

Stratified Negation

- Negation wrapped inside a recursion makes no sense.
- Even when negation and recursion are separated, there can be ambiguity about what the rules mean, and some one meaning must be selected.
- *Stratified negation* is an additional restraint on recursive rules (like safety) that solves both problems:
 1. It rules out negation wrapped in recursion.
 2. When negation is separate from recursion, it yields the intuitively correct meaning of rules.
- Stratification recently adopted in the SQL3 standard for recursive SQL.

Problem with Recursive Negation

Consider:

$$P(x) \leftarrow Q(x) \text{ AND NOT } P(x)$$

- $Q = \text{EDB} = \{1, 2\}$.
- Compute IDB P iteratively?
 - ❖ Initially, $P = \emptyset$.
 - ❖ Round 1: $P = \{1, 2\}$.
 - ❖ Round 2: $P = \emptyset$, etc., etc.

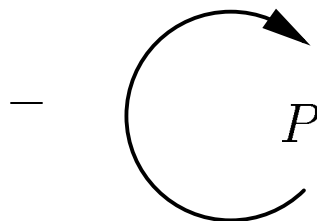
Strata

Intuitively: stratum of an IDB predicate = maximum number of negations you can pass through on the way to an EDB predicate.

- Must not be ∞ in “stratified” rules.
- Define *stratum graph*:
 - ❖ Nodes = IDB predicates.
 - ❖ Arc $P \rightarrow Q$ if Q appears in the body of a rule with head P .
 - ❖ Label that arc – if Q is in a negated subgoal.

Example

$P(x) \leftarrow Q(x) \text{ AND NOT } P(x)$

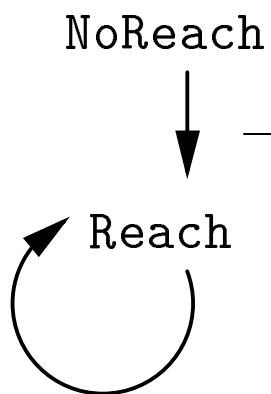


Example

$\text{Reach}(x) \leftarrow \text{Source}(x)$

$\text{Reach}(x) \leftarrow \text{Reach}(y) \text{ AND } \text{Arc}(y, x)$

$\text{NoReach}(x) \leftarrow \text{Target}(x)$
 $\text{AND NOT Reach}(x)$



Computing Strata

Stratum of an IDB predicate A = maximum number of $-$ arcs on any path from A in the stratum graph.

Examples

- For first example, stratum of P is ∞ .
- For second example, stratum of `Reach` is 0; stratum of `NoReach` is 1.

Stratified Negation

A Datalog program with recursion and negation is *stratified* if every IDB predicate has a finite stratum.

Stratified Model

If a Datalog program is stratified, we can compute the relations for the IDB predicates lowest-stratum-first.

Example

$\text{Reach}(x) \leftarrow \text{Source}(x)$

$\text{Reach}(x) \leftarrow \text{Reach}(y) \text{ AND } \text{Arc}(y, x)$

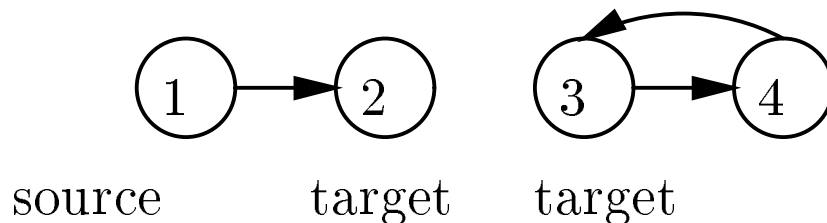
$\text{NoReach}(x) \leftarrow \text{Target}(x)$
 $\text{AND NOT Reach}(x)$

- EDB:

- ❖ $\text{Source} = \{1\}$.

- ❖ $\text{Arc} = \{(1, 2), (3, 4), (4, 3)\}$.

- ❖ $\text{Target} = \{2, 3\}$.



- First compute $\text{Reach} = \{1, 2\}$ (stratum 0).
- Next compute $\text{NoReach} = \{3\}$.

Is the Stratified Solution “Obvious”?

Not really.

- There is another model that makes the rules true no matter what values we substitute for the variables.
 - ❖ $\text{Reach} = \{1, 2, 3, 4\}$.
 - ❖ $\text{NoReach} = \emptyset$.
- Remember: the only way to make a Datalog rule false is to find values for the variables that make the body true and the head false.
 - ❖ For this model, the heads of the rules for Reach are true for all values, and in the rule for NoReach the subgoal $\text{NOT Reach}(x)$ assures that the body cannot be true.

SQL3 Recursion

WITH

stuff that looks like Datalog rules
an SQL query about EDB, IDB

- Rule =
[RECURSIVE] $R(<\text{arguments}>)$ AS
SQL query

Example

Find Sally's cousins, using EDB Par(child, parent).

WITH

Sib(x,y) AS

SELECT p1.child, p2.child

FROM Par p1, Par p2

WHERE p1.parent = p2.parent

AND p1.child <> p2.child,

RECURSIVE Cousin(x,y) AS

Sib

UNION

(SELECT p1.child, p2.child

FROM Par p1, Par p2, Cousin

WHERE p1.parent = Cousin.x

AND p2.parent = Cousin.y

)

SELECT y

FROM Cousin

WHERE x = 'Sally';

Plan for Describing Legal SQL3 recursion

1. Define “monotonicity,” a property that generalizes “stratification.”
2. Generalize stratum graph to apply to SQL queries instead of Datalog rules.
 - ❖ (Non)monotonicity replaces NOT in subgoals.
3. Define semantically correct SQL3 recursions in terms of stratum graph.

Monotonicity

If relation P is a function of relation Q (and perhaps other things), we say P is *monotone* in Q if adding tuples to Q cannot cause any tuple of P to be deleted.

Monotonicity Example

In addition to certain negations, an aggregation can cause nonmonotonicity.

```
Sells(bar, beer, price)
```

```
SELECT AVG(price)
FROM Sells
WHERE bar = 'Joe' 's Bar';
```

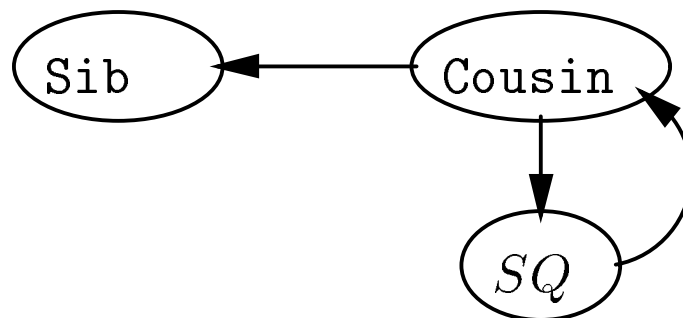
- Adding to `Sells` a tuple that gives a new beer Joe sells will usually change the average price of beer at Joe's.
- Thus, the former result, which might be a single tuple like (2.78) becomes another single tuple like (2.81), and the old tuple is lost.

Generalizing Stratum Graph to SQL

- Node for each relation defined by a “rule.”
- Node for each subquery in the “body” of a rule.
- Arc $P \rightarrow Q$ if
 - a) P is “head” of a rule, and Q is a relation appearing in the FROM list of the rule (not in the FROM list of a subquery).
 - b) P is head of a rule, and Q is a subquery directly used in that rule (not nested within some larger subquery).
 - c) P is a subquery, and Q is a relation or subquery used directly within P .
- Label the arc — if P is *not* monotone in Q .
- Requirement for legal SQL3 recursion: finite strata only.

Example

For the Sib/Cousin example, there are three nodes: Sib, Cousin, and SQ (the second term of the union in the rule for Cousin).



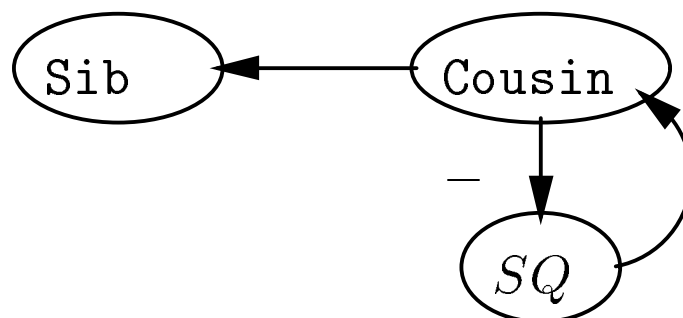
- No nonmonotonicity, hence legal.

A Nonmonotonic Example

Change the UNION to EXCEPT in the rule for Cousin.

```
RECURSIVE Cousin(x,y) AS
  Sib
  EXCEPT
  (SELECT p1.child, p2.child
   FROM Par p1, Par p2, Cousin
   WHERE p1.parent = Cousin.x
        AND p2.parent = Cousin.y
  )
```

- Now, Adding to the result of the subquery can delete Cousin facts; i.e., Cousin is nonmonotone in SG .



- Infinite number of $-$'s in cycle, so illegal in SQL3.

Another Example: NOT Doesn't Mean Nonmonotone

Leave `Cousin` as it was, but negate one of the conditions in the where-clause.

```
RECURSIVE Cousin(x,y) AS
  Sib
  UNION
  (SELECT p1.child, p2.child
   FROM Par p1, Par p2, Cousin
   WHERE p1.parent = Cousin.x
        AND NOT (p2.parent = Cousin.y)
  )
```

- You might think that SG depends negatively on `Cousin`, but it doesn't.
 - ❖ If I add a new tuple to `Cousin`, all the old tuples still exist and yield whatever tuples in SG they used to yield.
 - ❖ In addition, the new `Cousin` tuple might combine with old $p1$ and $p2$ tuples to yield something new.