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(customer,rest),
  • Likes(customer,soda), and
  • Sells(rest,soda,price).
◆ The rule is a query asking for “happy” customers --- those that frequent a rest that serves a soda that they like.

Anatomy of a Rule
Happy(d) <- Frequents(d,rest) AND Likes(d,soda) AND Sells(rest,soda,p)

Anatomy of a Rule
◆ 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

\[ Sells(\text{rest}, \text{soda}, p) \]

Interpreting Rules

- A variable appearing in the head is called **distinguished**;
- otherwise it is **nondistinguished**.

Example: Interpretation

\[ \text{Happy}(d) \leftarrow \text{Frequents}(d, \text{rest}) \text{ AND } \text{Likes}(d, \text{soda}) \text{ AND } Sells(\text{rest}, \text{soda}, p) \]

Interpretation: customer \( d \) is happy if there exist a rest, a soda, and a price \( p \) such that \( d \) frequents the rest, likes the soda, and the rest sells the soda at price \( p \).
Arithmetic sub-goals

In addition to relations as predicates, a predicate for a sub-goal of the body can be an arithmetic comparison.

- We write such sub-goals in the usual way, e.g.: \( x < y \).

Example: Arithmetic

A soda is “cheap” if there are at least two rests that sell it for under $1.

Figure out a rule that would determine whether a soda is cheap or not.

Example: Arithmetic

Cheap(soda) <-
Sells(rest1,soda,p1) AND
Sells(rest2,soda,p2) AND
p1 < 1.00 AND
p2 < 1.00 AND
rest1 <> rest2

Negated sub-goals

We may put “NOT” in front of a sub-goal, to negate its meaning.

Example: Negated sub-goals

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 \( y \).

\[ S(x,y) \leftarrow \text{Arc}(x,z) \land \text{Arc}(z,y) \land \neg \text{Arc}(x,y) \]

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 sub-goals. If the body becomes true, add the head’s tuple to the result.
Example: Variable-Based --- 1

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

- Arc(1,2) and Arc(2,3) are the only tuples in the Arc relation.
- Only assignments to make the first sub-goal Arc(x,z) true are:
  1. \( x = 1; z = 2 \)
  2. \( x = 2; z = 3 \)

Example: Variable-Based; \( x=1, z=2 \)

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

\[
\begin{array}{ccc}
  1 & 1 & 2 \\
  2 & 3 & 1 \\
\end{array}
\]

3 is the only value of \( y \) that makes all three sub-goals true.

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

Example: Variable-Based; \( x=1, z=2 \)

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

\[
\begin{array}{ccc}
  2 & 2 & 3 \\
  3 & 3 & 2 \\
\end{array}
\]

No value of \( y \) makes Arc(3,z) true.

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

Example: Variable-Based; \( x=2, z=3 \)

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

\[
\begin{array}{ccc}
  2 & 2 & 3 \\
  3 & 3 & 2 \\
\end{array}
\]

No value of \( y \)

Tuple-Based Assignment

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

\[ S(x,y) \leftarrow \text{Arc}(x,z) \land \text{Arc}(z,y) \land \neg \text{Arc}(x,y) \]

- Only possible values \( \text{Arc}(1,2), \text{Arc}(2,3) \)
- Four possible assignments to first two subgoals:

<table>
<thead>
<tr>
<th>Arc(x,z)</th>
<th>Arc(z,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(2,3)</td>
</tr>
<tr>
<td>(2,3)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>(2,3)</td>
<td>(2,3)</td>
</tr>
</tbody>
</table>

Only assignment with consistent \( z \)-value. Since it also makes \( \neg \text{Arc}(x,y) \) true, add \( S(1,3) \) to result.

These two rows are invalid since \( z \) can’t be \((1 \text{ and } 3)\) or \((3 \text{ and } 2)\) simultaneously.

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 following EDB find all the manufacturers of sodas Joe doesn’t sell:
  - \( \text{Sells}(\text{rest, soda, price}) \) and
  - \( \text{sodas(name, manf)} \).

\[
\text{JoeSells}(b) \leftarrow \text{Sells}('\text{Joe’s rest}', b, p) \\
\text{Answer}(m) \leftarrow \text{Sodas}(b,m) \\
\quad \text{AND NOT JoeSells}(b)
\]

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.
Expressive Power of Datalog

- But with recursion,
  - Datalog can express more than these languages.
  - Yet still not Turing-complete.

Recursive Example: Generalized Cousins

- EDB: Parent(c,p) = p is a parent of c.
- Generalized cousins: people with common ancestors one or more generations back.
- Note: We are all cousins according to this definition.

Recursive Example

Sibling(x,y) <- Parent(x,p)
  AND Parent(y,p)
  AND x<>y

Cousin(x,y) <- Sibling(x,y)

Cousin(x,y) <- Parent(x,xParent)
  AND Parent(y,yParent)
  AND Cousin(xParent,yParent)

Definition of Recursion

- Form a dependency graph whose nodes = IDB predicates.
- Arc X -> 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

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

Start:
IDB = ∅

Apply rules
to IDB, EDB

yes

no

Change
to IDB?

yes

no
done

Example: Evaluation of Cousin

◆Remember the rules:
Sibling(x,y) <- Parent(x,p) AND Parent(y,p) AND x<>y
Cousin(x,y) <- Sibling(x,y)
Cousin(x,y) <- Parent(x,xParent) AND Parent(y,yParent) AND Cousin(xParent,yParent)

Semi-naive 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.

Example: Evaluation of Cousin

◆We’ll proceed in rounds to infer
• Sibling facts (red)
• and Cousin facts (green).

Parent Data: Parent Above Child

Exercise:
1. List some of the parent-child relationships.
2. What is contained in the Sibling and Cousin data?

Parent Data: Parent Above Child

Exercise:
1. What do you expect after first round?
Round 1

Sibling and Cousin are presumed empty.
Sibling remains empty since it depends on Sibling and Cousin which is empty.
Exercise: What do you expect in the next round?

Round 2

Sibling facts remain unchanged because Sibling is not recursive.
First execution of the Cousin rule "duplicates" the Sibling facts as Cousin facts (shown in green).
Exercise: What do you expect in the next round?

Round 3

The execution of the non-recursive Cousin rule gives us nothing.
However, the recursive call gives us several pairs (shown in bold green).
Exercise: What do you expect in the next round?

Round 4

The execution of the non-recursive Cousin rule still gives us nothing.
However, the recursive call gives us several pairs (shown in even bold green).
Exercise: What do you expect in the next round?

Done!

Recursion Plus Negation

- "Naïve" and "Semi-Naïve" evaluation doesn't work when there are negated sub-goals.
  - Discovering IDB tuples on one route can decrease the IDB tuples on the next route.
  - Losing IDB tuples on one route can yield more tuples on the next route.
Recursion Plus Negation

♦ 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.

Problematic Recursive Negation

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

EDB: Q(1), Q(2)

Initial: \( P = \{ \} \) // From Q(1) & Q(2)
Round 1: \( P = \{(1), (2)\} \) // From NOT(P(1)) & NOT(P(2))
Round 2: \( P = \{ \} \) // From NOT(P(1)) & NOT(P(2))
Round 3: \( P = \{(1), (2)\} \) // From Q(1) & Q(2)
Round n: etc., etc. …

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.

Why Stratified Negation?

♦ Usually require that Negation be stratified to prevent the problem just described. Stratification does two things:
  ♦ Lets us evaluate the IDB predicates in a way that it converges.
  ♦ Lets us discover the “correct” solution in face of “many solutions.”

Safe Rules

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

Example: Unsafe Rules

♦ Each of the following is unsafe and not allowed:
  1. \( S(x) \leftarrow R(y) \)
     ♦ Because \( x \) appears as distinguished variable (\( S(x) \)) but does not appear in a non-negated sub-goal.
  2. \( S(x) \leftarrow R(y) \text{ AND NOT } R(x) \)
     ♦ Because \( x \) appears in negated sub-goal (\( R(x) \)) but does not appear in a sub-goal.
  3. \( S(x) \leftarrow R(y) \text{ AND } x < y \)
     ♦ Because \( x \) appears in an arithmetic sub-goal (\( R(x) \)) but does not appear in a non-negated sub-goal.
Example: Unsafe Rules

◆ In each case, an infinite number of values for x can satisfy the rule, even if R is a finite relation.

Strata

◆ Stratum:
  * Let us separate good negative recursive negation from bad.
  * 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.

Strata

◆ Stratified negation = “finite strata.”
◆ Notice in P(x) <- Q(x) 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 -> B if predicate A depends on B.
  * Label this arc “~” if the B sub-goal 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) <- Q(x) AND NOT P(x)

Infinite path due to loop: not stratified!
Another Example

Setting is graph: Nodes designated as source and target.
EDB consists of:
- Source in Source(x)
- Target in Target(x)
- Arcs between nodes in Arc(x,y)

Our problem is to find target nodes that are not reached from any source.

Rules: "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)

First 2 rules recursively define Reach:
- A node can be reached if it is a source or can be reached from a node connected to source.
- NoReach if it is a target that cannot be reached.

The Stratum Graph

Stratum 1:
<= 1 "–" arc on any path out.

Stratum 0:
No "–" arcs on any path out.

Since all strata are finite, this is an example of stratified negation.

Models

To discuss possible results
- Concept imported from Logic to Datalog
- Discussion is limited to Datalog application.
- 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.
- If moon were made of blue cheese, you will all flunk.

However, if the body is true, then the head must be true as well.
- If professor is human, you will get fair grades.

Minimal Models

A model should be minimal that if should not properly contain any other model
Intuitively, we don’t want to assert facts that do not have to be asserted
Minimal Models

- When there is no negation, a Datalog program has a unique minimal model
  - One given by naïve and semi-naïve evaluation
- With negation and recursion, there can be several minimal models
  - even if the program is stratified.
- Fortunately, we can compute the minimal model that makes sense
  - And that is the stratified model

The Stratified Model

- When the Datalog program is stratified:
  - We evaluate IDB predicates in stratum 0
    - There can be several predicates in stratum but they can’t depend negatively on themselves on any other IDB predicate strata.
  - Once evaluated, treat it as EDB for next strata.
  - Proceed iteratively until all IDB predicates are evaluated

Example: Multiple Models --- 1

Reach(x) <- Source(x)
Reach(x) <- Reach(y) AND Arc(y,x)
NoReach(x) <- Target(x) AND NOT Reach(x)

Stratum 0: Reach(1), Reach(2)
Stratum 1: NoReach(3)

1 is the only source; 2 and 3 are targets; 4 is an additional node.
Reach is fixed at 1 and 2. Since 1 and 2 can be reached, NoReach has one element in the set: 3.

Example: Multiple Models --- 2

Reach(x) <- Source(x)
Reach(x) <- Reach(y) AND Arc(y,x)
NoReach(x) <- Target(x) AND NOT Reach(x)

Another model: Reach(1), Reach(2), Reach(3), Reach(4); NoReach is empty.

SQL-99 Recursion

- Excellent example of Theory -> Practice
- Datalog recursion 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.
Example: SQL Recursion --- 1

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

WITH Sibling(x, y) AS
(SELECT p1.child, p2.child
FROM Parent p1, Parent p2
WHERE p1.parent = p2.parent AND p1.child <> p2.child)

Example: SQL Recursion --- 2

WITH ...
RECURSIVE Cousin(x, y) AS
(SELECT * FROM Sibling)
UNION
(SELECT p1.child, p2.child
FROM Parent p1, Parent p2, Cousin
WHERE p1.parent = Cousin.x AND p2.parent = Cousin.y)

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:
  \( * \ P = Q \ \text{UNION} \ R \).
  \( * \ P = \text{SELECT}_{p=16}(Q) \).
Example: Nonmonotonicity

If Sells(rest, soda, price) is our usual relation, then the result of the query:
SELECT AVG(price)
FROM Sells
WHERE rest = 'Joe''s Rest';
is not monotone in Sells.
Inserting a Joe's-Rest 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.

Arcs $P \rightarrow Q$:
1. $P$ is a relation in the FROM list (not of a subquery).
2. $P$ is a relation in the body of the subquery.
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:

```
Sib = ...
Cousin = ( ... FROM Sib )
UNION
( ... FROM Cousin )
```

Subquery S1
Subquery S2

No “-” at all, so surely stratified.

Nonmonotone Example

Change the UNION in the Cousin example to EXCEPT:

```
Sib = ...
Cousin = ( ... FROM Sib )
EXCEPT
( ... FROM Cousin )
```

Can delete a tuple from Cousin
Inserting a tuple into S2
The Graph

Sib

Cousin

S1

S2

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

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.