2. Datalog

2.1 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)

Relation parents(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):

\[
\begin{align*}
grandchild(GRANDPARENT, GRANDCHILD) & := \pi_{1,4}(\text{parents}[\text{CHILD} = \text{PARENT}]\text{parents}) \\
grandgrandchild(GRANDGRANDPARENT, GRANDGRANDCHILD) & := \pi_{1,4}(\text{parents}[\text{CHILD} = \text{GRANDPARENT}]\text{grandchild}) \\
\vdots \\
ancestor(ANCESTOR, NAME) & := \text{parents} \cup \text{grandchild} \cup \text{grandgrandchild} \cup \ldots
\end{align*}
\]
Possible Solution

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

```plaintext
begin
    result := \{\};
    newtuples := parents;
    while newtuples \not\subseteq result do
        begin
            result := result \cup newtuples;
            newtuples := \pi_{1,4}(newtuples[2 = 1]parents);
        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 `parents`
  ```plaintext
  ancestor(X,Y) :- parents(X,Y).
  ancestor(X,Z) :- parents(X,Y), ancestor(Y,Z).
  ```

- use the ancestor predicate to compute the ancestors of a certain person (`Hans`):
  ```plaintext
  hans_ancestor(X) :- ancestor(X,hans).
  ```

- compute the ancestors of a certain person (`Hans`) directly:
  ```plaintext
  hans_ancestor(X) :- parents(X,hans).
  hans_ancestor(X) :- hans_ancestor(Y), parents(X,Y).
  ```

Datalog - Syntax

- `relation_id`:
  - name of an existing relation of the database (`parents`) - 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:

- `=`, `\neq`, `>`, `<` 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

Restrictions on the Datalog Syntax

- `<relation_id>`:
  - name of an existing relation of the database (`parents`) - 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:
  - `=`, `\neq`, `>`, `<` 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

\[ R \ldots \text{datalog rule of the form } L_0 \leftarrow L_1, L_2, \ldots, L_n, \]

\[ L_i \ldots \text{literal of the form } p_i(t_1, \ldots, t_n) \]

\[ x_1, x_2, \ldots, x_ℓ \text{ variables in } R \]

\[ P \ldots \text{datalog program with the rules } R_1, R_2, \ldots, R_m \]

We construct

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

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 have:

\[ DB^* \text{ is a conjunction of ground atoms (i.e., the facts)} \]

\[ P^* \text{ is a conjunction of formulas with implication} \]

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 "\( P^* \land DB^* \)"

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

Example

Consider the database \( DB \) with relations \( \text{woman(NAME)} \), \( \text{man(NAME)} \), \( \text{parents(PARENT, CHILD)} \) and the datalog program:

\[ \text{grandpa}(X,Y) \leftarrow \text{man}(X), \text{parents}(X,Z), \text{parents}(Z,Y). \]

\[
\begin{array}{llll}
\text{woman } & \text{man } & \text{parents } \\
\text{Grete} & \text{Hans} & \text{Grete} & \text{Linda} \\
\text{Linda} & \text{Karl} & \text{Grete} & \text{Linda} \\
\text{Gerti} & \text{Michael} & \text{Karl} & \text{Gerti} \\
\end{array}
\]

We consider now

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

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

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

\[ DB \ldots \text{database with relations } REL_1, REL_2, \ldots, REL_k \]

We assign to each database relation \( REL \) the formula

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

- 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 commutative.

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

\[ DB^* = REL_1^* \land REL_2^* \land \cdots \land REL_k^*. \]
Let us compute $DB^*$, $P^*$ and $\text{cons}(P^* \land DB^*)$:

$$DB^* = REL_1^* \land \cdots \land REL_k^*$$

with $REL_i^*$ = conjunction of all facts

$$DB^* = \text{woman}(\text{grete}) \land \text{woman}(\text{linda}) \land \text{woman}(\text{gerti}) \land \text{man}(\text{hans}) \land \text{man}(\text{karl}) \land \text{man}(\text{michael}) \land $$

$$\text{parents}(\text{hans}, \text{linda}) \land \text{parents}(\text{grete}, \text{linda}) \land \text{parents}(\text{karl}, \text{michael}) \land \text{parents}(\text{linda}, \text{michael}) \land \text{parents}(\text{karl}, \text{gerti}) \land \text{parents}(\text{linda}, \text{gerti}) \land $$

$$P^* = R_1^* \land \cdots \land R_m^*$$

with $R_i^* = \forall X_1 \forall X_2 \ldots \forall X_L ((L_1 \land \cdots \land L_n) \Rightarrow L_0)$.

The new facts in $\text{cons}(P^* \land DB^*)$:

- grandpa(hans,michael), grandpa(hans,gerti).

The datalog program $P$ with

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

defines a new relation grandpa with the following tuples:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hans</td>
<td>Michael</td>
</tr>
<tr>
<td>Hans</td>
<td>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: parents(linda,michael), parents(linda,gerti)
- rule: siblings(michael,gerti) :- parents(linda,michael), parents(linda,gerti).

the following fact can be deduced:

$$\text{siblings}(\text{michael}, \text{gerti})$$

### 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 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:

\[ \text{sibling}(Y, Z) := \text{parents}(X, Y), \text{parents}(X, Z), Y \leftrightarrow Z. \]

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

\[
\begin{align*}
\text{sibling}(\text{grete}, \text{grete}) & : - \text{parents}(\text{grete}, \text{grete}), \text{parents}(\text{grete}, \text{grete}), \\
& \quad \text{grete} \leftrightarrow \text{grete} \ (X = Y = Z = \text{grete}) \\
\text{sibling}(\text{grete}, \text{linda}) & : - \text{parents}(\text{grete}, \text{grete}), \text{parents}(\text{grete}, \text{linda}), \\
& \quad \text{grete} \leftrightarrow \text{linda} \ (X = Y = \text{grete}, Z = \text{linda}) \\
\cdots & \\
\text{sibling}(\text{karl}, \text{karl}) & : - \text{parents}(\text{karl}, \text{karl}), \text{parents}(\text{karl}, \text{karl}), \\
& \quad \text{karl} \leftrightarrow \text{karl} \ (X = Y = Z = \text{karl})
\end{align*}
\]

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

- \( T_P^i(DB) \ldots \) set of facts, to which \( \langle T_P(DB) \rangle \) 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 \( T_P^i(DB) \):

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

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:

\[
\text{cons}(P^* \land DB^*) = T_P^i(DB)
\]
Proof of Theorem

Let $P$ be a program and $DB$ a database. We show
\[
\text{cons}(P^* \land DB^*) \subseteq T_P^\omega(DB).
\]

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

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

The inductive step. Suppose $T_P^i(DB) \subseteq \text{cons}(P^* \land DB^*)$ for $i \geq 0$. We show that $T_P^{i+1}(DB) \subseteq \text{cons}(P^* \land DB^*)$. Recall that $T_P^{i+1}(DB) = T_P(T_P^i(DB))$. Thus by the definition of $T_P$,
\[
T_P^{i+1}(DB) = T_P(DB) \cup \bigcup_{R \in P} \{L_0 \mid L_0 \vdash L_1, \ldots, L_n \in \text{Ground}(R, P, DB),
L_1, \ldots, L_n \in T_P^i(DB)\}
\]

(2) We show $\text{cons}(P^* \land DB^*) \subseteq T_P^\omega(DB)$. To this end, we prove that $L \not\in T_P^\omega(DB)$ implies $L \not\in \text{cons}(P^* \land DB^*)$, for any fact $L$. We thus simply show that $T_P^\omega(DB)$ 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 $L \not\in M$ is not a logical consequence of $F$ (as witnessed by $M$ itself).

By the induction hypothesis, $T_P^i(DB) \subseteq \text{cons}(P^* \land DB^*)$. Thus it remains to show that $L_0 \in \text{cons}(P^* \land DB^*)$ for any rule $R \in P$ such that there is $L_0 \vdash L_1, \ldots, L_n \in \text{Ground}(R, P, DB)$ with $L_1, \ldots, L_n \in T_P^i(DB)$.

Assume such a rule $R = L_0^0 \vdash L_1^1, \ldots, L_n^i$ in $P$, and suppose $\pi$ is the substitution of variables with constants such that applying $\pi$ to $R$ results in $L_0^0 \vdash L_1, \ldots, L_n$, i.e. $\pi(L_j^i) = L_j$ for $j \in \{0, \ldots, n\}$.

By construction, in $P^* \land DB^*$ we have the conjunct
\[
R^* = \forall x_1 \forall x_2 \ldots \forall x_l((L_1^i \land L_2^i \land \cdots \land L_n^i) \Rightarrow L_0^0).
\]

Thus, by employing the semantics of classical logic, for any variable substitution $\pi'$ such that $\{\pi'(L_1^1), \ldots, \pi'(L_n^i)\} \subseteq \text{cons}(P^* \land DB^*)$ we also have $\pi'(L_0^0) \in \text{cons}(P^* \land DB^*)$. Since $\pi$ is a substitution such that $\{\pi(L_1^i), \ldots, \pi(L_n^i)\} = \{L_1, \ldots, L_n\} \subseteq \text{cons}(P^* \land DB^*)$ by the induction hypothesis, we get $\pi(L_0^0) = L_0 \in \text{cons}(P^* \land DB^*)$.

$T_P^\omega(DB)$ is a model of $DB^*$ because $DB = T_P^\omega(DB) \subseteq T_P^\omega(DB)$ by the definition of $T_P^\omega(DB)$.

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

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

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

We know $\pi(L_0) \vdash \pi(L_1), \ldots, \pi(L_n) \in \text{Ground}(R, P, DB)$. Thus by the definition of $T_P$, $\pi(L_0) \in T_P(T_P^\omega(DB))$. Since $T_P(T_P^\omega(DB)) = T_P^\omega(DB)$ by the definition of $T_P^\omega(DB)$, we obtain $\pi(L_0) \in T_P^\omega(DB)$. 

Algorithm: INFER

INPUT: datalog program $P$, database $DB$
OUTPUT: $T_p^\infty(DB)$ ($= \text{cons}(P^* \land DB^*)$)

STEP 1. $GP := \bigcup_{R \in P} \text{Ground}(R; P, DB)$, (* $GP$ ... set of all ground instances *)
STEP 2. $OLD := \{\}; NEW := DB$;
STEP 3. while $NEW \neq OLD$ do begin
    $OLD := NEW$;
    $NEW := \text{ComputeTP}(OLD)$;
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 $DB$.

ancestor(X,Y) :- parents(X,Y).
ancestor(X,Z) :- parents(X,Y), ancestor(Y,Z).

Step 1. (INFER) build $GP$

$GP =$
    \begin{align*}
    \{ & \text{ancestor(grete,grete) :- parents(grete,grete),} \\
    & \text{ancestor(grete,linda) :- parents(grete,linda),} \\
    & \ldots, \\
    & \text{ancestor(grete,grete) :- parents(grete,grete),} \\
    & \text{ancestor(grete,grete) :- parents(grete,linda),} \\
    & \text{ancestor(linda,grete),} \\
    & \ldots \}.
    \end{align*}
(There are $6^2 + 6^3 = 252$ ground instances.)

Step 2. $OLD := \{\}, NEW := DB$;
Step 3. $OLD \neq NEW$
    Cycle 1: $OLD := DB, NEW := TP(OLD) = TP(DB)$
        $TP(OLD) = OLD \cup \{\text{ancestor(A,B) | parents(A,B) \in DB}\}$;
    Cycle 2: $OLD := TP(DB), NEW := TP(OLD) = TP(TP(DB))$
        $TP(OLD) = OLD \cup \{\text{ancestor(hans,michael), ancestor(hans,gerti),} \\
        \text{ancestor(grete,michael), ancestor(grete,gerti)}\}$.
    Cycle 3: $TP(OLD) = OLD$, there are no new facts

Step 4. Output of $OLD$.

The result corresponds to the extension of $DB$ with the new table ancestor.
1.6. 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**

\( \text{DEP}(P) \) 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 \( \text{DEP}(P) \) with a star \(" \star \), if \( q \) in the body is negated.

**Definition**

A datalog program \( P \) with negation is called valid if the graph \( \text{DEP}(P) \) has no directed cycle that contains an edge marked with \(" \star \).

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:

\[
\begin{align*}
\text{husband}(X) & :\text{-} \text{man}(X), \text{married}(X). \\
\text{bachelor}(X) & :\text{-} \text{man}(X), \text{not husband}(X).
\end{align*}
\]

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

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

is not stratified.

**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 $DB$, $P$:

$v(X,Y) :- r(X,X), r(Y,Y)$.
$u(X,Y) :- s(X,Y), s(Y,Z), \not v(X,Y)$.
$w(X,Y) :- \not u(X,Y), v(Y,X)$.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$w$</th>
<th>$w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>level 2</td>
</tr>
<tr>
<td>$s$</td>
<td>$v$</td>
<td>level 1</td>
</tr>
<tr>
<td>$r$</td>
<td>$r$, $s$, $v$</td>
<td>level 1</td>
</tr>
</tbody>
</table>
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

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.

p(X,Y) :- source(X), sink(Y), connection(X,Y).

connection(X,X) :- v(X).

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).
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.