2. Introduction to Datalog

Reinhard Pichler
Institut für Informationssysteme
Arbeitsbereich DBAI
Technische Universität Wien
10 March, 2015

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 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):

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

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

```
begin
    result := {};
    newtuples := parents;
    while newtuples ⊈ result do
        begin
            result := result ∪ newtuples;
            newtuples := π₁,4(newtuples[2 = 1]parents);
        end;
    ancestor := result
end.
```

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

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

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

Restrictions on the Datalog Syntax

- 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:

- =, <>, <=, >= 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 have:
\[ DB^* \text{ is a conjunction of ground atoms (i.e., the facts) and} \]
\[ P^* \text{ is a conjunction of formulas with implication} \]

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

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 $\text{cons}(P^* \land DB^*)$:

$$DB^* = \text{REL}_1^* \land \cdots \land \text{REL}_k^*$$ with $\text{REL}_i^* = \text{conjunction of all facts}$

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

$$P^* = R^*_1 \land \cdots \land R^*_m$$ with $R^*_i = \forall x_1 \forall x_2 \cdots \forall x_{\ell_i}(L_1 \land \cdots \land L_n) \Rightarrow L_0$.

$$P^* = \forall X \forall Y \forall Z : ((\text{man}(X) \land \text{parents}(X,Z) \land \text{parents}(Z,Y)) \Rightarrow \text{grandpa}(X,Y)).$$

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 $\text{grandpa}$ 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:

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

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 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:

\[
siblings(Y,Z):= \text{parents}(X,Y), \text{parents}(X,Z), Y<> Z.
\]

All 6³ 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}):

\[
siblings(grete,grete):= \text{parents}(grete,grete), \text{parents}(grete,grete), \\
grete<> grete \quad (X=Y=Z=grete)
\]

\[
siblings(grete,linda):= \text{parents}(grete,grete), \text{parents}(grete,linda), \\
grete<> linda \quad (X=Y=grete, Z=linda)
\]

\[
siblings(karl,karl):= \text{parents}(karl,karl), \text{parents}(karl,karl), \\
karl<> karl \quad (X=Y=Z=karl)
\]

**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 \( DB \).

\[
T_P(DB) = DB \cup \bigcup_{R \in P} \{ L_0 := L_1, \ldots, L_n \mid L_0 := L_1, \ldots, L_n \in \text{Ground}(R; P, DB), \nonumber \] \\
\]

\[
L_1, \ldots, L_n \in DB \}.
\]

\( T_P \) is called the immediate consequence operator.
Properties of $T_P(DB)$

- The set of facts is monotonically increasing e.g. $T_P^i(DB) \subseteq T_P^{i+1}(DB)$
- the sequence $(T_P^i(DB))$ converges finitely: there is $n$ with $T_P^n(DB) = T_P^m(DB)$ for all $m \geq n$.
- $T_P^ω(DB)$ is a set of facts, to which $(T_P^i(DB))$ 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 $T_P^ω(DB)$:

$$O[P] : DB \rightarrow T_P^ω(DB).$$

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: $\text{cons}(P^* \land DB^*) = T_P^ω(DB)$.
(2) We show $\text{cons}(P^* \land \text{DB}^*) \subseteq T_P^*(\text{DB})$. To this end, we prove that $L \not\in T_P^*(\text{DB})$ implies $L \not\in \text{cons}(P^* \land \text{DB}^*)$, for any fact $L$. We thus simply show that $T_P^*(\text{DB})$ is a model of $P^* \land \text{DB}^*$.

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).

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

It remains to show that $T_P^*(\text{DB})$ is also a model of $P^*$. Consider an arbitrary rule $R \in P$. We have to show that $T_P^*(\text{DB})$ 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 $T_P^*(\text{DB})$, i.e., $\{\pi(L_1), \ldots, \pi(L_n)\} \subseteq T_P^*(\text{DB})$.

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

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

Algorithm: INFER

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

STEP 1. $GP := \bigcup_{R \in P} \text{Ground}(R; P, \text{DB})$, (* $GP$ . . . 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 $DB$.

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

Step 1. (INFER) build $GP$

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

(There are $6^2 + 6^3 = 252$ ground instances.)

Step 2. OLD := {}, NEW := DB;

Step 3. OLD $\neq$ NEW

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

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

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

Step 4. Output of OLD.

The result corresponds to the extension of $DB$ with the new table

\[
\text{ancestor}
\]

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:
  \[
  \text{A relation } R \text{ must not be defined on the basis of its negation.}
  \]
- 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.

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:

husband(X) :- man(X), married(X).

bachelor(X) :- man(X), not 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 \( DB, P \):

\[
\begin{align*}
v(X,Y) & : = r(X,X), r(Y,Y). \\
u(X,Y) & : = s(X,Y), s(Y,Z), \text{not } v(X,Y). \\
w(X,Y) & : = \text{not } u(X,Y), v(Y,X).
\end{align*}
\]

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 \( i \).
  \( \Rightarrow \) 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 \( S \):
    \[
    d(X,Y) : = r(X,Y), \text{not } s(X,Y).
    \]
- **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 $Y$.

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

$$\text{connection}(X,X) :- v(X).$$

$$\text{connection}(X,Y) :- e(X,Z), \text{connection}(Z,Y).$$

$$n_{\text{source}}(X) :- e(Y,X).$$

$$\text{source}(X) :- v(X), \text{not} \ n_{\text{source}}(X).$$

$$n_{\text{sink}}(X) :- e(X,Y).$$

$$\text{sink}(X) :- v(X), \text{not} \ n_{\text{sink}}(X).$$

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.