Deductive Databases (MIF14 - UCBL)

Datalog Programs and their Evaluation

2020/2021

Outline

- Datalog: Facts and Rules
  - Evaluating Datalog programs (with no recursion)
- Recursive rules
  - Linear datalog vs Non-linear Datalog
  - Dependency Graph
  - Evaluating recursive rules (least fixed point)
  - The Naïve Evaluation Algorithm
- Safe/Unsafe rules

Datalog

- Logical query language for the relational model
- Consists of “if-then” rules made up of atoms:
  - relational : predicates corresponding to relations
  - arithmetic
- Basic concepts
  - predicate (relation)
  - term, constant, variable
  - goal, subgoal, head
  - rule, fact

Datalog Facts and Rules

Actor(pid,fname,lname)

Casts(pid,mid)

Movie(mid,name,year)

Find Movies made in 1940

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,1940)
```

Datalog Facts and Rules

Actor(pid,fname,lname)

Casts(pid,mid)

Movie(mid,name,year)

Find Actors who acted in Movies made in 1940

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,1940)

Q2(y):- Actor(z,f,i), Casts(z,x),
Movie(x,y,1940)

Datalog Facts and Rules

Actor(pid,fname,lname)

Casts(pid,mid)

Movie(mid,name,year)

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

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,1940)
```

```
Q2(f,i):- Actor(z,f,i), Casts(z,x),
          Movie(x,y,1940)
```

```
Q3(f,i):- Actor(z,f,i),
          Casts(z,x1),
          Movie(x1,y,1940),
          Casts(z,x2),
          Movie(x2,y,1910)
```

Datalog Programs

- A datalog program is a collection of one or more rules
  - Each rule expresses the idea that, from certain combinations of tuples in certain relations, we may **infer** that some other tuple must be in some other relation or in the query answer
- In a program, predicates can be either
  - EDB = Extensional Database = stored table
  - IDB = Intensional Database = relation defined by rules
- Never both!
- EDB cannot appear in the heads

EDBs and IDBs

Actor(pid,fname,lname)

Casts(pid,mid)

Movie(mid,name,year)

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

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,1940)

Q2(f,i):- Actor(z,f,i), Casts(z,x),
Movie(x,y,1940)

Q3(f,i):- Actor(z,f,i),
Casts(z,x1),
Movie(x1,y,1940),
Casts(z,x2),
Movie(x2,y,1910)

EDBs = Actor,Casts,Movie

IDBs = Q1,Q2,Q3

Evaluating Datalog Programs

- As long as there is no recursion
  - we can pick an order to evaluate the IDB predicates,
  - so that all the predicates in the body of its rules have already been evaluated
- If an IDB predicate has more than one rule
  - each rule contributes tuples to its relation

Datalog Program: An example

Sells(bar, beer, price)

Beers(name, manf)

Find the manufacturers of beers Joe doesn't sell

Facts

```
Sells('Adam's bar','firstBeer',1.2).
Sells('Adam's bar','bestBeer',3.0).
Sells('Joe's bar','bestBeer',2.5).
Beers('firstBeer','AAA').
Beers('bestBeer','CCC').
```

Rules

JoeSells(b) :- Sells('Joe''s Bar', b, p)

Answer(m) :- Beers(b,m),
¬ JoeSells(b)

Datalog Program: An example

- Step 1: Examine all `Sells` tuples with first component `'Joe''s Bar'`
  - Add the second component to `JoeSells`

Sells(bar, beer, price)

Beers(name, manf)

Find the manufacturers of beers Joe doesn't sell

Facts

```
Sells('Adam's bar','firstBeer',1.2).
Sells('Adam's bar','bestBeer',3.0).
Sells('Joe's bar','bestBeer',2.5).
Beers('firstBeer','AAA').
Beers('bestBeer','CCC').
```

```
JoeSells('bestBeer').
```

Rules

JoeSells(b) :- Sells('Joe''s Bar', b, p)

Answer(m) :- Beers(b,m),
¬ JoeSells(b)

Datalog Program: An example

- Step 2: Examine all `Beers` tuples `(b,m)`
  - If `b` is not in `JoeSells,` add `m` to `Answer`

Sells(bar, beer, price)
Beers(name, manf)

Find the manufacturers of beers Joe doesn't sell

Facts

```
Sells('Adam's bar','firstBeer',1.2).
Sells('Adam's bar','bestBeer',3.0).
Sells('Joe's bar','bestBeer',2.5).
Beers('firstBeer','AAA').
Beers('bestBeer','CCC').
```

```
JoeSells('bestBeer').
```

```
Answer('AAA').
```

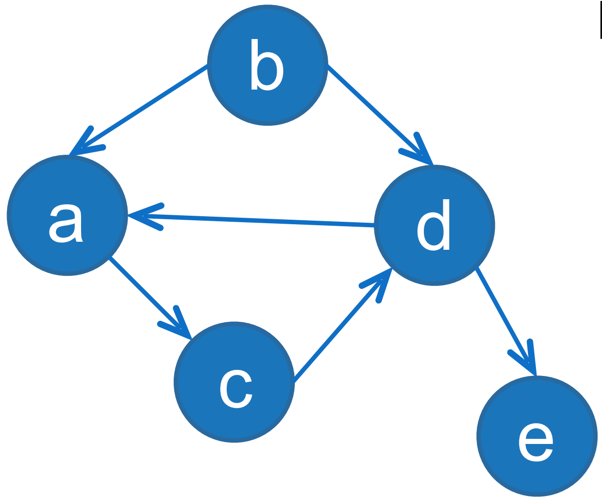
Rules

JoeSells(b) :- Sells('Joe''s Bar', b, p)

Answer(m) :- Beers(b,m),
¬ JoeSells(b)

Graph Example

- Write a Datalog program to find paths of length two
  - `P(v1,v2)`: the start and the end vertices

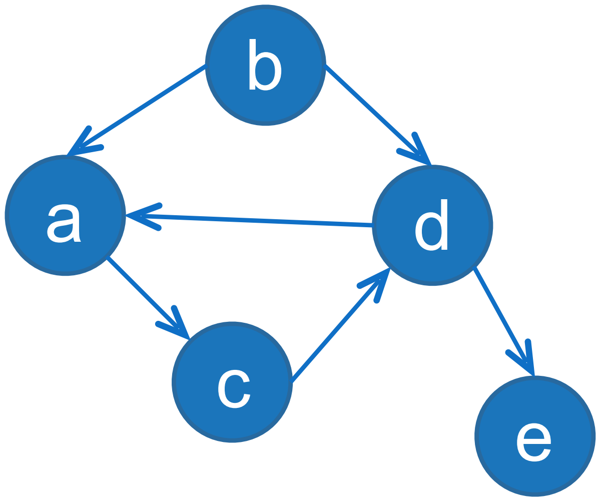


E (edge relation)

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

Graph Example

- Write a Datalog program to find paths of length two
  - `P(v1,v2)`: the start and the end vertices



E (edge relation)

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

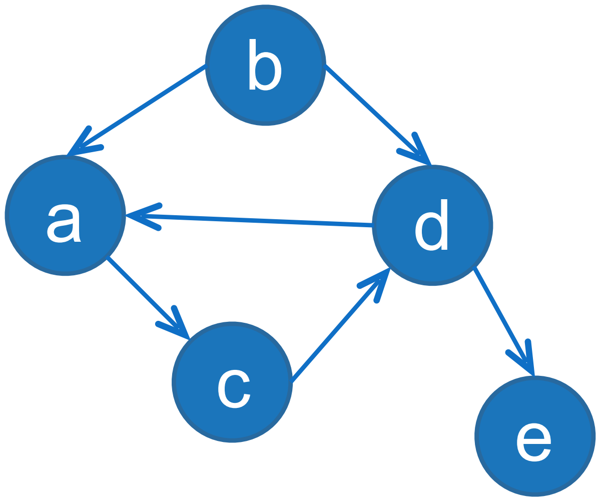
P

v1	v2
a	d
b	c
b	a
b	e
c	a
c	e
d	c

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

Graph Example

- Write a Datalog program to find all pairs of vertices `(u,v)` such that `v` is reachable from `u`

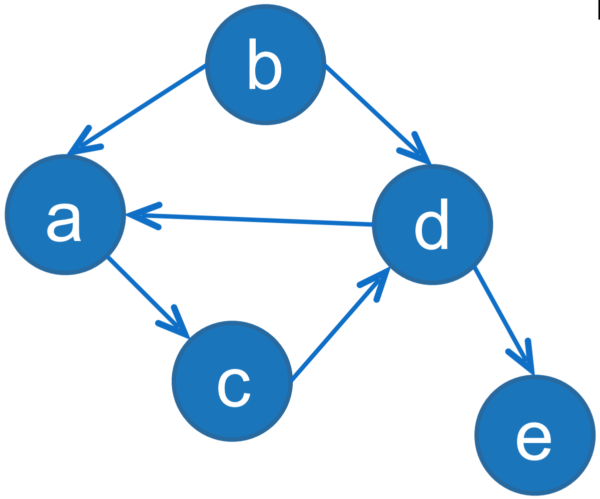


E (edge relation)

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

Recursive Rules

- Write a Datalog program to find all pairs of vertices `(u,v)` such that `v` is reachable from `u`



E (edge relation)

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

- Options 1 and 2 are linear
- Option 3 is non-linear

Option 1

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

Option 2

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

Option 3

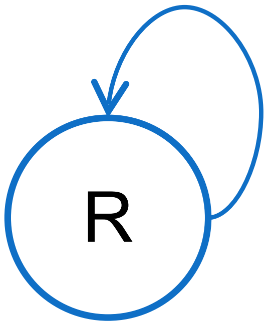
R(x, y) :- E(x, y)
R(x, y) :- R(x, z), R(z, y)

Linear Datalog

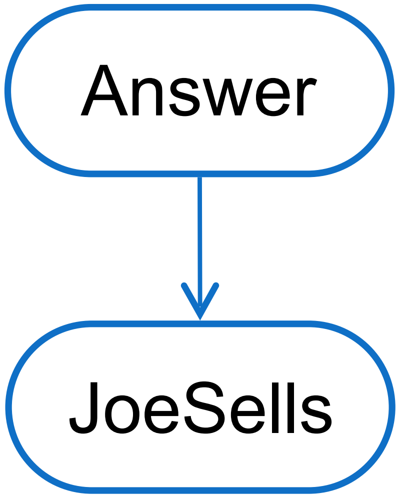
- Linear rule
  - at most one atom in the body that is recursive with the head of the rule
  - e.g. $R(x, y)$:- $E(x, z), R(z, y)$
- Linear datalog program
  - If all rules are linear
  - Like linear recursion

Dependency Graph

- Form a dependency graph whose nodes are IDB predicates
- An Arc $X \rightarrow Y$ if and only if there is a rule with X in the head and Y in the body
  - Cycle = recursion;
  - No cycle = no recursion



Graph Example



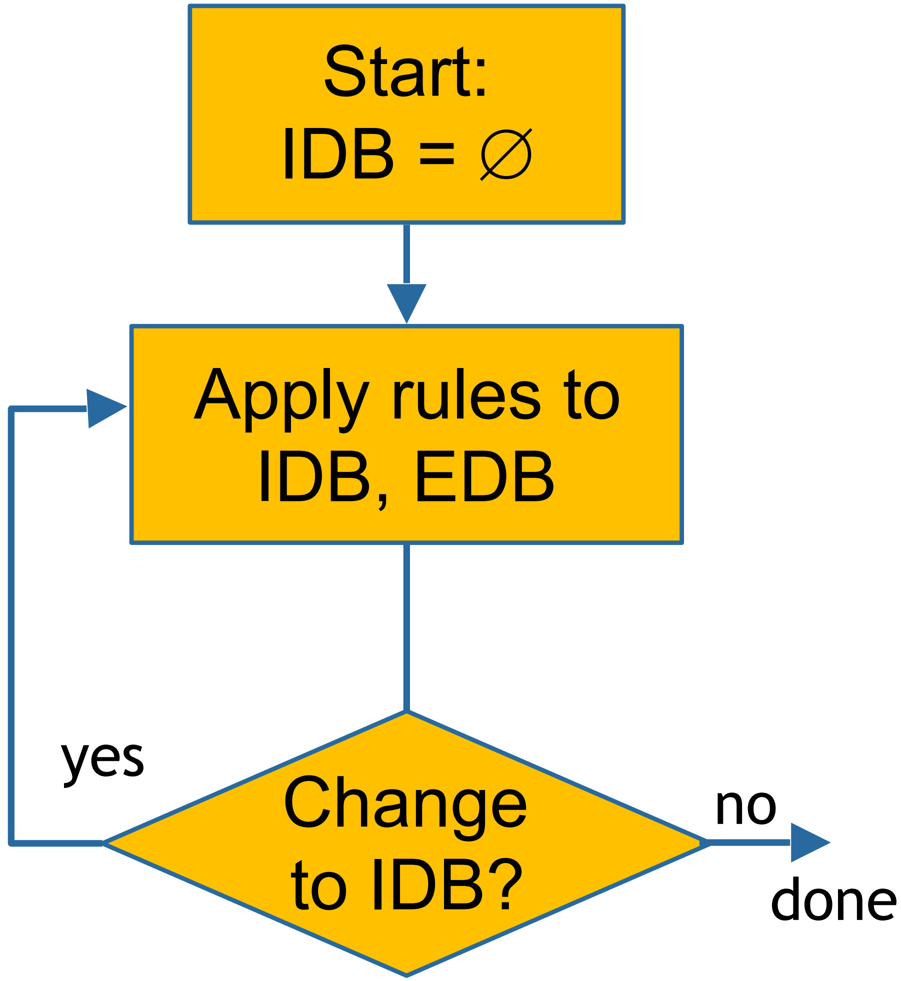
Bars and Beers Example

Evaluation of Recursive Rules

- The following works when there is no negation
  1. Start by assuming all IDB relations are empty
  2. Repeatedly evaluate the rules using the EDB and the previous IDB, to get a new IDB
  3. End when no change to IDB
- This set of IDB tuples is called the least fixed point of the rules

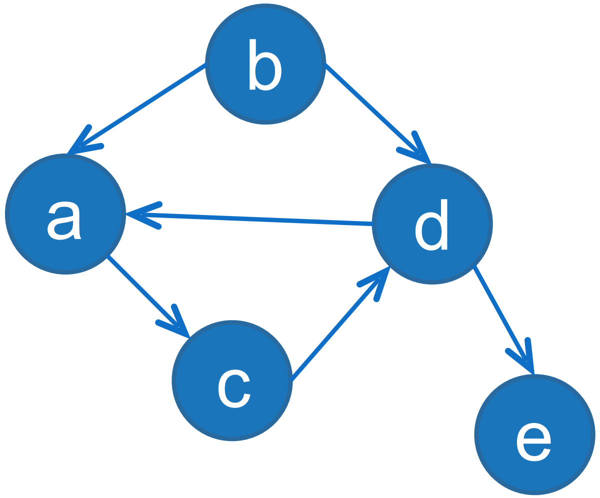
The "Naïve" Evaluation Algorithm

- In all subsequent iteration
  - check if any of the rules can be applied
  - do union of all the rules with the same head IDB



Naïve Evaluation: Graph Example

- `R(u,v)` vertices such that `v` is reachable from `u`



Iteration 1

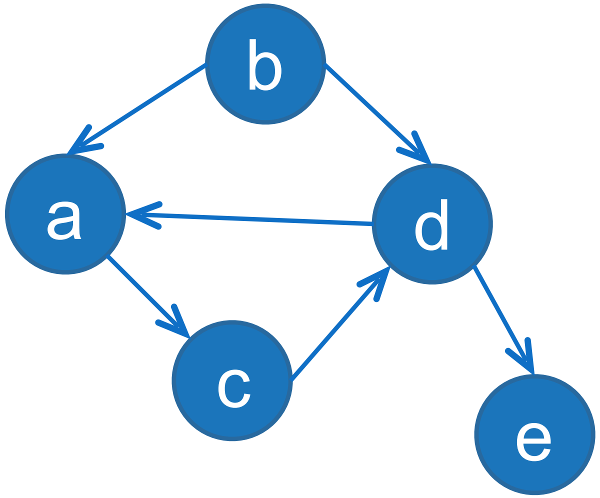
R = E = R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

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

Naïve Evaluation: Graph Example

- `R(u,v)` vertices such that `v` is reachable from `u`



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

R = E = R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

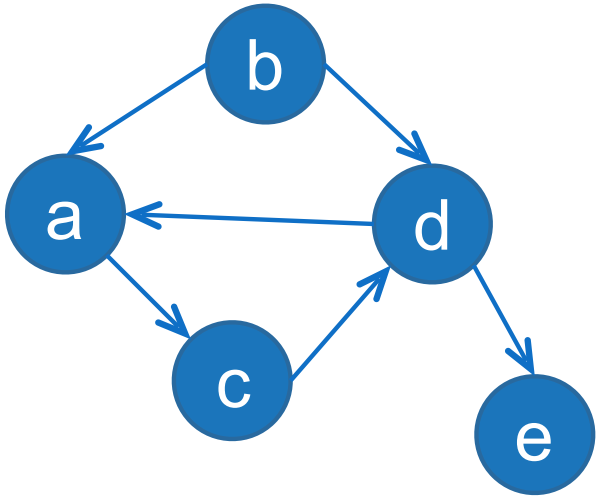
Iteration 2

R2 $\neq$ R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Naïve Evaluation: Graph Example

- `R(u,v)` vertices such that `v` is reachable from `u`



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

R = E = R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

R2 $\neq$ R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

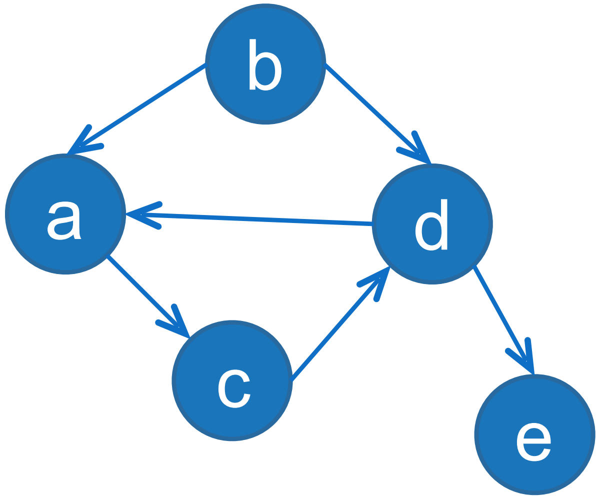
Iteration 3

R3 $\neq$ R2

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
d	d
c	c

Naïve Evaluation: Graph Example

- `R(u,v)` vertices such that `v` is reachable from `u`



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

R = E = R1

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e

$R2 \neq R1$

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 4

$R3 \neq R2$

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
d	d
c	c

$R4 = R3$

v1	v2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
d	d
c	c

STOP

Linear vs. Non-Linear Recursion

- Linear recursion is easier to implement
  - For linear recursion, just keep joining newly generated relation with the "base" relation
  - For non-linear recursion, need to join newly generated relation with all existing relation rows
- Non-linear recursion may take fewer steps to converge, but perform more work
  - Example: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
    - Linear recursion takes 4 steps
    - Non-linear recursion takes 3 steps
      - More work: e.g., $a \rightarrow d$ has two different derivations

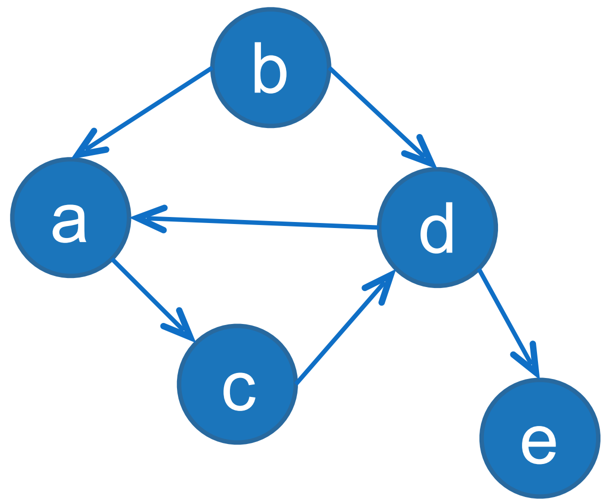
Termination of a Datalog Program

- A Datalog program always terminates – why?
  - Because the values of the variables are coming from the "active domain" in the input relations (EDBs)
- Active domain = (finite) values from the (possibly infinite) domain appearing in the instance of a database
  - e.g. age can be any integer (infinite), but active domain is only finitely many in R(id, name, age)
- Therefore the number of possible values in each of the IDBs is finite

Termination of a Datalog Program

- In the graph example `R(x,y)`, the values of `x` and `y` come from `{a,b,c,d,e}`
  - at most `5x5=25` tuples possible in the IDB `R(x,y)`
  - in any iteration, at least one new tuple is added in at least one IDB
  - Must stop after finite steps!

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



Problems with IDB Negation

- Model = set of IDB facts, plus the given EDB facts, that make the rules true for every assignment of values to variables
- When rules have negated IDB subgoals, there can be several minimal models

Minimal Models

- A minimal model should not properly contain any other model
  - Intuitively, we don’t want to assert facts that do not have to be asserted
- When there is no negation, a Datalog program has a unique minimal model
  - One given by naïve evaluation

Multiple Models: An example

Facts (tuples in the EDB)

```
Arc(1,2).
Arc(3,4).
Arc(4,3).
Source(1).
Target(2).
Target(3).
```

Rules

Reach(x):- Source(x)

Reach(x):- Reach(y), Arc(y,x)

NoReach(x):- Target(x), ¬Reach(x)

- Setting is graph: Nodes designated as source and target
- Our problem is to find target nodes that are not reached from any source

Multiple Models: An example

Facts (tuples in the EDB)

```
Arc(1,2).
Arc(3,4).
Arc(4,3).
Source(1).
Target(2).
Target(3).
```

Rules

Reach(x):- Source(x)

Reach(x):- Reach(y), Arc(y,x)

NoReach(x):- Target(x), ¬Reach(x)

- A possible model
  - `Reach(1), Reach(2), NoReach(3)`

Multiple Models: An example

Facts (tuples in the EDB)

```
Arc(1,2).
Arc(3,4).
Arc(4,3).
Source(1).
Target(2).
Target(3).
```

Rules

Reach(x):- Source(x)

Reach(x):- Reach(y), Arc(y,x)

NoReach(x):- Target(x), ¬Reach(x)

- Another possible model
  - `Reach(1), Reach(2), Reach(3), Reach(4)`
  - `NoReach` is empty
  - Remember: a rule is true whenever its body is false

Stratified Negation

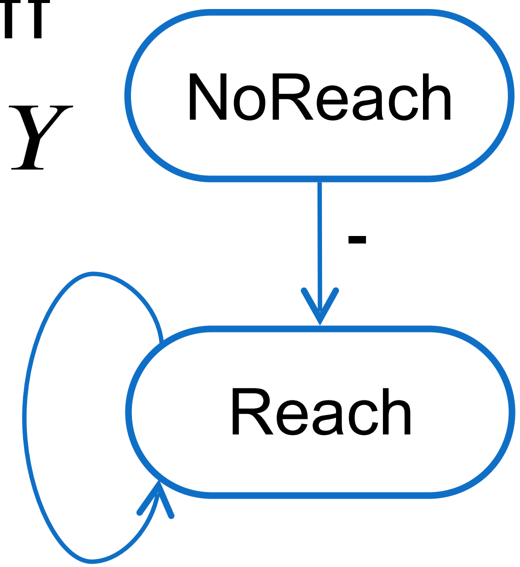
- Stratification is a constraint usually placed on Datalog with recursion and negation
  - It rules out negation wrapped inside recursion
  - Gives the sensible IDB relations when negation and recursion are separate
- Usually requires Negation to be stratified
- Stratification does two things:
  - Lets evaluate the IDB predicates in a way that it converges
  - Lets discover a “correct” solution in face of “many solutions”

Stratified Models

- Dependency graph describes how IDB predicates depend negatively on each other
- Stratified Datalog = no recursion involving negation
- Stratified model is a particular model that “makes sense” for stratified Datalog programs
  - Let us discover a “correct” solution in face of “many solutions”

Dependency Graph

- Form a dependency graph whose nodes are IDB predicates
- An Arc $X \rightarrow Y$ if and only if there is a rule with X in the head and Y in the body
- An Arc $X \rightarrow Y$ is labeled with - iff there is a subgoal with predicate Y that is a negated body

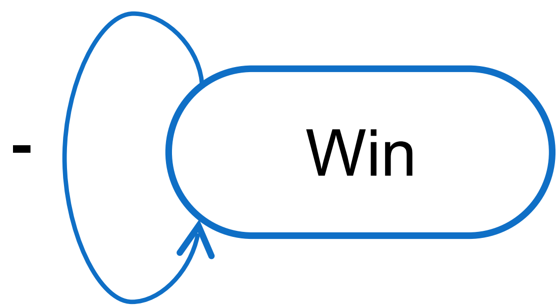


Datalog with Negation: Another example

Rule

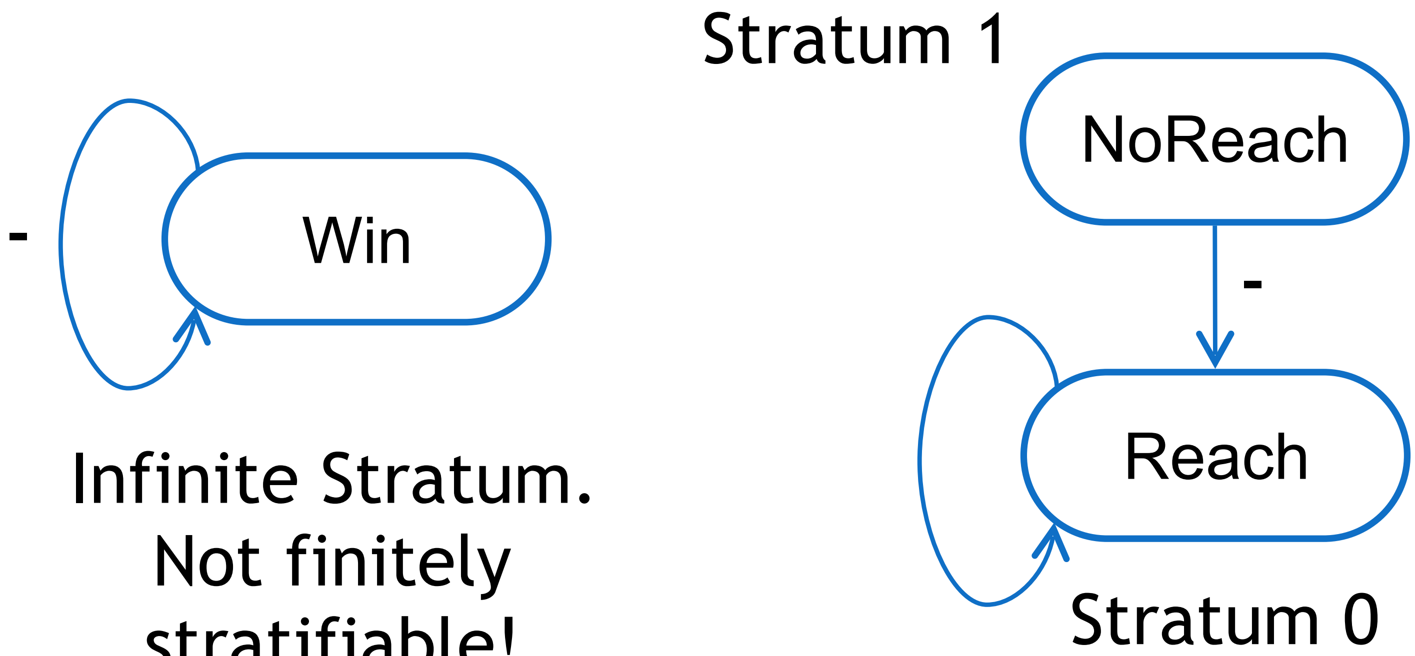
Win(x) :- Move(x, y), ¬Win(y)

- Represents games where you win by forcing your opponent to a position where they have no move



Strata

- The stratum of an IDB predicate is the largest number of –'s on a path from that predicate, in the dependency graph



Infinite Stratum.
Not finitely stratifiable!

Stratified Programs

- If all IDB predicates have finite strata
  - then the Datalog program is stratified
- If any IDB predicate has an infinite stratum,
  - then the program is unstratified, and
  - no stratified model exists

Stratified Models: Examples

- Let us consider the following rules

Rules

$R(x) \text{ :- } P(x), \neg Q(x)$

$Q(x) \text{ :- } \neg V(x), E(x)$

$P(x) \text{ :- } \neg S(x), E(x)$

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

$T(x) \text{ :- } Q(x)$

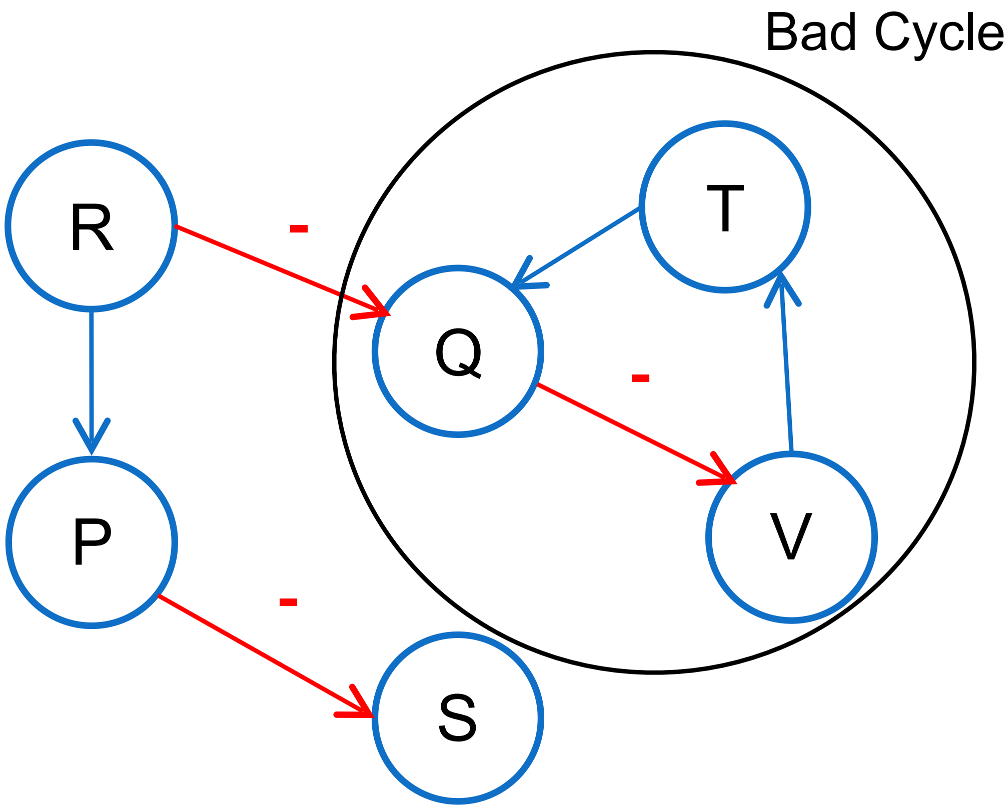
$S(x) \text{ :- } F(x)$

Stratified Models: Examples

- Let us consider the following rules

Rules

R(x) :- P(x), ¬Q(x)
Q(x) :- ¬V(x), E(x)
P(x) :- ¬ S(x), E(x)
V(x) :- T(x)
T(x) :- Q(x)
S(x) :- F(x)



Stratified Models: Examples

- Let us consider the following rules

Rules

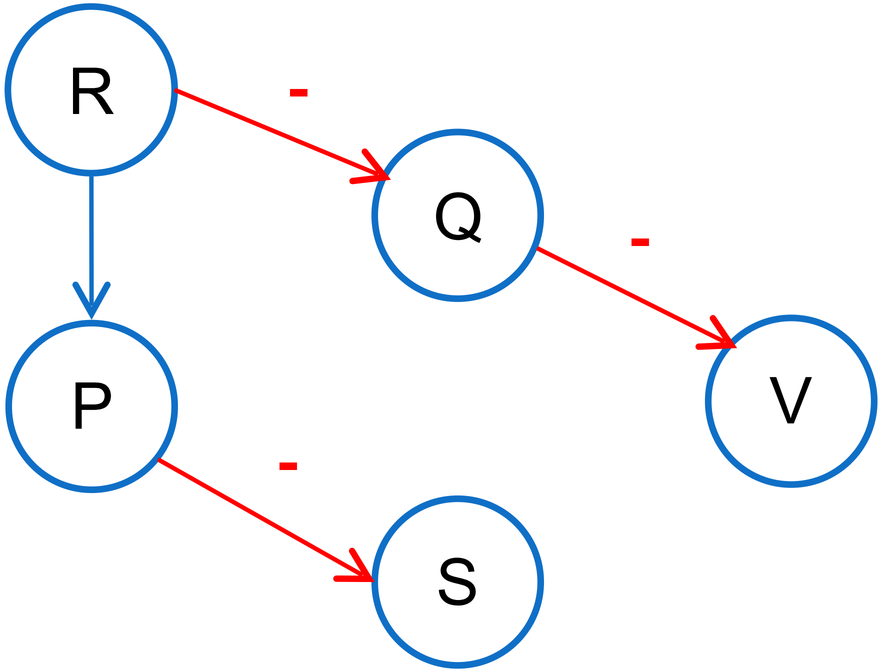
R(x) :- P(x), ¬Q(x)

Q(x) :- ¬V(x), E(x)

P(x) :- ¬ S(x), E(x)

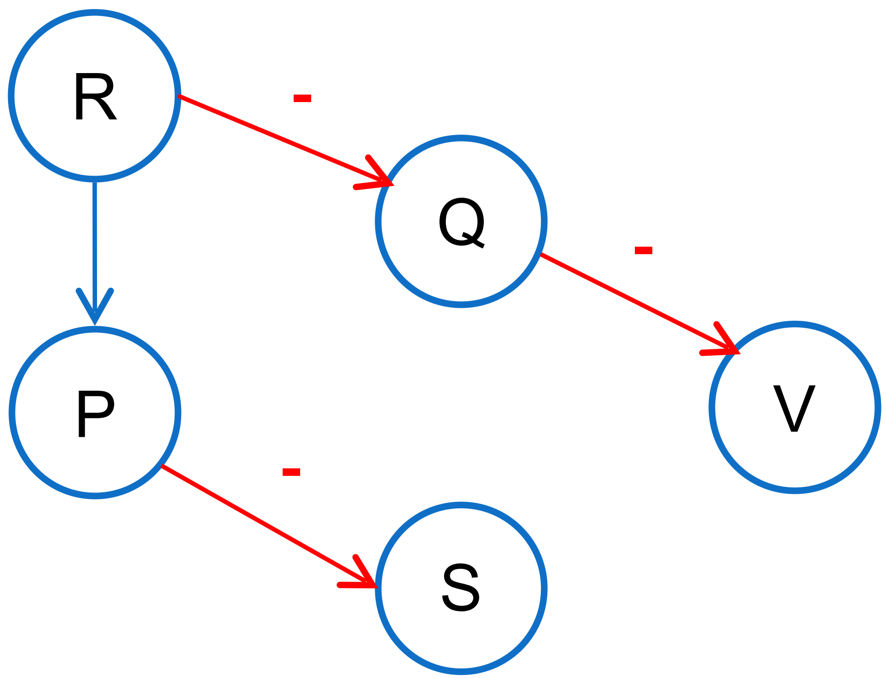
V(x) :- T(x)

S(x) :-F(x)



Stratified Models: Examples

- The program below is stratified.
  - Stratum 0 = {S, V}
  - Stratum 1 = {P, Q}
  - Stratum 2 = {R}



Rules

R(x) :- P(x), ¬Q(x)
Q(x) :- ¬V(x), E(x)
P(x) :- ¬ S(x), E(x)
V(x) :- T(x)
S(x) :-F(x)

Stratified Programs

- stratum 0:
  - do not depend on any negated IDB predicates
- stratum 1:
  - depend on negated IDB predicates from stratum 0;
- stratum 2:
  - depend on negated IDB predicates from stratum1,
- …
- stratum n:
  - depend on IDB predicates from stratum n-1.

Stratification

- A stratification of a datalog program P is a partitioning:

$$\Sigma = \{P_1, \ldots, P_m\}$$

of rule of P, into nonempty sets P_i such that

  - If $R \in P_i$, $R' \in P_j$, and $R \rightarrow R'$ is in the dependency graph, then $i \geq j$
  - If $R \in P_i$, $R' \in P_j$, and $R \rightarrow R'$ is in the dependency graph and is marked with "-", then $i > j$

- The sets $P_1, \ldots, P_m$ are called strata of P w.r.t. Σ

Evaluation of Stratified Programs

- Evaluate strata 0, 1,…in order.
- If the program is stratified, then any negated IDB subgoal has already had its relation evaluated
  - Safety assures that we can “subtract it from something”
  - Treat it as EDB
- Result is the stratified model

Evaluation of Stratified Programs

- Stratified Datalog programs have the following operational semantics:
  - First compute all IDB predicates in stratum 0 (using the usual fixpoint strategy)
  - …
  - Using IDB predicates from stratum n, compute IDB predicates from stratum n+1
- This produces unique minimal solutions for all IDB predicates

An example scenario

Person(pid,name,sex,married)

Find all single men

Facts = tuples in the database

```
Person(1,'Tom','M',0).
Person(2,'Alex','M',1).
Person(3,'Joe','F',1).
Person(4,'Mary','F',0).
Person(5,'John','M',0).
```

Rules

Man(x):- Person(id, x, 'M', m)

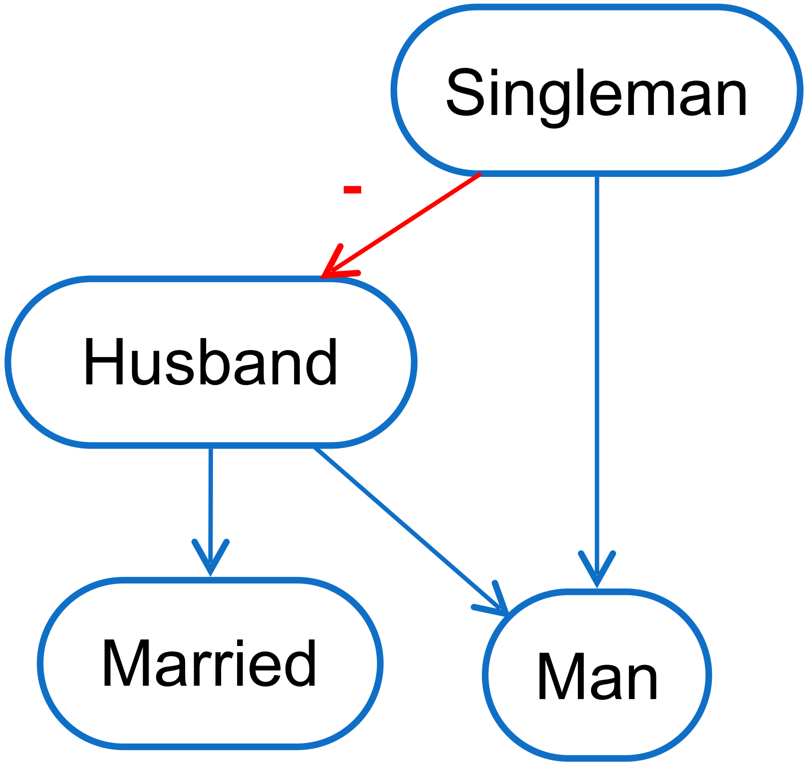
Married(x):- Person(id, x, s, 1)

Husband(x):- Man(x), Married(x)

Singleman(x):- Man(x), ¬Husband(x)

An example scenario: Dependency Graph

Person(pid,name,sex,married)



Find all single men

Rules

Man(x):- Person(id, x, ‘M’, m)

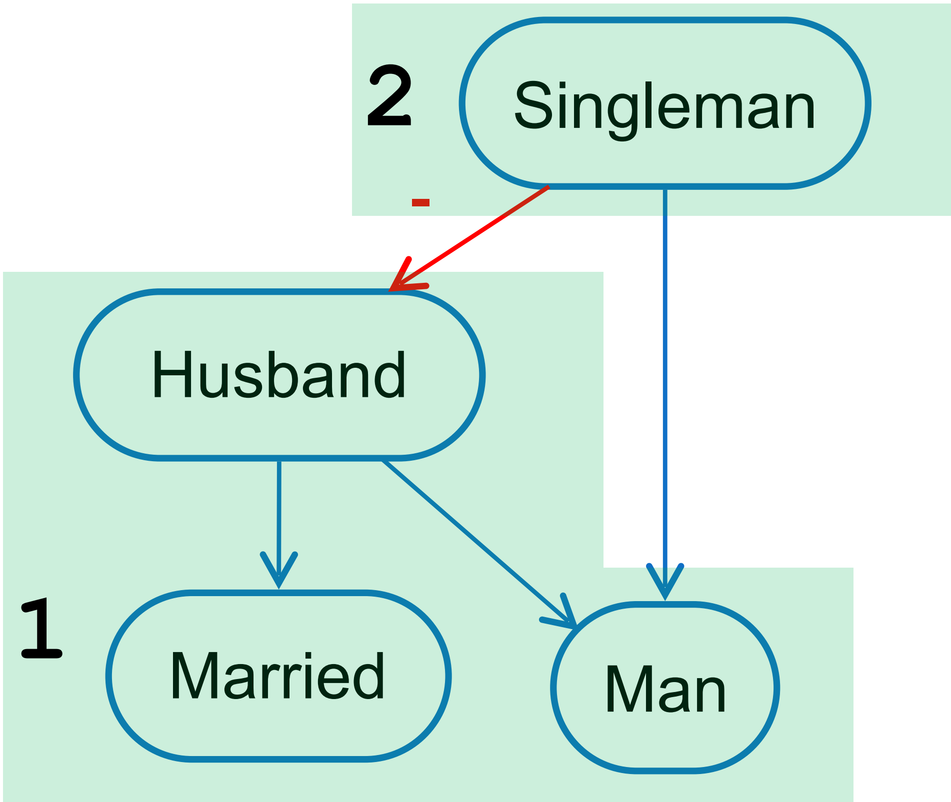
Married(x):- Person(id, x, s, 0)

Husband(x):- Man(x), Married(x)

Singleman(x):- Man(x), ¬Husband(x)

An example scenario: Stratification

Person(pid,name,sex,married)



Find all single men

Rules

Man(x):- Person(id, x, ‘M’, m)

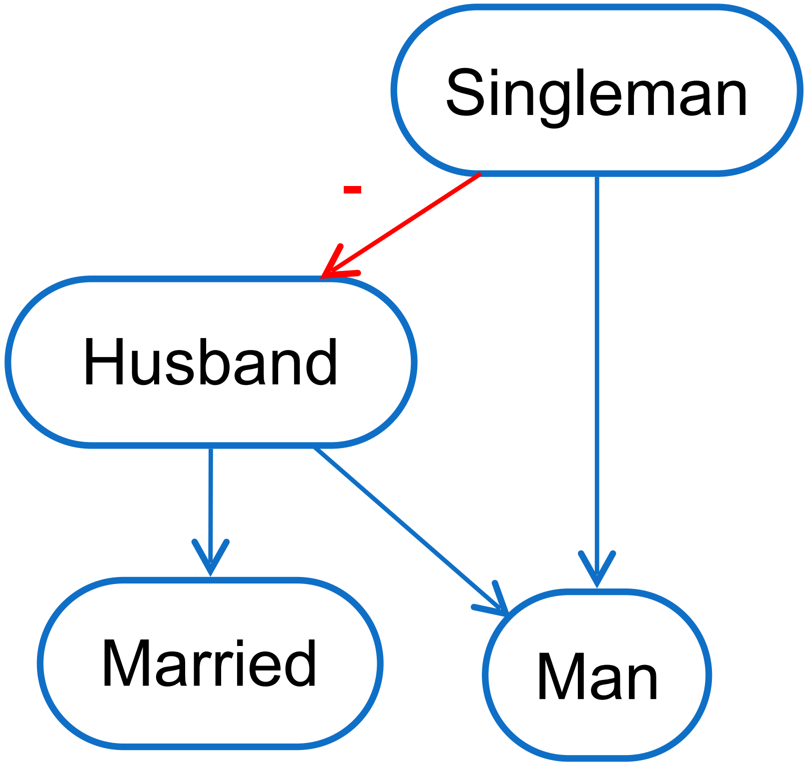
Married(x):- Person(id, x, s, 0)

Husband(x):- Man(x), Married(x)

Singleman(x):- Man(x), ¬Husband(x)

An example scenario: Partitioning

Person(pid,name,sex,married)



Find all single men

Rules

P1 = {Man(x):- Person(id, x, 'M', m)
Married(x):- Person(id, x, s, 0)
Husband(x):- Man(x), Married(x) }

P2 = {Singleman(x):- Man(x), ¬Husband(x) }

An example scenario: Evaluation

Person(pid,name,sex,married)

Find all single men

Facts = tuples in the database

```
Person(1,'Tom','M',0).
Person(2,'Alex','M',1).
Person(3,'Joe','F',1).
Person(4,'Mary','F',0).
Person(5,'John','M',0).
```

Rules

P1 = {Man(x):- Person(id, x, 'M', m)

Married(x):- Person(id, x, s, 1)

Husband(x):- Man(x), Married(x) }

P2 = {Singleman(x):- Man(x), ¬Husband(x) }

Evaluate P1: `Man('Tom'), Man('Alex'), Man('John'),`
`Married('Alex'), Married('Joe'), Husband('Alex')`

Evaluate P2: `Singleman('Tom'), Singleman('John')`

An example scenario: A model

Person(pid,name,sex,married)

Find all single men

Facts = tuples in the database

```
Person(1,'Tom','M',0).
Person(2,'Alex','M',1).
Person(3,'Joe','F',1).
Person(4,'Mary','F',0).
Person(5,'John','M',0).
```

Rules

P1 = {Man(x):- Person(id, x, 'M', m)

Married(x):- Person(id, x, s, 1)

Husband(x):- Man(x), Married(x) }

P2 = {Singleman(x):- Man(x), ¬Husband(x) }

Man('Tom'), Man('Alex'), Man('John'), Married('Alex'), Married('Joe') , Husband('Alex'), Singleman('Tom'), Singleman('John')