

# Datalog

Logical Rules

Recursion

SQL-99 Recursion

# Logic As a Query Language

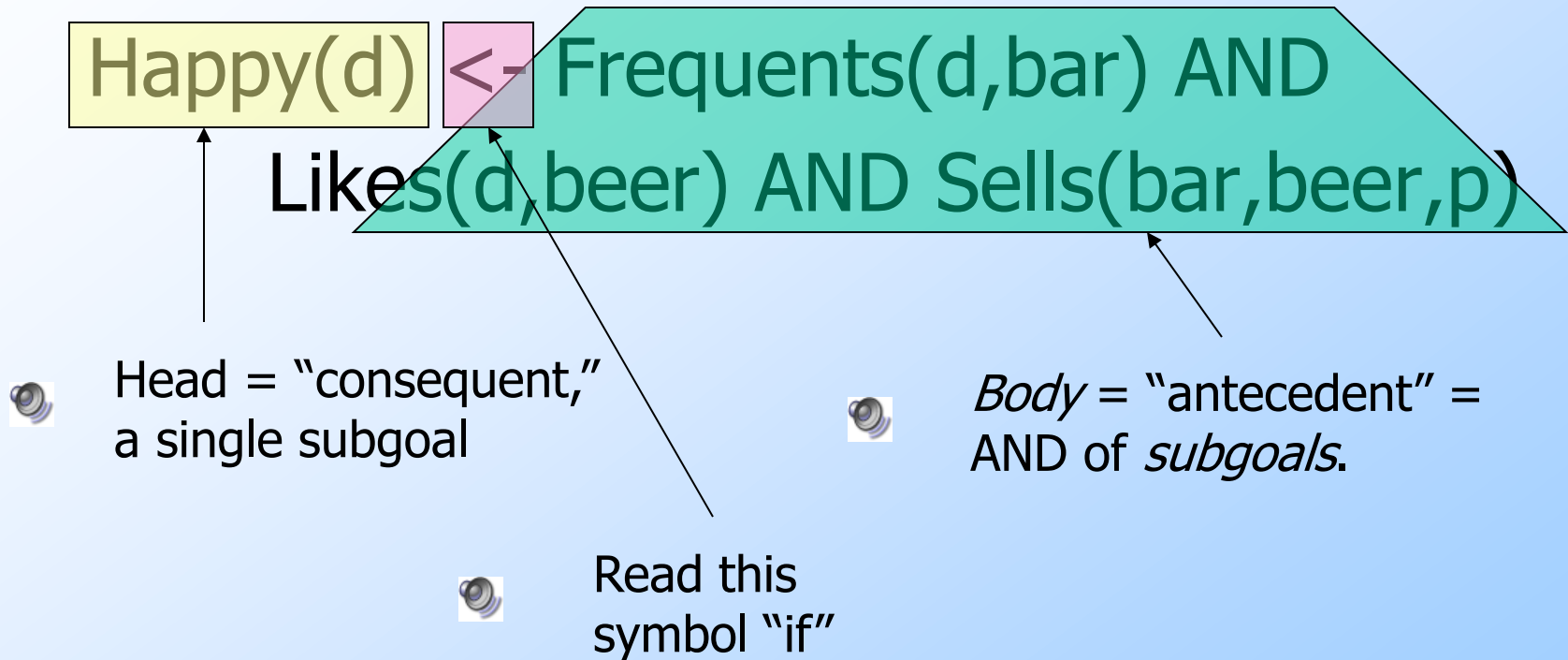
- ◆ If-then logical rules have been used in many systems.
  - ◆ Most important today: EII (Enterprise Information Integration).
- ◆ Nonrecursive rules are equivalent to the core relational algebra.
- ◆ Recursive rules extend relational algebra --- have been used to add recursion to SQL-99.

# A Logical Rule



- ◆ Our first example of a rule uses the relations `Frequents(drinker,bar)`, `Likes(drinker,beer)`, and `Sells(bar,beer,price)`.
- ◆ The rule is a query asking for “happy” drinkers --- those that frequent a bar that serves a beer that they like.

# Anatomy of a Rule



# Subgoals Are Atoms



- ◆ An *atom* is a *predicate*, or relation name with variables or constants as arguments.
- ◆ The head is an atom; the body is the AND of one or more atoms.
- ◆ Convention: Predicates begin with a capital, variables begin with lower-case.

# Example: Atom



Sells(bar, beer, p)

The predicate  
= name of a  
relation



Arguments are  
variables

# Interpreting Rules




- ◆ A variable appearing in the head is called *distinguished*; otherwise it is *nondistinguished*.
- ◆ Rule meaning: The head is true of the distinguished variables if there exist values of the nondistinguished variables that make all subgoals of the body true.

# Example: Interpretation



Happy(**d**) <- Frequents(**d**,**bar**) AND  
Likes(**d**,**beer**) AND Sells(**bar**,**beer**,**p**)

 Distinguished  
variable

 Nondistinguished  
variables

Interpretation: drinker *d* is happy if there exist a bar, a beer, and a price *p* such that *d* frequents the bar, likes the beer, and the bar sells the beer at price *p*.



# Arithmetic Subgoals



- ◆ In addition to relations as predicates, a predicate for a subgoal of the body can be an arithmetic comparison.
  - ◆ We write such subgoals in the usual way, e.g.:  $x < y$ .

# Example: Arithmetic



- ◆ A beer is “cheap” if there are at least two bars that sell it for under \$2.

Cheap(beer) <- Sells(bar1,beer,p1) AND  
Sells(bar2,beer,p2) AND p1 < 2.00  
AND p2 < 2.00 AND bar1 <> bar2

# Negated Subgoals



- ◆ We may put NOT in front of a subgoal, to negate its meaning.
- ◆ Example: Think of  $\text{Arc}(a,b)$  as arcs in a graph.
  - ◆  $S(x,y)$  says the graph is not transitive from  $x$  to  $y$ ; i.e., there is a path of length 2 from  $x$  to  $y$ , but no arc from  $x$  to  $y$ .

$S(x,y) \leftarrow \text{Arc}(x,z) \text{ AND } \text{Arc}(z,y)$   
 $\text{AND NOT } \text{Arc}(x,y)$

# Safe Rules



- ◆ A rule is *safe* if:
  1. Each distinguished variable,
  2. Each variable in an arithmetic subgoal,
  3. Each variable in a negated subgoal,  
also appears in a nonnegated,  
relational subgoal.
- ◆ We allow only safe rules.

# Example: Unsafe Rules



- ◆ Each of the following is unsafe and not allowed:
  1.  $S(x) \leftarrow R(y)$
  2.  $S(x) \leftarrow R(y) \text{ AND NOT } R(x)$
  3.  $S(x) \leftarrow R(y) \text{ AND } x < y$
- ◆ In each case, an infinity of  $x$ 's can satisfy the rule, even if  $R$  is a finite relation.

# Algorithms for Applying Rules



## ◆ Two approaches:

- 1. Variable-based* : Consider all possible assignments to the variables of the body. If the assignment makes the body true, add that tuple for the head to the result.
- 2. Tuple-based* : Consider all assignments of tuples from the non-negated, relational subgoals. If the body becomes true, add the head's tuple to the result.

# Example: Variable-Based --- 1

$S(x,y) \leftarrow \text{Arc}(x,z) \text{ AND } \text{Arc}(z,y)$   
 $\text{AND NOT } \text{Arc}(x,y)$

- ◆  $\text{Arc}(1,2)$  and  $\text{Arc}(2,3)$  are the only tuples in the Arc relation.
- ◆ Only assignments to make the first subgoal  $\text{Arc}(x,z)$  true are:
  1.  $x = 1; z = 2$
  2.  $x = 2; z = 3$

# Example: Variable-Based; $x=1, z=2$



$S(x,y) \leftarrow \text{Arc}(x,z) \text{ AND } \text{Arc}(z,y) \text{ AND NOT } \text{Arc}(x,y)$

1 3

1 2

2 3

1 3

3 is the only value of  $y$  that makes all three subgoals true.



Makes  $S(1,3)$  a tuple of the answer





# Example: Variable-Based; $x=2, z=3$

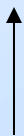


$S(x,y) \leftarrow \text{Arc}(x,z) \text{ AND } \text{Arc}(z,y) \text{ AND NOT } \text{Arc}(x,y)$

2                      2 3                      3                      2



Thus, no contribution  
to the head tuples;  
 $S = \{(1,3)\}$



No value of  $y$   
makes  $\text{Arc}(3,y)$   
true.

# Tuple-Based Assignment




- ◆ Start with the non-negated, relational subgoals only.
- ◆ Consider all assignments of tuples to these subgoals.
  - ◆ Choose tuples only from the corresponding relations.
- ◆ If the assigned tuples give a consistent value to all variables and make the other subgoals true, add the head tuple to the result.

# Example: Tuple-Based



$S(x,y) \leftarrow \text{Arc}(x,z) \text{ AND } \text{Arc}(z,y) \text{ AND NOT } \text{Arc}(x,y)$   
 $\text{Arc}(1,2), \text{Arc}(2,3)$

- ◆ Four possible assignments to first two subgoals:



Arc(x,z)	Arc(z,y)
(1,2)	(1,2)
(1,2)	(2,3)
(2,3)	(1,2)
(2,3)	(2,3)

Only assignment with consistent z-value. Since it also makes NOT Arc(x,y) true, add S(1,3) to result.

# Datalog Programs



- ◆ A *Datalog program* is a collection of rules.
- ◆ In a program, predicates can be either
  1. EDB = *Extensional Database* = stored table.
  2. IDB = *Intensional Database* = relation defined by rules.
- ◆ Never both! No EDB in heads.

# 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.

# Example: Datalog Program



- ◆ Using EDB Sells(bar, beer, price) and Beers(name, manf), find the manufacturers of beers Joe doesn't sell.

```
JoeSells(b) <- Sells('Joe''s Bar', b, p)
Answer(m) <- Beers(b,m)
               AND NOT JoeSells(b)
```

# Expressive Power of Datalog



- ◆ Without recursion, Datalog can express all and only the queries of core relational algebra.
  - ◆ The same as SQL select-from-where, without aggregation and grouping.
- ◆ But with recursion, Datalog can express more than these languages.
- ◆ Yet still not Turing-complete.

# Recursive Example

- ◆ EDB:  $\text{Par}(c,p) = p$  is a parent of  $c$ .
- ◆ Generalized cousins: people with common ancestors one or more generations back:

$\text{Sib}(x,y) \leftarrow \text{Par}(x,p) \text{ AND } \text{Par}(y,p) \text{ AND } x \neq y$

$\text{Cousin}(x,y) \leftarrow \text{Sib}(x,y)$

$\text{Cousin}(x,y) \leftarrow \text{Par}(x,xp) \text{ AND } \text{Par}(y,yp) \text{ AND } \text{Cousin}(xp,yp)$

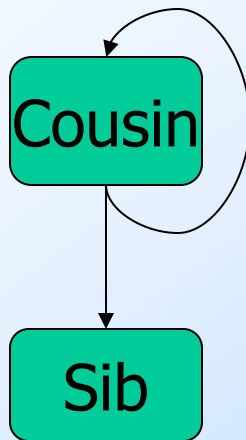


# Definition of Recursion

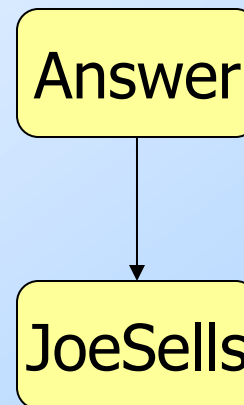


- ◆ Form a *dependency graph* whose nodes = IDB predicates.
- ◆ Arc  $X \rightarrow Y$  if and only if there is a rule with  $X$  in the head and  $Y$  in the body.
- ◆ Cycle = recursion; no cycle = no recursion.

# Example: Dependency Graphs



Recursive

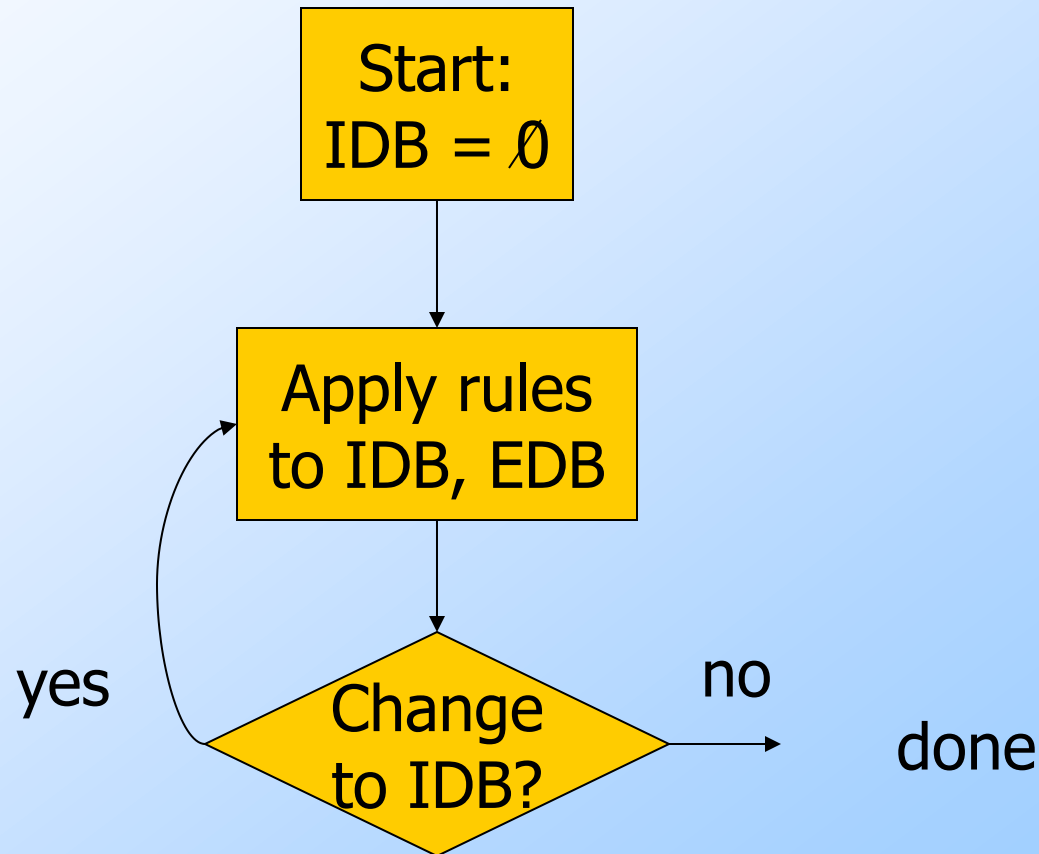


Nonrecursive

# Evaluating 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.

# The “Naïve” Evaluation Algorithm



# Example: Evaluation of Cousin

- ◆ We'll proceed in rounds to infer Sib facts (red) and Cousin facts (green).
- ◆ Remember the rules:

$\text{Sib}(x,y) \leftarrow \text{Par}(x,p) \text{ AND } \text{Par}(y,p) \text{ AND } x \neq y$

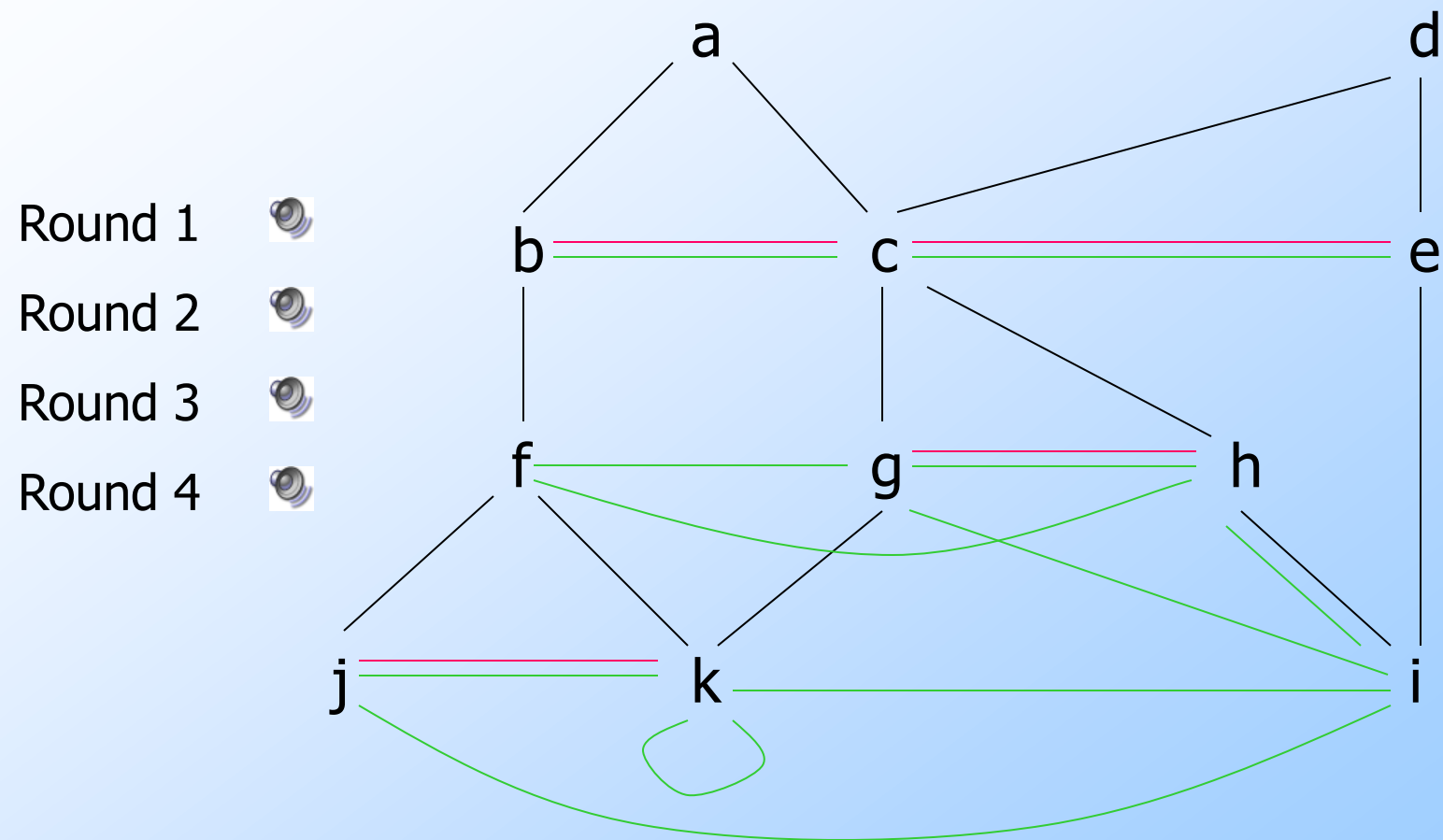
$\text{Cousin}(x,y) \leftarrow \text{Sib}(x,y)$

$\text{Cousin}(x,y) \leftarrow \text{Par}(x,xp) \text{ AND } \text{Par}(y,yp) \text{ AND } \text{Cousin}(xp,yp)$

# Seminaive Evaluation

- ◆ Since the EDB never changes, on each round we only get new IDB tuples if we use at least one IDB tuple that was obtained on the previous round.
- ◆ Saves work; lets us avoid rediscovering *most* known facts.
  - ◆ A fact could still be derived in a second way.

# Par Data: Parent Above Child



# Recursion Plus Negation



- ◆ “Naïve” evaluation doesn’t work when there are negated subgoals.
- ◆ In fact, negation wrapped in a recursion makes no sense in general.
- ◆ Even when recursion and negation are separate, we can have ambiguity about the correct IDB relations.



# 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.

# Problematic Recursive Negation



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

EDB:  $Q(1), Q(2)$

Initial:  $P = \{ \}$

Round 1:  $P = \{(1), (2)\}$

Round 2:  $P = \{ \}$

Round 3:  $P = \{(1), (2)\}, \text{ etc.}, \text{ etc. } \dots$

# Strata

- ◆ Intuitively, the *stratum* of an IDB predicate  $P$  is the maximum number of negations that can be applied to an IDB predicate used in evaluating  $P$ .
- ◆ Stratified negation = “finite strata.”
- ◆ Notice in  $P(x) \leftarrow Q(x) \text{ AND NOT } P(x)$ , we can negate  $P$  an infinite number of times deriving  $P(x)$ .

# Stratum Graph



- ◆ To formalize strata use the *stratum graph* :
  - ◆ Nodes = IDB predicates.
  - ◆ Arc  $A \rightarrow B$  if predicate  $A$  depends on  $B$ .
  - ◆ Label this arc “-” if the  $B$  subgoal is negated.

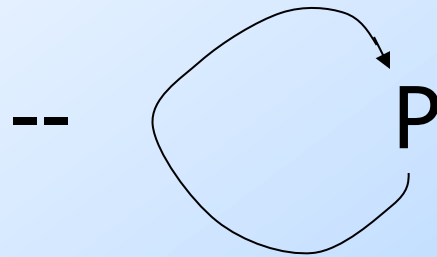
# Stratified Negation Definition

- ◆ The *stratum* of a node (predicate) is the maximum number of – arcs on a path leading from that node.
- ◆ A Datalog program is *stratified* if all its IDB predicates have finite strata.

# Example



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



# Another Example



- ◆ EDB = Source(x), Target(x), Arc(x,y).
- ◆ Rules for "targets not reached from any source":

Reach(x) <- Source(x)

Reach(x) <- Reach(y) AND Arc(y,x)

NoReach(x) <- Target(x)

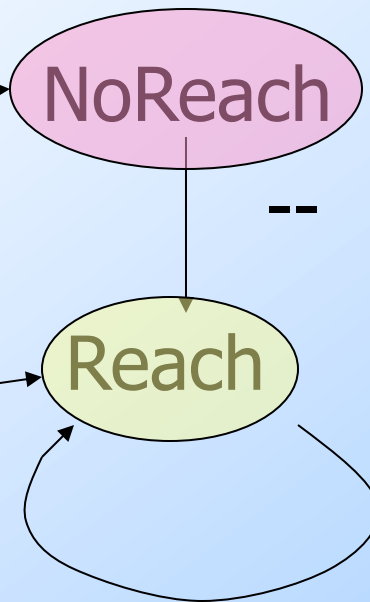
AND NOT Reach(x)

# The Stratum Graph



Stratum 1:  
 $\leq 1$  arc on  
any path out.

Stratum 0:  
No – arcs on  
any path out.





# Models

- ◆ A *model* is a choice of IDB relations that, with the given EDB relations makes all rules true regardless of what values are substituted for the variables.
  - ◆ Remember: a rule is true whenever its body is false.
  - ◆ But if the body is true, then the head must be true as well.

# Minimal Models

- ◆ When there is no negation, a Datalog program has a unique minimal model (one that does not contain any other model).
- ◆ But with negation, there can be several minimal models.
- ◆ The stratified model is the one that “makes sense.”

# The Stratified Model

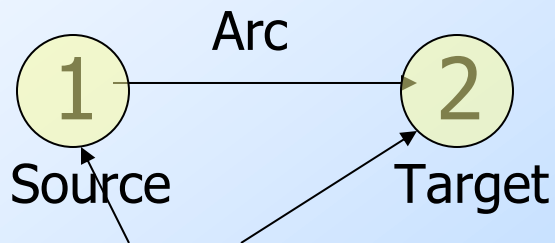
- ◆ When the Datalog program is stratified, we can evaluate IDB predicates lowest-stratum-first.
- ◆ Once evaluated, treat it as EDB for higher strata.

# Example: Multiple Models --- 1

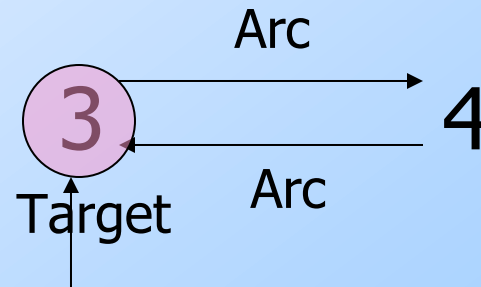
$\text{Reach}(x) \leftarrow \text{Source}(x)$

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

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



Stratum 0:  
 $\text{Reach}(1), \text{Reach}(2)$



Stratum 1:  
 $\text{NoReach}(3)$



# Example: Multiple Models --- 2

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

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

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



Another model!  $\text{Reach}(1)$ ,  $\text{Reach}(2)$ ,  
 $\text{Reach}(3)$ ,  $\text{Reach}(4)$ ;  $\text{NoReach}$  is empty.



# SQL-99 Recursion

- ◆ Datalog recursion has inspired the addition of recursion to the SQL-99 standard.
- ◆ Trickier, because SQL allows grouping-and-aggregation, which behaves like negation and requires a more complex notion of stratification.

# Form of SQL Recursive Queries

WITH

<stuff that looks like Datalog rules>

<an SQL query about EDB, IDB>

Rule =

[RECURSIVE] <name>(<arguments>)

AS <query>

# Example: SQL Recursion --- 1

- ◆ Find Sally's cousins, using SQL like the recursive Datalog example.
- ◆ Par(child,parent) is the EDB.

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;
```

Like Sib(x,y) <-  
Par(x,p) AND  
Par(y,p) AND  
x <> y





# Example: SQL Recursion --- 2



WITH ...

RECURSIVE Cousin(x,y) AS

(SELECT \* FROM Sib)

UNION

(SELECT p1.child, p2.child  
FROM Par p1, Par p2, Cousin  
WHERE p1.parent = Cousin.x AND  
p2.parent = Cousin.y);

Reflects Cousin(x,y) <-  
Sib(x,y)



Reflects  
Cousin(x,y) <-  
Par(x, xp) AND  
Par(y, yp) AND  
Cousin(xp, yp)



# Example: SQL Recursion --- 3



- ◆ With those definitions, we can add the query, which is about the “temporary view” Cousin(x,y):

```
SELECT y  
FROM Cousin  
WHERE x = 'Sally';
```

# Plan to Explain Legal SQL Recursion

1. Define “monotone,” a generalization of “stratified.”
2. Generalize stratum graph to apply to SQL.
3. Define proper SQL recursions in terms of the stratum graph.

# Monotonicity

- ◆ If relation  $P$  is a function of relation  $Q$  (and perhaps other relations), we say  $P$  is *monotone* in  $Q$  if inserting tuples into  $Q$  cannot cause any tuple to be deleted from  $P$ .
- ◆ Examples:
  - ◆  $P = Q \text{ UNION } R$ .
  - ◆  $P = \text{SELECT}_{a=10}(Q)$ .

# Example: Nonmonotonicity

- ◆ If  $\text{Sells}(\text{bar}, \text{beer}, \text{price})$  is our usual relation, then the result of the query:

```
SELECT AVG(price)
```

```
FROM Sells
```

```
WHERE bar = 'Joe''s Bar';
```

is not monotone in Sells.

- ◆ Inserting a Joe's-Bar tuple into Sells usually changes the average price and thus deletes the old average price.

# SQL Stratum Graph --- 2



## ◆ Nodes =

1. IDB relations declared in WITH clause.
2. Subqueries in the body of the “rules.”
  - ◆ Includes subqueries at any level of nesting.

# SQL Stratum Graph --- 2

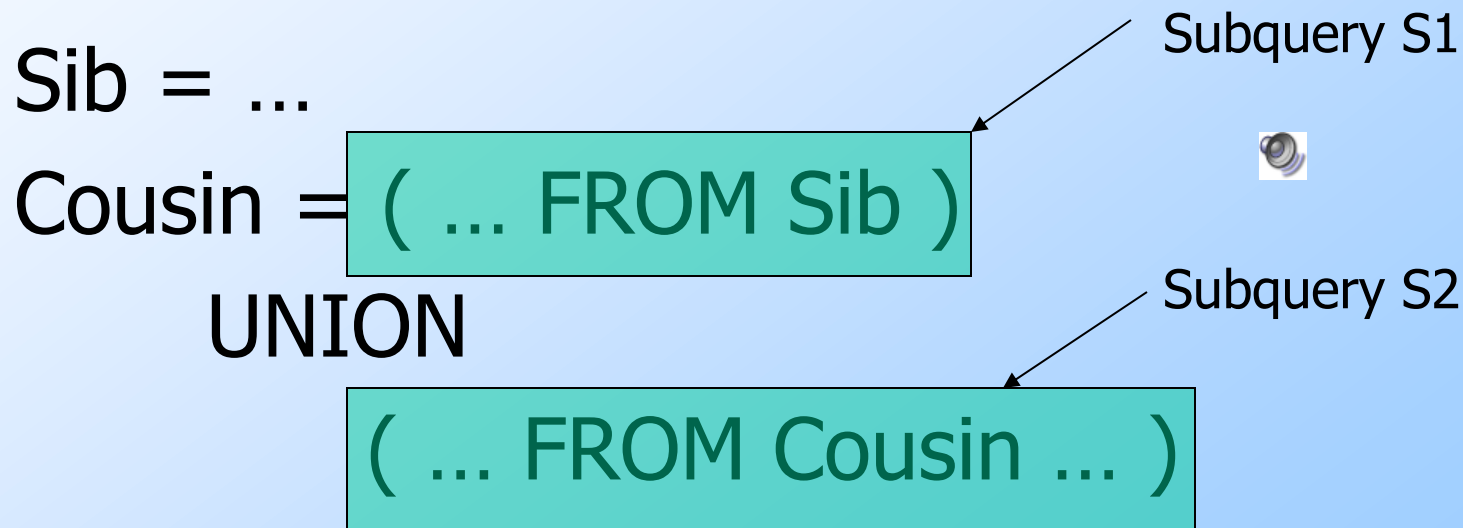


## ◆ Arcs $P \rightarrow Q$ :

1.  $P$  is a rule head and  $Q$  is a relation in the FROM list (not of a subquery).
  2.  $P$  is a rule head and  $Q$  is an immediate subquery of that rule.
  3.  $P$  is a subquery, and  $Q$  is a relation in its FROM or an immediate subquery (like 1 and 2).
- ◆ Put “—” on an arc if  $P$  is not monotone in  $Q$ .
  - ◆ Stratified SQL = finite #'s of —'s on paths.

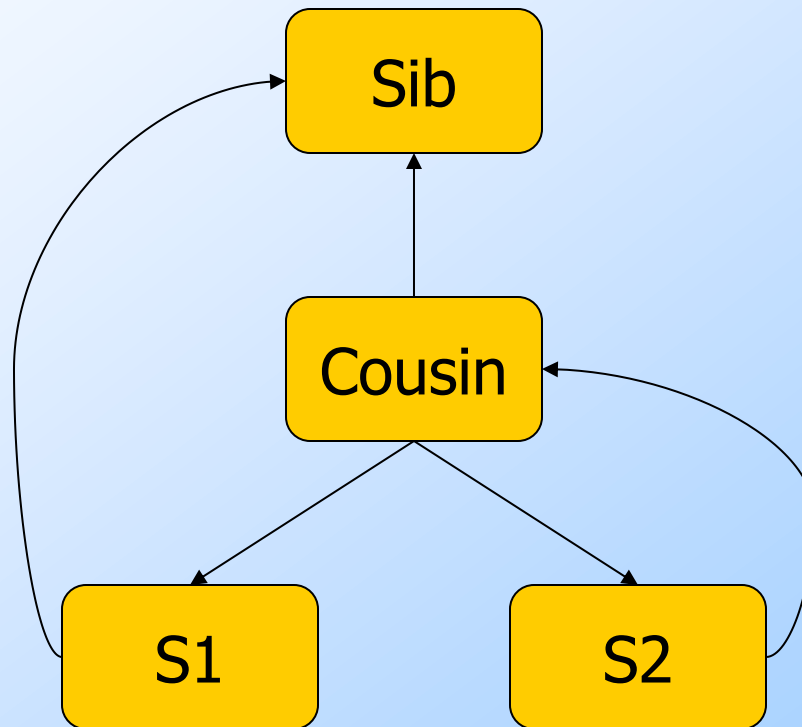
# Example: Stratum Graph

- ◆ In our Cousin example, the structure of the rules was:





# The Graph



No “—” at all,  
so surely stratified.



# Nonmonotone Example



- ◆ Change the UNION in the Cousin example to EXCEPT:

Sib = ...

Cousin = ( ... FROM Sib )

EXCEPT

( ... FROM Cousin ... )

Subquery S1

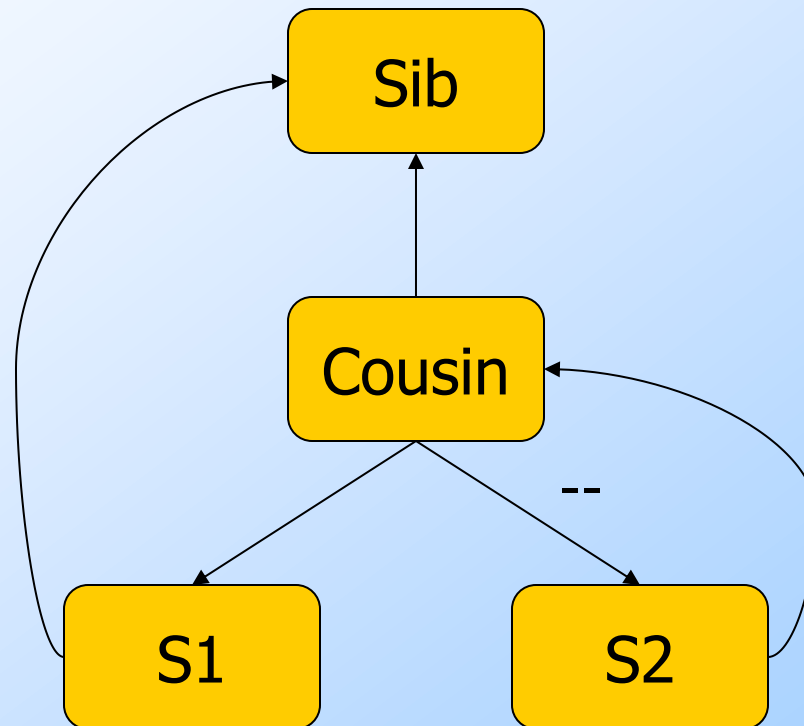
Subquery S2



Can delete a tuple from Cousin

Inserting a tuple into S2

# The Graph



An infinite number of --'s exist on cycles involving Cousin and S2.

# NOT Doesn't Mean Nonmonotone



- ◆ Not every NOT means the query is nonmonotone.
  - ◆ We need to consider each case separately.
- ◆ Example: Negating a condition in a WHERE clause just changes the selection condition.
  - ◆ But all selections are monotone.

# Example: Revised Cousin



```
RECURSIVE Cousin AS  
(SELECT * FROM Sib)  
UNION
```

Revised  
subquery S2

```
(SELECT p1.child, p2.child  
FROM Par p1, Par p2, Cousin  
WHERE p1.parent = Cousin.x AND  
NOT (p2.parent = Cousin.y)  
);
```

The only  
difference

# S2 Still Monotone in Cousin

- ◆ Intuitively, adding a tuple to Cousin cannot delete from S2.
- ◆ All former tuples in Cousin can still work with Par tuples to form S2 tuples.
- ◆ In addition, the new Cousin tuple might even join with Par tuples to add to S2.