Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs

Efthymia Tsamoura
Samsung AI Research

Víctor Gutiérrez-Basulto
Cardiff University

Angelika Kimmig
Cardiff University

Abstract

State-of-the-art inference approaches in probabilistic logic programming typically start by computing the relevant ground program with respect to the queries of interest, and then use this program for probabilistic inference using knowledge compilation and weighted model counting. We propose an alternative approach that uses efficient Datalog techniques to integrate knowledge compilation with forward reasoning with a non-ground program. This effectively eliminates the grounding bottleneck that so far has prohibited the application of probabilistic logic programming in query answering scenarios over knowledge graphs, while also providing fast approximations on classical benchmarks in the field.

1 Introduction

The significant interest in combining logic and probability for reasoning in uncertain, relational domains has led to a multitude of formalisms, including the family of probabilistic logic programming (PLP) languages based on the distribution semantics (Sato 1995) with languages and systems such as PRISM (Sato 1995), ICL (Poole 2008), ProbLog (De Raedt, Kimmig, and Toivonen 2007; Fierens et al. 2015) and PITA (Riguzzi and Swift 2011). State-of-the-art inference for PLP uses a reduction to weighted model counting (WMC) (Chavira and Darwiche 2008), where the dependency structure of the logic program and the queries is first transformed into a propositional formula in a suitable format that supports efficient WMC. While the details of this transformation differ across approaches, a key part of it is determining the relevant ground program with respect to the queries of interest, i.e., all groundings of rules that contribute to some derivation of a query. This grounding step has received little attention, as its cost is dominated by the cost of constructing the propositional formula in typical PLP benchmarks that operate on biological, social or hyperlink networks, where formulas are complex. However, it has been observed that the grounding step is the bottleneck that often makes it impossible to apply PLP inference in the context of ontology-based data access over probabilistic data (pOBDA) (Schoenfisch and Stuckenschmidt 2017; van Bremen, Dries, and Jung 2019), where determining the relevant grounding explores a large search space, but only small parts of this space contribute to the formulas.

We address this bottleneck, building upon the $Tc_{\mathcal{P}}$ operator (Vlasselaer et al. 2015), which integrates formula construction into forward reasoning for ground programs and is state-of-the-art for highly cyclic PLP programs. Our key contribution is a program transformation approach that allows us to implement forward inference using an efficient Datalog engine that directly operates on non-ground functor-free programs. We focus on programs without negation for simplicity, though the $Tc_{\mathcal{P}}$ operator has been studied for general probabilistic logic programs (Bogaerts and Van den Broeck 2015; Riguzzi 2016) as well; the extension to stratified negation following (Vlasselaer et al. 2016) is straightforward. We further build upon two well-known techniques from the Datalog community, namely semi-naive evaluation (Abiteboul, Hull, and Vianu 1995), which avoids recomputing the same consequences repeatedly during forward reasoning, and the magic sets transformation (Bancilhon et al. 1986; Beeri and Ramakrishnan 1991), which makes forward reasoning query driven. We adapt and extend both techniques to incorporate the formula construction performed by the $Tc_{\mathcal{P}}$ operator and implement our approach using VLog (Urbani, Jacobs, and Krötzsch 2016; Carral et al. 2019). Our experimental evaluation demonstrates that the resulting vProbLog system enables PLP inference in the pOBDA setting, answering each of the 14 standard queries of the LUBM benchmark (Guo, Pan, and Heflin 2011) over a probabilistic database of 19K facts in a few minutes at most, while most of these are infeasible for the existing ProbLog implementation of $Tc_{\mathcal{P}}$. Furthermore, for ten of the queries, vProbLog computes exact answers over 1M facts in seconds. At the same time, on three standard PLP benchmarks (Fierens et al. 2015; Renkens et al. 2014; Vlasselaer et al. 2016) where the bottleneck is formula construction, vProbLog achieves comparable approximations to the existing implementation in less time.

For details on proofs as well as additional background, we refer to the accompanying technical report (Tsamoura, Gutiérrez-Basulto, and Kimmig 2019).

2 Background

We provide some basics on probabilistic logic programming. We use standard notions of propositional logic and logic programming, cf. (Tsamoura, Gutiérrez-Basulto, and Kimmig 2019). We focus on the probabilistic logic programming language ProbLog (De Raedt, Kimmig, and Toivonen 2007; Fierens et al. 2015), and consider only function-free logic programs.

A *rule* (or *definite clause*) is a universally quantified expression of the form $\mathtt{h} :- \mathtt{b_1}, ..., \mathtt{b_n}$ where $\mathtt{h}$ and the $\mathtt{b_i}$ are atoms and the comma denotes conjunction. A *logic program* (or *program* for short) is a finite set of rules. A *ProbLog program* $\mathcal{P}$ is a triple $(\mathcal{R}, \mathcal{F}, \pi)$, where $\mathcal{R}$ is a *program*, $\mathcal{F}$ is a finite set of ground facts[1] and $\pi : \mathcal{F} \to [0,1]$ a function that labels facts with probabilities, which is often written using annotated facts $p :: f$ where $p = \pi(f)$. Without loss of generality, we restrict $\mathcal{R}$ to non-fact rules and include 'crisp' logical facts f in $\mathcal{F}$ by setting $\pi(f) = 1$. We also refer to a ProbLog program as probabilistic program. As common in probabilistic logic programming (PLP), we assume that the sets of predicates defined by facts in $\mathcal{F}$ and rules in $\mathcal{R}$, respectively, are disjoint. A ProbLog program specifies a probability distribution over its Herbrand interpretations, also called *possible worlds*. Every fact $f \in \mathcal{F}$ independently takes values `true` with probability $\pi(f)$ or `false` with probability $1 - \pi(f)$.

For the rest of the section we fix a probabilistic program $\mathcal{P} = (\mathcal{R}, \mathcal{F}, \pi)$. A *total choice* $C \subseteq \mathcal{F}$ assigns a truth value to every (ground) fact, and the corresponding logic program $C \cup \mathcal{R}$ has a unique least Herbrand model; the probability of this model is that of C. Interpretations that do not correspond to any total choice have probability zero. The *probability of a query* q is then the sum over all total choices whose program entails q:

$$\Pr(q) := \sum_{C \subseteq \mathcal{F}: C \cup \mathcal{R} \models q} \prod_{f \in C} \pi(f) \cdot \prod_{f \in \mathcal{F} \setminus C} (1 - \pi(f)) \,. \quad (1)$$

As enumerating all total choices entailing the query is infeasible, state-of-the-art ProbLog inference reduces the problem to that of weighted model counting. For a formula λ over propositional variables V and a weight function $w(\cdot)$ assigning a real number to every literal for an atom in V, the weighted model count is defined as

$$\mathrm{WMC}(\lambda) := \sum_{I \subseteq V: I \models \lambda} \prod_{a \in I} w(a) \cdot \prod_{a \in V \setminus I} w(\neg a) \,. \quad (2)$$

The reduction assigns $w(f) = \pi(f)$ and $w(\neg f) = 1 - \pi(f)$ for facts $f \in \mathcal{F}$, and $w(a) = w(\neg a) = 1$ for other atoms. For a query q, it constructs a formula λ such that for every total choice $C \subseteq \mathcal{F}$, $C \cup \{\lambda\} \models q$ if and only if $C \cup \mathcal{R} \models q$. While λ may use variables besides the facts, e.g., variables corresponding to the atoms in the program, their values have to be uniquely defined for each total choice.

In what follows we present (and adapt) notions and results by Vlasselaer et al. (2016).

[1]Note that the semantics is well-defined for countable $\mathcal{F}$, but assume (as usual in exact inference) that the finite support condition holds, which allows us to restrict to finite $\mathcal{F}$ for simplicity.

We start by noting that one way to (abstractly) specify such a λ is to take a disjunction over the conjunctions of facts in all total choices that entail the query of interest:

$$\bigvee_{C \subseteq \mathcal{F}: C \cup \mathcal{R} \models q} \; \bigwedge_{f \in C} f$$

We next extend the immediate consequence operator $T_\mathcal{P}$ for classic logic programs to construct parameterized interpretations associating a propositional formula with every atom.

Recall that the $T_\mathcal{P}$ operator is used to derive new knowledge starting from the facts. Let $\mathcal{P}$ be a logic program. For a Herbrand interpretation I, the $T_\mathcal{P}$ operator returns

$$T_\mathcal{P}(I) = \{\mathtt{h} \mid \mathtt{h} :- \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathcal{P} \text{ and } \{\mathtt{b_1}, \ldots, \mathtt{b_n}\} \subseteq I\}$$

The *least fixpoint* of this operator is the least Herbrand model of $\mathcal{P}$ and is the least set of atoms I such that $T_\mathcal{P}(I) \equiv I$. Let $T^k_\mathcal{P}(\emptyset)$ denote the result of k consecutive calls of $T_\mathcal{P}$, and $T^\infty_\mathcal{P}(\emptyset)$ the least fixpoint interpretation of $T_\mathcal{P}$.

Let $\mathsf{HB}(\mathcal{P})$ denote the set of all ground atoms that can be constructed from the constants and predicates occurring in a program $\mathcal{P}$. A *parameterized interpretation $\mathcal{I}$ of a probabilistic program $\mathcal{P}$* is a set of tuples (a, λ_a) with $a \in \mathsf{HB}(\mathcal{P})$ and λ_a a propositional formula over $\mathcal{F}$. We say that two parameterized interpretations $\mathcal{I}$ and $\mathcal{J}$ are *equivalent*, $\mathcal{I} \equiv \mathcal{J}$, if and only if they contain formulas for the same atoms and for all atoms a with $(a, \varphi) \in \mathcal{I}$ and $(a, \psi) \in \mathcal{J}$, $\varphi \equiv \psi$.

Before defining the $Tc_\mathcal{P}$ operator for probabilistic programs, we introduce some notation. For a parameterized interpretation $\mathcal{I}$ of $\mathcal{P}$, we define the set $B(\mathcal{I}, \mathcal{P})$ as

$$\begin{aligned} B(\mathcal{I}, \mathcal{P}) = \{(h\theta, \lambda_1 \wedge \ldots \wedge \lambda_n) \mid (h :- \mathtt{b_1}, \ldots, \mathtt{b_n}) \in \mathcal{P} \\ \wedge\, h\theta \in \mathsf{HB}(\mathcal{P}) \wedge \forall 1 \leq i \leq n : (b_i\theta, \lambda_i) \in \mathcal{I}\} \end{aligned}$$

Intuitively, $B(\mathcal{I}, \mathcal{P})$ contains for every grounding of a rule in $\mathcal{P}$ with head $h\theta$ for which all body atoms have a formula in $\mathcal{I}$ the pair consisting of the atom and the conjunction of these formulas. Note the structural similarity with $T_\mathcal{P}(I)$ above: the $\forall i$ condition in the definition of $B(\mathcal{I}, \mathcal{P})$ corresponds to the subset condition there, we include substitutions θ as our program is non-ground, and we store conjunctions along with the ground head.

Definition 1 ($Tc_\mathcal{P}$ operator) *Let $\mathcal{I}$ be a parameterized interpretation of $\mathcal{P}$. Then, the $Tc_\mathcal{P}$* operator is

$$Tc_\mathcal{P}(\mathcal{I}) = \{(a, \lambda(a, \mathcal{I}, \mathcal{P})) \mid a \in \mathsf{HB}(\mathcal{P}), \lambda(a, \mathcal{I}, \mathcal{P}) \not\equiv \bot\}$$

where

$$\lambda(a, \mathcal{I}, \mathcal{P}) = \begin{cases} a & \textit{if } a \in \mathcal{F} \\ \bigvee_{(a,\varphi) \in B(\mathcal{I}, \mathcal{P})} \varphi & \textit{if } a \in \mathsf{HB}(\mathcal{P}) \setminus \mathcal{F}. \end{cases}$$

The formula associated with a derived atom a in $Tc_\mathcal{P}(\mathcal{I})$ is the disjunction of $B(\mathcal{I}, \mathcal{P})$ formulae for all rules with head a, but only if this disjunction is not equivalent to the empty disjunction $\bot$. The latter is akin to not explicitly listing truth values for false atoms in regular interpretations.

We have the following correctness results. $Tc^i_\mathcal{P}(\emptyset)$ and $Tc^\infty_\mathcal{P}(\emptyset)$ are analogously defined as above.

Algorithm 1 $Tc_{\mathcal{P}}(\mathcal{I})$ for PLP $\mathcal{P}$

```
1: $I := \emptyset$; $B := \emptyset$
2: for each $h :- b_1, \ldots, b_n \in \mathcal{R}$ do
3:     for each $\theta$ with $h\theta \in \mathsf{HB}(\mathcal{P}) \wedge \forall i : (b_i\theta, \lambda_i) \in \mathcal{I}$ do
4:         $B := B \cup \{(h\theta, \lambda_1 \wedge \ldots \wedge \lambda_n)\}$
5: for each $f \in \mathcal{F}$ do
6:     $I := I \cup \{(f, f)\}$
7: for each $a$ with some $(a, \cdot) \in B$ do
8:     $I := I \cup \{(a, \bigvee_{(a,\varphi) \in B} \varphi)\}$
9: return $I$
```

Theorem 1 (Vlasselaer et al. 2016) *For a probabilistic program $\mathcal{P}$, let λ_a^i be the formula associated with atom a in $Tc_{\mathcal{P}}^i(\emptyset)$. For every atom $a \in \mathsf{HB}(\mathcal{P})$ and total choice $C \subseteq \mathcal{F}$, the following hold:*

1. For every iteration i, we have

$$C \models \lambda_a^i \quad \textit{implies} \quad C \cup \mathcal{R} \models a$$

2. There is an i_0 such that for every iteration $i \geq i_0$, we have

$$C \cup \mathcal{R} \models a \quad \textit{if and only if} \quad C \models \lambda_a^i$$

Thus, for every atom a, the λ_a^i reach a fixpoint λ_a^∞ exactly describing the possible worlds entailing a, and the $Tc_{\mathcal{P}}$ operator therefore reaches a fixpoint where for all atoms a, $\Pr(a) = \mathrm{WMC}(\lambda_a^\infty)$.[2]

Algorithm 1 shows how to naively compute $Tc_{\mathcal{P}}(\mathcal{I})$ following the recipe given in the definition. Lines 2–4 compute the set $B(\mathcal{I}, \mathcal{P})$, lines 5–6 add the formulas for facts and lines 7–8 the disjunctions that are different from $\perp$ (i.e., not empty) to the result. The fixpoint can then be easily computed, using the equivalence test as stopping criterion.

The ProbLog2 system provides an implementation of the $Tc_{\mathcal{P}}$ following (Vlasselaer et al. 2016), which proceeds in two steps. It first uses backward reasoning or SLD resolution to determine the relevant ground program for the query, i.e., all groundings of rules in $\mathcal{R}$ and all facts in $\mathcal{F}$ that contribute to some derivation of the query, and then iteratively applies the $Tc_{\mathcal{P}}$ on this program until it reaches the fixpoint or a user-provided timeout. In the latter case, the current probability is reported as lower bound. The implementation of the $Tc_{\mathcal{P}}$ updates formulas for one atom at a time, using a scheduling heuristic aimed at quick increases of probability with moderate increase of formula size.

Magic Sets Transformation. The two most common approaches to logical inference are *backward reasoning* or *SLD-resolution*, and *forward reasoning*. The magic sets transformation (Bancilhon et al. 1986; Beeri and Ramakrishnan 1991) is a well-known technique to make forward reasoning query-driven by simulating backward reasoning. The key idea behind this transformation is to introduce *magic* predicates for all derived predicates in the program, and to use these in the bodies of rules as a kind of guard that delays application of a rule during forward reasoning until the head predicate of the rule is known to be relevant for answering the query. To further exploit call patterns, different versions of such magic predicates can be used for the same original predicate; these are distinguished by *adornments*. In this work we exploit the fact that this transformation preserves query entailment.

[2]The finite support condition ensures this happens in finite time.

Algorithm 2 Semi-naive fixpoint computation of $Tc_{\mathcal{P}}$ for PLP $\mathcal{P}$

```
1: $\Delta\mathcal{I} := \{(f, f) \mid f \in \mathcal{F}\}$
2: $\mathcal{I} := \Delta\mathcal{I}$
3: repeat
4:     $\Delta\mathcal{I} := \Delta\, Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})$
5:     $\mathcal{I} := \Delta\mathcal{I} \cup \{(a, \lambda) \in \mathcal{I} \mid \neg\exists\lambda' : (a, \lambda') \in \Delta\mathcal{I}\}$
6: until $\Delta\mathcal{I} = \emptyset$
7: return $\mathcal{I}$
```

Theorem 2 (Beeri and Ramakrishnan 1991) *For any set $\mathcal{R}$ of rules with non-empty body, any set of facts $\mathcal{F}$, and any query q, $\mathcal{R} \cup \mathcal{F} \models q$ if and only if $\mathsf{magic}(\mathcal{R}, q) \cup \mathcal{F} \models q$, where $\mathsf{magic}(\mathcal{R}, q)$ is the program obtained by the magic sets transformation.*

3 Semi-naive Evaluation for $Tc_{\mathcal{P}}$

It is well-known that the computational cost of computing the fixpoint of the regular $T_{\mathcal{P}}$ operator can be lowered using semi-naive rather than naive evaluation (Abiteboul, Hull, and Vianu 1995). Intuitively, semi-naive evaluation focuses on efficiently computing the changes compared to the input interpretation rather than re-computing the full interpretation from scratch. We now discuss how we apply this idea to the $Tc_{\mathcal{P}}$ operator, where avoiding redundant work becomes even more important, as we have the added cost of compilation and more expensive fixpoint checks.

The high level structure of semi-naive evaluation for $Tc_{\mathcal{P}}$ is given in Algorithm 2. We start from the interpretation already containing the formulas for facts, that is, initially, the set $\Delta\mathcal{I}$ of formulas that just changed and the interpretation $\mathcal{I}$ derived so far contain exactly those pairs. The main loop then computes the set of pairs for which the formula changes in line 4, and updates $\mathcal{I}$ to contain these new pairs while keeping the old pairs for atoms whose formulas did not change. The fixpoint check in this setting simplifies to checking whether the set of changed formulas is empty.

Thus, the task of $\Delta\, Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})$ is to efficiently compute updated formulas for those derived atoms a whose formula changes compared to $\mathcal{I}$ given $\Delta\mathcal{I}$. This is outlined in Algorithm 3. Compared to $Tc_{\mathcal{P}}(\mathcal{I})$ in Algorithm 1, line 3 contains an additional condition: we only add those pairs for which at least one body atom has a changed formula, i.e., appears in $\Delta\mathcal{I}$, that is, the set D is a subset of the set B in naive evaluation. We no longer need to re-add fact formulas every time. Where naive evaluation simply formed all disjunctions from B to add to the result, in semi-naive evaluation, computing the result is slightly more involved. For each atom appearing in D, we compute the disjunction over D (line 6). If the atom did not yet have a formula, we add the disjunction to the output (line 11), otherwise, we disjoin

Algorithm 3 $\Delta\,Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})$ for PLP $\mathcal{P}$

```
1: $\Delta := \emptyset$; $D := \emptyset$
2: for each $h :- b_1, \ldots, b_n \in \mathcal{R}$ do
3:     for each $\theta$ with $h\theta \in \mathsf{HB}(\mathcal{P}) \wedge \forall i : (b_i\theta, \lambda_i) \in \mathcal{I}$
       $\wedge \exists i : (b_i\theta, \lambda_i) \in \Delta\mathcal{I}$ do
4:         $D := D \cup \{(h\theta, \lambda_1 \wedge \ldots \wedge \lambda_n)\}$
5: for each $a$ with some $(a, \cdot) \in D$ do
6:     $\beta_a := \bigvee_{(a,\varphi)\in D} \varphi$
7:     if $(a, \lambda_a) \in \mathcal{I}$ then          ▷ has previous formula $\lambda_a$
8:         $\gamma_a := \lambda_a \vee \beta_a$
9:         if $\gamma_a \not\equiv \lambda_a$ then $\Delta := \Delta \cup \{(a, \gamma_a)\}$
10:    else                                  ▷ no previous formula
11:        $\Delta := \Delta \cup \{(a, \beta_a)\}$
12: return $\Delta$
```

the disjunction with the old formula and only add this formula to the output if it is not equivalent to the previous one. Note that this performs the equivalence check per formula that needs to happen explicitly in naive evaluation.

Formally, lines 4 and 5 in Algorithm 2 implement the following operator.

Definition 2 (**$Sc_{\mathcal{P}}$ operator**) *Let $\mathcal{P}$ be a probabilistic logic program. Let $\mathcal{I}$ be a parameterized interpretation of $\mathcal{P}$, and $\Delta\mathcal{I} \subseteq \mathcal{I}$. Let $\mathcal{H} = \mathsf{HB}(\mathcal{P}) \setminus \mathcal{F}$. Then, the $\Delta\,Tc_{\mathcal{P}}$ operator is*

$$\begin{aligned}&\Delta\,Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})\\ &\quad= \{(a, \beta_a) \mid a \in \mathcal{H} \wedge \neg\exists\lambda : (a,\lambda) \in \mathcal{I} \wedge \beta_a \not\equiv \bot\}\\ &\quad\cup \{(a, \lambda \vee \beta_a) \mid a \in \mathcal{H} \wedge (a,\lambda) \in \mathcal{I} \wedge \lambda \not\equiv (\lambda \vee \beta_a)\}\end{aligned}$$

where

$$\beta_a = \bigvee_{(a,\varphi)\in D(\mathcal{I},\Delta\mathcal{I},\mathcal{P})} \varphi \qquad \textit{and}$$

$$\begin{aligned}D(\mathcal{I}, \Delta\mathcal{I}, \mathcal{P}) = \{(h\theta, \lambda_1 \wedge .. \wedge \lambda_n) \mid (h{:-}\, \mathtt{b_1}, .., \mathtt{b_n}) \in \mathcal{P}\\ \wedge h\theta \in \mathsf{HB}(\mathcal{P}) \wedge \forall 1 \leq i \leq n : (b_i\theta, \lambda_i) \in \mathcal{I}\\ \wedge \exists 1 \leq i \leq n : (b_i\theta, \lambda_i) \in \Delta\mathcal{I}\}\end{aligned}$$

The semi-naive $Tc_{\mathcal{P}}$-operator $Sc_{\mathcal{P}}$ is

$$\begin{aligned}Sc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I}) &= \Delta\,Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})\\ &\cup \{(a,\lambda) \in \mathcal{I} \mid \neg\exists\lambda' : (a,\lambda') \in \Delta\,Tc_{\mathcal{P}}(\mathcal{I}, \Delta\mathcal{I})\}\end{aligned}$$

We are now ready to prove correctness of the approach. Let $\mathcal{I}^0 = \{(f,f) \mid f \in \mathcal{F}\}$, let $Tc^i_{\mathcal{P}}(\mathcal{I}^0)$ be the result of i consecutive applications of the $Tc_{\mathcal{P}}$ starting from $\mathcal{I}^0$, and $Sc^i_{\mathcal{P}}(\mathcal{I}^0, \mathcal{I}^0)$ be the result of i consecutive applications of the $Sc_{\mathcal{P}}$ starting from $\mathcal{I}^0 = \Delta\mathcal{I}^0$ and using $\mathcal{I}^{i+1} = Sc_{\mathcal{P}}(\mathcal{I}^i, \Delta\mathcal{I}^i)$ and $\Delta\mathcal{I}^{i+1} = \Delta\,Tc_{\mathcal{P}}(\mathcal{I}^i, \Delta\mathcal{I}^i)$.

The next claim can be easily proven by induction.

Claim For all $i \geq 1$, $Tc^i_{\mathcal{P}}(\mathcal{I}^0) \equiv Sc^i_{\mathcal{P}}(\mathcal{I}^0, \mathcal{I}^0)$.

Semi-naive Fixpoint Computation using Rules

The key idea behind our efficient implementation of the semi-naive fixpoint computation for $Tc_{\mathcal{P}}$ is to introduce relations that capture the information computed in the steps of Algorithms 3 and 2, and rules that populate these relations, which can be executed using semi-naive evaluation functionality provided by an existing Datalog engine, cf. Section 5 below. We discuss the relations and rules here, abstracting from the specific syntax used in the implementation.

We focus on $\Delta\,Tc_{\mathcal{P}}$ first, and assume that, as common in semi-naive evaluation, the engine "knows" which tuples in $\mathcal{I}$ are in $\Delta\mathcal{I}$ as well. The input consists of atoms of the form `lambda(atom,formula)`, based on which we define three relations `d(head,conjunction)` representing the set D of head-conjunction pairs, `beta(head,disjunction)` representing pairs (a, β_a) computed in line 6, and `delta(head,disj)` representing the pairs in the output Δ.

The following types of rules populate these predicates. For every (potentially non-ground) rule `h :- b1,...,bn` in $\mathcal{P}$, we have a rule of the form

```
d(h,conj([F1,...,Fn])) :-
     lambda(b1,F1), ..., lambda(bn,Fn).
```

For every predicate `p/n` defined by rules in $\mathcal{P}$, we have

```
beta(p(X1,...,Xn),disj(L)) :-
   d(p(X1,...,Xn),_),
   findall(C, d(p(X1,...,Xn),C), L).
```

Finally, we have the general rules

```
delta(A,D) :- beta(A,D), not lambda(A,_).
delta(A,disj([D,F])) :-
    beta(A,D), lambda(A,F),
    not equivalent(disj([D,F]),F).
```

Here, `findall` is the usual Prolog predicate that collects all groundings of the first argument for which the second argument holds in a list and unifies the variable in the third argument with that list, `not` is negation as failure, and `equivalent` is a special predicate provided as an external Boolean function that when called on two ground terms returns true if the propositional formulas encoded by these terms are logically equivalent.

When computing the fixpoint, the relation `delta` provides the next $\Delta\mathcal{I}$, and we need to compute the next $\mathcal{I}$ from the current `lambda` and the `delta` as in line 5 of Algorithm 2. We next show how we integrate this step into the rules. The key idea is to extend the `lambda` predicate with an additional argument that contains a unique identifier for every atom-formula pair added to the relation, to mark pairs that contain outdated formulas for an atom by adding the identifier to a new relation `outdated`, and to incrementally populate these two relations across the fixpoint computation while dropping all intermediate relations after each iteration.

The most recent formula for an atom `a` is now given by the conjunctive query `lambda(a,F,I), not outdated(I)`. We adapt the rules defining `d` to use this pattern in the body, keep the rules for `beta` unchanged, and replace the rules for `delta` with the following set of rules:

```
lambda(A,D,A) :-
  beta(A,D), not lambda(A,_,_).
aux(A,disj([D,F]),I) :-
  beta(A,D), lambda(A,F,I), not outdated(I),
  not equivalent(disj([D,F]),F).
lambda(A,F,u(I)) :- aux(A,F,I).
outdated(I) :- aux(A,F,I).
```

The first of these covers the case where atom `A` did not have a formula yet, in which case we use the atom itself as identifier. The second rule defines an auxiliary relation whose elements are an atom, the updated formula for the atom, and the identifier for the atom's previous formula, where the latter is used in the last two rules to generate a new identifier and mark the old one as outdated.

At the fixpoint, we perform a final projection step to eliminate identifiers using `lambda(A,F) :- lambda(A,F,I), not outdated(I).`

4 Magic Sets for $Tc_{\mathcal{P}}$

So far, we have discussed how to efficiently compute the full fixpoint of the $Tc_{\mathcal{P}}$ operator. However, in practice, we are often only interested in specific queries. In this case, regular $T_{\mathcal{P}}$ often uses the magic set transformation to restrict the fixpoint computation to the atoms relevant to the query. We now show that we can apply the same transformation in our setting to make the $Tc_{\mathcal{P}}$ goal-directed, and then introduce an optimization of our approach for magic programs.

We fix a probabilistic program $\mathcal{P} = (\mathcal{R}, \mathcal{F}, \pi)$.

Definition 3 (Magic Sets for PLP) *The* magic transform of $\mathcal{P}$ *with respect to a query* q *is the program* $\mathcal{M} = (\mathsf{magic}(\mathcal{R}, q), \mathcal{F}, \pi)$*, where* $\mathsf{magic}(\cdot,\cdot)$ *is as in Theorem 2.*

That is, we apply the regular magic set transformation to the rules with non-empty body and keep the facts and labeling.

Theorem 3 *Let* q *be a query. The formula* $\lambda(\mathcal{M}, q)$ *associated with* q *in the fixpoint of* $Tc_{\mathcal{M}}$ *is equivalent to the formula* $\lambda(\mathcal{P}, q)$ *associated with* q *in the fixpoint of* $Tc_{\mathcal{P}}$.

This is a direct consequence of Theorems 1 (Point 2) and 2.

While the above makes our approach query directed, compiling formulas for magic atoms may introduce significant overhead. We therefore now define an optimized version of goal-directed $Tc_{\mathcal{P}}$ that avoids compilation for magic atoms, and show that this computes correct formulas.

Definition 4 (magic-$Tc_{\mathcal{P}}$ operator) *Let* $q \in \mathsf{HB}(\mathcal{P})$ *be the query of interest, and* $\mathcal{I}$ *a parameterized interpretation of the magic transform* $\mathcal{M}$ *of* $\mathcal{P}$*. Then, the magic-*$Tc_{\mathcal{P}}$ *operator* $Mc_{\mathcal{P}}(\mathcal{I})$ *is defined as*

$$\{(a, \mu(a, \mathcal{I}, \mathcal{M})) \mid a \in \mathsf{HB}(\mathcal{M}) \land \mu(a, \mathcal{I}, \mathcal{M}) \not\equiv \bot\}$$

where

$$\mu(a, \mathcal{I}, \mathcal{M}) = \begin{cases} \lambda(a, \mathcal{I}, \mathcal{M}) & \text{if } a \in \mathsf{HB}(\mathcal{P}) \\ \top & \text{if "magic case"} \end{cases}$$

where λ *is as in Definition 1 above, and "magic case" means that* $a \in \mathsf{HB}(\mathcal{M}) \setminus \mathsf{HB}(\mathcal{P})$ *and there is a rule* $h :- b_1, \ldots, b_n$ *in* $\mathcal{M}$ *and a grounding substitution* θ *such that* $a = h\theta$ *and for each* b_i *there is a* λ_i *such that* $(b_i\theta, \lambda_i) \in \mathcal{I}$.

That is, if a is not a magic atom (which includes the facts), we use the same update operations as for the regular $Tc_{\mathcal{M}}$ operator with the magic program, but for magic atoms, we set the formula $\top$ if (and only if) there is a rule that can derive the atom from the current interpretation $\mathcal{I}$.

Theorem 4 *For a query* q*, let* τ_q^i *be the formula associated with* q *in* $Mc_{\mathcal{P}}^i(\emptyset)$*. For every total choice* $C \subseteq \mathcal{F}$*, there is an* i_0 *such that for every iteration* $i \geq i_0$*,* τ_q^i *exists and*

$$C \cup \mathcal{R} \models q \quad \textit{if and only if} \quad C \models \tau_q^i$$

The proof relies on two intermediate results: Formulas computed by $Tc_{\mathcal{P}}$ are always lower bounds for those computed by $Mc_{\mathcal{P}}$; and for any atom in the original program (i.e., excluding the magic atoms), the formulas computed by $Mc_{\mathcal{P}}$ only include correct choices relative to the original program. These combined with Theorem 1 (Point 2) provide us with the desired result.

5 Implementation

We use VLog (Urbani, Jacobs, and Krötzsch 2016; Carral et al. 2019) to implement the principles introduced in Section 3, and refer to this implementation as vProbLog[3]. We use VLog because it has been shown to be efficient, is open source, supports negation (which is used by some of our transformed rules) and provides very efficient rule execution functions that we can invoke directly.

As VLog is a Datalog engine, we cannot directly use the rules as introduced in Section 3. Instead, we encode functors into predicate or constant names wherever possible and use a procedural variant of the second set of rules to avoid findall. We implement the "equivalent" function as an external function over Sentential decision diagrams (SDD) (Darwiche 2011), using the SDD package developed at UCLA[4].

More precisely, given program $\mathcal{P}$, query q and iteration parameter d, vProbLog performes the following steps:

1. Apply the magic set transformation to $\mathcal{R}$ and q.
2. Partition $\mathsf{magic}(\mathcal{R}, q)$ into three sets, where $\mathcal{R}_1$ contains the rules whose bodies only use facts, $\mathcal{R}_2$ contains all other rules that do not depend on cyclic derivations, and $\mathcal{R}_3$ the remaining ones, and apply the transformation of Section 3 to all three.
3. Materialize the fixpoint, i.e., execute the transforms of $\mathcal{R}_1$ and $\mathcal{R}_2$ in sequence, iteratively execute the transform of $\mathcal{R}_3$ d times (or until fixpoint if $d = \infty$), and finally use the projection rule to eliminate bookkeeping identifiers.
4. For each query answer in the materialization, compute the WMC of its SDD.

This gives us vProbLog$^{\mathsf{plain}}$, an implementation of $Tc_{\mathcal{M}}$, i.e., without the optimizations introduced in Section 4. To implement the modified operator $Mc_{\mathcal{P}}$, we note that instead of explicitly setting the formula of a magic atom $\top$ in our transformed program, we can simply not store magic atoms in the `lambda` relation but instead keep them as usual wherever they appear. We refer to this as vProbLog$^{\mathsf{opt}}$.

6 Experimental Evaluation

We perform experiments using the two versions of vProbLog as well as the implementation of $Tc_{\mathcal{P}}$ provided

[3]https://bitbucket.org/tsamoura/vproblog/

[4]http://reasoning.cs.ucla.edu/sdd/

by the ProbLog2 system[5], which we refer to as ProbLog2, to answer the following questions:

Q1 Does vProbLog$^{\mathsf{opt}}$ outperform vProbLog$^{\mathsf{plain}}$ in terms of running times and scalability?

Q2 How does the performance of vProbLog$^{\mathsf{opt}}$ compare to that of ProbLog2?

Q3 How scalable is vProbLog$^{\mathsf{opt}}$?

As vProbLog aims to improve the reasoning phase, but essentially keeps the knowledge compilation phase unchanged, we distinguish two types of benchmarks: those where the bottleneck is reasoning, i.e., propositional formulas are relatively small, but a large amount of reasoning may be required to identify the formulas, and those where the bottleneck is knowledge compilation, i.e., propositional formulas become complex quickly. As a representative of the former, we use a probabilistic version of **LUBM** (Guo, Pan, and Heflin 2011), a setting known to be challenging for ProbLog (Schoenfisch and Stuckenschmidt 2017; van Bremen, Dries, and Jung 2019); for the latter type, we use three benchmarks from the ProbLog literature (Fierens et al. 2015; Renkens et al. 2014; Vlasselaer et al. 2016) that essentially are all variations of network connectivity queries:

LUBM We create a probabilistic version of the **LUBM** benchmark by adding a random probability from $[0.01, 1.0]$ to each fact in the database, using three databases of increasing size (approximately 19K tuples in **LUBM001**, 1M tuples in **LUBM010** and 12M tuples in **LUBM100**). We drop all rules that introduce existentials, as these are not supported in our setting, and use the 14 standard queries (with default join order).

WebKB We use the **WebKB**[6] dataset restricted to the 100 most frequent words (Davis and Domingos 2009) and with random probabilities from $[0.01, 0.1]$, using all pages from the Cornell database. This results in a dataset with 63 ground queries.

Smokers We use random power law graphs with increasing numbers of persons for the standard ‘Smokers’ social network domain, and non-ground query `asthma(_)`.

Genes We use the biological network of Ourfali et al. 2007 and 50 of its connection queries on gene pairs.

To answer **Q1**, we compare running times of both versions of vProbLog on **LUBM001**. The first two blocks of Table 1 report times for materialization and WMC as well as their sum, with a two minute timeout on the former. The differences in materialization time clearly show that compiling formulas for magic atoms can introduce significant overhead and should thus be avoided. We thus answer **Q1** affirmatively, and only use vProbLog$^{\mathsf{opt}}$ (or vProbLog for short) below.

To answer **Q2**, we consider all benchmarks, with a two minute timeout per query for ProbLog2. In the middle of Table 1, we list the times ProbLog2 reports for grounding, compiling SDDs, and total time (which also includes data loading) per query on **LUBM001**. The *answer* column indicates whether ProbLog2 returns exact probabilities (fixpoint detected), lower bounds (timeout with partial formula for query, before detecting fixpoint) or no result (timeout without formula for query). The lower bounds reported for q10 and q13 are practically the final probabilities, but ProbLog2 has not detected this yet at timeout. ProbLog2 reaches the timeout during grounding for q02, q07, q08 and q09, for which the default join order first builds a Cartesian product of two or three type relations, which is the worst possible join order for SLD resolution.

Comparing ProbLog2’s times to the times for vProbLog$^{\mathsf{opt}}$ to the left, we observe that vProbLog materializes fixpoints, which includes compiling formulas (using the same SDD tool) in often significantly less time than it takes ProbLog2 to just determine the relevant ground program, with the exception being the database lookup query q14, where running times are similar. These results clearly demonstrate the benefits of exploiting Datalog techniques in terms of speed.

On the PLP benchmarks, ProbLog2 can only compute lower bounds for most queries, and vProbLog cannot fully materialize the fixpoint, as SDDs quickly become too large to handle. For vProbLog, we therefore select for each benchmark a fixed number of iterations close to the feasibility borderline. Intuitively, this restricts the length of paths explored, though that length does not equal the number of iterations due to our program transformations, and vProbLog thus always computes lower bounds. Here, we are thus interested to see whether vProbLog can achieve comparable or better lower bounds to ProbLog2 in less time.

For **WebKB**, seven of the 63 queries are easy, as the corresponding pages have incoming link chains of length at most two, whereas formulas explode for all others. ProbLog2 computes probabilities for the easy queries in at most seven seconds, and bounds after timeout for the others, whereas vProbLog computes bounds using four iterations for all queries in at most seven seconds. All differences between lower bounds are smaller than 0.02, with ProbLog2 achieving higher bounds on 47 queries and smaller bounds on 11.

For **Smokers**, ProbLog2 quickly computes exact answers for all networks up to size 12, and reaches the timeout on all networks from size 19 onwards. We therefore consider 10 networks for each size from 10 to 20 persons, using three iterations for vProbLog. Table 2 lists for each size the number of scenarios (out of ten) solved exactly by ProbLog2, along with minimum, average and maximum total times for both systems (excluding timeout cases). We again observe a clear time advantage for vProbLog with increasing size. Furthermore, the bounds provided by vProbLog are close to actual probabilities (exact for 89 queries, at most 0.002 lower for 615 queries) where ProbLog2 computes those, and close to ProbLog2’s lower bounds otherwise (up to 0.05 higher on 570 queries, and at most 0.002 lower for 312).

For **Genes**, ProbLog2 reaches the timeout and thus computes lower bounds for all queries. As those range from 0.0108 to 0.9999, we run vProbLog for 1, 3, 5, 6 and 7 iterations to explore the effect of the approximation in more detail. 5 iterations are infeasible for two queries, 7 iterations

[5]https://dtai.cs.kuleuven.be/problog/ version 2.1.0.37, option `-k fsdd`

[6]http://www.cs.cmu.edu/webkb/

Table 1: Results for **LUBM**: time for materialization, weighted model counting and total time for vProbLog$^{\mathsf{plain}}$ (two minute timeout for materialisation) and vProbLog$^{\mathsf{opt}}$ on **LUBM001** (Q1), grounding time, compilation time and total time for ProbLog2 (two minute overall timeout, marked x) as well as type of answer (exact or lower bound) provided if any (Q2), and times for vProbLog$^{\mathsf{opt}}$ on **LUBM010** and **LUBM100** (Q3). All times in seconds.

	LUBM001										**LUBM010**			**LUBM100**		
	vProbLog$^{\mathsf{plain}}$			vProbLog$^{\mathsf{opt}}$			ProbLog2				vProbLog$^{\mathsf{opt}}$			vProbLog$^{\mathsf{opt}}$		
	mat	wmc	total	mat	wmc	total	ground	comp	total	answer	mat	wmc	total	mat	wmc	total
q01	0.1	0.1	0.1	0.0	0.1	0.1	0.7	1.2	40.9	exact	0.5	0.2	0.7	6.7	1.8	8.5
q02	0.5	0.0	0.5	0.1	0.0	0.1	x			-	6.9	0.6	7.5			
q03	0.1	0.1	0.2	0.0	0.1	0.2	6.3	x		exact	0.2	0.3	0.5	2.3	1.9	4.1
q04	x		x	3.6	0.5	4.2	49.6	x		-	5.2	0.7	5.8			
q05	x		x	6.3	11.3	17.5	56.4	x		-	7.5	9.8	17.3			
q06	x		x	41.8	113.4	155.2	47.7	x		-						
q07	6.3	0.9	7.3	3.4	1.0	4.4	x			-	5.1	1.1	6.2			
q08	x		x	60.6	115.2	175.8	x			-						
q09	x		x	22.2	2.8	25.0	x			-						
q10	1.6	0.1	1.6	0.5	0.1	0.6	50.6	x		bound	1.2	0.2	1.5			
q11	0.5	5.3	5.8	0.3	3.4	3.7	0.4	0.9	40.7	exact	0.3	3.5	3.8	0.4	4.9	5.3
q12	x		x	15.5	0.2	15.7	47.2	x		-	17.9	0.4	18.3			
q13	2.1	0.0	2.1	1.0	0.0	1.1	57.9	x		bound	12.7	0.6	13.3			
q14	0.2	80.2	80.4	0.2	89.4	89.6	1.2	0.2	87.7	exact	3.1					

Table 2: **Smokers**: ProbLog2: number of scenarios (of 10) solved exactly in 2 minutes, min/avg/max times in seconds over those; vProbLog: min/avg/max times in seconds over all networks to compute lower bounds with 3 iterations

	ProbLog2				vProbLog		
size	exact	min	avg	max	min	avg	max
10	10	0.1	0.6	3.3	0.5	0.6	1.0
11	10	0.1	1.0	5.7	0.5	0.6	0.8
12	10	0.2	10.9	93.1	0.6	0.8	0.9
13	9	0.2	15.7	55.9	0.6	0.8	1.2
14	8	0.7	34.9	106.7	0.3	1.2	3.6
15	3	2.7	34.2	65.6	0.8	1.3	2.5
16	3	12.0	24.6	41.8	0.7	1.6	3.3
17	2	1.7	8.4	15.1	0.8	2.1	4.5
18	1	23.6	23.6	23.6	0.9	2.7	5.8
19	0				1.1	4.6	12.1
20	0				1.3	3.9	13.4

for another five queries. vProbLog running times for the last feasible iteration of each query are below five seconds per query for all but two queries, which take 13 and 78 seconds, respectively. We refer to the first non-zero bound for query q vProbLog reaches as f_q, to the last (feasible) one as l_q, and to ProbLog2's bound as p_q. We observe $p_q < f_q$ for 23 queries, $f_q \leq p_q < l_q$ for 15, and $l_q \leq p_q$ for 12. The actual differences $p_q - l_q$ vary from -0.88 to 0.20, indicating that this is a diverse set of queries where no single approximation strategy suits all, but also that vProbLog often achieves much higher bounds in less time.

Based on these results, as an answer to **Q2**, we conclude that vProbLog often speeds up logical inference time significantly, which enables inference in scenarios that have previously been infeasible due to the grounding bottleneck. At the same time, for benchmarks where the bottleneck is knowledge compilation, its use of magic sets provides a natural scheduling strategy for formula updates that achieves similar bounds to those of ProbLog2, but often in less time.

To answer **Q3**, we consider the three **LUBM** databases. Table 1 lists running times of vProbLog$^{\mathsf{opt}}$ (for materialization, weighted model counting, and total of both) without iteration limit per query in seconds. Blank entries indicate that the corresponding phase did not finish due to problems with SDDs. The three queries that are answered on the largest database all have one element answers from relatively narrow classes with a direct link to a specific constant, which limits both the number of answers and the size of formulas. In contrast, the three queries that cannot be solved on the medium size database either ask for a broad class (all students, q06) or triples that include members of such a class and two related objects (q08 and q09), which means both more answers on larger databases and more complex SDDs.

Taking all results together, our answer to **Q3** is that vProbLog directly benefits from the scalability of VLog for logical reasoning, but as all WMC-based approaches is limited by the complexity of formula manipulation.

7 Related Work

A related formalism is the probabilistic version of Datalog by Bárány et al. 2017, but it does not adopt the distribution semantics and there is no existing system. pOBDA can be also viewed as a variation of PLP formalisms (Jung and Lutz 2012). The exact relation to PLP broadly depends on the ontology language of choice, usually based on description logics or existential rules, see (Borgwardt, Ceylan, and Lukasiewicz 2018) for a recent survey.

Finally, we note that magic sets have been also applied to other extensions of Datalog, e.g. with aggregates, equality or disjunctive Datalog (Alviano, Greco, and Leone 2011; Benedikt, Motik, and Tsamoura 2018; Alviano et al. 2012).

8 Conclusions

We have adapted and extended the well-known Datalog techniques of semi-naive evaluation and magic sets to avoid the grounding bottleneck of state-of-the-art inference in probabilistic logic programming, contributed a prototype implementation based on VLog, and experimentally demonstrated the benefits in terms of scalability on both traditional PLP benchmarks and a query answering scenario that previously has been out of reach for ProbLog. Immediate future work includes extending the approach to stratified negation and eliminating the need to pre-determine the depth parameter by iterative expansion, which will provide a proper anytime algorithm. Beyond this, we intend to perform experiments on additional benchmarks in the knowledge graph and ontology setting, including ontologies based on description logics and existential rules, to study ways to support probabilistic programs with functors, as well as alternative ways to trade-off time spent on logical reasoning vs compilation, e.g., by restricting the number of fixpoint checks performed.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Alviano, M.; Faber, W.; Greco, G.; and Leone, N. 2012. Magic Sets for disjunctive Datalog programs. *Artificial Intelligence* 187:156–192.

Alviano, M.; Greco, G.; and Leone, N. 2011. Dynamic Magic Sets for Programs with Monotone Recursive Aggregates. In *LPNMR*.

Bancilhon, F.; Maier, D.; Sagiv, Y.; and Ullman, J. D. 1986. Magic sets and other strange ways to implement logic programs. In *PODS*.

Bárány, V.; ten Cate, B.; Kimelfeld, B.; Olteanu, D.; and Vagena, Z. 2017. Declarative probabilistic programming with datalog. *ACM Trans. Database Syst.* 42(4):22:1–22:35.

Beeri, C., and Ramakrishnan, R. 1991. On the power of magic. *J. Log. Program.* 10(3&4):255–299.

Benedikt, M.; Motik, B.; and Tsamoura, E. 2018. Goal-driven query answering for existential rules with equality. In *AAAI Conference on Artificial Intelligence*.

Bogaerts, B., and Van den Broeck, G. 2015. Knowledge compilation of logic programs using approximation fixpoint theory. *TPLP* 15(4-5):464–480.

Borgwardt, S.; Ceylan, İ. İ.; and Lukasiewicz, T. 2018. Recent advances in querying probabilistic knowledge bases. In *International Joint Conference on Artificial Intelligence*.

Carral, D.; Dragoste, I.; González, L.; Jacobs, C.; Krötzsch, M.; and Urbani, J. 2019. VLog: A rule engine for knowledge graphs. In *International Semantic Web Conference*.

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172(6-7):772–799.

Darwiche, A. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Davis, J., and Domingos, P. 2009. Deep Transfer via Second-Order Markov Logic. In *International Conference on Machine Learning (ICML)*.

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence*.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D. S.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming (TPLP)* 15(3):358–401.

Guo, Y.; Pan, Z.; and Heflin, J. 2011. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2-3).

Jung, J. C., and Lutz, C. 2012. Ontology-based access to probabilistic data with OWL QL. In *International Semantic Web Conference*.

Ourfali, O.; Shlomi, T.; Ideker, T.; Ruppin, E.; and Sharan, R. 2007. SPINE: a framework for signaling-regulatory pathway inference from cause-effect experiments. *Bioinformatics* 23(13):359–366.

Poole, D. 2008. The Independent Choice Logic and Beyond. In De Raedt, L.; Frasconi, P.; Kersting, K.; and Muggleton, S., eds., *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer. 222–243.

Renkens, J.; Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2014. Explanation-based approximate weighted model counting for probabilistic logics. In *AAAI Conference on Artificial Intelligence*.

Riguzzi, F., and Swift, T. 2011. The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty. *Theory and Practice of Logic Programming* 11(4–5):433–449.

Riguzzi, F. 2016. The distribution semantics for normal programs with function symbols. *Int. J. Approx. Reasoning* 77:1–19.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming (ICLP)*.

Schoenfisch, J., and Stuckenschmidt, H. 2017. Analyzing real-world SPARQL queries and ontology-based data access in the context of probabilistic data. *Int. J. Approx. Reasoning* 90:374–388.

Tsamoura, E.; Gutiérrez-Basulto, V.; and Kimmig, A. 2019. Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs (technical report). *CoRR* abs/1911.07750.

Urbani, J.; Jacobs, C. J. H.; and Krötzsch, M. 2016. Column-oriented datalog materialization for large knowledge graphs. In *AAAI Conference on Artificial Intelligence*.

van Bremen, T.; Dries, A.; and Jung, J. C. 2019. Ontology-mediated queries over probabilistic data via probabilistic logic programming. In *International Conference on Information and Knowledge Management (CIKM)*. ACM.

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2015. Anytime inference in probabilistic logic programs with Tp-compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2016. Tp-compilation for inference in probabilistic logic programs. *International Journal of Approximate Reasoning* 78:15 – 32.