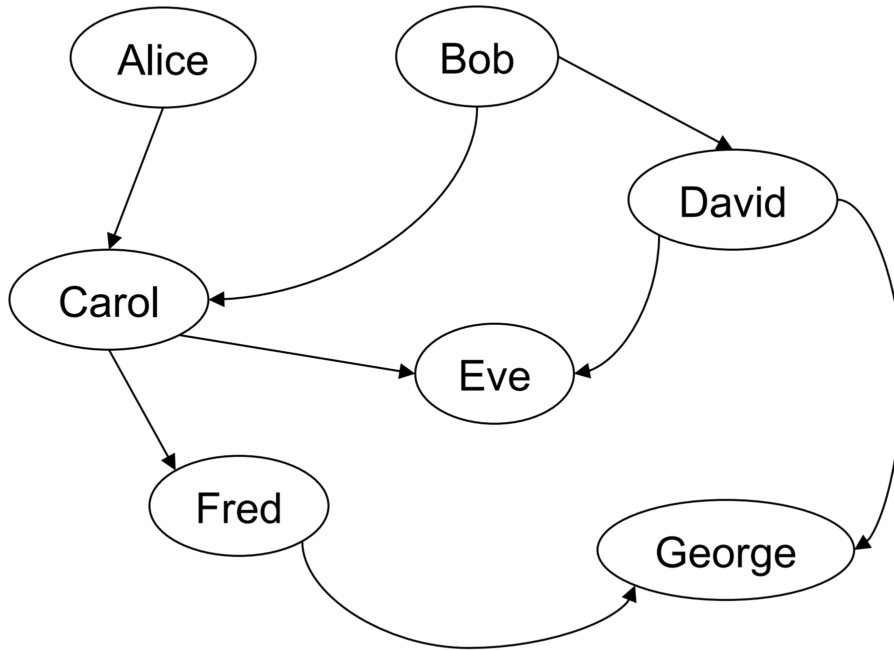
ParentChild

p	c
Alice	Carol
Bob	Carol
Bob	David
Carol	Eve
…	

ParentChild(p,c)

Count Descendants

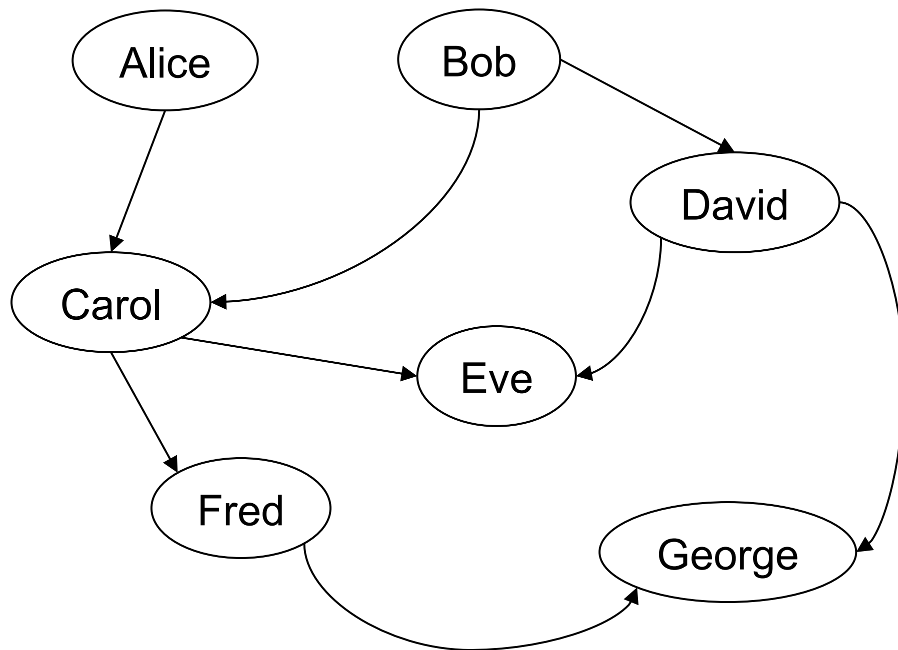
For each person, count his/her descendants



ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



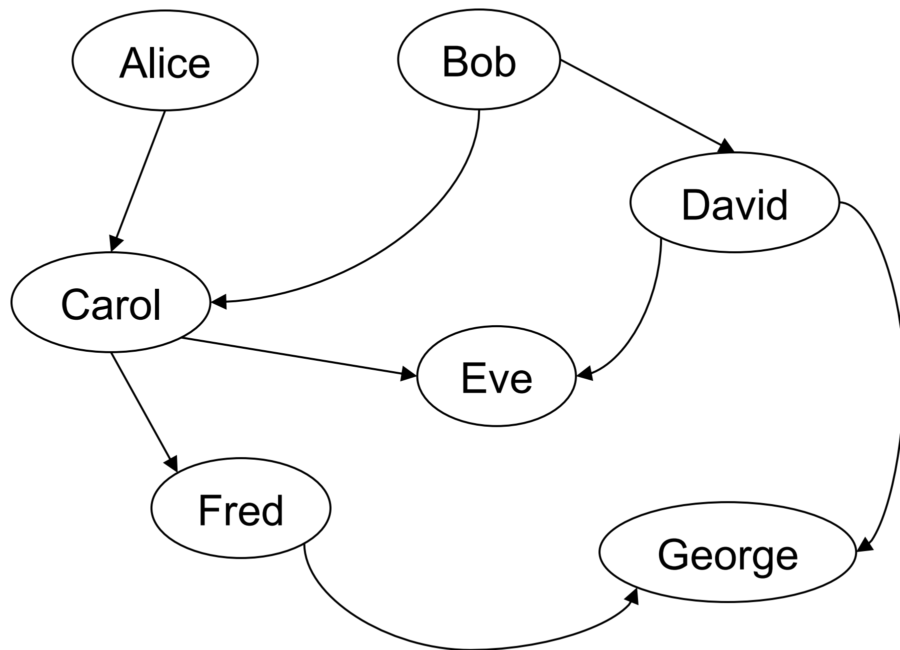
Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

ParentChild(p,c)

Count Descendants

For each person, count his/her descendants



Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

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

Count Descendants

Compute transitive closure of ParentChild

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

Count Descendants

Compute transitive closure of ParentChild

```
// 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).
```

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,_) }.
```

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,_) }.
```

Count Descendants

How many descendants does Alice have?

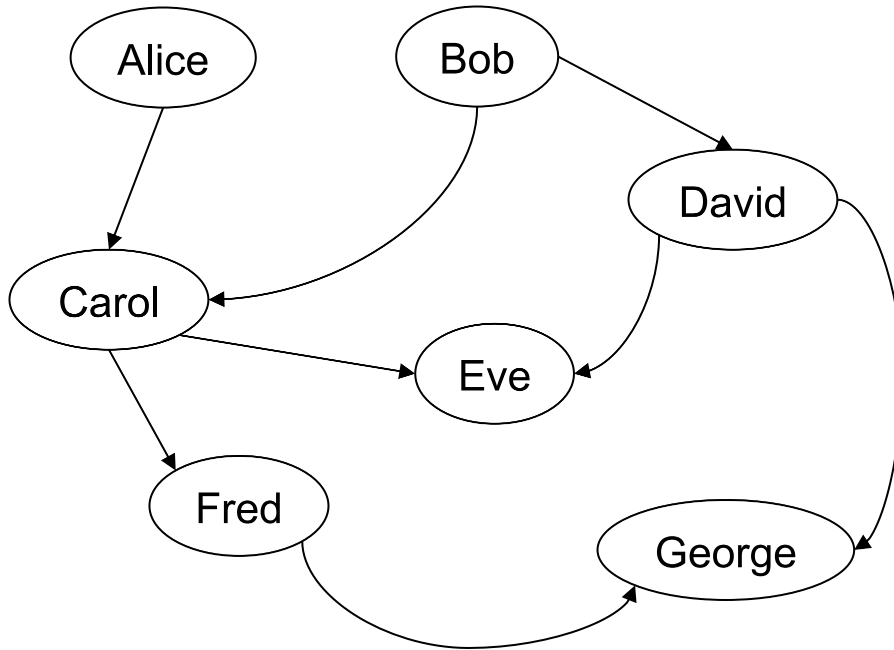
```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.

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

Negation: use “!”

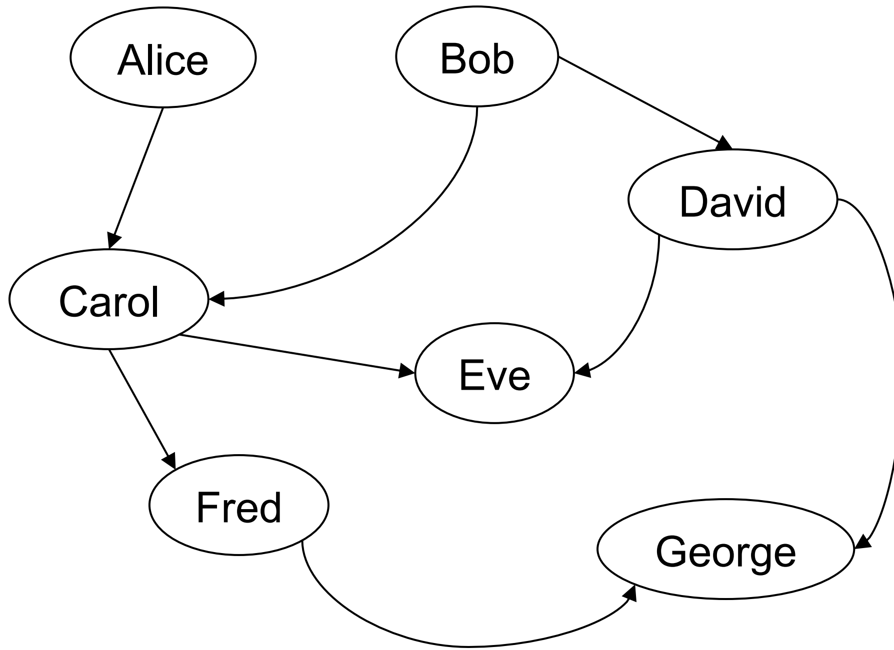
Find all descendants of Bob that are not descendants of Alice



ParentChild(p,c)

Negation: use “!”

Find all descendants of Bob that are not descendants of Alice



Answer

x
David

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

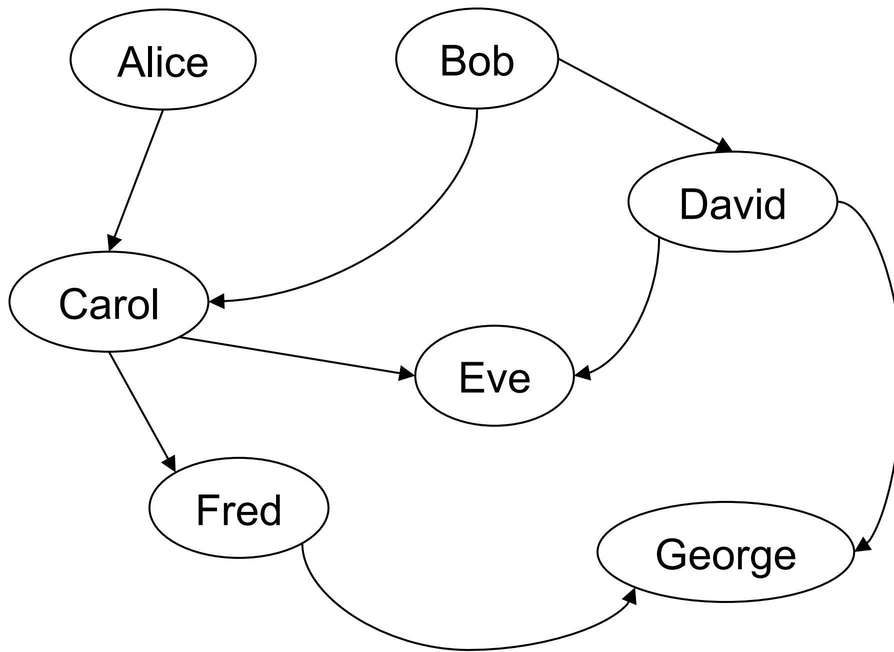
```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// Compute the answer: notice the negation
Q(x) :- D("Bob",x), !D("Alice",x).
```

ParentChild(p,c)

Same Generation

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



SG

p1	p2
Carol	David
Eve	George
Fred	George
Fred	Eve

Same Generation

Compute pairs of people at the same generation

```
// common parent
```

ParentChild(p,c)

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

ParentChild(p,c)

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)
```

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
```

Stratified Datalog

Recursion conflicts with non-monotone queries

- Example: what does this mean?

```
Happy(Bob):- !Happy(Alice).
Happy(Alice) :- !Happy(Bob).
```

- A program is *stratified* if it can be partitioned into *strata,* such that every IDB predicate in a non-monotone position has been defined in an earlier stratum

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

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

Stratum 2

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

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

```
T(p,c) :- D(p,_), c = count : { D(p,_) }.
Q(d) :- T(p,d), p = "Alice".
```

Stratum 2

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

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

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

Stratum 2

May use !D

```
Happy(Bob):- !Happy(Alice).
Happy(Alice) :- !Happy(Bob).
```

Non-stratified

Outline

- Naïve Evaluation Algorithm
- Extensions
- Semi-naïve Algorithm

Problem with the Naïve Algorithm

- Same facts are discovered repeatedly
- The *semi-naïve* algorithm reduces the number of rediscovered facts
- Uses *incremental view maintenance*

Incremental View Maintenance

- Consider a materialized view:
  - Defined it: V = Query(R, S, T, …)
  - Compute it and store the result

Incremental View Maintenance

- Consider a materialized view:
  - Defined it: V = Query(R, S, T, …)
  - Compute it and store the result
- Suppose some table gets updated:
  - $R \leftarrow R \cup \Delta R$

Incremental View Maintenance

- Consider a materialized view:
  - Defined it: V = Query(R, S, T, …)
  - Compute it and store the result
- Suppose some table gets updated:
  - $R \leftarrow R \cup \Delta R$
- We want to update the view
  - $V \leftarrow V \cup \Delta V$

Incremental View Maintenance

- Consider a materialized view:
  - Defined it: $V = Query(R, S, T, \ldots)$
  - Compute it and store the result
- Suppose some table gets updated:
  - $R \leftarrow R \cup \Delta R$
- We want to update the view
  - $V \leftarrow V \cup \Delta V$

IVM Problem: compute $\Delta V = \Delta Query(\Delta R, R, S, T, \ldots)$

IVM

Example 1:

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

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

IVM

Example 1:

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

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

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

IVM

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) ?

IVM

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 $\Delta V(x,y)$?

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

We use datalog convention
to represent the union
of three queries

IVM

Example 3:

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

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

IVM

Example 3:

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

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

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

Semi-Naïve Evaluation Algorithm

Key idea:

- Use IVM in the inner loop of the naïve algorithm

Applies *only* to one monotone stratum

Semi-Naïve Evaluation Algorithm

$IDB := \emptyset$
repeat
  $IDB := USPJ(IDB)$
until no more change

Naive

Semi-Naïve Evaluation Algorithm

$IDB := \emptyset$
repeat
  $IDB := USPJ(IDB)$
until no more change

Naive

Semi-naive

$IDB := \Delta := NonRecursiveUSPJ$
repeat
  $\Delta := \Delta USPJ(IDB, \Delta) - IDB$
  if $\Delta = \emptyset$ **then** break
  $IDB := IDB \cup \Delta$

Example

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

$T := \emptyset$;

repeat

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

until [no more change]

Naïve

Example

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

$T := \emptyset;$

repeat

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

until [no more change]

Naïve

Semi-naïve

$T := \Delta T := R;$

repeat

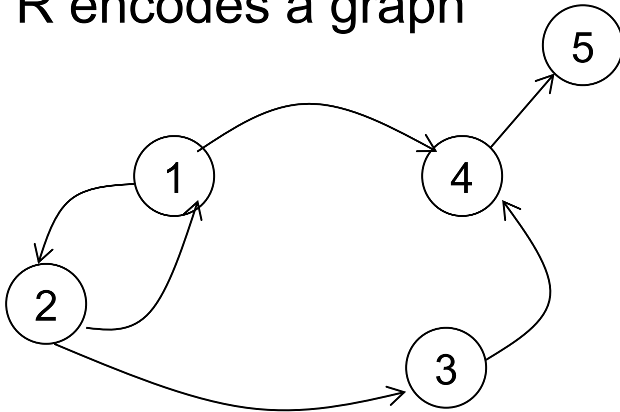
$\quad \Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$

$\quad$ **if** $\Delta T = \emptyset$ **then** break

$\quad T := T \cup \Delta T;$

Example

R encodes a graph



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

$T := \Delta T := R;$

repeat

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

if $\Delta T = \emptyset$ **then** break

$T := T \cup \Delta T;$

R=

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

Initially:

ΔT=

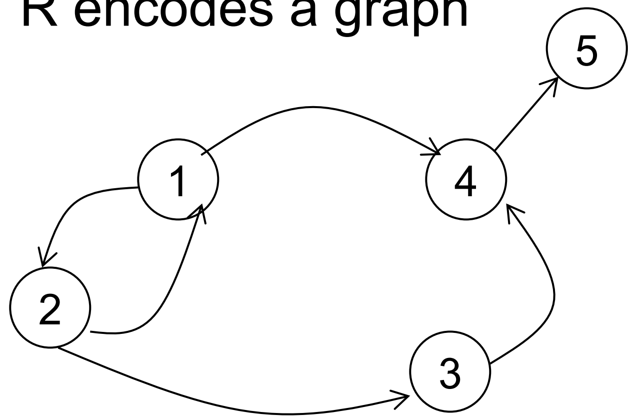
1	2
1	4
2	1
2	3
3	4
4	5

T=

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

Example

R encodes a graph



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

$T := \Delta T := R;$
repeat
$\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
if $\Delta T = \emptyset$ **then** break
$T := T \cup \Delta T;$

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

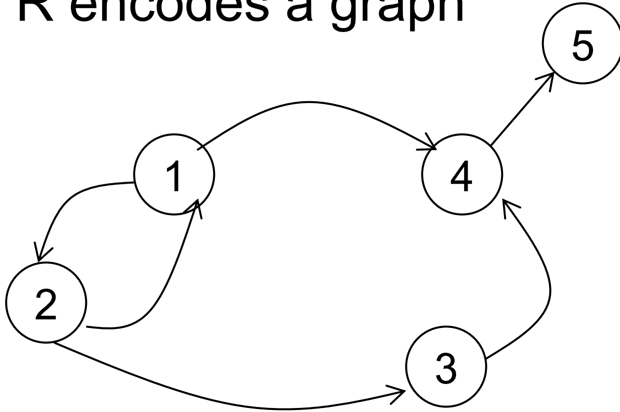
1	1
1	3
1	5
2	2
2	4
3	5

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

R encodes a graph



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

$T := \Delta T := R;$
repeat
$\quad \Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
$\quad$ **if** $\Delta T = \emptyset$ **then** break
$\quad T := T \cup \Delta T;$

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

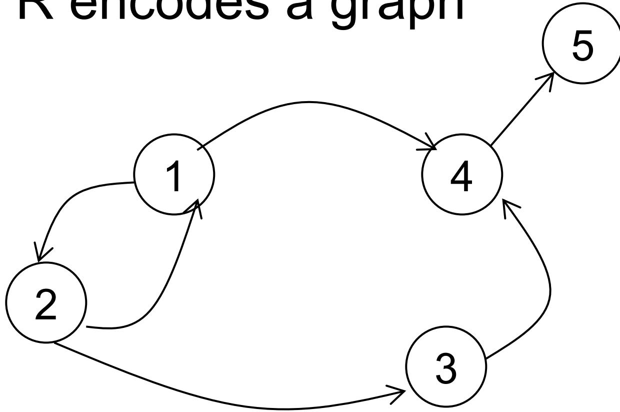
2	5

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

R encodes a graph



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

$T := \Delta T := R;$
repeat
$\Delta T := \Pi_{x,y}(R \bowtie \Delta T) - T;$
if $\Delta T = \emptyset$ **then** break
$T := T \cup \Delta T;$

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

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

Other Datalog Optimizations

- Every USPJ query can be optimized using traditional optimizers
- Semi-naïve evaluation
- Magic sets: a powerful mean to push predicates past recursion
- Asynchronous execution: start iteration t+1 before iteration t has finished