Recap and Schedule

- Till Now:
  - Query Languages: Relational Algebra; SQL
- Today:
  - Datalog – A logical query language.
  - SQL Recursion.

Logical Query Languages

Motivation:

1. Logical rules extend more naturally to recursive queries than does relational algebra.
    - Used in SQL recursion.
2. Logical rules form the basis for many information-integration systems and applications.

Tables as Predicates

- The basic premise of logical data model is to view tables as *predicates*. E.g.,
- Table R(A,B,C) = {<1,2,4>, <1,2,3>, <1,1,1>} (i.e, a table with three tuples).
- Predicate R(x,y,z) is true/false depending on whether <x,y,z> is a tuple in R. Therefore,
  - R(1,2,4) is TRUE
  - R(1,3,3) is FALSE

Datalog Terminology

Likes(drinker, beer); Sells(bar, beer, price); Frequents(drinker, bar)

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

- Above = *rule*.
- Left side = *head*.
- Right side = **AND** of *subgoals* = *body*

- Head and subgoals consist of:
  - *Predicates:* Relation name or arithmetic predicate
  - *Arguments*: Variables or constants.
- Subgoals (not head) may optionally be negated by **NOT**.
  - "NOT X(..)" is true iff "X(..)" if false, and vice-versa.

Meaning of Rules

Head is true if:

- Some values for variables make all the subgoals true.
- Natural join of subgoals and project the head variables (for simple rules – without negations or arithmetic predicates)

Example

Previous rule equivalent to

$$\text{Happy(d)} = \pi_{\text{drinker}}(\text{Frequents} \bowtie \text{Likes} \bowtie \text{Sells})$$

Datalog Rule: AND, OR, NOT

- The "AND" in the rule is implicit. So, we just write:

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

- Multiple rules signify an "OR" or union. So,

  Good(d) <- Likes(d, "Miller")
  Good(d) <- Likes(d, "Bud")

  A drinker "d" is "Good" if he likes Bud or Miller.

- Subgoals (not head) may optionally be negated by **NOT**.
  - "NOT X(..)" is true iff "X(..)" if false, and vice-versa.

  Sad(d) <- Frequents(d,bar), NOT Sells(d, "Miller")

  - A drinker "d" is "Sad" if he frequents SOME bar that does not sell Miller.

Evaluation of (Non-Recursive) Rules

- Consider all possible assignments of values to variables.
- For each assignment:
    - If all subgoals are true, add the head to the result relation.

[Most important slide]

Evaluation: Example

S(x,y) <- R(x,z) AND R(z,y) AND NOT R(x,y)

R =

A	B
1	2
2	3

First subgoal true for:

1. $x \rightarrow 1, z \rightarrow 2$.
2. $x \rightarrow 2, z \rightarrow 3$.

Case (1): $y \rightarrow 3$ makes 2nd and 3rd subgoals true.

- Thus, add $(x,y) = (1,3)$ to relation S.

Evaluation: Example

S(x,y) <- R(x,z) AND R(z,y) AND NOT R(x,y)

R =

A	B
1	2
2	3

2. $x \rightarrow 2, z \rightarrow 3.$

Case (2): No *y* value makes the 2nd subgoal true.

- Thus, no other tuple added to *S*.

- S = {(1,3)}

Safety

Examples

- **S(x) <- R(y)**
- **S(x) <- NOT R(x)**
- **S(x) <- R(y) AND x < y**

In each case, the result is infinite. even if *R* is finite.

Safety: If *x* appears in either

1. The head,
2. A negated subgoal, or
3. An arithmetic comparison,

then *x* must also appear in a nonnegated, relational subgoal the body.

Datalog Programs

- A collection of rules is a *Datalog program*.
- Relations divide into two classes:
  - EDB = *extensional database* = relation in DB.
  - IDB = *intensional database* = relation defined by one or more rules.

- A relation must be IDB or EDB, not both.
  - Thus, EDB cannot appear as a head.

SQL to Datalog Conversion

Beers(<u>name</u>, manf); Sells(<u>bar</u>, <u>beer</u>, price)

Find the manufacturers of the beers Joe sells.

```
SELECT    manf
FROM      Beers
WHERE     name IN(SELECT     beer
                  FROM       Sells
                  WHERE      bar = 'Joe''s Bar');
```

Equivalent Datalog program

```
JoeSells(b)   <-   Sells('Joe''s Bar', b, p)
Answer(m)   <-    JoeSells(b) AND Beers(b,m)
```

EDBs: **Beers, Sells**

IDBs : **JoeSells, Answer**

Class Exercise: Graphs

- Node(v) : v is a node.
- Edge(x, y): (x,y) is an edge

- Write Datalog Programs for:
  - Find pairs of nodes connected by a path of length 2.
  - Find isolated nodes (no edges).
  - Find pair of nodes that are *at least* 4 hops apart.

Expressive Power of Datalog

- Nonrecursive Datalog = Core relational algebra.
- Datalog simulates SQL select-from-where without aggregation and grouping.
- Recursive Datalog (next) expresses queries that cannot be expressed in SQL.
- But none of these languages have full expressive power (*Turing completeness*).

Recursion

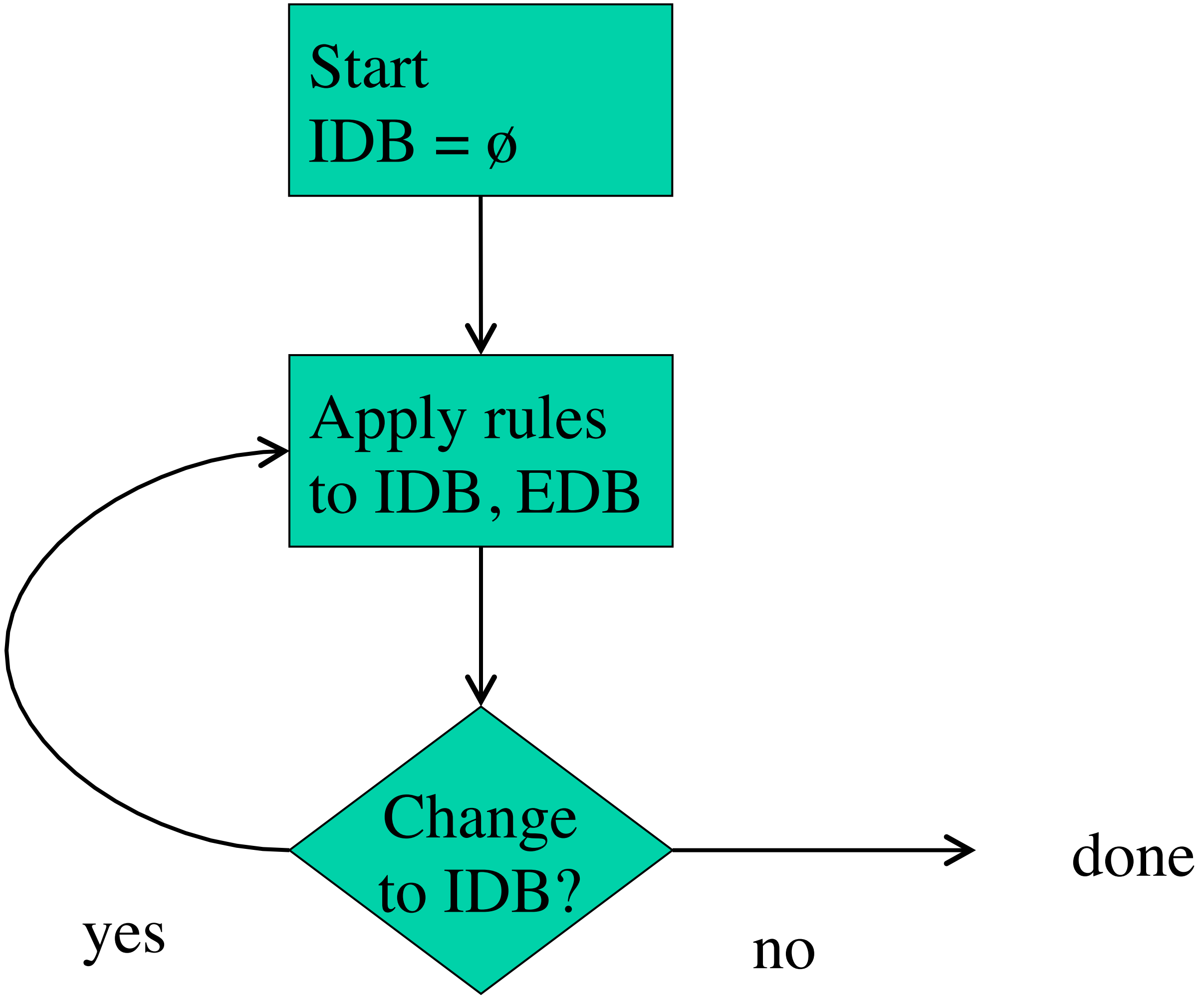
Example

```
Sib(x,y)       <- Par(x,p) AND Par(y,p) AND x <> y
Cousin(x,y)    <- Sib(x,y)
Cousin(x,y)    <- Par(x,xp), Par(y,yp), Cousin(xp,yp)
```

- IDB predicate *P* *depends* on predicate *Q* if there is a rule with *P* in the head and *Q* in a subgoal.
- Draw a graph:
  - Nodes = IDB predicates
  - Arcs = *P* → *Q* means *P* depends on *Q*.
- Cycles if and only if recursive.

Evaluation of Recursive Rules

(Assume NO NEGATIONS)



Start
IDB = ø

Apply rules
to IDB, EDB

Change
to IDB?

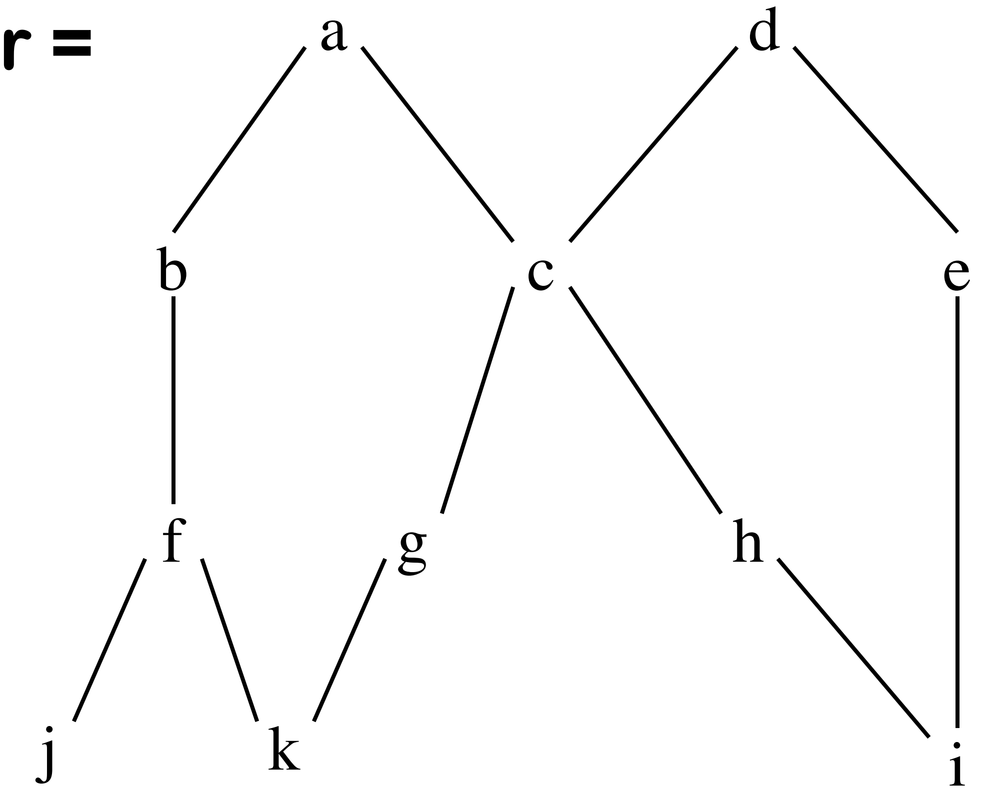
yes

no

done

Example

EDB **Par =**



Note: **Sib** and **Cousin** are symmetric

- We mention only (*x*,*y*) when both (*x*,*y*) & (*y*,*x*) are meant.

	Sib	**Cousin**
Initial	$\emptyset$	$\emptyset$
Round 1 add:	$(b,c),(c,e)$ $(g,h),(j,k)$	$\emptyset$
Round 2 add:		$(b,c),(c,e)$ $(g,h),(j,k)$
Round 3 add:		$(f,g),(f,h)$ $(g,i),(h,i)$ (i,k)
Round 4 add:		(k,k) (i,j)

Negation + Recursion: I

1. Negation “within” a recursion makes no sense.

E.g.: **P(x) <- Q(x), NOT P(x)**

- $Q = \{1,2\}$.
- Compute P iteratively?
  - Initially, $P = \varnothing$.
  - Round 1: $P = \{1,2\}$. Stopping here doesn’t seem right.
  - Round 2: $P = \varnothing$? Leads to a never-ending loop.

Negation + Recursion: II

2. However, the below is fine:

    P(x) <- Q(x), P(x), NOT R(x)

    The above is just equivalent to:

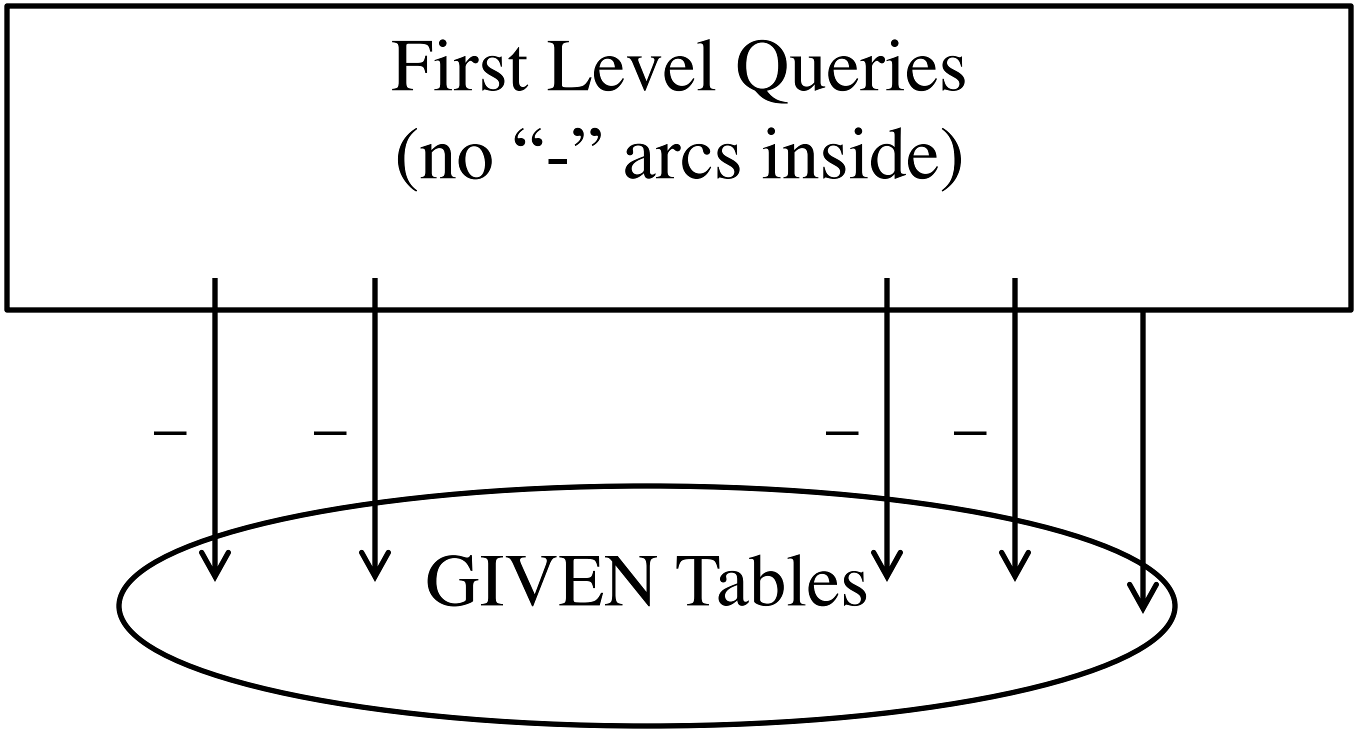
    P(x) <- P(x), T(x)

    where T = Q – R.

So, when is negation ok with recursion?

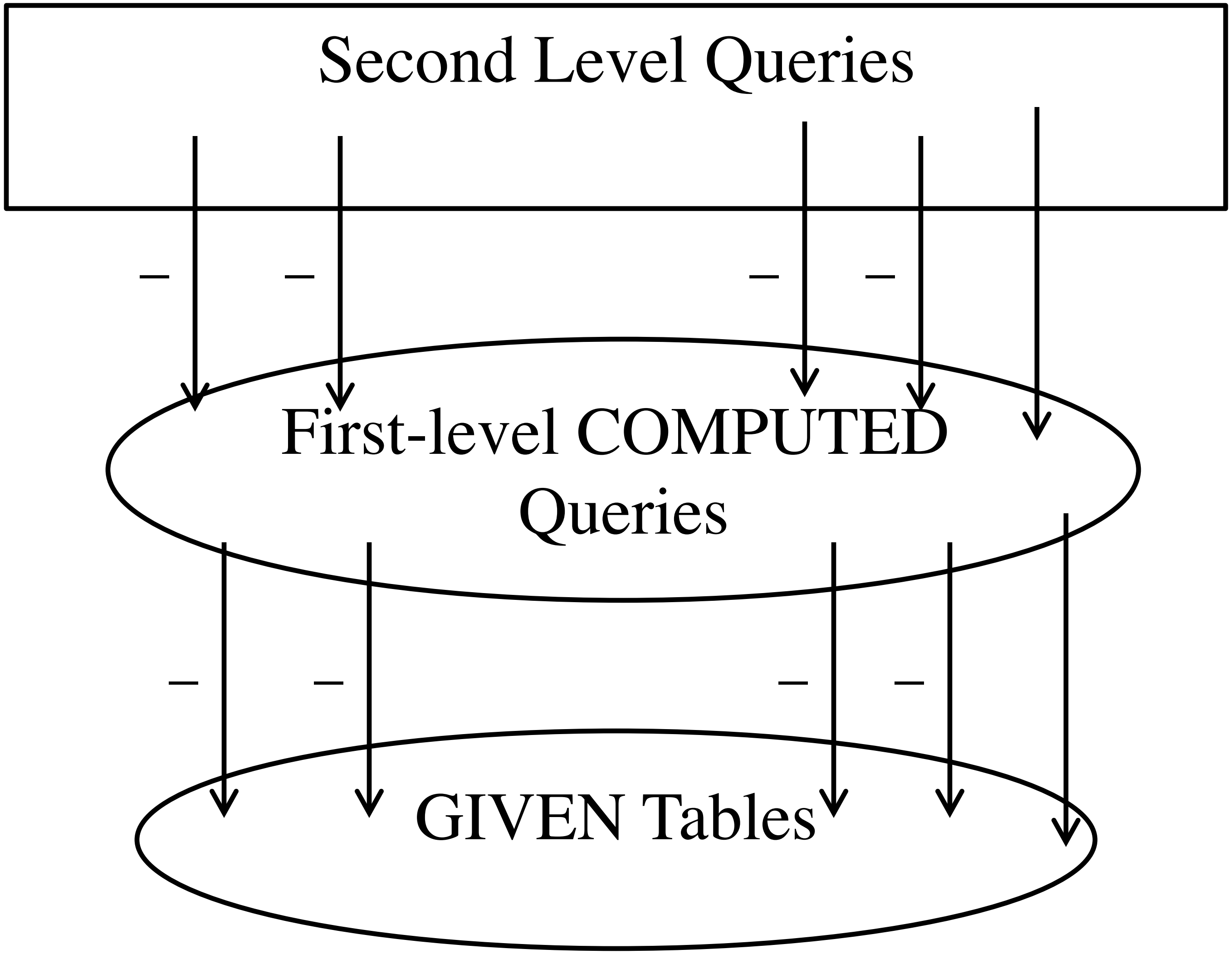
- Stratified negation. It is an additional restraint on recursive rules (like safety) that tells us what programs are valid and how to evaluate them.

Stratified Negation: Intuition



If the negation (i.e., NOT) is only over the EDBs, then the program can be evaluated using the classic approach (replacing NOT R by "R complement")

Stratified Negation: Intuition



Second Level Queries

First-level COMPUTED Queries

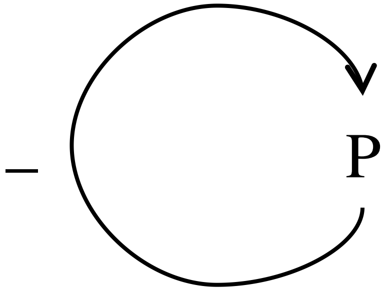
GIVEN Tables

Stratum Graph

- 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) <- Q(x) AND NOT P(x)



Computing Strata (Levels)

- *Stratum* of an IDB predicate *A* = maximum number of "–" arcs on any path from *A* in the stratum graph.
  - In Example 1, stratum of *P* is ∞.
  - In Example 2, stratum of **Reach** is 0; of **NoReach** is 1.

Stratified Negation

- A Datalog program is *stratified* if every IDB predicate has a finite stratum.

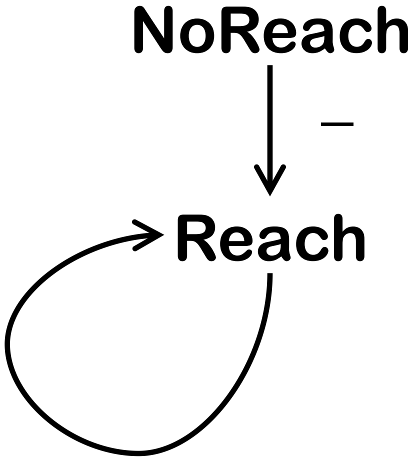
Stratified Model of Computation

- For stratified Datalog programs, we can compute the relations for the IDB predicates lowest-stratum-first.

Example

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

NoReach(x) <- Target(x) AND NOT Reach(x)



- EDB:
  - **Source** = {1}
  - **Arc** = {(1,2), (3,4), (4,3)}
  - **Target** = {2,3}

- Compute **Reach** = {1,2
- Compute **NoReach** = {3}

Datalog Recap

- Non-Recursive Datalog
  - Equivalent to SQL without aggregates.
  - Evaluate by considering all attribute-value combinations (see slide #7).
- Recursive Datalog (without NOTs)
  - Evaluate “iteratively” until no change.
- Datalog with recursion and NOTs
  - Assign “levels” to defined IDBs, and evaluate them one level at a time.

Class Exercise: Graphs

- Nodes(G, v) : Graph G has a node v
- Edge(G, x, y): Graph G has an edge (x,y)

- Write Datalog Programs for:
  - Find all nodes reachable from “A” in each graph
  - Find all “connected” graphs
  - Find distance between each pair of nodes in each graph. Distance = length of *shortest* path

SQL Recursion

SQL Recursion: 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';
```

Stratified SQL: Intuition

- The key problem with “recursion + negation” in Datalog was:
  - Negation over a *changing* (during evaluation) table can result in “invalidation of a previously valid explanation” (and hence, *deletion* of already-added tuples from IDBs) – leading to an infinite loop.
- In SQL, the set of operations that can result in such deletions – are many. We identify them using the concept of “monotonicity”.

Stratified SQL: High Level Plan

1. Define “monotonicity,” to identify operators that may result in deletions of added tuples.
2. Generalize stratum graph to apply to SQL queries.
   - Non-monotonicity replaces Datalog NOT
3. Define legal SQL recursions in terms of stratum graph.

Monotonicity Example

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.

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

- Adding a tuple to Sells may *change* a tuple in result.
- Hence, the old tuple in result may be lost.
- Thus, the above query is non-monotonic in Sells.

Monotonicity: More Examples

- Q = R – S.
  - Q is monotonic in R, but is non-monotonic in S.
- "NOT" in where clause doesn't imply non-monotonicity. E.g.,

  Select *
  From R
  Where NOT R.a = 5.

  The above query is monotonic in R.

NOT Doesn't Mean Nonmonotone

```
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))
```

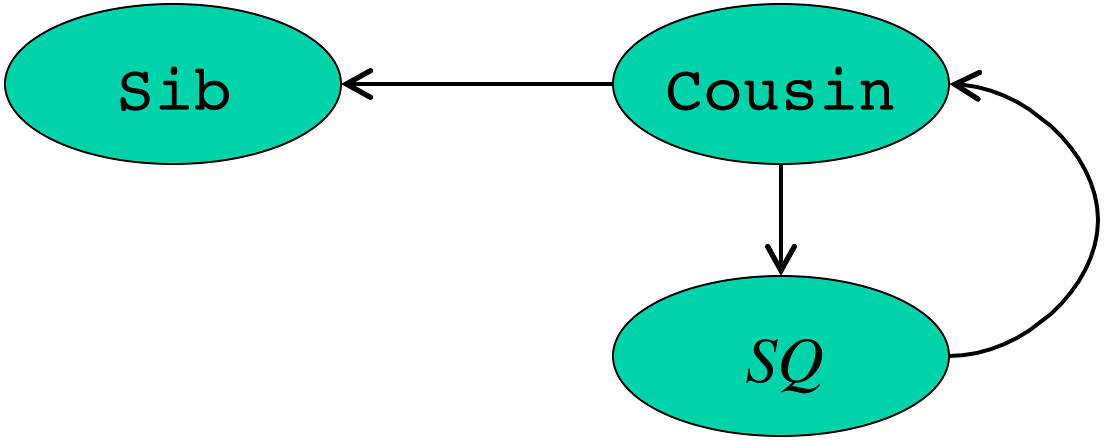
- Does *SQ* depend negatively on **Cousin**? No.
  - An added tuple to **Cousin** doesn't delete tuples from SQ.
  - May add new tuples to SQ.

Generalizing Stratum Graph to SQL

- Nodes in the stratum graph: relations, subqueries.
- Arc $P \rightarrow Q$ if P "depends" on Q.
- Label the arc – if P is *not* monotone in Q.
- Requirement for legal SQL recursion: finite strata.

Example

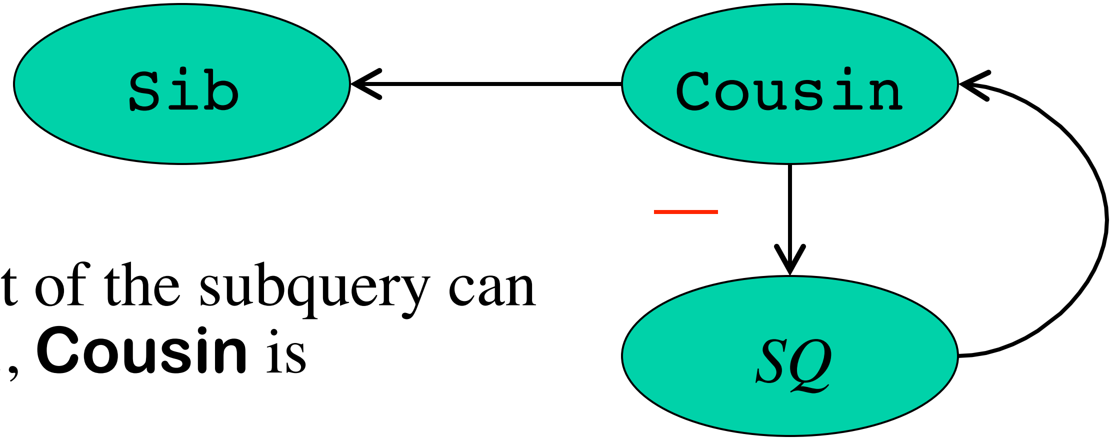
- For the Sib/Cousin example, there are three nodes: **Sib, Cousin**, and *SQ* (the second term of the union in the rule for **Cousin**).



- No nonmonotonicity, hence legal.

A Nonmonotonic Example

```
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 *SQ*.
- Infinite number of –' s in cycle, so illegal in SQL.

SQL Recursion

Syntax

WITH

<stuff that looks like Datalog rules>

<an SQL query about EDB, IDB>

- Rule =

  [**RECURSIVE**] *R*(<arguments>) **AS**

  <SQL query>