Provenance for Probabilistic Logic Programs

Shaobo Wang
Georgetown University
sw1001@georgetown.edu

Hui Lyu
University of Pennsylvania
huilyu@seas.upenn.edu

Jiachi Zhang
Georgetown University
jz598@georgetown.edu

Chenyuan Wu
Beijing Jiao Tong University
16221213@bjtu.edu.cn

Xinyi Chen
Shanghai Jiao Tong University
cxinyic@sjtu.edu.cn

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Susan B. Davidson
University of Pennsylvania
susan@seas.upenn.edu

Chen Chen
Megagon Labs
chen@megagon.ai

ABSTRACT

Despite the emergence of *probabilistic logic programming* (PLP) languages for data driven applications, there are currently no debugging tools based on provenance for PLP programs. In this paper, we propose a novel provenance model and system, called P3 (Provenance for Probabilistic logic Programs) for analyzing PLP programs. P3 enables four types of provenance queries: traditional explanation queries, queries for finding the set of most important derivations within an approximate error, top-K most influential queries, and modification queries that enable us to modify tuple probabilities with fewest modifications to program or input data. We apply these queries into real-world scenarios and present theoretical analysis and practical algorithms for such queries. We have developed a prototype of P3, and our evaluation on real-world data demonstrates that the system can support a wide-range of provenance queries with explainable results. Moreover, the system maintains provenance and execute queries efficiently with low overhead.

1 INTRODUCTION

In many data intensive applications, there has been a paradigm shift towards probabilistic and statistical reasoning. In some cases, it is in support of programs that rely on probability distributions. Probabilistic reasoning is used as a basis to trade off performance and accuracy, when collecting and aggregating readings from sensors. In other cases, probabilistic reasoning is dictated by use of external libraries, typically involving programs that rely on outputs of machine learning libraries which are intrinsically probabilistic.

Consequently, over the past few years, there is an emergence of *probabilistic logic programming* (PLP) languages. Many of these languages use database-style declarative conjunctive rules that are loosely based on Datalog semantics with extensions to handle probability. These include SLP [21], Datalog$_P$ [7], PRISM [28], ICL [23], ProbLog [24]. A common thread across these systems is that they allow both data and rules to be probabilistic, and are based on Sato's distribution semantics [27] among possible worlds under a given probability distribution. These languages cover the bulk of PLP languages in use today.

In addition, some PLP languages are used as machine learning models, such as Markov Logic Networks (MLN) [26] and Probabilistic Soft Logic (PSL) [3]. These PLP programs are combined with probabilistic graphical models (PGM), or converted to weighted Boolean formulas [6], for inference and learning. Beyond these languages, BAYONET [9], which introduces their own flavor of probabilistic network programming language.

Despite the proliferation of declarative PLP languages, there are currently no tools that enable us to debug and analyze programs. Given the declarative feature, a natural question to ask is whether *data provenance* [11] can be used for debugging these declarative data-driven systems. However, prior provenance work falls short in enabling debugging capabilities for PLP programs because they are geared primarily towards traditional relational databases. The most obvious candidate is provenance in probabilistic databases [25]. However, these systems do not work for PLP programs, given that only tuples are labeled with independent probabilities while the operators in SQL remain deterministic. This is unlike PLP programs where the rules (and hence the operators used for executing these rules) are probabilistic. Consequently, systems that support provenance in probabilistic databases do not work on queries or programs with uncertainty built into the algorithm rather than the underlying data.

In this paper, we present a model and system called P3 (Provenance for Probabilistic logic Programs). P3 enables a novel form of provenance, which we term *probabilistic provenance*. P3 is aimed at the first class of declarative PLP programs described above. All of these languages consist of programs that are a union of weighted conjunctive rules with recursion (without negation), and adhere to the possible world semantics. A representative example of this language that we use throughout our paper is ProbLog, although the approach can be generalized to similar languages. In ProbLog, tuples and rules are labeled with probabilities. ProbLog-like languages encompasses a wide range of PLP programs that involve reasoning with uncertainty over data. To the best of our knowledge, our work is the first one providing a comprehensive case study on using probabilistic provenance for PLP analysis.

By allowing PLP programs to be analyzable using probabilistic provenance, P3 enables a whole series of novel provenance queries not previously possible, including (1) showing the derivation graphs that explain tuples and their probability values; (2) finding the set of most important derivations for a derived tuple based on its provenance; (3) finding the top-K most influential variables (including base tuples and rules) for a derived tuple based on its provenance; and (4) supporting modification queries, where we can answer how to modify variables' probabilities to

efficiently change the derived tuple probability to a target score with small cost.

The key contributions of this paper are as follows:

- **Probabilistic provenance model.** We propose a provenance based approach to reason PLP programs with semantics similar to ProbLog. Provenance is maintained through both graph-based representation (provenance graph) and algebraic representation (provenance polynomials).
- **Probabilistic provenance queries.** We demonstrate how the provenance model can enable provenance queries to answer explanation, derivation, influence and modification questions about derived tuples. We further demonstrate how these queries can generate meaningful results in the presence of cycles in recursive rules.
- **Implementation and evaluation.** We have implemented P3, and evaluated the system over use cases based on real-world data. Through our use cases, we also demonstrate how P3 can help debug and fix errors in ProgLog programs. Our results demonstrate that P3 can enable a range of novel provenance queries with low overhead at maintenance and query time.

2 BACKGROUND

Given our choice of ProbLog as a representative example, we first provide an introduction to the salient features of the language. ProbLog's syntax is based on Datalog, with the main difference being that all base tuple clauses and rule clauses are labeled with probabilities. A ProbLog program specifies a probability distribution over all possible non-probabilistic subprograms of the ProbLog program. The semantics of ProbLog is defined by the success probability of a query, which is the probability that the queried tuple succeeds among these subprograms.

```
rid p1: H() :- B1(),B2(),...,Bn().
tid p2: B1().
```

Figure 1: ProbLog syntax

Figure 1 summarizes the ProbLog syntax. There are two types of clauses: a weighted conjunctive rule (first line in Figure 1 where `H()` means the rule head, and `B1(),B2(),..., Bn()` are relations in the rule body), and a probabilistic base tuple (second line in Figure 1). Each conjunctive rule has a `rid`, labeled with a probability (`p1` in Figure 1) of being true. Each base tuple has a `tid`, labeled with a probability score of existence (`p2` in Figure 1).

2.1 Running Example

As a running example used throughout the paper, we consider the `Acquaintance` ProbLog program shown in Figure 2. The program computes all pairs of people who may know each other.

```
r1 0.8: know(P1,P2) :-
        live(P1,C), live(P2,C), P1!=P2.
r2 0.4: know(P1,P2) :-
        like(P1,L), like(P2,L), P1!=P2.
r3 0.2: know(P1,P3) :-
        know(P1,P2), know(P2,P3), P1!=P3.
t1 1.0: live("Steve","DC").
t2 1.0: live("Elena","DC").
t3 1.0: live("Mary","NYC").
t4 0.4: like("Steve","Veggies").
t5 0.6: like("Elena","Veggies").
t6 1.0: know("Ben","Steve").
```

Figure 2: `Acquaintance` ProbLog rules and base tuples

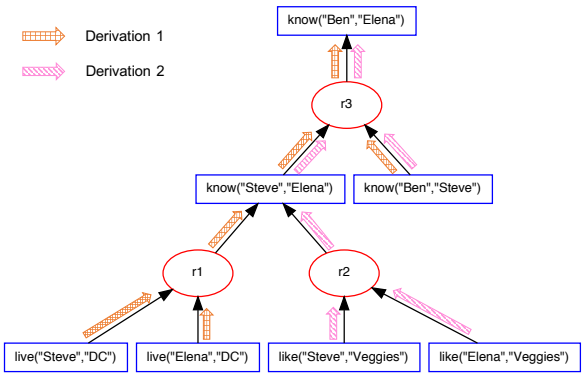

Figure 3: Provenance graph of `know("Ben","Elena")` with annotated derivations

In the `Acquaintance` program, people who live in the same place (`r1`) or like the same item (`r2`) may know each other with some probability. The `Acquaintance` relationship can also be transitive with some probability (`r3`). Given the program in Figure 2, our P3 system demonstrates how we can derive more `Acquaintance` relations based on existing relations. For example, we can infer whether `know("Ben","Elena")` can be derived by querying the tuple, and also derive the probability that Ben knows Elena.

Beyond inferring probability values, P3 will enable the following classes of queries:

- **Probability explanations.** Figure 3 is a provenance graph which explains how `know("Ben","Elena")` is derived. There are two derivations in Figure 3, annotated by two types of arrows. They share some paths to derive the tuple. We can use provenance queries to answer which derivation contributes more to the tuple probability.
- **Change modification.** After getting the probability score of `know("Ben","Elena")`, we may be dissatisfied with its score, and would like to increase or decrease it, either by changing base tuple values or rule weights. To minimize disruption, we would like make the fewest possible changes. A provenance query may be used to find out the variable that influences the derivation of `know("Ben", "Elena")` most, so that if we would like to modify its probability values with the fewest number of variable changes, we can start with the most influential variables. Likewise, we can also use provenance queries to determine the fewest number of modifications to rules.

2.2 ProbLog Semantics

As is shown in Figure 1, each clause c_i (whether a base tuple or rule) is labeled with a probability p_i. These probabilities are mutually independent. A ProbLog program $T = \{p_1 : c_1, \cdots, p_n : c_n\}$ defines a probability distribution over logic programs $L \subseteq L_T = \{c_1, \cdots, c_n\}$ as follows:

$$P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i) \tag{1}$$

where L denotes one non-probabilistic subprogram, also called one possible world. The success probability $P(q|T)$ of a query q in a ProbLog program T is defined as:

$$P(q|L) = \begin{cases} 1, & \exists \theta : L \models q\theta \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$P(q, L|T) = P(q|L) \cdot P(L|T) \tag{3}$$

$$P(q|T) = \sum_{L \subseteq L_T} P(q, L|T) \tag{4}$$

The success probability of query q corresponds to the probability that the query q has a proof, given the probability distribution over logic programs. As mentioned in [24], the success probability of a ProbLog query can be computed as the probability of a Boolean monotone DNF (Disjunctive Normal Form) formula of binary variables being true, which is an NP-hard problem [29]. The DNF formula is obtained by SLD-resolution [8], and then represented by binary decision diagrams (BDDs) [4] in order to compute its probability efficiently.

The reason we choose ProbLog semantics to be the focused of this paper is as follows:

- Sato's distribution semantics or possible worlds semantics [5] is widely used in probabilistic databases, where the base tuple score has a meaningful semantics - probability. In ProbLog, the rule weight also means the probability that the rule is true.
- Provenance in probabilistic databases is intuitively easy to be extended in ProbLog's syntax and semantics.
- Alternative PLP languages (such as PRISM, ICL, SLP, and Datalog$_P$) that adhere to the Sato's distribution semantics impose extra constraints such as mutual exclusion constraint for rule bodies of the same rule head, and do not allow for recursion. However, ProbLog is more general and expressive, and it supports recursive rules which is necessary in network data applications. Hence, provenance support for ProbLog naturally carries forward to these alternative languages.

Note that while we focus on ProbLog, given ProbLog's generality, the concepts that we introduce in this paper can be generally applied to any declarative probabilistic programs that are based on union of conjunctive rules.

3 PROVENANCE MODEL

In this section, we introduce the provenance model used in P3. Traditionally, provenance can have both graph representation (provenance graph) and algebraic representation (provenance polynomials). P3 provides both types of representations, and as we will see later, different representations support different types of provenance queries.

3.1 Provenance Graph

Given a ProbLog-like PLP program, we model its provenance as a directed graph $G(V, E)$, which describes the data dependencies (see Figure 3 as an example).

The vertex set V in G consists of tuple vertices and rule execution vertices. An oval denotes a rule execution vertex, and a rectangle denotes a tuple vertex. We also annotate the associated probability for each tuple vertex and rule execution vertex. For a concise graph representation, we omit the probability values in the example provenance graph, but they are maintained as an associated attribute and can be queried.

The edge set E consists of unidirectional edges that represent data dependencies between tuple vertices and rule execution vertices. An edge is always pointing from a tuple vertex t to a rule execution vertex r, or from a rule execution vertex r to a tuple vertex t. The former indicates t is an input of r, and the latter means t is derived through r. The provenance graph G shows the complete derivation history of teh PLP program. For a queried tuple q (derivable from the PLP program), it provenance is the a subgraph of G rooted by q's corresponding tuple vertex.

For PLP programs containing recursive rules, it may generate cycles during the derivations. A cycle appears when a derived tuple can also be an input tuple in one of its own derivations. However, we prove that the cycles in the graph can be removed without affecting computing the success probability of the queried tuple q (See Section 3.3 for details).

Figure 3 shows a simple provenance graph for `Acquaintance` example. Intuitively, we can find two derivations of tuple `know("Ben","Elena")` in the graph. `know("Ben","Elena")` is derived through rule `r3` when both `know("Ben","Steve")` and `know("Steve","Elena")` are true. Then there are two ways to derive `know("Steve","Elena")`: through `r1` or through `r2`. The two derivations also share some paths.

3.2 Provenance Maintenance

The provenance maintenance for PLP shares similarities with that of ExSPAN engine [31]: we perform an automatic rule rewrite of the PLP program to create and maintain provenance information at runtime as a side-computation along with the evaluation of the original PLP program. During this process, our system naturally maintains provenance as a graph: the *direct* dependencies of the tuple and rule executions (i.e., the edges in the provenance graph) are captured and stored in relational tables.

More concretely, each rule `rid p H() :- B1(),...,Bn().` is rewritten into three rules at compile time [1]:

```
H() :- B1(),...,Bn().
prov(H(),p,rid) :- B1(),...,Bn().
rule(rid,(B1(),...,Bn())) :- B1(),...,Bn().
```

The first rule performs the original tuple derivation; the second rule records the dependency between the rule execution and its input tuples (i.e., `B1(), ..., Bn()`); the final rule records that the derived tuple `H()` has a derivation from this particular rule execution. In addition, we further identify rules labeled with probabilities and associate the rule execution vertices with their corresponding probabilities. For example, consider rule `r1` in Figure 2, its execution results in two dependencies (captured in the `prov` and `rule` table respectively): `know(P1,P2)`$\xleftarrow{0.8}$`r1` and `r1`$\leftarrow$`(live(P1,C), live(P2,C))`. These are reflected in Derivation 1 in Figure 3.

The maintenance of provenance only adds a reasonable constant cost for each rule evaluation, and is expected to have limited impacted on the scalability of the program or application.

3.3 Provenance Querying

With the captured direct dependencies (i.e., the edges in the provenance graph), we can extract the complete provenance of a queried tuple y by recursively traversing the graph: starting from y, we traverse the graph by following the edges until we reach base tuples. The returned provenance is presented in a pre-determined representation. Generally, the graph traversal allows us to extract any provenance representation defined as a provenance semiring [11].

Provenance polynomials. In this paper, provenance polynomial is adopted as the basis for answering more complex queries (such as identifying the most influential base tuples, etc), for

[1] For performance considerations, additional optimization is adopted in the actual implementation to ensure that the rule body, which is the same for all the three rules, only needs to be evaluated once.

its close connection to probability calculation. The provenance polynomial for a queried tuple q in ProbLog is a Boolean formula $\lambda(q)$, where each literal denotes one individual tuple or rule. The literals are Boolean variables and each has some probability of being true. There are two binary operators "$\cdot$" and "+" in $\lambda(q)$. Specifically, "$\cdot$" denotes conjunctive use of multiple tuple or rules for the derivation of a derived tuple, and "+" denotes union of alternative derivations for the same derived tuple.

For example, if tuple q is derived from a conjunctive rule r_1 which takes tuple t_1 and t_2 as input, then the provenance polynomial of this derivation is $r_1 \cdot t_1 \cdot t_2$. Suppose that q has another derivation from rule r_2 which takes tuple t_3 and t_4 as input, then the complete provenance polynomial $\lambda(q)$ is $r_1 \cdot t_1 \cdot t_2 + r_2 \cdot t_3 \cdot t_4$.

Consider the `Acquaintance` example (see Figure 2), the provenance polynomial of the derived tuple `know("Ben","Elena")` is r_3 · `know("Ben","Steve")` · `know("Steve","Elena")`, where `know("Steve","Elena")` is a derived tuple that can be further expanded. The complete provenance polynomial of tuple `know("Ben","Elena")` is the following:

r_3 · `know("Ben","Steve")` ·
(r_1 · `live("Steve","DC")` · `live("Elena","DC")` +
r_2 · `like("Steve","Veggie")` · `like("Elena","Veggie")`)

This provenance polynomial represents the two derivations of `know("Ben","Elena")`.

Success probability of provenance polynomial. Noting that this example provenance polynomial consists of only base tuple literals and rule literals, we can, at least in theory, compute the success probability of the provenance polynomial. We simply treat the provenance polynomial as a formula of random variables; we can then calculate its success probability by plugging in the probabilities of the base tuples and the rules, which are provided as input to a ProbLog program. More formally,

$$\mathbf{P}\,[\lambda(q)\,] = \mathbf{E}[\,\widetilde{\lambda}(q)\,] \tag{5}$$

where $\widetilde{\lambda}$ denotes the arithmetization form of λ. $\mathbf{P}[]$ denotes success probability, and $\mathbf{E}[]$ denotes expectation.

However, computing the success probability of an arbitrary provenance polynomial is an NP-hard problem [29]. For instance, the success probability of formula $a+b$ is not the sum of the probability of a and the probability of b because of Inclusion–Exclusion principle. Therefore, in practice, we use Monte-Carlo sampling [14] approach to estimate the probability value.

Handling cycles. For a ProbLog program containing recursive rules, a cycle may appear in the provenance graph where a derived tuple can also be an input tuple for one of its own derivation. This raises issues when we retrieve provenance polynomials from the provenance graph – the provenance polynomials may potentially be arbitrarily long (by infinitely expanding the graph traversal through cycles) or contain literals that correspond to derived tuples (by stopping the graph traversal at these derived tuples). This would greatly affect the calculation of the success probability of the provenance polynomial.

However, we show that any such cycles can be removed from the provenance graph without affecting the success probability of the provenance polynomial. Generally, if the queried tuple q has cycles in its provenance graph, its provenance can be written as a polynomial in the following form:

$$\lambda(q) = (P_E + P_I) \cdot \lambda(q) + P'_E + P'_I \tag{6}$$

where P_E and P'_E each denotes a polynomial that only contains base tuples and rule literals. P_I and P'_I each denotes a polynomial containing other derived tuple literals (e.g., the provenance graph contains other cycles that do not involve q). The polynomial $(P_E + P_I) \cdot \lambda(q)$ indicates that some derivations of q depend on the existence of q itself.

Given the existence of cycles, q has infinitely many derivations: some derivations can traverse around a cycle multiple times. We define a series of formula that progressively include derivations with multiple-round cycles:

$\lambda^0(q)$ includes derivations containing no cycles. We have

$$\lambda^0(q) = P'_E + P'_I \tag{7}$$

$\lambda^1(q)$ includes derivations containing at-most-one-round cycles, that is, it includes all derivations in $\lambda^0(q)$ but also derivations containing exact one-round cycles. We have

$$\begin{aligned}
\mathbf{P}[\,\lambda^1(q)\,] &= \mathbf{P}[\,(P_E + P_I) \cdot \lambda^0(q) + P'_E + P'_I\,] && (8)\\
&= \mathbf{P}[\,(P_E + P_I) \cdot (P'_E + P'_I) + P'_E + P'_I\,] && (9)\\
&= \mathbf{P}[\,(1 + P_E + P_I) \cdot (P'_E + P'_I)\,] && (10)\\
&= \mathbf{P}[\,P'_E + P'_I\,] \text{ (Absorption Law)} && (11)
\end{aligned}$$

Thus, we have $\mathbf{P}[\,\lambda^0(q)\,] = \mathbf{P}[\,\lambda^1(q)\,]$. Likewise, we have $\mathbf{P}[\,\lambda^0(q)\,] = \mathbf{P}[\,\lambda^1(q)\,] = \mathbf{P}[\,\lambda^2(q)\,] = \cdots = \mathbf{P}[\,\lambda^\infty(q)\,] = \mathbf{P}[\,P'_E + P'_I\,]$. Therefore, if we are only concerned about calculating its success probability, we can simplify provenance polynomial for q as:

$$\lambda(q) = P'_E + P'_I \tag{12}$$

Furthermore, to simplify P'_I, if any derived tuple q_k in P'_I that has cycles in its derivation graph, we can replace $\lambda(q_k)$ with $\lambda^0(q_k)$. Then we have

$$\lambda(q) = P'_E + P'_I \mid (\forall \lambda(q_k), \lambda(q_k) = \lambda^0(q_k)) \tag{13}$$

In cases where $\lambda^0(q_k)$ also includes $\lambda(q)$, we can replace $\lambda(q)$ with $P'_E + P'_I \mid (\forall \lambda(q_k), \lambda(q_k) = \lambda^0(q_k))$ in $\lambda^0(q_k)$, and then remove $\lambda^0(q_k)$ following the same process (Equation (6) to Equation (12)). Recursively, the final provenance polynomial for q consists of only base tuple literals and rule literals.

4 REASONING PLP WITH PROVENANCE

The provenance of a queried tuple can be used in a variety of ways to reason a PLP program. For instance, users may want to know how a tuple is derived (Explanation Query), which derivation contributes most to achieving the probability of the derived tuple (Derivation Query), which base tuple has the most influence on the derivation of the queried tuple (Influence Query), and how to change the base tuples to achieve a given target value (Modification Query). These questions can help users reason and debug a PLP program. A summary of the provenance query types discussed in this paper is shown in Table 1.

4.1 Explanation Query

An *Explanation Query* returns the complete derivations of the queried tuple. The success probability of the queried tuple as well as any intermediate derived tuples can be computed efficiently using Monte-Carlo simulation. Consider the `Acquaintance` program (see Figure 2), an example Explanation Query is:

Query 1: Show the derivations of the derived tuple `know("Ben","Elena")`.

By querying the provenance of `know("Ben","Elena")`, we get provenance polynomial λ(`know("Ben","Elena")`) =

Table 1: Summary of provenance query types

Query Type	Operation
Explanation Query	Illustrate the derivations graph of the queried tuple for explanation
Derivation Query	Find the set of most important derivations of the queried tuple given some approximation error
Influence Query	Show the (top-K) most influential variables (base tuples and rules) of the queried tuple
Modification Query	Modify the variables to achieve a target probability value with minimal change

r_3 · know("Ben","Steve") ·
(r_1 · live("Steve","DC") · live("Elena","DC") +
r_2 · like("Steve","Veggie") · like("Elena","Veggie"))

Its success probability $\mathbf{P}[\,\lambda(\texttt{know("Ben","Elena")})\,] = 0.18$, meaning there is a 18% probability Ben knows Elena. The provenance polynomial also corresponds to the visual provenance graph shown in Figure 3 and explained in Section 3.1: tuple know("Ben","Elena") has two derivations that share parts of their paths.

4.2 Derivation Query

Sometimes, the complete explanation can be too long or too complex for users to (intuitively) understand; instead, users may be interested in a more compact explanation that still retains a *close-enough* success probability. For example, in the Acquaintance scenario, suppose two person Alice and Bob share many common hobbies (and thus may know each other from the same hobby groups). But if Alice's and Bob's addresses show that they are next-door neighbors, then this serves as a strong explanation for why Alice knows Bob; it easily trumps the share-similar-hobbies explanation which can be long and tedious.

More generally, after computing the success probability of a queried tuple, it is intuitive to ask: (a) which derivations contributed most to the success probability? (b) can we find a small set of important derivations to achieve an approximate probability? The *Derivation Query* is used to answer these types of questions. Formally, given a provenance polynomial λ, the Derivation Query returns a *sufficient provenance* λ^S:

$$|\,\mathbf{P}[\,\lambda\,] - \mathbf{P}[\,\lambda^S\,]\,| \leq \epsilon \tag{14}$$

where λ^S consists of a subset of the monomials in λ, and ϵ is a user-specified error limit. As an example, consider the Acquaintance program. An example Derivation Query is:

Query 2: What are the most important derivations of know("Ben","Elena"), assuming an error limit of ϵ?

The original provenance polynomial is given by Query 1, which consists of two monomials, corresponding to the two derivations of know("Ben","Elena"). The returned sufficient provenance varies with the value of ϵ: When ϵ is set to 0.001, the sufficient provenance remains the same, as removing either of the monomials would yield a success probability change greater than ϵ. After we increase ϵ to 0.01, the returned sufficient provenance λ^S(know("Ben","Elena")) =

r_3 · know("Ben","Steve") ·
r_1 · live("Steve","DC") · live("Elena","DC")

It removes one derivation and presents the most important derivation: The fact that Steve and Elena live in the same city contributes more to the derivation of know("Steven","Elena") (and then, in turn, the derivation of know("Ben","Elena"), since rule r1 has a significantly higher probability than r2.

Compute sufficient provenance. For a queried tuple q, each monomial in the provenance DNF formula $\lambda(q)$ corresponds to one derivation path of q. The probability of each monomial is computed by multiplying all the probabilities of literals in the monomial. So it is easy to find the most important derivation of q. However, the probability of λ is not the sum of the probability of each monomial in λ, since these monomials can be correlated. In fact, finding the smallest sufficient provenance of λ, i.e., the $\epsilon-$approximate polynomial with the minimal number of monomials, is NP-hard [25].

A naïve way to compute sufficient provenance is to sort the monomials in the provenance polynomial according to their probabilities in a descending order. We can then progressively remove monomials that have the lowest probabilities, until the error limit ϵ is reached. This doesn't guarantee the smallest sufficient provenance but it provides an approximation. In our evaluation in Section 6, this naïve approach performs surprisingly well.

Alternatively, prior work [25] on approximate lineage for probabilistic databases proposed an algorithm to efficiently find an ϵ-approximation of provenance polynomial. Our system extends it for PLP programs. Briefly, the algorithm finds a sufficient provenance of a $k-$literal polynomial λ in the following steps:

- **Step 1.** It first finds an arbitrary match of λ: A match of λ consists of a set of *independent* monomials in λ. Since these monomials are independent, the success probability of the match can be efficiently computed.
- **Step 2.** If the match is already an $\epsilon-$approximation of λ, then the match is returned as the final result.
- **Step 3.** Otherwise, the monomials in λ are partitioned into groups, where the monomials in each group share at least one literal. Therefore, each group can be rewritten into the form $l \cdot (m_1 + m_2 + ... + m_k)$, where l is the literal shared by all monomials in the group.
- **Step 4.** The algorithm can then recursively find the sufficient provenance for $m_1 + m_2 + ... + m_k$, which is a (k-1)$-$literal polynomial. The algorithm is guaranteed to terminate at 1-literal polynomial.

Although this algorithm is more efficient than the naïve approach, it relies heavily on the choices of the match (in Step 1) and the groups (in Step 3). In some cases, it provides little reduction in the size of the provenance polynomial.

4.3 Influence Query

In addition to derivations of a queried tuple, users may also be interested in the influence of each literal on the queried tuple. An *Influence Query* returns the most influential literals (i.e., rule weight or base tuple probability) of a given derived tuple.

Intuitively, the influence of a literal x_i on a provenance polynomial λ measures the impact on the success probability of λ when the value of x_i changes. For example, a counterfactual base literal would have a large influence, because setting it to be false would invalidate the derived tuple. We adopt the definition proposed in a prior work [13]: Consider λ as a multiple-variable function, the influence of x_i is the partial derivative $\frac{\partial \lambda}{\partial x_i}$.

Definition 4.1. **Literal influence [13].** The influence of a literal x_i on the provenance polynomial λ, denoted $\text{Inf}_{x_i}(\lambda)$, is:

$$\text{Inf}_{x_i}(\lambda) = \frac{\partial \lambda}{\partial x_i} = \mathbf{P}[\lambda|_{x_i=1}] - \mathbf{P}[\lambda|_{x_i=0}] = \mathbf{E}[\widetilde{\lambda}|_{x_i=1} - \widetilde{\lambda}|_{x_i=0}] \quad (15)$$

where $\widetilde{\lambda}$ denotes the arithmetization form of λ. $\mathbf{P}[]$ denotes probability, and $\mathbf{E}[]$ denotes expectation. $\mathbf{P}[\lambda] = \mathbf{E}[\widetilde{\lambda}]$.

Based on this definition, P3 can answer Influence Query efficiently by using Monte-Carlo sampling approach to estimate $\mathbf{E}[\widetilde{\lambda}|_{x_i=1} - \widetilde{\lambda}|_{x_i=0}]$. Consider the `Acquaintance` program. An example Influence Query is:

Query 3: What are the most influential literals to the success probability of `know("Ben","Elena")`?

Table 2: Results of Influence Query

Variable	Influence Value
r_3	0.896
r_1	0.2
t_6	0.1792

Table 2 lists the top-3 most influential literal. It shows that rule `r3` is the most influential. It makes intuitive sense: Rule `r3` is the critical recursive rule that allows for the generation of multi-hop `know` relationships, including the case for `Ben` and `Elena` who know each other through their direct relationships with `Steve`.

4.4 Modification Query

For a queried tuple, our system can answer which derivations contribute most, and which individual variables are the most influential ones. Users may further explore how to effectively modify the base variables to adjust the queried tuple's success probability, for example, when the success probability of the queried tuple is suspiciously low (or high). The *Modification Query* aims to answer this type of question. We use the results from the Influence Query as a basis to answer Modification Queries.

Based on Equation (5), we can easily get

$$\mathbf{P}[\lambda_t] = \text{Inf}_{x_i}(\lambda) \cdot p(x_i) + \mathbf{P}[\lambda|_{x_i=0}] \quad (16)$$

It means that for any variable x_i, the success probability of the derived tuple is positively correlated to $p(x_i)$, and the positive coefficient is the influence value of x_i. Equation (16) effectively considers $\mathbf{P}[\lambda]$ as a function of $p(x_i)$.

Query 4: Given a target success probability of the queried tuple, how should we modify the base literals to achieve the targe with minimal cost?

Here, we consider the cost is defined as:

$$\text{Cost} = \sum_i |\Delta p(x_i)| \quad (17)$$

which is the summation of the probability change of each modified literal. To find a good solution for Query 5, We follow a heuristic-based greedy algorithm. The algorithm selects the most influential variable, and change its value to reach the target probability. Sometimes, even changing the most influential variable to the maximum value of 1 (or minimum value of 0) is still not enough. In this case, we continue to find the most influential variable in the remaining variables, until the target probability is reached. The most influential variable in each iteration corresponds to the highest slope to change, so the total cost of modification of variables is minimized. The strategy returned by this heuristic-based algorithm is not guaranteed to be an optimal solution of minimal cost, but, by leveraging the result of Influence Query, it works well empirically.

Take the `Acquaintance` program as an example. The original success probability of `know("Ben","Elena")` is 0.18. If we hope to modify the value to be above 0.5, the answer returned by P3 is that we should change variable r_3 to 0.56. The total cost of the change is 0.36. Further evaluation of the effectiveness of the heuristic-based algorithm is elaborated in Section 5.2.

5 CASE STUDY

The `Acquaintance` program is a simple example to illustrate how provenance is utilized to reason the evaluation results of ProbLog programs. In this section, we present two case studies to showcase how users can benefit from P3's practical reasoning capabilities. Performance evaluations will follow in the next section.

Visual Question Answering. Our first use case describes a scenario from multi-modal learning in the machine learning community, called Visual Question Answering (VQA). The learning task is to answer user's questions regarding a presented photo, e.g., identifying a (partially blocked) object in the photo. In this scenario, we use provenance to provide some insights from human-explainable perspective, e.g., to identify the most influential features that lead to the learning result.

Mutual Trust in Social Network. Our second use case is similar to the `Acquaintance` example, which, at the core, computes probabilistic recursive rules. In this scenario, we use sampled data from real-world trust network, and computes pair-wise mutual trust between each pair of peers in the network. We use provenance to identify critical *direct* trust relationships.

5.1 Visual Question Answering

This use case involves supporting the Visual Question Answering (VQA) [2] task using a probabilistic logic program. In VQA, the system answers a natural language question about an image. This task comgines natural language, image processing, and logical reasoning. Prior work on PSL based VQA program [1] can decompose and reason VQA task in terms of logic rules, but lacks the ability to explain the derivation procedure for the answer. Here, we rewrite the VQA-PSL program in ProbLog syntax, and provides provenance queries to explain VQA answers.

The input of this use case is a set of tuples parsed from both the image (`hasImg` relation) and the question (`hasQ` relation) with some probability. The similarity between words are also base tuples (`sim` relation). The input also includes `word` relation tuples. It is the set of answers with prior confidence scores as probability. The scores can come from some sources, e.g. a dictionary with word frequency.

The VQA-PSL program [1] is written into an equivalent VQA ProbLog program shown in Figure 5. Each rule is associated with a weight probability, which can be assigned any reasonable values. The four rules in Figure 5 explain how the final answers (`ans` relation) can be derived step-by-step combining image information, question parsing information, and words similarity knowledge.

In Figure 5, rule `r1` extends the `hasImg` relation by replacing items with their synonyms (but with a diminishing score). For example, `hasImg(V,"apple","in","background")` can be extended to `hasImg(V,"fruit","in","background")`, as `apple` is similar to `fruit` (indicated by `sim("apple","fruit")`).

Rule `r2` states that a word in the dictionary automatically becomes a candidate answer. However, it may then be out-weighted

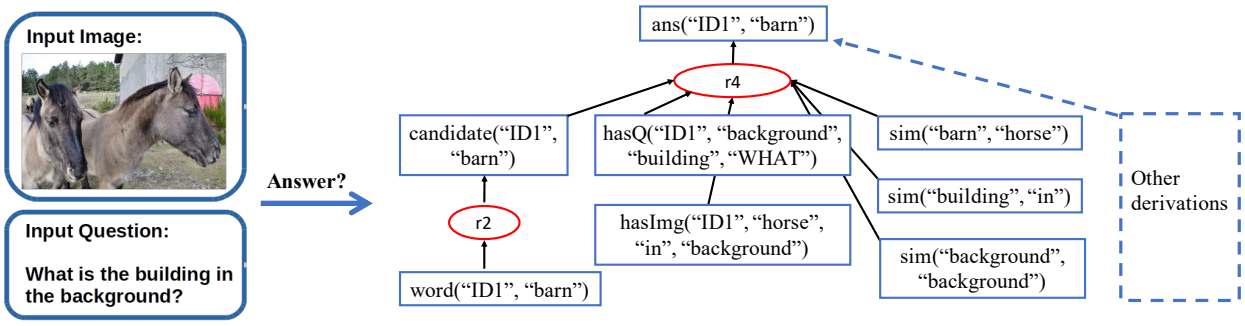

Figure 4: Abstract provenance graph of `ans("ID1","barn")`. The complete most important derivation is shown.

by results generated by rule `r3` which further considers how the candidate answer is correlated with user's question and the objects appeared in the photo.

Finally, rule `r4` takes all the candidate answers and evaluates their correlation with the question and the objects in the photo. Stronger correlation will have higher scores. Note that we assigned equal weight to all words in the dictionary such that the predicted result is unbiased.

```
r1 w1:hasImgAns(V,Z,X1,R1,Y1) :-
      word(V,Z), hasImg(V,X1,R1,Y1),
      sim(Z,X1), sim(Z,Y1).
r2 w2:candidate(V,Z) :- word(V,Z).
r3 w3:candidate(V,Z) :- word(V,Z),
      hasQ(V,X,R,Y), hasImgAns(V,Z,X1,R1,Y1),
      sim(R,R1), sim(Y,Y1), sim(X,X1).
r4 w4:ans(V,Z) :- candidate(V,Z),
      hasQ(V,X,R,"WHAT"), hasImg(V,Z1,R1,X1),
      sim(Z,Z1), sim(R,R1), sim(X,X1).
```

Figure 5: VQA ProbLog program

Given an image and a question shown in Figure 4, the input tuples can be obtained using a image captioning system and natural language processing system. The word similarity inputs are obtained from Word2Vec library. The evaluation of the ProbLog program returns the tuple `ans("ID1","barn")` as the the result with the highest confidence, meaning that the building in the presented photo (identified by "ID1") is determined as a barn. We query the provenance of `ans("ID1","barn")` to check whether the returned provenance can give meaningful explanations.

Query 1A: Show the derivations of `ans("ID1", "barn")`. Since the provenance graph for the queried tuple has many branches, to enhance readability, we only display a condensed graph shown in Figure 4 instead of the complete fine-grained provenance graph. The actual complete graph can be generated by our system if the user hopes to take a closer look at it.

In Figure 4, we show the most important derivation to `ans("ID1", "barn")`, and leave out other derivations. In the most important derivation, word "barn" is selected as a candidate (rule `r2`) given that "barn" is included in the dictionary. Besides, "barn" is strongly correlated to the horse appeared in the photo, adding some other words similarity relation, `ans("ID1", "barn")` is derived through rule `r4`. There are other derivations that can also derive `ans("ID1", "barn")`, but they contribute less to the final probability of `ans("ID1", "barn")`.

Query 1B: Show the most influential base tuple to `ans("ID1", "barn")`. Our next query concerns the base tuple that influences the final learning result the most. In other words, we would like to understand which factor would affect

Figure 6: Image of horses in front of a church

the predicted result the most if its value was changed. We performed the influence query on the returned provenance result, then identified that the base tuple `word("ID1","barn")` is the most influential one. This is reasonable, however, less interesting as it is simply stating the obvious truth that a word must be considered as a potential answer to affect the result. As the program mainly depended on the information from `hasImg()` and `sim()` for prediction, we decided to find the most influential tuples in these two relations respectively.

We further identified the base tuple `hasImg("ID1", "horse", "in", "background")` as the most influential one in `hasImg()`. Its influence value was approximately 0.005. Effectively, this result states that the building being identified as a barn is largely influenced by the fact that there is a horse in the background. In `sim()`, the most influential tuple was `sim("barn", "horse")` with 0.03 influence value. Again, it confirmed that horse was the key factor that would affect the result the most. Both tuples coincided with human intuition while supporting the feasibility of our approach as well.

Query 1C: Show the most influential base tuple after making modifications to the input image. Our next query demonstrates the use of provenance queries for debugging. Intuitively, we expected that `ans("ID1","church")` would have the highest probability if we replaced the horses in the photo with a cross. We replaced `hasImg("ID1","horse","in","background")` with `hasImg("ID1","cross","in","background")` to mimic the modified photo aforementioned. However, the result was not as what we expected. We observed that `ans("ID1","barn")` still appeared in the evaluation result with the highest probability value. For debugging purposes, we ran the query on Figure 6 and captured the image information shown in Table 3.

We then updated the word similarity using Word2Vec. However, `ans("ID1","barn")` was still predicted as the most probable answer (i.e., the answer with the high probability). Demonstrating the generality of our approach, we use the previous queries to debug this issue. First, we ran the Derivation Query to show the most important derivations of `ans("ID1", "barn")` and `ans("ID1", "church")`. We found out that "barn" had

Table 3: Captured image information

Information	Prob.
horse color brown	1
horse in field	0.88
cloud in sky	0.85
building with roof	0.5
cross on building	1

very high similarities with other objects in the image ("cross": 0.30, "horse": 0.35, "cloud": 0.33), while "church" had much lower values ("cross": 0.09, "horse": 0.19, "cloud": 0.01). Surprisingly, `sim("church", "cross")` was much lower than `sim("barn", "cross")`. This explains exactly why the program did not predict the answer correctly.

Next, we would like to see how to increase the probability of `ans("ID1", "church")` with minimum cost. Our goal was to make "Church" the answer with the highest probability. So we ran the Influence Query, and selected the unique tuples that only appeared in the provenance of `ans("ID1", "church")`. The top 3 unique most influential tuples are shown in the following table:

Table 4: Top 3 unique influential tuple for `ans("ID1", "church")`

Tuple	Influence
`sim("church", "cross")`	0.04
`sim("church", "horse")`	0.02
`sim("church", "cloud")`	0.01

It is reasonable to have `sim("church", "cross")` as the most influential tuple. We set the probability of `ans("ID1", "barn")` as the target probability value for `sim("church", "cross")` and further computed the increment using the Modification Query, which returns a result value of 0.42. The value of `sim("church", "cross")` is updated to 0.51. Again, this met with our expectation that `sim("church", "cross")` should be greater than `sim("barn", "cross")`.

This use case demonstrates how multiple queries can be used in tandem to explain and debug unexpected answers caused by input data errors in probabilistic logic programs.

5.2 Mutual Trust in Social Network

For this graph network reachability use case, we use the Bitcoin OTC trust weighted signed network dataset[2]. This is a who-trusts-whom network of people who trade using Bitcoin on a platform called Bitcoin OTC [15, 16]. Each weighted edge in the network graph represents the trust between two users with the degree of trust as weight value. To fit the data in our probabilistic setting, we re-scale the weights of edges from [-10,10] to [0,1] to represent the probability score of trust.

In this use case, the input tuples are `trust` relations between people. For example, tuple `trust(1,2)` of score 0.7 means Person 1 trusts Person 2 with probability 0.7. We introduce the ProbLog program shown in Figure 7 to find `trustPath` and `mutualTrustPath` tuples that can be derived. Rule `r1` is a base case that each `trust` relation tuple is also a one-hop `trustPath` relation tuple. Rule `r2` is a recursive rule used to derive all the reachable `trustPath` tuples. Rule `r3` defines `mutualTrustPath` tuples that can be derived when the `trustPath` between any two nodes are bi-directional. The `Trust` ProbLog program can help finding any indirect trust relations between people, which is of significant interest during actual transactions.

```
r1 1.0: trustPath(P1,P2) :- trust(P1,P2).
r2 1.0: trustPath(P1,P3) :-
        trust(P1,P2), trustPath(P2,P3),
        P1!=P3.
r3 0.8: mutualTrustPath(P1,P2) :-
        trustPath(P1,P2),trustPath(P2,P1).
```

Figure 7: `Trust` ProbLog program

The original network graph is quite large, so for quality check purpose, we use a sample of 30 nodes to evaluate the provenance query results.

Query 2A: Show the derivations of `mutualTrustPath(1,6)`. The answer returned by P3 system is the provenance graph for queried tuple `mutualTrustPath(1,6)`, shown in Figure 8. The provenance graph shows Person 1 and Person 6 mutually trust each other because of the existence of trust paths from 6 to 1 (denoted by `trustPath(1,6)`) and from 1 to 6 (denoted by `trustPath(6,1)`). It further shows that there is one single derivation for `trustPath(6,1)` where Person 2 is the middle man that connects this trust path; on the other hand, `trustPath(1,6)` has two derivations, namely through path $1 \rightarrow 2 \rightarrow 6$ or path $1 \rightarrow 13 \rightarrow 2 \rightarrow 6$.

Query 2B: Show the most influential base tuple for `mutualTrustPath(1,6)`.
Next, we perform a query to understand which base tuples, i.e., the `trust` tuples, are most critical to the mutual trust between Person 1 and Perosn 6. We initialize the base tuples with probability values shown as follows (we omit the ones that are not involved in the derivation of `mutualTrustPath(1,6)`):

Table 5: Initial probability values of base tuples

Literal	Prob.	Literal	Prob.
`trust(1,2)`	0.9	`trust(1,13)`	0.65
`trust(2,1)`	0.9	`trust(2,6)`	0.75
`trust(6,2)`	0.7	`trust(13,2)`	0.6

Given this initialization, the P3 system returns that `trust(6,2)` is the most influential literal with an influence value of 0.51. The second most influential literal is `trust(2,6)` with an influence value of 0.48. Observing the provenance structure in Figure 8, we find that this result comply with human's intuition: `trust(6,2)` is influential[3] because its existence is the basis of the trust path from Person 6 to Person 1; `trust(2,6)` is also influential, more so than `trust(1,2)` and `trust(1,13)`, for the existence of the trust path from Person 1 to Person 6.

More intuitively, the result indicates that if Person 6 wants to increase the mutual trust to Person 1 without directly reaching him, strengthening the trust to Person 2 might be the most effective approach.

Query 2C: Show the optimal way to increase the probability of `mutualTrustPath(1,6)`. Here, we assume that each literal's probability can reach to 1.0 as maximum. The original **P**[mutualTrustPath(1, 6)] is 0.3524. Now we want to approximately double it to a target value 0.7. The strategy returned by P3 is as follows:

[2] https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html

[3] `trust(6,2)` is considered to have a higher influence value than `trust(2,1)` due to the initial probability assignment: whether `trust(6,2)` exist or not has a higher influence because its existence would *almost* imply the existence of the trust path from 6 to 1 as **P**[`trust(2,1)`] is very close to 1.

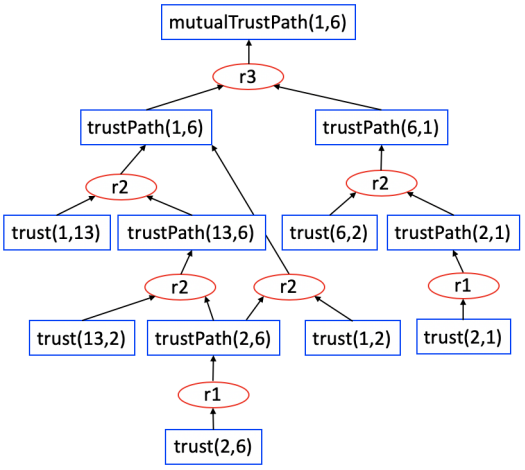

Figure 8: Provenance graph of queried tuple `mutualTrustPath(1,6)`

Table 6: Optimal strategy (total change = 0.58)

	Literal	Change	Overall prob.
Step 1	`trust(6,2)`	0.7 → 1.0	0.51
Step 2	`trust(2,6)`	0.75 → 1.0	0.68
Step 3	`trust(2,1)`	0.9 → 0.93	0.7

To show the effectiveness of our heuristic-based algorithm, we compare the cost, in terms of total change in base tuple's probability values, of its strategy with another randomly generated strategy where a random base tuple is chosen to update in each step. The result of a random strategy is shown as follows:

Table 7: Random strategy (total change = 1.36)

	Literal	Change	Overall prob.
Step 1	`trust(1,13)`	0.65 → 1.0	0.37
Step 2	`trust(13,2)`	0.6 → 1.0	0.38
Step 3	`trust(6,2)`	0.7 → 1.0	0.54
Step 4	`trust(1,2)`	0.9 → 1.0	0.55
Step 5	`trust(2,6)`	0.75 → 0.96	0.7

We find that the strategy provided by P3 significantly outperforms the random strategy (0.58 vs 1.36 in total change). We make similar observations for other Modification Queries.

6 EVALUATION

We developed a prototype of P3 based on enhancements to the ExSPAN provenance engine [31]. While ExSPAN was a distributed provenance engine designed for networks, we focused on using ExSPAN primarily in a single-node centralized setting. ExSPAN provides provenance for Datalog programs and provides basic provenance explanations. We enhanced the system to implement ProbLog, collect provenance, and support the three additional types of P3 queries (derivation, influence, and modification).

Experimental setup. Our experiments were performed on a Dell PowerEdge R730 server equipped with dual Intel Xeon E5-2640 CPUs and 32GB memory running Ubuntu 16.04 LTS 64-bit operating system.

As our workload, we evaluated the program shown in Figure 7. The output derivation of interest was the `mutualTrustPath` relation, over which we would run provenance queries. We selected this program as it had all the features of our use case. Meanwhile,

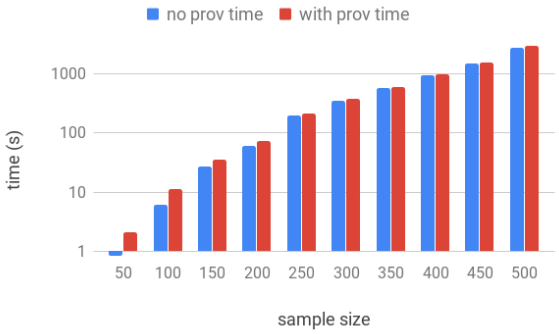

Figure 9: Running time with and without provenance

there existed real-world data that we could use for our experiments. Specifically, we used the Bitcoin OTC data set described in Section 5.2. The data set consisted of a directed graph network with 5,881 nodes (bitcoin users) and 35,592 edges (trust relations). We sampled the graph with different sizes of nodes, and evaluated the performance of provenance queries elaborated in Section 4 for each sample.

6.1 Provenance Maintenance and Querying

To evaluate the provenance maintenance and querying performance, we sampled subgraphs from the original trust network with 50, 100, 150, ..., 500 nodes. For each sample, we randomly chose a small set of seed nodes, and expanded the graph by performing a breadth-first search within the trust network from these seed nodes, until the graph reaches a given number of nodes. We then collected all traversed edges. For each size, we repeated our experiments 10 times with different samples and calculated the average running time.

Maintenance. Our first set of experiments aimed to measure the overhead of provenance maintenance. We compared the running time between running the `Trust` program with and without provenance maintenance. The overhead is shown in Figure 9. We observe that the evaluation time, both with and without provenance, increases exponentially as the sample size grows, which complies with our expectation. In addition, as provenance is maintained at the runtime as a side-computation along each rule evaluation, the provenance maintenance incurs a small overhead. We observed that, in average, the maintenance time is less than 10% of the total running time, and will not impact the asymptotic scalability of PLP programs. Thus, the provenance maintenance is efficient and its overhead in P3 for ProbLog-like PLP programs is low enough to be accepted.

Query. We next measured the overhead to obtain the provenance of a queried tuple, as a generic **explanation query**. Figure 10 summarizes our evaluation results. We fixed the hop limit to 4, which means we only retrieved the mutual trust paths whose length are no greater than 4 hops. Limiting the path length can remove overly long derivation paths from the provenance and expedite the querying process: The number of derivation paths grows exponentially with the path length, however, the long paths contribute little to the derivability of the queried tuple (intuitively, people are less likely to trust a relationship that has many hops). As shown on the figure, the query time is roughly on the same order of magnitude compared to the maintenance time, but grows slower for larger-sized graphs owing to the use of hop limits. Overall, the provenance querying is reasonably efficient

Figure 10: Provenance query time compared with maintenance time. Hop limit is set to 4

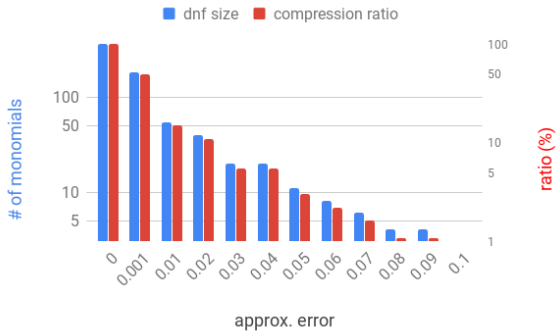

Figure 11: Compression ratio of sufficient provenance

to be practical, and can be further optimized by parallelizing the provenance graph traversal.

6.2 Performance for Different Queries

We have evaluated the overhead of provenance maintenance for a PLP program, and the execution time of generic **explanation query** (i.e., to obtain the provenance for a queried tuple). In this section, we perform a set of experiments to measure the performance of answering other types of provenance queries elaborated in Section 4.

Derivation Query. To evaluate this type of query, we sampled graphs consisting of 150 nodes and 150 edges from the trust network (we repeated the experiment 10 times with different sampled graphs). We queried all possible mutual paths between two specific users and set the hop limit to 6. We ran the query on each sampled graph and calculated the average. For each returned provenance polynomial λ, varying the approximation error ϵ leads to sufficient provenance of different sizes – a more forgiving approximation error (i.e., larger ϵ) leads to a smaller-sized sufficient provenance. We evaluated the compression ratio of sufficient provenance with approximation error from 0.1% to 10% (X% means X percent of $\mathbf{P}[\lambda]$). Compression ratio is defined as the number of monomials in the sufficient provenance divided by the number of monomials in the original provenance polynomials. Our evaluation result is shown in Figure 11. We observed that, as expected, sufficient provenance leads to significant compression: it warrants a 50% deduction in provenance size even with merely 0.1% approximation error; 10% approximation error allows approximately 99.8% deduction.

We further evaluated the computation time of our naïve approach; we observe that the computation time was consistently under 1 second. In fact, it decreased tremendously as we increased the approximation error. This is because the computation of Derivation Queries heavily relies on Monte-Carlo simulation (for evaluating whether more monomials should be removed), and larger approximation error shifts the search for optimal sufficient provenance towards shorter polynomials and therefore a shorter overall query time.

Influence Query. Our next set of experiments evaluates the performance of Influence Queries. We used the same set of sampled graphs as the Derivation Query, that is, each sampled graph consists of 150 nodes and 150 edges. We observe that the average time for computing the influence of all literals (and identifying the most influential literal) is 9.6 seconds; according to the definition of influence presented in Section 4, the computation time of influence queries highly depends on the size of the provenance, more specifically, the number of monomials in the provenance polynomial and the number of distinct literals. We consider two optimizations that significantly reduce the computation time of influence queries.

Parallelize Monte-Carlo simulation. Monte-Carlo simulation repeatedly evaluates the truth value of a Boolean polynomial given a random value assignment of the variables (which can be either true or false). This process is embarrassingly parallel and can greatly benefit from using hardwares such as GPUs. We therefore evaluated the computation time of the Influence Query using a parallel implementation of Monte-Carlo simulation. The experiment was run on a workstation equipped with an Intel i7 9800X CPU, 64GB memory and four Nvidia GTX 1080 Ti GPUs. Table 8 summarizes the time required to compute the influence value sequentially or in parallel.

Table 8: Comparison of influence query time

Seq total	Seq per-literal	Para total	Para per-literal
9.60	0.14	0.85	0.01

We observe that the parallel implementation provides a 10x speed improvement owing to the high degree of parallelization, and reduces the total computation time to under 1 second.

Preprocess using sufficient provenance. We noticed that most literals have a negligibly small influence value. If our goal is to identify the most influential literal, we might be able to avoid computing the influences of most literals. Our approach is to use sufficient provenance as a preprocessing step, such that the influence query runs on a much small provenance polynomial. To evaluate the effectiveness of this method, we first examined whether the returned sufficient provenance still retains the most influential literals. We computed the sufficient provenance when allowing different approximation errors, and compared the top-5 most influential literals to the ones computed from the original provenance. As shown in Figure 12, the rank of the top-5 most influential literals remained the same when the error limit was less than 2%, and then started to fluctuate as the error limit increased. However, the most influential tuple remained the same even the error limit was as high as 10%.

Second, we evaluated the computation time of Influence Query after the preprocessing step retrieves the sufficient provenance. Figure 13 shows the evaluation results with varying approximation error limits. We observe that, since the number of monomials decreased exponentially as we increased the error limit, the computation time of Influence Query also decreased exponentially.

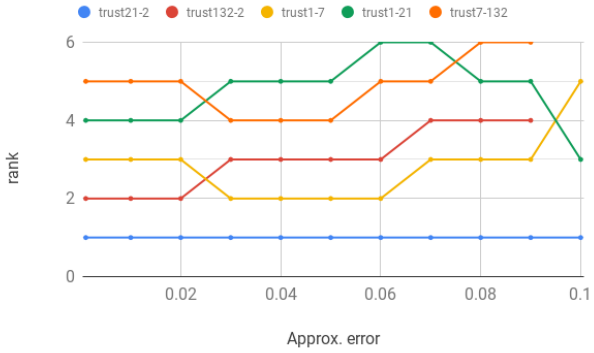

Figure 12: Rank change of top-5 most influential literals

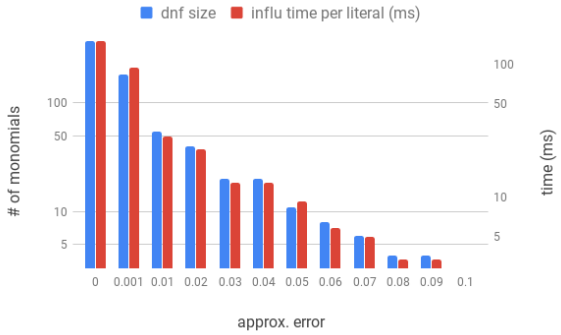

Figure 13: Influence query time for a single literal

We further measured the total execution time of computing influence queries with sufficient provenance. The result is shown in Figure 14. We observed that, for large provenance polynomial, allowing even a small approximation error would reduce the query time tremendously. We compared it with Figure 12 and observed that, when the approximation error limit was set appropriately (around 2% in this case), the returned top influential literals remained unchanged while the computation time could be reduced substantially. For example, we observed an order of magnitude reduction in computation time when setting the error limit to 2%.

Modification Query In our last set of experiments, we evaluated the performance of modification query where we change the provenance probability to a target value with minimum cost. Recall that the changed tuple in each step is the most influential one to the provenance. We tested the Modification query on a given provenance polynomial that consists of 366 monomials and 65 distinct literals. The probability of the provenance was 0.873, and we wanted to reduce it to 0.373. We conducted the experiment using three methods: sequential without sufficient provenance, parallel without sufficient provenance, and sequential with sufficient provenance (with an error limit of 0.01). All the three method returned the same change sequence: a) modify the probability of `trust21-2` by -0.75 (i.e., set its probability to 0), and b) modify the probability of `trust132-2` by -0.196. The computation time are shown in Table 9.

Table 9: Compare running times of modification query

Sequential	Parallel	Seq. with suff. prov.
20.66	1.55	2.44

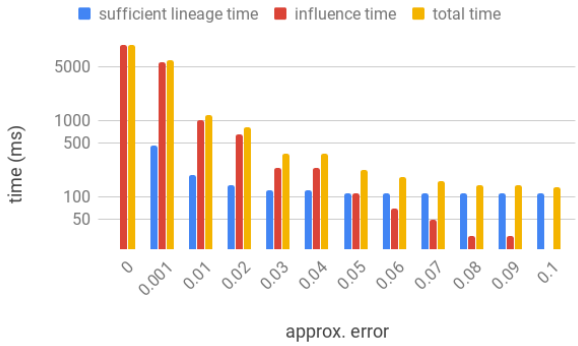

Figure 14: Influence query on sufficient provenance

We observed that sufficient provenance can give the same result while its computation time is in the same order of magnitude compared to the hardware-aided parallel processing.

7 RELATED WORK

We briefly summarize related works in provenance for relational databases (regular and probabilistic), as well as a reference to other probabilistic logic programs.

7.1 Data Provenance

Data provenance is widely used in declarative logic programs like Datalog and NDlog (Network Datalog) [17–19]. Provenance information is stored using *graph representation* and *algebraic representation* [11, 31]. The graph representation is also called provenance graph, and the algebraic representation is commonly encoded as provenance polynomials.

Tuple-level provenance graph is a fine-grained derivation graph. In ExSPAN [31], the provenance graph is acyclic. There are two types of vertices in the graph. One is the tuple vertex, and the other is the rule execution vertex. Each tuple vertex is either a base tuple or a derived tuple; each rule execution vertex denotes an instance of a rule execution. The edges in the provenance graph are unidirectional, and represent data flows between tuple vertices and rule execution vertices.

Provenance can also be represented as *provenance polynomials* [11] which is encoded as an algebraic expression with two binary operations: addition "+" and multiplication "·". Each base tuple is encoded as one literal in the polynomials. Specifically, "+" indicates the combination of tuples with union and projection operations, and "·" denotes a natural join over tuples.

As we will describe later in our paper, P3 also uses ExSPAN style execution to maintain and process provenance, with several extensions in order to support a new provenance model.

7.2 Provenance in Probabilistic Databases

In probabilistic databases community, lineage is the synonym of provenance. Each tuple has an associated probability score in probabilistic databases. For SQL-like queries in probabilistic databases, provenance can support explanations for the queried tuple probabilities. For instance, Trio [30] is an innovative database management system (DBMS) based on an extended relational model called *Uncertainty-Lineage Databases* (ULDBs) to handle the uncertainty of data and data lineage. It extends the traditional model by adding a confidence value (probability of being

true) for each tuple. Trio introduces a SQL-based language called *TriQL* for querying confidences and lineage in ULDBs. In our paper, for PLP programs, we do not consider SQL-like queries.

In addition, for probabilistic databases, approximation of lineage such as *sufficient lineage* [25] is used for only keeping track of the most important derivations for the derived tuple given some approximation error ϵ. Kanagal et al. presented a further discussion and provided an efficient approach for sensitivity and explanation analysis [13]. This approach works on read-once lineages from conjunctive queries without self-joins. However, read-once is not a universal property of the provenance polynomials extracted from PLP programs.

7.3 Probabilistic Logic Programs

Milch et al. introduced the BLOG language [20] to define probabilistic models with unknown objects and identity uncertainty. Unlike BLOG based on first-order logic, Goodman et al. developed Church[10], a LISP-like language based on lambda calculus to describe stochastic generative processes. Furthermore, Pfeffer introduced an object-oriented language called Figaro for probabilistic programming. Beside these languages, for probabilistic inference, Alchemy[26] is a system based on Markov logic representation to construct knowledge bases. Niu et al. and Gribkoff et al. introduced Tuffy[22] and SlimShot[12] respectively such that MLN inference can perform on large scale data sets. Compared to the aforementioned studies, our contribution is on providing a feasible solution to perform quantitative query evaluations for debugging purposes.

8 CONCLUSIONS AND FUTURE WORK

This paper proposes P3, a platform and system for capturing provenance in probabilistic logic programs. P3 maintains the provenance information using both graph representation (directed acyclic provenance graph) and algebraic representation (provenance polynomials as Boolean DNF formulas). P3 enables a wide range of novel query types, including explanation query, derivation query, influence query, and modification query. We conduct the theoretical analysis of P3's provenance model and queries. Our evaluation on a P3 prototype demonstrate the feasibility of P3 across multiple use cases with low overhead.

Moving forward, we are working on expanding the scope of PLP programs supported by P3. We plan to extend provenance model and system to support first-order PLP programs with negation, and machine-learning style inference [3, 6, 26]. As we support more language features, we would also like to broaden our use cases to include learning-based applications, such as explainable recommendation through relational learning. Finally, we are exploring ways to improve our performance further, leveraging parallel query execution across multiple machines in a cluster.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. This paper is partially funded from NSF grants CNS-1453392, CNS-1513734, CNS-1513679, NSF CNS-1703936, and CNS-1704189. This research was additionally supported by ONR under grant N00014-18-1-2618 and by DARPA under grants HR0011-17-C-0047 and HR0011-16-C-0056. The findings and opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Somak Aditya, Yezhou Yang, and Chitta Baral. 2018. Explicit Reasoning over End-to-End Neural Architectures for Visual Question Answering. In *AAAI*.
[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. 2015. VQA: Visual Question Answering. In *ICCV*. 2425–2433.
[3] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. 2015. Hinge-Loss Markov Random Fields and Probabilistic Soft Logic. *Journal of Machine Learning Research* 18 (2015), 109:1–109:67.
[4] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (Aug. 1986), 677–691.
[5] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient query evaluation on probabilistic databases. *PVLDB* (2004), 523–544.
[6] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP* 15 (2015), 358–401.
[7] Norbert Fuhr. 1995. Probabilistic Datalog&Mdash;a Logic for Powerful Retrieval Methods. In *SIGIR*. ACM, New York, NY, USA, 282–290.
[8] Jean H Gallier. 1988. Logic for Computer Science Foundations of Automatic Theorem Proving. (1988).
[9] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic Inference for Networks. In *PLDI*. ACM, New York, NY, USA, 586–602. https://doi.org/10.1145/3192366.3192400
[10] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. 2008. Church: A language for generative models. *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008* 2008, 220–229.
[11] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. 31–40.
[12] Eric Gribkoff and Dan Suciu. 2016. SlimShot: in-database probabilistic inference for knowledge bases. *Proceedings of the VLDB Endowment* 9 (03 2016), 552–563.
[13] Bhargav Kanagal, Jian Li, and Amol Deshpande. 2011. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*. 841–852.
[14] Richard M. Karp and Michael Luby. 1983. Monte-Carlo Algorithms for Enumeration and Reliability Problems. In *SFCS*. IEEE Computer Society, 56–64.
[15] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and VS Subrahmanian. 2018. Rev2: Fraudulent user prediction in rating platforms. In *WSDM*. ACM, 333–341.
[16] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. 2016. Edge weight prediction in weighted signed networks. In *ICDM*. IEEE, 221–230.
[17] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*. 97–108.
[18] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing Declarative Overlays. In *SOSP*. ACM, 75–90.
[19] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*.
[20] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel Ong, and Andrey Kolobov. 2005. BLOG: Probabilistic Models with Unknown Objects. *IJCAI* 2005 (01 2005).
[21] Stephen Muggleton et al. 1996. Stochastic logic programs. *ILP* 32 (1996), 254–264.
[22] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. 2011. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *VLDB Endowment* 4, 6 (2011), 373–384.
[23] David Poole. 2008. The independent choice logic and beyond. In *Probabilistic Inductive Logic Programming*. Springer, 222–243.
[24] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *IJCAI*.
[25] Christopher Ré and Dan Suciu. 2008. Approximate lineage for probabilistic databases. In *PVLDB*. 797–808.
[26] Matthew Richardson and Pedro Domingos. 2006. Markov Logic Networks. *Mach. Learn.* 62, 1-2 (Feb. 2006), 107–136.
[27] Taisuke Sato. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *ICLP*.
[28] Taisuke Sato and Yoshitaka Kameya. 2008. New Advances in Logic-based Probabilistic Modeling by PRISM. In *Probabilistic Inductive Logic Programming*, Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton (Eds.). Springer-Verlag, Berlin, Heidelberg, 118–155.
[29] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.
[30] Jennifer Widom. 2008. Trio: A System for Data, Uncertainty, and Lineage. In *Managing and Mining Uncertain Data*. Springer.
[31] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. 2010. Efficient Querying and Maintenance of Network Provenance at Internet-scale. In *SIGMOD*. ACM, 615–626.