Recursion

- 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, arc from $P$ to $Q$ means $P$ depends on $Q$.
- If there is a cycle then the program is recursive.

Recursive Example

\[
\begin{align*}
\text{Sib}(x,y) & \leftarrow \text{Par}(x,p) \text{ AND } \text{Par}(y,p) \text{ AND } x \succ y \\
\text{Cousin}(x,y) & \leftarrow \text{Sib}(x,y) \\
\text{Cousin}(x,y) & \leftarrow \text{Par}(x,xp) \text{ AND } \text{Par}(y,yp) \text{ AND } \text{Cousin}(xp,yp)
\end{align*}
\]

Evaluating Recursive Rules

- Iterative fixed-point evaluation:

\[
\begin{array}{c}
\text{Start} \\
\text{IDB} = \emptyset \\
\text{Apply rules} \\
\text{to IDB, EDB} \\
\text{Changes} \\
\text{to IDB} \\
\text{Yes} \quad \text{No} \\
\text{Done}
\end{array}
\]

Example

- EDB Par =

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e} \\
\text{f} \\
\text{g} \\
\text{h} \\
\text{i} \\
\text{j} \\
\text{k}
\end{array}
\]
Iterations

<table>
<thead>
<tr>
<th>Sib</th>
<th>Cousin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>Ø</td>
</tr>
<tr>
<td>Round 1 add:</td>
<td>(b,c), (c,e)</td>
</tr>
<tr>
<td></td>
<td>(g,h), (j,k)</td>
</tr>
<tr>
<td>Round 2 add:</td>
<td>(b,c), (c,e)</td>
</tr>
<tr>
<td></td>
<td>(g,h), (j,k)</td>
</tr>
<tr>
<td>Round 3 add:</td>
<td>(f,g), (f,h), (g,i)</td>
</tr>
<tr>
<td></td>
<td>(h,i), (j,k)</td>
</tr>
<tr>
<td>Round 4 add:</td>
<td>(k,k), (i,j)</td>
</tr>
</tbody>
</table>

Negation and Recursion

- 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 one meaning must be selected.

Stratified Negation

- 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 (the stratified model).

Problem with Recursive Negation

- Consider:
  \[ P(x) \leftarrow Q(x) \land \neg P(x) \]
- \( Q = \text{EDB} = \{1,2\} \).
- 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 infinity 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

\[ P(x) \leftarrow Q(x) \land \neg P(x) \]
Another Example

- Given \( \text{Source}(\text{node}) \), \( \text{Target}(\text{node}) \), \( \text{Arc}(\text{node1}, \text{node2}) \).
- Which target nodes cannot be reached from any source node?
  \[
  \text{Reach}(x) \leftarrow \text{Source}(x)
  \]
  \[
  \text{Reach}(x) \leftarrow \text{Reach}(y) \text{ AND Arc}(y,x)
  \]
  \[
  \text{NoReach}(x) \leftarrow \text{Target}(x) \text{ AND NOT Reach}(x)
  \]

Computing Strata

- Stratum of an IDB predicate \( A = \) maximum number of negative arcs on any path from \( A \) in the stratum graph.
- Examples:
  - For first example, stratum of \( P \) is \( \infty \).
  - For second example, stratum of \( \text{Reach} \) is 0; stratum of \( \text{NoReach} \) is 1.

Stratified Negation

- A Datalog program with recursion and negation is stratified if every IDB predicate has a finite stratum.
- If a Datalog program is stratified, we can compute the relations for the IDB predicates lowest-stratum-first.
  - This is the stratified model.

Example

\[
\text{Reach}(x) \leftarrow \text{Source}(x)
\]
\[
\text{Reach}(x) \leftarrow \text{Reach}(y) \text{ AND Arc}(y,x)
\]
\[
\text{NoReach}(x) \leftarrow \text{Target}(x) \text{ AND NOT Reach}(x)
\]
- EDB:
  - \( \text{Source} = \{1\} \).
  - \( \text{Arc} = \{(1,2), (3,4), (4,3)\} \).
  - \( \text{Target} = \{2,3\} \).

Example (continued)

- First compute \( \text{Reach} = \{1,2\} \) (stratum 0).
- Next compute \( \text{NoReach} = \{3\} \).
- Is the stratified solution obvious?
- There is another model that makes the rules true no matter what values we substitute for the variables.
  - \( \text{Reach} = \{1,2,3,4\} \).
  - \( \text{NoReach} = \emptyset \).

Example (continued)

- 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 \( \text{Reach} \) are true for all values, and in the rule for \( \text{NoReach} \) the subgoal \( \text{NOT Reach}(x) \) assures that the body cannot be true.
**SQL3 Recursion**

- WITH
  - stuff that looks like Datalog rule an SQL query about EDB, IDB
- Rule =
  [RECURSIVE] R(<arguments>) AS SQL query

**Example**

- Find Sally's cousins, using 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
  (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)

  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.
   - Nonmonotonicity 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.
  SELECT AVG(price)
  FROM Sells
  WHERE bar = 'Spoon';
- Adding to Sells a tuple that gives a new beer Spoon sells will usually change the average price of beer at Spoon.
- 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
  - \( P \) 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), as argument of a UNION, etc.
  - \( P \) is head of a rule, and \( Q \) is a subquery directly used in that rule (not nested within some larger subquery).
  - \( P \) is a subquery, and \( Q \) is a relation or subquery used directly within \( P \) (analogous to (a) and (b) for rule heads).
- 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.

```sql
RECURSIVE Cousin(x,y) AS
  (SELECT * from 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.

Example (continued)

- 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.

```sql
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))
```

Example (continued)

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