## Announcements

- **Individual Assignment #1:**
  - Due Friday.
- **This week's schedule:**
  - **Lectures**
    - Mon, 3:10pm
    - **Wed 9am (discussion slot)**
    - Wed 3:10pm
  - **Discussion**
    - **Fri 3:10pm (lecture slot)**



## Datalog: Safety

- **Safety**:
  - every variable **occurs positively** in the body! (why?)
    - … more precisely, in a positive relational atom.
- In particular,
  - every head variable occurs positively in the body
  - every negated (body) variable occurs positively in the body

↓ the rule is then range-restricted (safe)



Reminder from last week: rules must be safe!

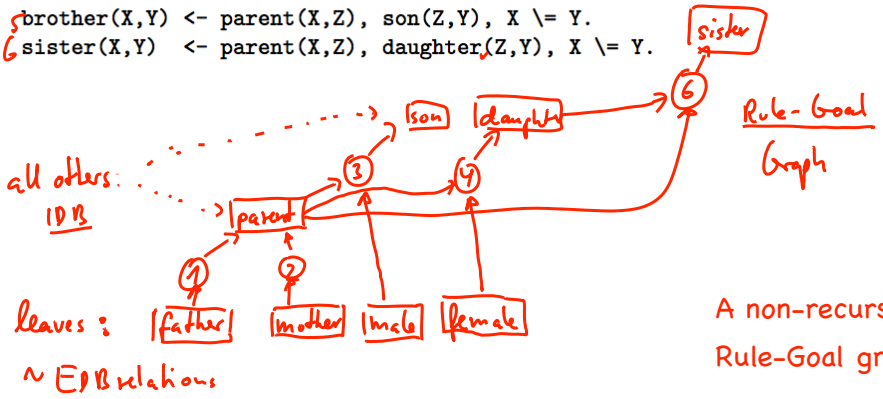
(Safe rules can be translated into RA queries; possibly with loops, in case of recursive rules.)

## Living in the family: Rule-Goal Graph

```
1 parent(X,Y) <- father(X,Y).
2 parent(X,Y) <- mother(X,Y).

3 son(X,Y)      <- parent(Y,X), male(Y).
4 daughter(X,Y) <- parent(Y,X), female(Y).

5 brother(X,Y) <- parent(X,Z), son(Z,Y), X \= Y.
6 sister(X,Y)  <- parent(X,Z), daughter(Z,Y), X \= Y.
```




A non-recursive Rule-Goal graph

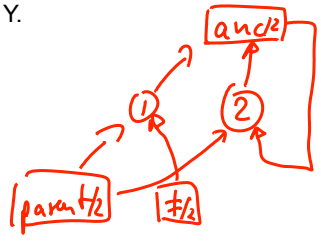


## Rule-Goal Graph (with recursion)

```
grandparent(X, Y) :-
      parent(X, Z), parent(Z, Y).
father(X, Y) :-
      parent(X, Y), male(X).
mother(X, Y) :-
      parent(X, Y), female(X).
brother(X, Y) :-
      parent(P, X), parent(P, Y), male(X), X != Y.
sister(X, Y)  :-
       parent(P, X), parent(P, Y), female(X), X != Y.

1 ancestor(X, Y) :- parent(X, Y), X≠Y.
2 ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

anc := parent
Repeat
  anc := parent ⋈ anc
         $2=$1
Until no change



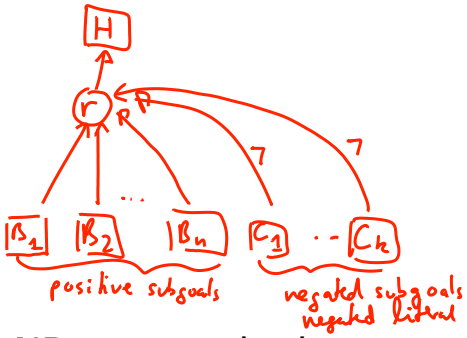

Bottom-up Fixpoint Evaluation



**A recursive Rule-Goal graph.**

## Rule Goal Graphs

- For each rule
  - (r) H ← B1, B2, …, Bn, ¬C1, … ¬Ck
- … create the following edges:




Def Datalog program P is recursive, if RG-graph(P) has a cycle; else non-recursive

- NB: negated edges; recursion

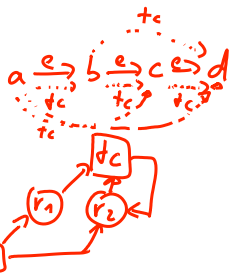
Note: The "goal nodes" in the graph are identified by the predicate name (the arity is assumed to be part of the name).



## Fixpoint Semantics: Bottom-up Evaluation

- Given a database instance I, by Tp(I) we mean the *immediate consequences* of evaluating the rules of P on I.
- We get an increasing sequence
  - $I_0$, $I_1$, $I_2$, $I_3$, …  I_{n+1} := I_n U Tp(I_{n})
  - … until we have that $I_k = I_{k+1}$
  - ➔Fixpoint is the desired answer
- Optimizations:
  - Evaluate rules bottom-up, taking into account the rule-goal graph
  - Semi-naïve evaluation
  - Magic Sets

## More on Datalog Evaluation



```
e(a,b). e(b,c). e(c,d).        % FACTS (EDB-relation: e/2)

tc(X,Y) :- e(X,Y).             % RULES (IDB-relation: tc/2)
tc(X,Y) :- e(X,Z), tc(Z,Y).
```

Plan: 1. exec $r_1$ (completely) 2. Repeat exec $r_2$ until no new tc atoms

| $n$ | $I_n$ |
|---|---|
| 0 | $\emptyset$ |
| 1 | $\{\mathbf{e(a,b)}, \mathbf{e(b,c)}, \mathbf{e(c,d)}\}$ ← facts (EDB) – rule 0 |
| 2 | $\{e(a,b), e(b,c), e(c,d)\} \cup \{\mathbf{tc(a,b)}, \mathbf{tc(b,c)}, \mathbf{tc(c,d)}\}$ ← $r_2$ copy e/2 into tc/2 |
| 3 | $\{e(a,b), e(b,c), e(c,d)\} \cup \{tc(a,b), tc(b,c), tc(c,d), \mathbf{tc(a,c)}, \mathbf{tc(b,d)}\}$ $r_2$ |
| 4 | $\{e(a,b), e(b,c), e(c,d)\} \cup \{tc(a,b), tc(b,c), tc(c,d), tc(a,c), tc(b,d), \mathbf{tc(a,d)}\}$ |
| 5 | $\{\ldots\}$ = ↑ no change! |

⇒ if the longest path in e/2 has length $n$, then $O(n)$ rounds are needed!

(Exercise: how about the following rule? `tc(X,Y) ← tc(X,Z), tc(Z,Y).`)



**Note: Only O(log(n)) rounds are needed in a graph of diameter n. But the intermediate results of the join tc(X,Z), tc(Z,Y) are huge, and benchmarks have shown that this double recursive tc/2 rule is much less efficient than the single recursive (r2) above.**

## Recursion vs Aggregation, Negation

- Negation before or after recursion: OK

(1) E(X,Y) :- A(X,Y), not B(X,Y). // $E := A \setminus B$
(2) TC(X,Y) :- E(X,Y). } $TC := E^+$
(3) TC(X,Y) :- E(X,Z), TC(Z,Y). }
(4) nTC(X,Y) :- node(X), node(Y), not TC(X,Y).

node(x) ∈ E(x,_), node(x) ∈ E(_,x) → (0)

↑ complement of TC ($=E^+$) ↷ one cannot reach y from x via $E^{(+)}$

"Wrong schedule": (1) (4) {(2),(3)} until no change

"Correct schedule": (0) (1) (2) Repeat (3) until no change (4)



**If we were to apply the "wrong schedule", then we would incorrectly assume that TC/2 is empty, so the complement nTC/2 would be "too big"**

## Lowest Common Ancestor

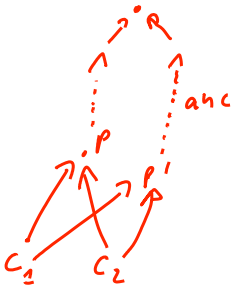
```
% anc(Node, Ancestor)
anc(C,P) :- parent(C,P).
anc(X,Y) :- anc(X,Z), anc(Z,Y).

% ca(NodeX,NodeY, CommonAncestor)
ca(X,Y,A) :-
        anc(X,A),
        anc(Y,A),
        X≠Y.

not_lca(X,Y,P) :-
        pc(P,C),
        ca(X,Y,P),
        ca(X,Y,C).

lca(X,Y,L) :-
        ca(X,Y,L),
        not not_lca(X,Y,L).
```

anc = parent+






A common ancestor P is "out", if there is a lower ("better") common ancestor C.

A common ancestor L of X and Y,



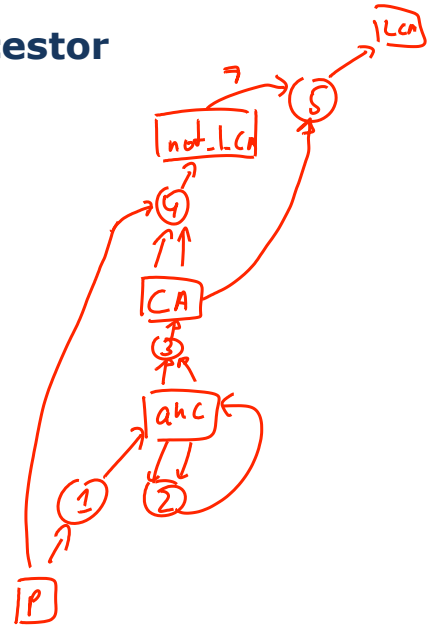
## Lowest Common Ancestor

```
  % anc(Node, Ancestor)
1 anc(C,P) :- parent(C,P).
2 anc(X,Y) :- anc(X,Z), anc(Z,Y).

  % ca(NodeX,NodeY, CommonAncestor)
3 ca(X,Y,A) :-
        anc(X,A),
        anc(Y,A).

4 not_lca(X,Y,P) :-
        pc(P,C),
        ca(X,Y,P),
        ca(X,Y,C).

5 lca(X,Y,L) :-
        ca(X,Y,L),
        not not_lca(X,Y,L).
```






Notes: The "rule schedule" is obtained by "firing" (evaluating) the rules according to the order given by the Rule-Goal graph: 1; 2+; 3; 4; 5. In particular, we can NOT evaluate rule 5 (with negation) before we have computed not_lca/3 completely, which in turn requires to compute ca/3 first via 3, which in turn requires computation of anc/2 via repeated evaluation of rule 2.

## Recursion vs Aggregation, Negation

- Rule-goal graph has **no negative** cycles ➔
  - Can be "stratified" into layers (strata)
  - Evaluate lower strata, then move to higher ones
  - All recursion/loops are monotone
- But recursion "through negation" (or "through aggregation") is problematic!
  - Rule-goal graph has **negative cycles**
  - p(X) :- q(X), not p(X) … madness …
    - What does this rule even mean? If p(X) isn't true, then it is true?
  - win(X) :- move(X,Y), not win(Y) … (sanity:)
    - On the other hand: this rule makes some sense! Computes whether X is won (or lost/drawn) in a game defined by move(X,Y)