Database Theory
VU 181.140, SS 2018

2. Introduction to Datalog

Reinhard Pichler

Institute of Logic and Computation
DBAI Group
TU Wien

13 March, 2018
Outline

2. Datalog
   2.1 Motivation
   2.2 Datalog - Syntax
   2.3 Restrictions on the Datalog Syntax
   2.4 Logical Semantics of Datalog
   2.5 Operational Semantics of Datalog
   2.6 Datalog with negation
   2.7 Stratification
Motivation

- SQL, relational algebra, relational calculus (both tuple and domain relational calculus) are “relational complete”, i.e., they have the full expressive power of relational algebra.
- But: many interesting queries cannot be formulated in these languages
- Example: no recursive queries (SQL now offers a recursive construct)
Example

- Relation `parent(PARENT, CHILD)`, gives information on the parent-child relationship of a certain group of people.
- Problem: look for all ancestors of a certain person.
- Solution: define relation `ANCESTOR(X, Y)`: X is ancestor of Y by generating one generation after the other (one join and one projection each) and finally merge all generations (union):

  \[
  \text{grandparent(GRANDPARENT, GRANDCHILD) := } \pi_{1,4}(\text{parent}[\text{CHILD} = \text{PARENT}]\text{parent})
  \]

  \[
  \text{grandgrandparent(GRANDGRANDPARENT,GRANDGRANDCHILD) := } \pi_{1,4}(\text{parent}[\text{CHILD} = \text{GRANDPARENT}]\text{grandparent})
  \]

  \[
  \ldots
  \]

  \[
  \text{ancestor(ANCESTOR, NAME) := parent } \cup \text{ grandparent } \cup \text{ grandgrandparent } \cup \ldots
  \]
Possible Solution

- Use of a programming language with an embedded relational complete query language:

```plaintext
begin
result := {};
newtuples := parent;
while newtuples ∉ result do
begin
result := result ∪ newtuples;
newtuples := π_{1,4}(newtuples[2 = 1]parent);
end;
ancestor := result
end.
```

- procedural, needs knowledge of a programming language, leaves little room for query optimization.
Better Solution: Datalog

- Prolog-like logical query language,
- allows recursive queries in a declarative way
- Example:
  - compute all ancestors on the basis of the relation parent
    \[
    \text{ancestor}(X,Y) \leftarrow \text{parent}(X,Y).
    \text{ancestor}(X,Z) \leftarrow \text{parent}(X,Y), \text{ancestor}(Y,Z).
    \]
  - use the ancestor predicate to compute the ancestors of a certain person (Hans):
    \[
    \text{hans\_ancestor}(X) \leftarrow \text{ancestor}(X,\text{hans}).
    \]
  - compute the ancestors of a certain person (Hans) directly:
    \[
    \text{hans\_ancestor}(X) \leftarrow \text{parent}(X,\text{hans}).
    \text{hans\_ancestor}(X) \leftarrow \text{hans\_ancestor}(Y), \text{parent}(X,Y).
    \]
Datalog - Syntax

\[
\begin{align*}
\text{<datalog_program>} & \ ::= \text{<datalog_rule>} \mid \\
& \quad \quad \quad \quad \quad \quad \quad \quad \text{<datalog_program><datalog_rule>}
\end{align*}
\]

\[
\begin{align*}
\text{<datalog_rule>} & \ ::= \text{<head>} \,:-\, \text{<body>}
\end{align*}
\]

\[
\begin{align*}
\text{<head>} & \ ::= \text{<literal>}
\text{<body>} & \ ::= \text{<literal>} \mid \text{<body>, <literal>}
\end{align*}
\]

\[
\begin{align*}
\text{<literal>} & \ ::= \text{<relation_id>}(\text{<list_of_args>})
\text{<list_of_args>} & \ ::= \text{<term>} \mid \text{<list_of_args>, <term>}
\text{<term>} & \ ::= \text{<symb_const>} \mid \text{<symb_var>}
\text{<symb_const>} & \ ::= \text{<number>} \mid \text{lcc} \mid \text{lcc}<\text{string}>
\text{<symb_var>} & \ ::= \text{<ucc>} \mid \text{ucc}<\text{string}>
\end{align*}
\]

(lcc = lower_case_character; ucc = upper_case_character)
Restrictions on the Datalog Syntax

<relation_id>:

- name of an existing relation of the database (parent) - can be used only in rule bodies
- name of a new relation defined by the datalog program (ancestor)
- has always the same number of arguments.

comparison predicates:

=, <>, <, > are treated like known database relations.

variables:

- each variable that appears in the head of a rule has to be bound in the body
- variables that appear as arguments of comparison predicates must appear in the same body in literals without comparison predicates

A datalog query is also called datalog program
Logical Semantics of Datalog

We consider

\[ R \ldots \text{ datalog rule of the form } L_0 \leftarrow L_1, L_2, \ldots, L_n, \]
\[ L_i \ldots \text{ literal of the form } p_i(t_1, \ldots, t_{n_i}) \]
\[ x_1, x_2, \ldots, x_\ell \text{ variables in } R \]
\[ P \ldots \text{ datalog program with the rules } R_1, R_2, \ldots, R_m \]

We construct

\[ R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L_1 \land L_2 \land \cdots \land L_n) \Rightarrow L_0). \]

We assign to each datalog program \( P \) the (semantically) well-defined formula \( P^* \) as follows:

\[ P^* = R_1^* \land R_2^* \land \cdots \land R_m^* \]
We consider now

\[ \text{REL} \ldots \text{a relation of the database.} \]

\[ \langle t_1, \ldots, t_n \rangle \ldots \text{a tuple of the relation REL.} \]

\[ \text{rel}(t_1, \ldots, t_n) \ldots \text{a fact} \]

\[ \text{DB} \ldots \text{database with relations REL}_1, \text{REL}_2, \ldots, \text{REL}_k \]

We assign to each database relation REL the formula

\[ \text{REL}^* = \text{conjunction of all facts} \]

- a relation is an unordered set of tuples
- the assignment REL \(\mapsto\) REL* is therefore not uniquely defined.
- take an arbitrary order (e.g. lexicographical order) since conjunction is associative and commutative.

We assign to each database DB the (semantically) well-defined formula DB* as follows:

\[ DB^* = \text{REL}_1^* \land \text{REL}_2^* \land \cdots \land \text{REL}_k^*. \]
We have:

\[ DB^* \] is a conjunction of ground atoms (i.e., the facts) and
\[ P^* \] is a conjunction of formulas with implication

Let \( G \) be a conjunction of facts and formulas with implication. Then the set \( \text{cons}(G) \) of facts that follow from \( G \) is uniquely defined. In other words, we have \( \text{cons}(G) = \{A \mid A \text{ is a fact with } G \models A\} \).

**Definition**

The semantics of a datalog program \( P \) is defined as the function \( M[P] \), that assigns to each database \( DB \) the set of all facts that follow from the formula “\( P^* \land DB^* \)”

\[
M[P] : DB \to \text{cons}(P^* \land DB^*)
\]
Example

Consider the database DB with relations woman(NAME), man(NAME), parent(PARENT, CHILD) and the datalog program:

\[
\text{grandpa}(X,Y) :- \text{man}(X), \text{parent}(X,Z), \text{parent}(Z,Y).
\]

<table>
<thead>
<tr>
<th>woman (NAME)</th>
<th>man (NAME)</th>
<th>parent (PARENT, CHILD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grete</td>
<td>Hans</td>
<td>Hans Linda</td>
</tr>
<tr>
<td>Linda</td>
<td>Karl</td>
<td>Grete Linda</td>
</tr>
<tr>
<td>Gerti</td>
<td>Michael</td>
<td>Karl Michael</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linda Michael</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Karl Gerti</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linda Gerti</td>
</tr>
</tbody>
</table>
Let us compute $DB^*$, $P^*$ and $\text{cons}(P^* \land DB^*)$:

$DB^* = REL_1^* \land \cdots \land REL_k^*$ with $REL_i^* = \text{conjunction of all facts}$

\[
DB^* = \text{woman}(\text{grete}) \land \text{woman}(\text{linda}) \land \text{woman}(\text{gerti}) \land \\
\text{man}(\text{hans}) \land \text{man}(\text{karl}) \land \text{man}(\text{michael}) \land \\
\text{parent}(\text{hans}, \text{linda}) \land \text{parent}(\text{grete}, \text{linda}) \land \\
\text{parent}(\text{karl}, \text{michael}) \land \text{parent}(\text{linda}, \text{michael}) \land \\
\text{parent}(\text{karl}, \text{gerti}) \land \text{parent}(\text{linda}, \text{gerti}).
\]

$P^* = R_1^* \land \cdots \land R_m^*$ with $R_i^* = \forall x_1 \forall x_2 \ldots \forall x_\ell((L_1 \land \cdots \land L_n) \Rightarrow L_0)$.

\[
P^* = \forall X \forall Y \forall Z : ((\text{man}(X) \land \text{parent}(X, Z) \land \text{parent}(Z, Y)) \Rightarrow \text{grandpa}(X, Y)).
\]
The new facts in $\text{cons}(P^* \land DB^*)$:

$\text{grandpa}(\text{hans}, \text{michael}), \text{grandpa}(\text{hans}, \text{gerti})$.

The datalog program $P$ with

$P = \text{grandpa}(X, Y) : \text{man}(X), \text{parent}(X, Z), \text{parent}(Z, Y)$

defines a new relation $\text{grandpa}$ with the following tuples:

<table>
<thead>
<tr>
<th>grandpa (X   Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hans  Michael</td>
</tr>
<tr>
<td>Hans    Gerti</td>
</tr>
</tbody>
</table>
Operational Semantics of Datalog

- Datalog rules are seen as inference rules,
- a fact that appears in the head of a rule can be deduced, if the facts in the body of the rule can be deduced.

**Example:**

facts: parent(linda,michael), parent(linda,gerti)
rule: siblings(michael,gerti) :-
      parent(linda,michael), parent(linda,gerti).

the following fact can be deduced:

siblings(michael,gerti)
Rules with variables

- A rule $R$ with variables represents all variable-free rules we get from $R$ by substituting the variables with the constant symbols.
- The constant symbols are taken from the database $DB$ and the program $P$.
- A variable-free rule resulting from such a substitution is called a ground instance of $R$ with respect to $P$ and $DB$.
- We write $\text{Ground}(R, P, DB)$ to denote the set of all ground instances over $P$ and $DB$ of $R$. 
Example:

Compute all relations between siblings with the following rule:

\[
\text{siblings}(Y, Z) : - \text{parent}(X, Y), \text{parent}(X, Z), Y \leftrightarrow Z.
\]

All \(6^3\) ground instances of this rule with respect to \(P\) and \(DB\) from above are (Note that there are 6 constant symbols: \{grete, linda, gerti, hans, michael, karl\}):

\[
\begin{align*}
\text{siblings(} & \text{grete, grete}) : - \text{parent(} & \text{grete, grete}), \text{parent(} & \text{grete, grete}), \\
& & \text{grete} \leftrightarrow \text{grete} & (X = Y = Z = \text{grete}) \\
\text{siblings(} & \text{grete, linda}) : - \text{parent(} & \text{grete, grete}), \text{parent(} & \text{grete, linda}), \\
& & \text{grete} \leftrightarrow \text{linda} & (X = Y = \text{grete}, Z = \text{linda}) \\
& & \ldots & \ldots \\
\text{siblings(} & \text{karl, karl}) : - \text{parent(} & \text{karl, karl}), \text{parent(} & \text{karl, karl}), \\
& & \text{karl} \leftrightarrow \text{karl} & (X = Y = Z = \text{karl}) \\
\end{align*}
\]
Idea: execution of a datalog program $P$ on a database $DB$:
iterative deduction of facts until saturation is reached
(fixpoint)

Formalization: define a fixpoint operator

- define Operator $T_P(DB)$: augments $DB$ with all facts, that can be deduced in one step by applying the rules from $P$ to $DB$.

$$T_P(DB) = DB \cup \bigcup_{R \in P} \{L_0 \mid L_0 \leftarrow L_1, \ldots, L_n \in \text{Ground}(R; P, DB),
\quad L_1, \ldots, L_n \in DB\}$$

- $T_P$ is called the immediate consequence operator.
- $T_P^i(DB) = T_P(T_P^{i-1}(DB))$ iterated application of $T_P$. 
\begin{align*}
T^0_P(DB) &= DB \\
T^1_P(DB) &= T_P(T^0_P(DB)) = T_P(DB) \\
&= DB \cup \bigcup_{R \in P} \{ L_0 \mid L_0 : -L_1, \ldots, L_n \in \text{Ground}(R; P, DB), \quad L_1, \ldots, L_n \in DB \} \\
T^2_P(DB) &= T_P(T^1_P(DB)) = T_P(T_P(DB)) \\
&\quad \quad \quad \cdots \quad \cdots \\
T^i_P(DB) &= T_P(T^{i-1}_P(DB)) = T_P(\ldots T_P(DB)) \\
&\quad \quad \quad \cdots \quad \cdots
\end{align*}
Properties of $T_P(DB)$

- The set of facts is monotonically increasing i.e.:

$$T_P^i(DB) \subseteq T_P^{i+1}(DB)$$

- the sequence $\langle T_P^i(DB) \rangle$ converges finitely:
  there exists $n$ with $T_P^m(DB) = T_P^n(DB)$ for all $m \geq n$.

- $T_P^\omega(DB)$ ... set of facts, to which $\langle T_P^i(DB) \rangle$ converges is the result of the application of $P$ to $DB$.

- The operational semantics of a datalog program $P$ assigns to each database $DB$ the set of facts $T_P^\omega(DB)$:

$$O[P] : DB \rightarrow T_P^\omega(DB).$$

**Theorem (Equivalence of semantics)**

Assume a program $P$. Then it holds that $M[P] = O[P]$. In other words, for any database $DB$, we have: $\text{cons}(P^* \land DB^*) = T_P^\omega(DB)$.
Proof of Theorem

Let $P$ be a program and $DB$ a database. We show

$$cons(P^* \land DB^*) = T^\omega_P(DB).$$

(1) We first show $T^\omega_P(DB) \subseteq cons(P^* \land DB^*)$. By induction on $i$, we show that $T^i_P(DB) \subseteq cons(P^* \land DB^*)$ for every $i \geq 0$. Note that this includes the case where $i = \omega$.

**Base case.** Assume $i = 0$. Take a fact $L \in T^0_P(DB)$. Then by definition of $T^0_P(DB)$, $L \in DB$. By definition, $DB^*$ is a conjunction of literals and $L$ occurs in it. Hence, by classical logic, $L \in cons(P^* \land DB^*)$.

**The inductive step.** Suppose $T^i_P(DB) \subseteq cons(P^* \land DB^*)$ for $i \geq 0$. We show that $T^{i+1}_P(DB) \subseteq cons(P^* \land DB^*)$. Recall that

$$T^{i+1}_P(DB) = T_P(T^i_P(DB)).$$

Thus by the definition of $T_P$, $T^i_P(DB)$

$$T^{i+1}_P(DB) = T^i_P(DB) \cup \bigcup_{R \in P} \{L_0 \mid L_0 \vdash L_1, \ldots, L_n \in \text{Ground}(R, P, DB), L_1, \ldots, L_n \in T^i_P(DB)\}.$$
By the induction hypothesis, $T^i_P(DB) \subseteq cons(P^* \land DB^*)$. Thus it remains to show that $L_0 \in cons(P^* \land DB^*)$ for any rule $R \in P$ such that there is $L_0 \leftarrow L_1, \ldots, L_n \in \text{Ground}(R, P, DB)$ with $L_1, \ldots, L_n \in T^i_P(DB)$.

Assume such a rule $R = L_0' \leftarrow L_1', \ldots, L_n'$ in $P$, and suppose $\pi$ is the substitution of variables with constants such that applying $\pi$ to $R$ results in $L_0 \leftarrow L_1, \ldots, L_n$, i.e. $\pi(L_j') = L_j$ for $j \in \{0, \ldots, n\}$.

By construction, in $P^* \land DB^*$ we have the conjunct

$$R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell((L_1' \land L_2' \land \cdots \land L_n') \Rightarrow L_0').$$

Thus, by employing the semantics of classical logic, for any variable substitution $\pi'$ such that $\{\pi'(L_1'), \ldots, \pi'(L_n')\} \subseteq cons(P^* \land DB^*)$ we also have $\pi'(L_0') \in cons(P^* \land DB^*)$. Since $\pi$ is a substitution such that $\{\pi(L_1'), \ldots, \pi(L_n')\} = \{L_1, \ldots, L_n\} \subseteq cons(P^* \land DB^*)$ by the induction hypothesis, we get $\pi(L_0') = L_0 \in cons(P^* \land DB^*)$. 

Pichler 13 March, 2018
(2) We show $\text{cons}(P^* \land DB^*) \subseteq T^*_P(DB)$. To this end, we prove that $L \notin T^*_P(DB)$ implies $L \notin \text{cons}(P^* \land DB^*)$, for any fact $L$. We thus simply show that $T^*_P(DB)$ is a model of $P^* \land DB^*$.

This suffices because of the following simple property: if $M$ is a model of a formula $F$, then any fact $L \notin M$ is not a logical consequence of $F$ (as witnessed by $M$ itself).
$T^*_P(DB)$ is a model of $DB^*$ because $DB = T^0_P(DB) \subseteq T^*_P(DB)$ by the definition of $T^*_P(DB)$.

It remains to show that $T^*_P(DB)$ is also a model of $P^*$. Consider an arbitrary rule $R \in P$. We have to show that $T^*_P(DB)$ is a model of $R^*$ with $R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L_1 \land L_2 \land \cdots \land L_n) \Rightarrow L_0)$.

Consider an arbitrary (ground) variable assignment $\pi$ on the variables $x_1, \ldots, x_\ell$. The only non-trivial case is that all facts $\pi(L_1), \ldots, \pi(L_n)$ are true in $T^*_P(DB)$, i.e., $\{\pi(L_1), \ldots, \pi(L_n)\} \subseteq T^*_P(DB)$.

We have to show that then also $\pi(L_0)$ is true in $T^*_P(DB)$, i.e., $\pi(L_0) \in T^*_P(DB)$.

We know $\pi(L_0) :\neg \pi(L_1), \ldots, \pi(L_n) \in \text{Ground}(R, P, DB)$. Thus by the definition of $T_P$, $\pi(L_0) \in T_P(T^*_P(DB))$. Since $T_P(T^*_P(DB)) = T^*_P(DB)$ by the definition of $T^*_P(DB)$, we obtain $\pi(L_0) \in T^*_P(DB)$.
Algorithm: INFER

**INPUT**: datalog program $P$, database $DB$

**OUTPUT**: $T^\omega_P(DB)$ ($= cons(P^* \land DB^*)$)

**STEP 1.** $GP := \bigcup_{R \in P} Ground(R; P, DB)$,  
(* $GP$ . . . set of all ground instances *)

**STEP 2.** $OLD := \{\}; NEW := DB$;

**STEP 3.** while $NEW \neq OLD$ do begin  
  $OLD := NEW; NEW := ComputeTP(OLD);$  
end;

**STEP 4.** output $OLD$. 
Subroutine ComputeTP

**INPUT**: Set of facts $OLD$

**OUTPUT**: $T_P(OLD)$

**STEP 1.** $F := OLD$;

**STEP 2.** for each rule $L_0 :- L_1, \ldots, L_n$ in $GP$ do
  if $L_1, \ldots, L_n \in OLD$
  then $F := F \cup \{ L_0 \}$;

**STEP 3.** return $F$;
Example

Apply the following program $P$ to calculate all ancestors of the above given database $DB$.

\[
\text{ancestor}(X,Y) :- \text{parent}(X,Y).
\]
\[
\text{ancestor}(X,Z) :- \text{parent}(X,Y), \text{ancestor}(Y,Z).
\]

**Step 1. (INFER) build $GP$**

\[
GP = \{ \text{ancestor}(\text{grete},\text{grete}) :- \text{parent}(\text{grete},\text{grete}), \\
\text{ancestor}(\text{grete},\text{linda}) :- \text{parent}(\text{grete},\text{linda}), \\
\ldots, \\
\text{ancestor}(\text{grete},\text{grete}) :- \text{parent}(\text{grete},\text{grete}), \\
\text{ancestor}(\text{grete},\text{grete}), \\
\text{ancestor}(\text{grete},\text{grete}) :- \text{parent}(\text{grete},\text{linda}), \\
\text{ancestor}(\text{linda},\text{grete}), \\
\ldots \}.
\]

(There are $6^2 + 6^3 = 252$ ground instances.)
Step 2. \( \text{OLD} := \{\} \), \( \text{NEW} := \text{DB} \);

Step 3. \( \text{OLD} \neq \text{NEW} \)

Cycle 1: \( \text{OLD} := \text{DB} \), \( \text{NEW} := \text{TP(OLD)} = \text{TP(DB)} \)
\[
\text{TP(OLD)} = \text{OLD} \cup \{\text{ancestor}(A, B) \mid \text{parent}(A, B) \in \text{DB}\};
\]

Cycle 2: \( \text{OLD} := \text{TP(DB)} \), \( \text{NEW} := \text{TP(OLD)} = \text{TP( TP(DB) )} \)
\[
\text{TP(OLD)} =
\text{OLD} \cup \{\text{ancestor}(\text{hans, michael}), \text{ancestor}(\text{hans, gerti}), \text{ancestor}(\text{grete, michael}), \text{ancestor}(\text{grete, gerti})\}.
\]

Cycle 3: \( \text{TP(OLD)} = \text{OLD} \), there are no new facts

Step 4. Output of \( \text{OLD} \).

The result corresponds to the extension of \( \text{DB} \) with the new table \text{ancestor}
<table>
<thead>
<tr>
<th>parent</th>
<th>(PARENT</th>
<th>CHILD)</th>
<th>ancestor</th>
<th>(ANCESTOR</th>
<th>NAME)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hans</td>
<td>Linda</td>
<td></td>
<td>Hans</td>
<td>Linda</td>
<td></td>
</tr>
<tr>
<td>Grete</td>
<td>Linda</td>
<td></td>
<td>Grete</td>
<td>Linda</td>
<td></td>
</tr>
<tr>
<td>Karl</td>
<td>Michael</td>
<td></td>
<td>Karl</td>
<td>Michael</td>
<td></td>
</tr>
<tr>
<td>Linda</td>
<td>Michael</td>
<td></td>
<td>Linda</td>
<td>Michael</td>
<td></td>
</tr>
<tr>
<td>Karl</td>
<td>Gerti</td>
<td></td>
<td>Karl</td>
<td>Gerti</td>
<td></td>
</tr>
<tr>
<td>Linda</td>
<td>Gerti</td>
<td></td>
<td>Linda</td>
<td>Gerti</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hans</td>
<td>Michael</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hans</td>
<td>Gerti</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Grete</td>
<td>Michael</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Grete</td>
<td>Gerti</td>
<td></td>
</tr>
</tbody>
</table>
Datalog with negation

- Without negation, datalog is not relational complete because set difference \((R - S)\) cannot be expressed.
- We introduce the negation (not) in bodies of rules.
- Restriction on the application of the negation:
  
  *A relation \(R\) must not be defined on the basis of its negation.*

- Check for this constraint: with graph-theoretic methods.
Graph representation

Let $P$ be a datalog program with negated literals in the body of rules

**Definition: dependency graph**

$DEP(P)$ is defined as the directed graph, with:
- nodes ... predicates (predicate symbols) $p$ in $P$,
- edges ... $p \rightarrow q$, if there exists a rule in $P$ where $p$ is the head atom and $q$ appears in the body (meaning: “$p$ depends on $q$”).

Mark an edge $p \rightarrow q$ of $DEP(P)$ with a star “*”, if $q$ in the body is negated.

**Definition**

A datalog program $P$ with negation is called valid if the graph $DEP(P)$ has no directed cycle that contains an edge marked with “*”.

Such programs are called **stratified**, since they can be divided into strata with respect to the negation.
Example

The following program $P$ with the rules:

- $\text{husband}(X) :- \text{man}(X), \text{married}(X)$.
- $\text{bachelor}(X) :- \text{man}(X), \text{not} \ \text{husband}(X)$.

is stratified.
The program $P$ with the rules:

husband(X) :- man(X), not bachelor(X).
bachelor(X) :- man(X), not husband(X).

is not stratified.
Stratification

Definition

A stratum is composed by the maximal set of predicates for which the following holds:

1. if a predicate $p$ appears in the head of a rule, that contains a negated predicate $q$ in the body, then $p$ is in a higher stratum than $q$.

2. if a predicate $p$ appears in the head of a rule, that contains an unnegated (positive) predicate $q$ in the body, then $p$ is in a stratum at least as high as $q$. 
Algorithm

INPUT: A set of datalog rules.
OUTPUT: the decision whether the program is stratified and the classification of the predicates into strata.

METHOD:

1. initialize the strata for all predicates with 1.
2. do for all rules $R$ with predicate $p$ in the head:
   - if (i) the body of $R$ contains a negated predicate $q$, (ii) the stratum of $p$ is $i$, and (iii) the stratum of $q$ is $j$ with $i \leq j$, then set $i := j + 1$.
   - if (i) the body of $R$ contains an unnegated predicate $q$, (ii) the stratum of $p$ is $i$, and (iii) the stratum of $q$ is $j$ with $i < j$, then set $i := j$.

until:

- status is stable $\Rightarrow$ program is stratified.
- stratum $n > \#$ predicates $\Rightarrow$ not stratified.
Example

Consider $R, S$ relations of the database $DB, P$:

$$v(X,Y) :- r(X,X), r(Y,Y).$$
$$u(X,Y) :- s(X,Y), s(Y,Z), \text{not} \ v(X,Y).$$
$$w(X,Y) :- \text{not} \ u(X,Y), \ v(Y,X).$$
Semantics of datalog with negation

**Note:** when calculating the strata of a datalog program with negation the following holds:

- **Step 1:** computation of all relations of the first stratum.
- **Step i:** computation of all relations that belong to stratum $i$.

⇒ the relations computed in step $i - 1$ are known in step $i$.

Semantics of datalog with negation is therefore uniquely defined.

Computation of $P$ from the last example above:

- **Step 1:** compute $V$ from $R$
- **Step 2:** compute $U$ from $S$ and $V$
- **Step 3:** compute $W$ from $U$ and $V
Properties of datalog with negation

- Datalog with negation is relational complete:
  - The difference $D = R - S$ of two (e.g. binary) relations $R$ and $S$:
    $d(X,Y) \leftarrow r(X,Y), \textsf{not } s(X,Y)$.

- Syntactical restrictions of datalog with negation:
  all variables that appear in the body within a negated literal must also appear in a positive (unnegated) literal.
Example

Let $DB$ be a database that contains information on graphs, with relations $v(X)$, saying $X$ is a node and $e(X,Y)$ saying there is an edge from $X$ to $Y$. Write a datalog program that computes all pairs of nodes $(X,Y)$, where $X$ is a source, $Y$ is a sink and $X$ is connected to $Y$.

\begin{verbatim}
\text{p}(X,Y) :- source(X), sink(Y), connection(X,Y).

connection(X,X) :- v(X).
connection(X,Y) :- e(X,Z), connection(Z,Y).

n_source(X) :- e(Y,X).
source(X) :- v(X), not n_source(X).

n_sink(X) :- e(X,Y).
sink(X) :- v(X), not n_sink(X).
\end{verbatim}
n_source: b, c, e, f
n_sink: a, b, c, d
connection: (a,a), \ldots, (f,f), (a,b), (a,c), (a,e), (a,f), (b,c), (b,e), (c,e), (d,c), (d,e)
source: a, d
sink: e, f
p: (a,e), (a,f), (d,e)
Learning objectives

- Motivation for Datalog (recursive queries)
- Syntax of Datalog
- Semantics of Datalog:
  - logical semantics,
  - operational semantics.
- Datalog with negation:
  - the need for negation,
  - the notions of dependency graph and stratification,
  - semantics of Datalog with negation.