CS 784: Foundations of Data Management *Instructor: Paris Koutris*

Lecture 10: Datalog with Negation

In this lecture we will study the addition of negation to Datalog. We start with an example.

Example 1

Suppose that we want to compute the complement of graph transitive closure (i.e. output all edges (a, b) such that there is no directed path from a to b). We can compute this using the following Datalog program:

```
V(x)    :- R(x,y).
V(y)    :- R(x,y).
T(x,y)  :- R(x,y).
T(x,y)  :- T(x,z), R(z,y).
TC(x,y):- V(x), V(y), not T(x,y).
```

Notice that we now allow negation in the body of the rules. Similar to the case of Conjunctive Queries with negation, we have to be careful that the negation is *safe*: this means that every variable that appears in a negated atom must also appear in a positive atom. We also do not allow negation to appear in the head of a rule. We call this new language Datalog$^\neg$.

The semantics of (positive) Datalog are well-defined: there exists a unique minimum model, and we can apply the fixpoint algorithm to iteratively find this model. However, when we add negation, finding the right semantics becomes a harder task. We investigate this next.

The first attempt to define the semantics would be by extending the fixpoint semantics. The immediate consequence operator can still be defined as with positive Datalog (it will be an RA expression with negation). However, the fixpoint is not well-defined in the case of Datalog with negation! Consider the following examples:

Example 2

Consider the following program in Datalog$^\neg$: $R(x)$:- $S(x), \neg R(x)$, and suppose that the EDB S has input $\{S(1)\}$. In this case, neither $\{R(1)\}$ nor $\{\}$ are fixpoints! In fact, the above program does not have any fixpoint if we follow the standard definition.

Example 3

Consider the following program in Datalog$^\neg$: $R(x)$:- $S(x), \neg T(x)$ and $T(x)$:- $S(x), \neg R(x)$, with input $\{S(1)\}$. In this case, we have two fixpoints: $\{R(1)\}$ and $\{T(1)\}$. None of the two fixpoints is contained in the other, so they are both minimal fixpoints!

The above two examples show that defining semantics for Datalog$^\neg$ is a challenging task. We will see next several different ways that we can define them.

10.1 Semi-positive Datalog

In *semi-positive Datalog*, we are allowed to have negation only in front of EDB relations. To define the semantics for semi-positive Datalog, we interpret $\neg R(\vec{x})$ as follows: $\neg R(\vec{x})$ is true if and only if $\vec{x}$ is in the active domain and $\vec{x}$ is not in R. The standard iterative computation works for semi-positive Datalog, since the negated EDB relations can essentially be replaced by a positive relation (which is the complement).

Example 4

Consider the following Datalog$^\neg$ program:

```
RC(x,y) :- V(x), V(y), not R(x,y).
T(x,y)  :- RC(x,y).
T(x,y)  :- T(x,z), RC(z,y).
```

where the EDB R is: $\{R(1,2), R(2,3), R(1,1), R(1,3)\}$. We can compute the complement $\bar{R}$ of R as: $\{R(2,1), R(2,2), R(3,3), R(3,2), R(3,1)\}$. We can now replace R with $\bar{R}$ in the Datalog program with the complementary input, and run it simply as a positive Datalog program.

10.2 Stratified Datalog

A stratified Datalog program with negation is a generalization of the idea behind semi-positive Datalog. In semi-positive Datalog, the negation is only in front of EDB relations. We can now try to add negation in front of an IDB predicate that is the output of a semi-positive program (when there is no mutual recursion), and so on. Formally, we define a *stratification* of the Datalog program as follows.

Definition 1: Stratification

A *stratification* of a Datalog$^\neg$ program P is an ordered partition of P into Datalog subprograms $P^1, P^2, \ldots, P^m$ such that for any two rules r, r' with heads R, R' that belong in partitions $P^i, P^{i'}$ respectively the following hold:

- If $R = R'$ then $i = i'$ (i.e., both rules are in the same partition).
- If R appears positive in the body of r', then $i \leq i'$.
- If R appears negatively in the body of r', then $i < i'$.

Example 5

Consider the Datalog$^\neg$ program from the first example. This program can be stratified in 3 Datalog subprograms as follows:

```
(P1) V(x)    :- R(x,y).
(P1) V(y)    :- R(x,y).
(P2) T(x,y)  :- R(x,y).
(P2) T(x,y)  :- T(x,z), R(z,y).
(P3) TC(x,y):- V(x), V(y), not T(x,y).
```

Notice that (P2) defines the IDB T, and (P3) defines TC; TC has a negated IDB in the body of a rule that defines it, but this IDB appears in the previous stratum (P2).

A Datalog$^\neg$ program can have many possible stratifications. For instance, in the above example we could switch the order of (P1), (P2) and still get a stratification. Also, not every Datalog$^\neg$ program can be stratified! For instance, the program $R(x)$:- $S(x), \neg R(x)$ is *not* stratifiable!

If a Datalog$^\neg$ program can be stratified, we can use the stratification to provide well-defined semantics. The idea is simple: we start from the first stratum (partition), which will be a semi-positive program, and compute its fixpoint. We can now "freeze" the IDB predicates defined in the first stratum, and use them as EDB relations in the second stratum. We continue this process until we traverse all strata. This is called the *stratified semantics*.

How do we determine if a Datalog program is stratifiable? We construct the *precedence graph*: whenever an IDB predicate R appears positive in a rule with head S, we add a directed edge (R, S) with label $+$. Whenever an IDB predicate R appears negative in a rule with head S, we add an edge (R, S) with label $-$. We can now use the precedence graph to look for a stratification

Proposition 1

A Datalog$^\neg$ program is *stratifiable* if and only if its precedence graph has no directed cycles with a negative edge.

If the precedence graph indeed has no directed cycles with a negative edge, any topological ordering of the graph will give a stratification of the program. What happens in the case a Datalog$^\neg$ program admits many possible stratifications?

Theorem 1

Let P be a stratifiable program in Datalog$^\neg$. Then:

- The result of the stratified semantics is the same, independent of which stratification of P we choose.
- If P is positive, the stratified semantics output the unique minimum model.

10.3 Well-Founded Semantics

As we discussed before, there are Datalog$^\neg$ programs that cannot be stratified. We now discuss how to define semantics for these types of programs as well, which we call *well-founded semantics*.

As a running example, we will consider the Datalog$^\neg$ program that describes the *win-move* game. In this game, we have a directed graph given by the relation $moves(x, y)$. Two players make alternatively moves in the graph according to the relation *moves*. A player loses if she has no moves to make. We want to compute the predicate $win(x)$, which means that the player has a winning strategy if she moves from vertex x. The following Datalog$^\neg$ program defines the relation:

```
win(x) :- moves(x,y), not win(y).
```

We will consider the following instance of the relation *moves*:

$$\{moves(1,2), moves(2,1), moves(1,3), moves(3,4)\}$$

We start by defining the *stable model* of a Datalog$^\neg$ program. Intuitively, a set I of IDB facts is a stable model if, by applying the rules of P we get exactly I; however, we can only use non-membership of a fact to I when we apply the rules.

To formally define a stable model, we need to define the reduct of a program w.r.t. a set of facts I.

Definition 2: Reduct

The *reduct* P of a Datalog$^\neg$ program P w.r.t. a set of facts I is obtained as follows:

1. Create all possible instantiations (groundings) of the rules in P.
2. Delete all instantiations that contain $\neg R(\vec{a})$, where $R(\vec{a}) \in I$.
3. Delete all remaining negative literals in the bodies of the remaining rules.

The reduct of a Datalog$^\neg$ program w.r.t. I is always a positive Datalog program. Thus, we can always define the fixpoint of the reduct.

Definition 3: Stable Model

I is a *stable model* of a Datalog$^\neg$ program P if the fixpoint of its reduct w.r.t. to I is exactly I.

Example 6

Let $I = \{win(3), win(1)\}$. To compute the reduct, we first write the instantiations of the rules:

```
win(1) :- moves(1,2), not win(2).
win(2) :- moves(2,1), not win(1).
win(1) :- moves(1,3), not win(3).
```

```
win(3) :- moves(3,4), not win(4).
```

Since $win(1), win(3)$ are in I, we remove rules 2 and 3; and we remove the negated literals from the remaining rules. The result is the reduct:

```
win(1) :- moves(1,2).
win(3) :- moves(3,4).
```

If we run the fixpoint for this program, we obtain that the output is $\{win(1), win(3)\}$, which is exactly I. Thus, $I = \{win(3), win(1)\}$ is a stable model!

A Datalog$^\neg$ program can have several (exponentially many) stable models. For our running example, there are two stable models:

$$\{win(3), win(1)\}, \{win(3), win(2)\}$$

It can also be the case that there are no stable models! To overcome these problem, the well-founded semantics classify each possible answer to facts that are true, false, and facts that are *unknown*. In other words, we want to find a *3-valued* model to describe the answer to the program. Formally, each possible fact in the active domain will be mapped to one of three values: $0, 1, 1/2$ (0 means certainly not in the output, 1 means certainly in the output, and $1/2$ means unknown/uncertain).

Definition 4

Given a Datalog$^\neg$ program P, the *well-founded* model of P is the minimal 3-valued model such that: (i) the true facts belong to all stable models of P, and (ii) the false facts belong in no stable model of P.

If a fact appears in some stable models (but not all), then we classify it as unknown (with value $1/2$). For our running example, the well-founded model has one true fact ($win(3)$), and one false fact ($win(4)$); the remaining facts ($win(1), win(2)$) are unknown; intuitively, they are the ones that lead to a draw.

To compute the well-founded model, we start by assuming that the facts are all false, hence $I^0 = \emptyset$. In the first iteration, we compute the reduct of P w.r.t to I^0, and compute the fixpoint, which will be I^1. In the second iteration, we compute the reduct of P w.r.t. to I^1 and then find the fixpoint I^2, and so on. We call this procedure the *alternating fixpoint*.

The above process does not have a fixpoint, but it has two fixpoints! The results of even iterations are monotonically increasing, so $I^0 \subseteq I^2 \subseteq I^4 \subseteq \ldots$ Meanwhile, the results of the odd iterations are monotonically decreasing, so $I^1 \supseteq I^3 \supseteq I^5 \supseteq \ldots$ What happens is that every instance I^{2k} is an underestimation of the true facts, and an overestimation of the false facts. Similarly, every instance I^{2k+1} is an overestimation of the false facts and an underestimation of the true facts. The fixpoint of the even iterations will give us the set of all true facts, and the fixpoint of the odd iterations the set of all false facts. The remaining facts will be labeled as unknown!

Example 7

Consider our running example. Then we have:

$$
\begin{aligned}
I^0 &= \emptyset \\
I^1 &= \{win(1), win(2), win(3)\} \\
I^2 &= \{win(3)\} \\
I^3 &= \{win(1), win(2), win(3)\} \\
I^4 &= \{win(3)\}
\end{aligned}
$$

At this point we have reached the fixpoint. The true fact is $win(3)$, the false fact is $win(4)$, and the remaining are unknown.

References

[Alice] S. ABITEBOUL, R. HULL and V. VIANU, "Foundations of Databases."