A Datalog Primer

Jef Wijsen

March 19, 2020

Abstract

This document introduces the syntax and semantics of the language *"datalog with stratified negation,"* and shows that programs in this language can be executed in polynomial time. After studying this document, students should be able to write their own programs in this language.

1 Schema and Database Instance

Datalog is a query language for relational databases. A *relational database schema* (or *schema* for short) is a finite set of relation names with associated arities. We write R/n to indicate that R is a relation name of arity n.

Example 1.1. In our running example, we will use a schema that contains $\mathit{Knows}/2$ and $\mathit{Owns}/2$. $\square$

A *fact* is an expression $R(c_1, \dots, c_n)$ where each c_i is a constant. A *database instance* I is a finite set of facts.

Example 1.2. The set $I = \{\mathit{Knows}(\texttt{Jeb}, \texttt{Don}), \mathit{Knows}(\texttt{Don}, \texttt{Jeb}), \mathit{Knows}(\texttt{An}, \texttt{Don}), \mathit{Knows}(\texttt{Ed}, \texttt{An}), \mathit{Owns}(\texttt{Don}, \texttt{iPad}), \mathit{Owns}(\texttt{Don}, \texttt{iPod}), \mathit{Owns}(\texttt{Jeb}, \texttt{iPod})\}$ is a database instance. $\square$

The term *EDB predicate* is often used as a synonym for "relation name of the relational database schema." The letter E in *EDB* stands of *Extensional.* I like to think about it as *Existential*: that what really *Exists* in the database.

Example 1.3. *Knows* and *Owns* are EDB predicates. $\square$

As a minor comment, in mathematical logic, one often uses the term *predicate symbol* for what I call a *predicate* in this document.

2 Datalog Syntax

We assume a set of *IDB predicates* with associated arities. The IDB predicates are distinct from the EDB predicates. The letter I in *IDB* stands of *Intensional.* I like to think about it as *Inferred*, because datalog is all about *Inferring* IDB facts from a database instance of EDB facts.

A fact $P(c_1, \dots, c_n)$ is an *EDB fact* if P is an EDB predicate, and is an *IDB fact* if P is an IDB predicate. An *atom* is an expression $P(t_1, \dots, t_n)$ where each t_i is a constant or a variable. The letter t refers to *term*, which is a variable or a constant.

A *datalog rule* is an expression

$$\underbrace{H(\vec{t_0})}_{\textit{rule head}} \quad \leftarrow \quad \underbrace{R_1(\vec{t_1}), \dots, R_\ell(\vec{t_\ell}), \neg R_{\ell+1}(\vec{t}_{\ell+1}), \dots, \neg R_m(\vec{t_m})}_{\textit{rule body}} \tag{1}$$

where

- H is an IDB predicate,
- each R_i is an EDB predicate or an IDB predicate, and
- each $\vec{t_i}$ is a sequence of terms (of appropriate length).

Note that the same predicate can occur more than once in a rule. Within a rule, we distinguish between the *head* and the *body*, as indicated in (1). Every rule must be *safe*:

Safety Requirement: Every variable that occurs in a rule (in its head or its body), must occur in a nonnegated atom of the rule body.

Informally, the safety requirement guarantees that every variable can be restricted to a finite domain.

Significantly, EDB predicates cannot be used in rule heads. A *datalog program* $\mathcal{P}$ is a finite set of rules, which can be written in any order.

Example 2.1. Assume IDB predicates *Happy*/1 and *Unhappy*/1.

$$\begin{aligned}
\mathit{Happy}(x) &\leftarrow \mathit{Owns}(x, \mathtt{iPad}), \mathit{Owns}(x, \mathtt{iPod})\\
\mathit{Happy}(x) &\leftarrow \mathit{Knows}(x, y), \mathit{Happy}(y)\\
\mathit{Happy}(x) &\leftarrow \mathit{Knows}(x, y), \neg\mathit{Knows}(x, \mathtt{Don})\\
\mathit{Unhappy}(x) &\leftarrow \mathit{Owns}(x, y), \neg\mathit{Happy}(x)
\end{aligned}$$

It can be easily seen that all rules are safe. In the last rule, the variable x is restricted to the constants that occur at the first position of some *Owns*-fact. □

Informally, the semantics is as follows. An *instantiation* of a rule is obtained by replacing all variables with constants. Such an instantiation is called a *ground rule*. Then, whenever the body of a ground rule is true, we infer that its rule head is also true. The body of a ground rule is true if every nonnegated fact is true, and every negated fact is false.

Example 2.2. Consider the program of Example 2.1. An instantiation of the first rule is:

$$\mathit{Happy}(\mathtt{Don}) \leftarrow \mathit{Owns}(\mathtt{Don}, \mathtt{iPad}), \mathit{Owns}(\mathtt{Don}, \mathtt{iPod})$$

Since $\mathit{Owns}(\mathtt{Don}, \mathtt{iPad})$ and $\mathit{Owns}(\mathtt{Don}, \mathtt{iPod})$ are in the database instance, they are both true. Therefore, the body is true, and we can infer that $\mathit{Happy}(\mathtt{Don})$ is true. An instantiation of the second rule is:

$$\mathit{Happy}(\mathtt{Jeb}) \leftarrow \mathit{Knows}(\mathtt{Jeb}, \mathtt{Don}), \mathit{Happy}(\mathtt{Don})$$

Since $\mathit{Knows}(\mathtt{Jeb}, \mathtt{Don})$ is in the database instance, and $\mathit{Happy}(\mathtt{Don})$ has been inferred to be true, we can infer that $\mathit{Happy}(\mathtt{Jeb})$ is true. An instantiation of the third rule is:

$$\mathit{Happy}(\mathtt{Ed}) \leftarrow \mathit{Knows}(\mathtt{Ed}, \mathtt{An}), \neg\mathit{Knows}(\mathtt{Ed}, \mathtt{Don})$$

Since $\mathit{Knows}(\mathtt{Ed}, \mathtt{An})$ is in the database instance, and $\mathit{Knows}(\mathtt{Ed}, \mathtt{Don})$ is false (because it is not in the database instance), we can infer that $\mathit{Happy}(\mathtt{Ed})$ is true. □

3 Stratified Negation

3.1 The Problem of "Circular" Negation

If P is an EDB predicate, then the truth of a fact $P(c_1, \ldots, c_n)$ is naturally defined as follows:

- $P(c_1, \ldots, c_n)$ is true if it is in the database instance; and
- $P(c_1, \ldots, c_n)$ is false (and therefore $\neg P(c_1, \ldots, c_n)$ is true) if $P(c_1, \ldots, c_n)$ is not in the database instance.

This way of defining truth is also called the *Closed World Assumption (CWA)*.

If P is an IDB predicate, then it is trickier to determine the truth of a fact $P(c_1, \ldots, c_n)$. Consider the following program:

$$\begin{aligned}
\mathit{Man}(x) &\leftarrow \mathit{Owns}(x, y), \neg\mathit{Female}(x)\\
\mathit{Female}(x) &\leftarrow \mathit{Owns}(x, y), \neg\mathit{Man}(x)
\end{aligned}$$

We can obtain the following ground rules:

$$\begin{aligned}
\mathit{Man}(\mathtt{Jeb}) &\leftarrow \mathit{Owns}(\mathtt{Jeb}, \mathtt{iPod}), \neg\mathit{Female}(\mathtt{Jeb})\\
\mathit{Female}(\mathtt{Jeb}) &\leftarrow \mathit{Owns}(\mathtt{Jeb}, \mathtt{iPod}), \neg\mathit{Man}(\mathtt{Jeb})
\end{aligned}$$

Note incidentally that every program without variables is necessarily safe. Since $\mathit{Owns}(\texttt{Jeb}, \texttt{iPod})$ is true (because it is in the database), we can simplify these ground rules as follows:

$$\begin{aligned} \mathit{Man}(\texttt{Jeb}) &\leftarrow \neg \mathit{Female}(\texttt{Jeb}) \\ \mathit{Female}(\texttt{Jeb}) &\leftarrow \neg \mathit{Man}(\texttt{Jeb}) \end{aligned}$$

This program is cumbersome: the first rule allows us to infer $\mathit{Man}(\texttt{Jeb})$ if we cannot infer $\mathit{Female}(\texttt{Jeb})$; and the second rule allows us to infer $\mathit{Female}(\texttt{Jeb})$ if we cannot infer $\mathit{Man}(\texttt{Jeb})$. This leads to an impossible situation: the first rule cannot be executed before the second rule, and the second rule cannot be executed before the first rule. Significantly, a datalog program does not depend on the order in which its rules are listed. So the previous program is really the same as:

$$\begin{aligned} \mathit{Female}(\texttt{Jeb}) &\leftarrow \neg \mathit{Man}(\texttt{Jeb}) \\ \mathit{Man}(\texttt{Jeb}) &\leftarrow \neg \mathit{Female}(\texttt{Jeb}) \end{aligned}$$

3.2 Stratified Negation

A *stratification* of a datalog program assigns a nonnegative integer, called *stratum*, to every IDB predicate of the program such that for every program rule

$$H(\vec{t_0}) \quad \leftarrow \quad R_1(\vec{t_1}), \dots, R_\ell(\vec{t_\ell}), \neg R_{\ell+1}(\vec{t}_{\ell+1}), \dots, \neg R_m(\vec{t_m})$$

the following holds:

Stratification Requirement:

- for every $i \in \{1, \dots, \ell\}$, if R_i is an IDB predicate, then the stratum of R_i is smaller than or equal to the stratum of H; and
- for every $i \in \{\ell + 1, \dots, m\}$, if R_i is an IDB predicate, then the stratum of R_i is strictly smaller than the stratum of H.

That is, if the stratum of the head predicate H is the integer h, then all IDB predicates in the rule body must have a stratum that is $\leq h$. Moreover, negated IDB predicates in the rule body must have a stratum $< h$.

Not all datalog programs have a stratification. In particular, we claim that there is no stratification for:

$$\begin{aligned} \mathit{Man}(x) &\leftarrow \mathit{Owns}(x, y), \neg \mathit{Female}(x) \\ \mathit{Female}(x) &\leftarrow \mathit{Owns}(x, y), \neg \mathit{Man}(x) \end{aligned}$$

To prove our claim, assume that we assign stratum m to *Man*, and stratum f to *Female*. Then, the Stratification Requirement applied to the first rule implies $f < m$. Likewise, the Stratification Requirement applied to the second rule implies $m < f$. But no stratification can possibly satisfy both $f < m$ and $m < f$.

We say that a datalog program is *stratified* if it has a stratification. From now on, we will reject all programs that are not stratified. If we have a stratification, we will assume without loss of generality that the strata are numbered $0, 1, 2 \dots$

Example 3.1. We compute a stratification for the program:

$$\begin{aligned} \mathit{Happy}(x) &\leftarrow \mathit{Owns}(x, \texttt{iPad}), \mathit{Owns}(x, \texttt{iPod}) \\ \mathit{Happy}(x) &\leftarrow \mathit{Knows}(x, y), \mathit{Happy}(y) \\ \mathit{Happy}(x) &\leftarrow \mathit{Knows}(x, y), \neg \mathit{Knows}(x, \texttt{Don}) \\ \mathit{Unhappy}(x) &\leftarrow \mathit{Owns}(x, y), \neg \mathit{Happy}(x) \end{aligned}$$

The Stratification Requirement only talks about IDB predicates, and therefore we can ignore the EDB predicates *Owns* and *Knows*. Let the stratum of *Happy* be h, and the stratum of *Unhappy* be u. Because of the second program rule, we must have $h \leq h$, which is vacuously satisfied. Because of the last program rule, we must have $h < u$. We obtain a valid stratification by letting $h = 0$ and $u = 1$. This stratification is summarized in the following table:

$$\begin{array}{ll} \text{stratum 0:} & \textit{Happy} \\ \text{stratum 1:} & \textit{Unhappy} \end{array}$$

□

3.3 Semantics of Stratified Negation

Assume we have computed a stratification for a stratified datalog program $\mathcal{P}$, and let m be the greatest stratum in our stratification. For every $h \in \{0, 1, \ldots, m\}$, we write $\mathcal{P}^{(h)}$ for the set of rules whose head predicate has stratum h.

Example 3.2. For the program $\mathcal{P}$ of Example 3.1, the subprograms $\mathcal{P}^{(0)}$ and $\mathcal{P}^{(1)}$ are as follows:

$$\begin{array}{lcl} \mathcal{P}^{(0)} & = & \left\{ \begin{array}{lcl} \mathit{Happy}(x) & \leftarrow & \mathit{Owns}(x, \texttt{iPad}), \mathit{Owns}(x, \texttt{iPod}) \\ \mathit{Happy}(x) & \leftarrow & \mathit{Knows}(x, y), \mathit{Happy}(y) \\ \mathit{Happy}(x) & \leftarrow & \mathit{Knows}(x, y), \neg \mathit{Knows}(x, \texttt{Don}) \end{array} \right. \\ \mathcal{P}^{(1)} & = & \left\{ \begin{array}{lcl} \mathit{Unhappy}(x) & \leftarrow & \mathit{Owns}(x, y), \neg \mathit{Happy}(x) \end{array} \right. \end{array}$$

□

The datalog program is then executed in the following order:

- Execute $\mathcal{P}^{(0)}$.
- Execute $\mathcal{P}^{(1)}$.
- Execute $\mathcal{P}^{(2)}$.
- ...
- Execute $\mathcal{P}^{(m)}$.

When we say "Execute $\mathcal{P}^{(h)}$," we mean to repeatedly instantiate all rules of $\mathcal{P}^{(h)}$ in all possible ways. That is, repeatedly compute ground rules by replacing variables with constants. Whenever the body of such a ground rule is true, infer that its rule head is also true. When the body of a rule in $\mathcal{P}^{(h)}$ contains a negated atom $\neg R_i(\vec{t_i})$, then the Stratification Requirement tells us that R_i is either an IDB predicate with stratum $< h$, or an EDB predicate. Therefore, when we execute $\mathcal{P}^{(h)}$, all rules that can compute R_i-facts have already been executed, and we can safely say that a fact $\neg R_i(\vec{c})$ is true if $R_i(\vec{c})$ has not been inferred. In other words, when executing $\mathcal{P}^{(h)}$, we can treat negated predicates as if they were EDB predicates. We will have more to say about program execution in Section 6.

Note that the rules of a stratified datalog program can be listed in any order. The stratification is the responsibility of the *datalog engine* that executes the program.

Example 3.3. The following program computes the complement of the transitive closure of *Knows*.

$$\begin{array}{rcl} \mathit{Person}(x) & \leftarrow & \mathit{Knows}(x, y) \\ \mathit{Person}(y) & \leftarrow & \mathit{Knows}(x, y) \\ \mathit{TransitivelyKnows}(x, y) & \leftarrow & \mathit{Knows}(x, y) \\ \mathit{TransitivelyKnows}(x, y) & \leftarrow & \mathit{Knows}(x, z), \mathit{TransitivelyKnows}(z, y) \\ \mathit{NotTransitivelyKnows}(x, y) & \leftarrow & \mathit{Person}(x), \mathit{Person}(y), \neg \mathit{TransitivelyKnows}(x, y) \end{array}$$

Let the strata of *Person*, *TransitivelyKnows*, and *NotTransitivelyKnows* be p, t, and n respectively. The Stratification Requirement requires $p \leq n$ and $t < n$. These inequalities are satisfied by the following stratification:

$$\begin{array}{ll} \text{stratum 0:} & \textit{Person}, \textit{TransitivelyKnows} \\ \text{stratum 1:} & \textit{NotTransitivelyKnows} \end{array}$$

This stratification gives the following programs $\mathcal{P}^{(0)}$ and $\mathcal{P}^{(1)}$:

$$\mathcal{P}^{(0)} = \left\{ \begin{array}{rcl} \mathit{Person}(x) & \leftarrow & \mathit{Knows}(x,y) \\ \mathit{Person}(y) & \leftarrow & \mathit{Knows}(x,y) \\ \mathit{TransitivelyKnows}(x,y) & \leftarrow & \mathit{Knows}(x,y) \\ \mathit{TransitivelyKnows}(x,y) & \leftarrow & \mathit{Knows}(x,z), \mathit{TransitivelyKnows}(z,y) \end{array} \right.$$

$$\mathcal{P}^{(1)} = \left\{ \mathit{NotTransitivelyKnows}(x,y) \leftarrow \mathit{Person}(x), \mathit{Person}(y), \neg\, \mathit{TransitivelyKnows}(x,y) \right.$$

Therefore, a datalog engine will first execute the rules of $\mathcal{P}^{(0)}$ until no more facts can be inferred. Then, and only then, $\mathcal{P}^{(1)}$ is executed. Therefore, when $\mathcal{P}^{(1)}$ is executed, all *TransitivelyKnows*-facts have already been computed, and therefore we know which *TransitivelyKnows*-facts are not true.

Note incidentally that a program can have more than one valid stratification. Here is another stratification for this program:

stratum 0:	*TransitivelyKnows*
stratum 1:	*Person*, *NotTransitivelyKnows*

It can be shown that the output of a stratified datalog program does not depend on the stratification that is chosen to execute the program: all stratifications result in the same output. □

To conclude, every stratified datalog program can be regarded as a sequence of smaller datalog programs, one for each stratum, that only use "classical" negation.

Example 3.4. A bus company called *Red Cy* uses a binary EDB predicate *Red*/2 to store its direct bus connections. Moreover, a predicate *RedCanceled*/2 stores the connections that are currently canceled due to a virus outbreak. For example, $I = \{\mathit{Red}(\texttt{Mons},\texttt{Ath}), \mathit{Red}(\texttt{Ath},\texttt{Dour}), \mathit{Red}(\texttt{Dour},\texttt{Mons}), \mathit{Red}(\texttt{Mons},\texttt{Huy}), \mathit{Red}(\texttt{Ans},\texttt{Mons}), \mathit{Red}(\texttt{Huy},\texttt{Ans}), \mathit{Red}(\texttt{Ans},\texttt{Spa}), \mathit{Red}(\texttt{Spa},\texttt{Huy}), \mathit{RedCanceled}(\texttt{Ans},\texttt{Mons})\}$. We want to define an IDB predicate *CanAlwaysReturn*/1 such that $\mathit{CanAlwaysReturn}(x)$ holds true if on any journey that starts from x, one can always return to x. For our example database instance, *CanAlwaysReturn*(`Mons`) is false, because when one travels from `Mons` to `Huy`, one cannot return to `Mons`. On the other hand, *CanAlwaysReturn*(`Huy`) is true: it is possible to return to `Huy` from every station currently reachable from `Huy`.

When computing a set in datalog, it is sometimes easier to first compute the complement of that set. Let us introduce *CannotAlwaysReturn*/1 with the meaning that $\mathit{CannotAlwaysReturn}(x)$ is false if and only if $\mathit{CanAlwaysReturn}(x)$ is true. So our program will contain the following rules:

$$\begin{array}{rcl} \mathit{CanAlwaysReturn}(x) & \leftarrow & \mathit{Station}(x), \neg \mathit{CannotAlwaysReturn}(x) \\ \mathit{Station}(x) & \leftarrow & \mathit{Red}(x,y) \\ \mathit{Station}(y) & \leftarrow & \mathit{Red}(x,y) \end{array}$$

The IDB predicate *Station* is useful for making the first rule safe. We are now ready to give the rules for $\mathit{CannotAlwaysReturn}(x)$, which is true if one can travel from x to some station y, but there is no journey from y back to x.

$$\begin{array}{rcl} \mathit{RedTrip}(x,y) & \leftarrow & \mathit{Red}(x,y), \neg \mathit{RedCanceled}(x,y) \\ \mathit{RedTrip}(x,y) & \leftarrow & \mathit{Red}(x,z), \mathit{RedTrip}(z,y), \neg \mathit{RedCanceled}(x,z) \\ \mathit{CannotAlwaysReturn}(x) & \leftarrow & \mathit{RedTrip}(x,y), \neg \mathit{RedTrip}(y,x) \end{array}$$

A stratification of this program is as follows:

stratum 0:	*Station*, *RedTrip*
stratum 1:	*CannotAlwaysReturn*
stratum 2:	*CanAlwaysReturn*

This stratification gives the following programs $\mathcal{P}^{(0)}$, $\mathcal{P}^{(1)}$, and $\mathcal{P}^{(2)}$:

$$\mathcal{P}^{(0)} = \left\{ \begin{array}{rcl} \mathit{Station}(x) & \leftarrow & \mathit{Red}(x,y) \\ \mathit{Station}(y) & \leftarrow & \mathit{Red}(x,y) \\ \mathit{RedTrip}(x,y) & \leftarrow & \mathit{Red}(x,y), \neg \mathit{RedCanceled}(x,y) \\ \mathit{RedTrip}(x,y) & \leftarrow & \mathit{Red}(x,z), \mathit{RedTrip}(z,y), \neg \mathit{RedCanceled}(x,z) \end{array} \right.$$

$$\mathcal{P}^{(1)} = \left\{ \mathit{CannotAlwaysReturn}(x) \leftarrow \mathit{RedTrip}(x,y), \neg \mathit{RedTrip}(y,x) \right.$$

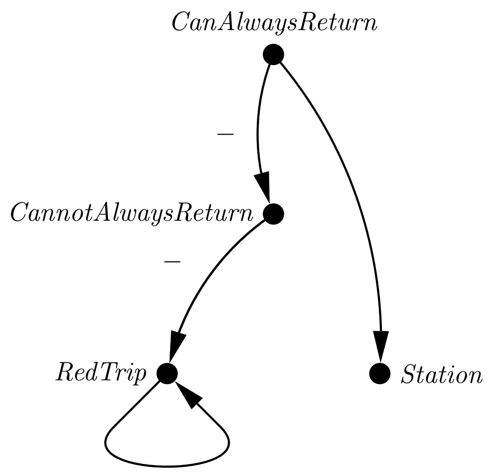
$$\mathcal{P}^{(2)} = \left\{ \mathit{CanAlwaysReturn}(x) \leftarrow \mathit{Station}(x), \neg \mathit{CannotAlwaysReturn}(x) \right.$$

Nonetheless, programmers can write the rules of their program in any order, and the output is independent of the chosen order. □

4 Program Dependency Graph

The *program dependency graph (PDG)* of a datalog program is a directed edge-labeled graph whose vertices are the IDB predicates of the program. There is a directed edge from H to P if there is a rule such that H is the head predicate, and P is an IDB predicate that occurs in the body; such an edge is labeled $-$ if P occurs within the scope of $\neg$.

For example, for the program of Example 3.4, the PDG is as follows:



It is easy to show the following property:

> A program $\mathcal{P}$ is stratified if and only if its PDG has no directed cycle that contains an edge with label $-$.

Consequently, in the PDG of a stratified datalog program, any directed (possibly infinite) path that starts from some vertex can traverse at most finitely many edges with label $-$. It is then easy to show the following:

> Let f be the function that assigns to each IDB predicate P the greatest number of edges with label $-$ on any directed path that starts from P. We have just argued that this number is finite. Then, f is a valid stratification. For example, in the preceding PDG, f assigns 2 to *CanAlwaysReturn*, 1 to *CannotAlwaysReturn*, and 0 to both *RedTrip* and *Station*.

5 Datalog Languages

In the literature, the name *"datalog"* is usually used only for programs that do not contain negation. For programs with stratified negation, one commonly uses *"stratified datalog"* or *"datalog with stratified negation."*

"Datalog with $\neq$" refers to datalog without negation, but with $\neq$. In "datalog with stratified negation," we can compute a *NotEqual*/2 predicate, and therefore "datalog with stratified negation and $\neq$" is not more powerful than "datalog with stratified negation." For example,

$$\begin{array}{rcl} \mathit{Equal}(x,x) & \leftarrow & \mathit{Person}(x) \\ \mathit{NotEqual}(x,y) & \leftarrow & \mathit{Person}(x), \mathit{Person}(y), \neg\mathit{Equal}(x,y) \end{array}$$

A (stratified) datalog program is called *linear* if in the body of each rule, there is at most one occurrence of an IDB predicate that has the same stratum as the rule head.[1] For example, the following rule cannot

[1]Note that, by the Stratification Requirement, such an IDB predicate cannot occur within the scope of $\neg$ in the rule body.

occur in a linear datalog program:

$$\mathit{TransitivelyKnows}(x,y) \quad\leftarrow\quad \mathit{TransitivelyKnows}(x,z), \mathit{TransitivelyKnows}(z,y). \tag{2}$$

A linear datalog program can contain the following rule:

$$\mathit{TransitivelyKnows}(x,y) \quad\leftarrow\quad \mathit{Knows}(x,z), \mathit{TransitivelyKnows}(z,y). \tag{3}$$

It is generally a good idea to solve a problem by a linear datalog program whenever this is possible, because such programs can be executed more efficiently than nonlinear programs. Informally, (2) is less efficient than (3) because it uses two recursive calls.

A program is in *semi-positive datalog* if negation only applies to EDB predicates.

6 Analysis of Program Execution

In Section 3.3, we pointed out that a program $\mathcal{P}$ is executed by successively executing $\mathcal{P}^{(0)}$, $\mathcal{P}^{(1)}$, $\mathcal{P}^{(2)}$,.... In this section, we discuss in more detail the execution of these subprograms. In particular, for any fixed stratified datalog program $\mathcal{P}$, we will discuss the complexity of the following computational problem:

INPUT: A database instance.

OUTPUT: All IDB facts that can be inferred by $\mathcal{P}$.

We will show that this problem is in polynomial time in the size of the input. Note that $\mathcal{P}$ itself is not part of the input.

6.1 Guiding Example

We start by an example explaining the execution of $\mathcal{P}^{(0)}$ in Example 3.4. This execution is captured by the following pseudo-code:

```
    $Station^0$ := $\emptyset$
    $RedTrip^0$ := $\emptyset$
    $i$ := $0$
loop
    $Station^{i+1}$ := $\{x \mid \exists y Red(x,y)\}$
    $Station^{i+1}$ := $\{y \mid \exists x Red(x,y)\}$
    $RedTrip^{i+1}$ := $\{x,y \mid Red(x,y) \land \neg RedCanceled(x,y)\}$
    $RedTrip^{i+1}$ := $\{x,y \mid \exists z \left(Red(x,z) \land RedTrip^i(z,y) \land \neg RedCanceled(x,z)\right)\}$
    $i$ := $i+1$
until a fixed point is reached
```

This pseudo-code initializes the IDB predicates as empty. The loop then repeatedly assigns new values to the IDB predicates. The queries at the right-hand of := are relational calculus queries, which can be implemented in relational algebra or SQL, and executed in polynomial time (and even in logarithmic space). For example, the query

$$\{x,y \mid \exists z \left(\mathit{Red}(x,z) \land \mathit{RedTrip}^i(z,y) \land \neg\mathit{RedCanceled}(x,z)\right)\} \tag{4}$$

is equivalent to (using numbers for attributes):

```
SELECT   Red.1, RedTrip^i.2
FROM     Red, RedTrip^i
WHERE    Red.2 = RedTrip^i.1
AND      NOT EXISTS (  SELECT   *
                       FROM     RedCanceled
                       WHERE    RedCanceled.1 = Red.1
                       AND      RedCanceled.2 = Red.2 )
```

It is easy to see:

- $\mathit{RedTrip}^0 \subseteq \mathit{RedTrip}^1$; and

Red

1	2
Mons	Ath
Ath	Dour
Dour	Mons
Mons	Huy
Ans	Mons
Huy	Ans
Ans	Spa
Spa	Huy

RedCanceled

1	2
Ans	Mons

$Station^0$

1

$RedTrip^0$

1	2

$Station^1$

1
Mons
Ath
Dour
Huy
Ans
Spa

$RedTrip^1$

1	2
Mons	Ath
Ath	Dour
Dour	Mons
Mons	Huy
Huy	Ans
Ans	Spa
Spa	Huy

$Station^2$

1
Mons
Ath
Dour
Huy
Ans
Spa

$RedTrip^2$

1	2
Mons	Ath
Ath	Dour
Dour	Mons
Mons	Huy
Huy	Ans
Ans	Spa
Spa	Huy
Mons	Dour
Ath	Mons
Dour	Ath
Dour	Huy
Mons	Ans
Huy	Spa
Ans	Huy
Spa	Ans

⋮ ⋮

Figure 1: First two iterations in a fixed point computation.

- if $\mathit{RedTrip}^{i} \subseteq \mathit{RedTrip}^{i+1}$, then $\mathit{RedTrip}^{i+1} \subseteq \mathit{RedTrip}^{i+2}$.

Informally, the second item is true because in (4), $\mathit{RedTrip}^{i}$ does not occur within the scope of a negation. From these two items, it follows:

$$\mathit{RedTrip}^{0} \subseteq \mathit{RedTrip}^{1} \subseteq \mathit{RedTrip}^{2} \subseteq \mathit{RedTrip}^{3} \subseteq \cdots$$

The first steps of this computation are shown in Figure 1. Since there are finitely many stations, this sequence must necessarily reach a fixed point. Note that the sequence for *Station* already reaches a fixed point after one iteration.

We now argue that a fixed point will be reached after a number of steps that is polynomial in the size of the database instance. To this end, assume that the database instance has size ℓ. Then, there can be at most ℓ distinct stations, and at most ℓ^2 distinct *RedTrip*-facts. Therefore, it must be the case that $\mathit{RedTrip}^{\ell^2} = \mathit{RedTrip}^{\ell^2+1}$.

6.2 General Treatment

We briefly sketch how the treatment of Section 6.1 generalizes to some $\mathcal{P}^{(h)}$, where h is any stratum. A pseudo-code can be constructed as in Section 6.1, as follows. Assume that $\mathcal{P}^{(h)}$ contains a rule

$$H(\vec{t_0}) \quad\leftarrow\quad R_1(\vec{t_1}),\dots,R_\ell(\vec{t_\ell}),\neg R_{\ell+1}(\vec{t}_{\ell+1}),\dots,\neg R_m(\vec{t_m}),$$

where the IDB predicates of stratum h are H, $R_1, R_2, \dots, R_k$ $(1 \leq k \leq \ell)$, while $R_{k+1},\dots,R_m$ are either IDB predicates of a smaller stratum or EDB predicates. In particular, $R_{\ell+1},\dots,R_m$ cannot be IDB predicates of stratum h because of the Stratification Requirement. Then, the **loop** of the pseudo-code contains an instruction:

$$H^{i+1} \quad := \quad \left\{ \vec{t_0} \;\middle|\; \exists\vec{x}\left(\begin{array}{l} R_1^{i}(\vec{t_1}) \wedge \cdots \wedge R_k^{i}(\vec{t_k}) \wedge \\ R_{k+1}(\vec{t}_{k+1}) \wedge \cdots \wedge R_\ell(\vec{t_\ell}) \wedge \neg R_{\ell+1}(\vec{t}_{\ell+1}) \wedge \cdots \wedge \neg R_m(\vec{t_m}) \end{array} \right) \right\}, \qquad (5)$$

where $\vec{x}$ contains the variables that are not in $\vec{t_0}$.

Since the IDB predicates of stratum h (i.e., $R_1^i,\dots,R_k^i$ in the query (5)) do not occur within the scope of a negation, the pseudo-code must necessarily reach a fixed point. After how many steps will this fixed point be reached? Let n be the arity of H, and assume a database instance of size ℓ. Since such a database instance can contain at most ℓ constants, we can generate at most ℓ^n distinct H-facts from this database, which is a polynomial number (because n is fixed). It follows that a fixed point will be reached in polynomial time in the size of the database. To conclude, the program $\mathcal{P}^{(h)}$ runs in polynomial time. Since the number of strata is constant (it is not part of the input), every stratified datalog program can be evaluated in polynomial time.

7 Datalog Engine

A datalog engine called DLV can be downloaded at `http://www.dlvsystem.com/`. To run the program of Example 3.4, we create the text file `brol.txt` as shown in Figure 2. Constants start with a lowercase letter, and variables with an uppercase letter. More information on the DLV syntax can be found at `http://www.dlvsystem.com/html/DLV_User_Manual.html#AEN928`. Note that DLV supports a larger language than stratified datalog with $\neq$. For example, DLV accepts programs in which negation is not stratified. However, in the course *Bases de Données II*, the aim is to use only stratified datalog. When we execute our program by means of the command

```
dlv -filter=CanAlwaysReturn brol.txt
```

we get the following output:

```
{CanAlwaysReturn(huy), CanAlwaysReturn(ans), CanAlwaysReturn(spa)}
```

For help, use the command `dlv -help`.

A common situation is where we want to separate the database from the datalog program, so that the same database can be used with different programs. The following command executes the program `pgm.txt` on the database in `db.txt`:

```
dlv db.txt pgm.txt
```

```
% This is file brol.txt
% The database.
Red(mons, ath).
Red(ath, dour).
Red(dour, mons).
Red(mons, huy).
Red(ans, mons).
Red(huy, ans).
Red(ans, spa).
Red(spa, huy).
RedCanceled(ans, mons).

% The program.
CanAlwaysReturn(X) :- Station(X), not CannotAlwaysReturn(X).
Station(X) :- Red(X,Y).
Station(Y) :- Red(X,Y).
Redtrip(X,Y) :- Red(X,Y), not RedCanceled(X,Y).
Redtrip(X,Y) :- Red(X,Z), Redtrip(Z,Y), not RedCanceled(X,Z).
CannotAlwaysReturn(X) :- Redtrip(X,Y), not Redtrip(Y,X).
```

Figure 2: The program of Example 3.4 in DLV syntax.