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) (Schoenisch 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 $T_{CP}$ 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 function-free programs. We focus on programs without negation for simplicity, though the $T_{CP}$ 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 $T_{CP}$ 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 $T_{CP}$. 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 Kim- mig 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 \( h : \neg b_1, \ldots, \neg b_n \) where \( h \) and the \( b_i \) are atoms and the comma denotes conjunction. A logic program (or program for short) is a finite set of rules. A ProbLog program \( P \) 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 \( \text{true} \) with probability \( \pi(f) \) or \( \text{false} \) with probability \( 1 - \pi(f) \).

For the rest of the section we fix a probabilistic program \( P = (\mathcal{R}, \mathcal{F}, \pi) \). 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 \( q \):

\[
\Pr(q) := \sum_{C \subseteq \mathcal{F} : C \cup \mathcal{R} \models q} \prod_{f \in C} \pi(f) \cdot \prod_{f \notin C} (1 - \pi(f)).
\]

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 \( \lambda \) 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

\[
WMC(\lambda) := \sum_{I \subseteq V : |I| = \lambda} w(a) \cdot \prod_{a \in V \setminus I} w(\neg a).
\]

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 \( \lambda \) 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 \( \lambda \) 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 \( \lambda \) 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 \( T_P \) for classic logic programs to construct parameterized interpretations associating a propositional formula with every atom.

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

\[
T_P(I) = \{ h : \neg b_1, \ldots, b_n \in \mathcal{P} \text{ and } \{ b_1, \ldots, b_n \} \subseteq I \}
\]

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

Let \( \text{HB}(\mathcal{P}) \) 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 \( I \) of a probabilistic program \( P \) is a set of tuples \((a, \lambda_a) \) with \( a \in \text{HB}(\mathcal{P}) \) and \( \lambda_a \) a propositional formula over \( \mathcal{F} \). We say that two parameterized interpretations \( I \) and \( J \) are equivalent, \( I \equiv J \), if and only if they contain formulas for the same atoms and for all atoms \( a \) with \((a, \varphi) \in I \) and \((a, \psi) \in J, \varphi \equiv \psi \).

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

\[
B(I, \mathcal{P}) = \{(h\theta, \lambda_1 \land \ldots \land \lambda_n) | (h: \neg b_1, \ldots, b_n) \in \mathcal{P}
\land \neg h\theta \in \text{HB}(\mathcal{P}) \land \forall 1 \leq i \leq n : (b_i\theta, \lambda_i) \in I\}
\]

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

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

\[
TC_P(I) = \{(a, \lambda(a, I, \mathcal{P})) | a \in \text{HB}(\mathcal{P}), \lambda(a, I, \mathcal{P}) \neq \bot\}
\]

where

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

The formula associated with a derived atom \( a \) in \( TC_P(I) \) is the disjunction of \( B(I, \mathcal{P}) \) 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. \( TC_P(\emptyset) \) and \( TC^\infty_P(\emptyset) \) are analogously defined as above.
The key idea behind this transformation is to introduce reasoning query-driven by simulating backward reasoning. The transformation (Bancilhon et al. 1986; Beeri and Ramakrishnan 1991) is a well-known technique to make forward reasoning 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 $T_{CP}$ operator, where avoiding redundant work becomes even more important, as we have the added cost of compilation and more expensive fixpoint checks.

### 3 Semi-naive Evaluation for $T_{CP}$

It is well-known that the computational cost of computing the fixpoint of the regular $T_{CP}$ operator can be lowered using semi-naive rather than naive evaluation (Abiteboul, Hull, and Vianu 1995). The algorithm 2 provided in this setting simplifies to checking whether the set of changed formulas is empty.

#### 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.

#### 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 $\text{magic}(\mathcal{R}, q) \cup \mathcal{F} \models q$, where $\text{magic}(\mathcal{R}, q)$ is the program obtained by the magic sets transformation.

---

**Algorithm 1** $T_{CP}(\mathcal{I})$ for PLP $\mathcal{P}$

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

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

1. $\Delta \mathcal{I} := \{(f, f) \mid f \in \mathcal{F}\}$
2. $\mathcal{I} := \Delta \mathcal{I}$
3. repeat
4.     $\Delta \mathcal{I} := \Delta T_{CP}(\mathcal{I}, \Delta \mathcal{I})$
5.     $\mathcal{I} := \Delta \mathcal{I} \cup \{(a, \lambda) \in \mathcal{I} \mid \exists \lambda' : (a, \lambda') \in \Delta \mathcal{I}\}$
6. until $\Delta \mathcal{I} = \emptyset$
7. return $\mathcal{I}$
Formally, lines 4 and 5 in Algorithm 2 implement the following operator.

\[ \beta_a := \bigvee_{(a,\varphi) \in D(\Delta,\Delta)} \varphi \] and

\[ D(\Delta,\Delta,\mathcal{P}) = \{(h\theta, \lambda_1 \land \ldots \land \lambda_n) \mid (h, b_1, \ldots, b_n) \in \mathcal{P} \land h\theta \in \text{HB}(\mathcal{P}) \land \forall i \leq i \leq n : (b_i, \lambda_i) \in \mathcal{I} \land \exists i \leq i \leq n : (b_i, \lambda_i) \in \Delta \} \]

The semi-naive \( T_{CP} \)-operator \( SC_{P} \) is

\[ SC_{P}(\mathcal{I}, \Delta, \mathcal{I}) = \Delta T_{CP}(\mathcal{I}, \Delta, \mathcal{I}) \]

\[ \cup \{(a, \lambda) \mid a \in \mathcal{I} \land \neg \exists \lambda' : (a, \lambda') \in \Delta T_{CP}(\mathcal{I}, \Delta, \mathcal{I}) \} \]

We are now ready to prove correctness of the approach. Let \( \mathcal{I}^0 = \{(f, f) \mid f \in F\} \), let \( \Delta T_{CP}(\mathcal{I}^0) \) be the result of \( i \) consecutive applications of the \( T_{CP} \) starting from \( \mathcal{I}^0 \), and \( SC_{P}(\mathcal{I}^0, \mathcal{I}^0) \) be the result of \( i \) consecutive applications of the \( SC_{P} \) starting from \( \mathcal{I}^0 = \Delta T_{CP}(\mathcal{I}^0) \) and using \( \mathcal{I}^i+1 = SC_{P}(\mathcal{I}^i, \mathcal{I}^i) \) and \( \Delta T_{CP}(\mathcal{I}^i, \mathcal{I}^i) \) the next claim can be easily proved by induction.

Claim For all \( i \geq 1 \), \( T_{CP}(\mathcal{I}^0) \equiv SC_{P}(\mathcal{I}^0, \mathcal{I}^0) \)

Semi-naive Fixpoint Computation using Rules

The key idea behind our efficient implementation of the semi-naive fixpoint computation for \( T_{CP} \) 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 \( \Delta T_{CP} \) first, and assume that, as common in semi-naive evaluation, the engine "knows" which tuples in \( \mathcal{I} \) are in \( \Delta T \) as well. The input consists of atoms of the form \( \lambda(\text{atom},\text{formula}) \), based on which we define three relations \( \text{d(head,conjunction)} \) representing the set \( D \) of head-conjunction pairs, \( \text{beta(head,disjunction)} \) representing pairs \( (a, \beta_a) \) computed in line 6, and \( \text{delta(head,disj)} \) representing the pairs in the output \( \Delta \).

The following types of rules populate these predicates. For every (potentially non-ground) rule \( h := b_1, \ldots, b_n \in \mathcal{P} \), we have a rule of the form

\[ \text{d}(h, \text{conj}([F_1, \ldots, F_n])) := \lambda(b_1, F_1), \ldots, \lambda(b_n, F_n). \]

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

\[ \text{beta}(p(X_1, \ldots, X_n), \text{disj}(L)) := \text{d}(p(X_1, \ldots, X_n), _), \text{findall}(C, \text{d}(p(X_1, \ldots, X_n)), C), L). \]

Finally, we have the general rules

\[ \text{delta}(A, D) := \text{beta}(A, D), \text{not lambda}(A, _). \]

\[ \text{delta}(A, \text{disj}([D, F])) := \text{beta}(A, D), \lambda(A, F), \text{not equivalent}(\text{disj}([D, F])), F). \]

Here, \text{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 \text{delta} provides the next \( \Delta \), and we need to compute the next \( \mathcal{I} \) from the current \( \lambda \) and the \( \text{delta} \) 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 \text{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 \( \lambda(\text{atom},\text{formula}), \text{not outdated}(I) \). We adapt the rules defining \( d \) to use this pattern in the body, keep the rules for \( \text{beta} \) unchanged, and replace the rules for \( \text{delta} \) with the following set of rules:

\[ \lambda(A, D, A) := \text{beta}(A, D), \text{not lambda}(A, _). \]

\[ \text{aux}(A, \text{disj}([D, F]), I) := \text{beta}(A, D), \lambda(A, F), I, \text{not outdated}(I), \text{not equivalent}(\text{disj}([D, F]), F). \]

\[ \lambda(A, F, u(I)) := \text{aux}(A, F, I). \]

\[ \text{outdated}(I) := \text{aux}(A, F, I). \]
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 $\text{lambda}(\lambda, F) := \text{lambda}(\lambda, F, I)$, not outdated($I$).

4 Magic Sets for $Tc_P$

So far, we have discussed how to efficiently compute the full fixpoint of the $Tc_P$ operator. However, in practice, we are often only interested in specific queries. In this case, regular $Tc$ 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 $Tc_P$ goal-directed, and then introduce an optimization of our approach for magic programs.

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

Definition 3 (Magic Sets for PLP) The magic transform of $P$ with respect to a query $q$ is the program $M = (\text{magic}(R, q), F, \pi)$, where $\text{magic}(:, :)$ 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 $\lambda(M, q)$ associated with $q$ in the fixpoint of $Tc_M$ is equivalent to the formula $\lambda(P, q)$ associated with $q$ in the fixpoint of $Tc_P$.

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 $Tc_P$ that avoids compilation for magic atoms, and show that this computes correct formulas.

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

$\{ (a, \mu(a, I, M)) | \ a \in HB(M) \land \mu(a, I, M) \neq \bot \}$

where

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

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

That is, if $a$ is not a magic atom (which includes the facts), we use the same update operations as for the regular $Tc_M$ 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 $I$.

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

$C \cup R \models q \text{ if and only if } C \models \tau^i_q$.

The proof relies on two intermediate results: Formulas computed by $Tc_P$ are always lower bounds for those computed by $Mc_{Tc_P}$; and for any atom in the original program (i.e., excluding the magic atoms), the formulas computed by $Mc_{Tc_P}$ 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. 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 functions 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) (Darchwile 2011), using the SDD package developed at UCLA.

More precisely, given program $P$, query $q$ and iteration parameter $d$, vProbLog performs the following steps:

1. Apply the magic set transformation to $R$ and $q$.
2. Partition $\text{magic}(R, q)$ into three sets, where $R_1$ contains the rules whose bodies only use facts, $R_2$ contains all other rules that do not depend on cyclic derivations, and $R_3$ the remaining ones, and apply the transformation of Section 3 to all three.
3. Materialize the fixpoint, i.e., execute the transforms of $R_1$ and $R_2$ in sequence, iteratively execute the transform of $R_3$ $d$ 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 vProbLogplain, an implementation of $Tc_M$, i.e., without the optimizations introduced in Section 4. To implement the modified operator $Mc_{Tc_P}$, 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 vProbLogopt.

6 Experimental Evaluation

We perform experiments using the two versions of vProbLog as well as the implementation of $Tc_P$ provided

3https://bitbucket.org/tsamoura/vproblog/
4http://reasoning.cs.ucla.edu/sdd/
by the ProbLog2 system\textsuperscript{5}, which we refer to as ProbLog2, to answer the following questions:

**Q1** Does vProbLog\textsuperscript{opt} outperform vProbLog\textsuperscript{plain} in terms of running times and scalability?

**Q2** How does the performance of vProbLog\textsuperscript{opt} compare to that of ProbLog2?

**Q3** How scalable is vProbLog\textsuperscript{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 Hefflin 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 LUBM\text{001}, 1M tuples in LUBM\text{10} and 12M tuples in LUBM\text{100}). 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\textsuperscript{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 \texttt{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 LUBM\text{001}. 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\textsuperscript{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 groundings, compiling SDDs, and total time (which also includes data loading) per query on LUBM\text{001}. 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\textsuperscript{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
Table 1: Results for LUBM: time for materialization, weighted model counting and total time for vProbLog\textsuperscript{plain} (two minute timeout for materialisation) and vProbLog\textsuperscript{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\textsuperscript{opt} on LUBM100 and LUBM100 (Q3). All times in seconds.

<table>
<thead>
<tr>
<th>LUBM001</th>
<th>vProbLog\textsuperscript{plain} prob</th>
<th>vProbLog\textsuperscript{opt} prob</th>
<th>ProbLog2</th>
<th>vProbLog\textsuperscript{opt} prob</th>
<th>LUBM100</th>
<th>vProbLog\textsuperscript{opt} prob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mat wmc total</td>
<td>mat wmc total</td>
<td>ground</td>
<td>comp total answer</td>
<td>mat wmc total</td>
<td></td>
</tr>
<tr>
<td>q01</td>
<td>0.1 0.1 0.1</td>
<td>0.0 0.1 0.1</td>
<td>0.7</td>
<td>1.2 40.9 exact</td>
<td>0.5 0.2 0.7</td>
<td></td>
</tr>
<tr>
<td>q02</td>
<td>0.5 0.0 0.5</td>
<td>0.1 0.0 0.1</td>
<td>x</td>
<td>x</td>
<td>6.9 0.6 7.5</td>
<td></td>
</tr>
<tr>
<td>q03</td>
<td>0.1 0.1 0.2</td>
<td>0.0 0.1 0.2</td>
<td>6.3</td>
<td>3 x exact</td>
<td>0.2 0.3 0.5</td>
<td></td>
</tr>
<tr>
<td>q04</td>
<td>x x</td>
<td>3.6 0.5 4.2</td>
<td>49.6</td>
<td>x</td>
<td>5.2 0.7 5.8</td>
<td></td>
</tr>
<tr>
<td>q05</td>
<td>x x</td>
<td>6.3 11.3 17.5</td>
<td>56.4</td>
<td>x</td>
<td>7.5 9.8 17.3</td>
<td></td>
</tr>
<tr>
<td>q06</td>
<td>x x</td>
<td>41.8 113.4 155.2</td>
<td>47.7</td>
<td>x</td>
<td>2.3 1.9 4.1</td>
<td></td>
</tr>
<tr>
<td>q07</td>
<td>6.3 0.9 7.3</td>
<td>3.4 1.0 4.4</td>
<td>x</td>
<td>5.1 1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q08</td>
<td>x x</td>
<td>60.6 115.2 175.8</td>
<td>x</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q09</td>
<td>x x</td>
<td>22.2 2.8 25.0</td>
<td>x</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q10</td>
<td>1.6 0.1 1.6</td>
<td>0.5 0.1 0.6</td>
<td>50.6</td>
<td>x bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q11</td>
<td>0.5 5.3 5.8</td>
<td>0.3 3.4 3.7</td>
<td>0.4</td>
<td>0.9 40.7 exact</td>
<td>0.3 3.5 3.8</td>
<td></td>
</tr>
<tr>
<td>q12</td>
<td>x x</td>
<td>15.5 0.2 15.7</td>
<td>47.2</td>
<td>x</td>
<td>17.9 0.4 18.3</td>
<td></td>
</tr>
<tr>
<td>q13</td>
<td>2.1 0.0 2.1</td>
<td>1.0 0.0 1.1</td>
<td>57.9</td>
<td>x bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q14</td>
<td>0.2 80.2 80.4</td>
<td>0.2 89.4 89.6</td>
<td>1.2</td>
<td>0.2 87.7 exact</td>
<td>3.1</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>size</th>
<th>exact</th>
<th>min</th>
<th>avg</th>
<th>max</th>
<th>probLog2</th>
<th>min</th>
<th>avg</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mat</td>
<td>wmc</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.1</td>
<td>0.6</td>
<td>3.3</td>
<td>0.6</td>
<td></td>
<td>0.5</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>11</td>
<td>0.1</td>
<td>1.0</td>
<td>5.7</td>
<td>0.6</td>
<td></td>
<td>0.5</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>12</td>
<td>0.2</td>
<td>10.9</td>
<td>93.1</td>
<td>0.6</td>
<td></td>
<td>0.6</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>13</td>
<td>0.2</td>
<td>15.7</td>
<td>55.9</td>
<td>0.6</td>
<td></td>
<td>0.6</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>14</td>
<td>0.7</td>
<td>34.9</td>
<td>106.7</td>
<td>0.3</td>
<td></td>
<td>0.6</td>
<td>1.2</td>
<td>3.6</td>
</tr>
<tr>
<td>15</td>
<td>3.2</td>
<td>34.2</td>
<td>65.6</td>
<td>0.8</td>
<td></td>
<td>1.3</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3.2</td>
<td>12.0</td>
<td>24.6</td>
<td>0.7</td>
<td></td>
<td>1.6</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2.1</td>
<td>8.1</td>
<td>15.1</td>
<td>0.8</td>
<td></td>
<td>2.1</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1.7</td>
<td>23.6</td>
<td>23.6</td>
<td>0.9</td>
<td></td>
<td>2.7</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.1</td>
<td>4.6</td>
<td>12.1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.3</td>
<td>3.9</td>
<td>13.4</td>
</tr>
</tbody>
</table>

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

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 for Programs with Monotone Recursive Aggregates. In LPNMR.
Borgwardt, S.; Ceylan, İ. İ.; and Lukasiewicz, T. 2018. Recent advances in querying probabilistic knowledge bases. In International Joint Conference on Artificial Intelligence.