Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy

DAVID ZHAO, University of Sydney
PAVLE SUBOTIĆ, Amazon
BERNHARD SCHOLZ, University of Sydney

Logic programming languages such as Datalog have become popular as Domain Specific Languages (DSLs) for solving large-scale, real-world problems, in particular, static program analysis and network analysis. The logic specifications that model analysis problems process millions of tuples of data and contain hundreds of highly recursive rules. As a result, they are notoriously difficult to debug. While the database community has proposed several data provenance techniques that address the *Declarative Debugging Challenge* for Databases, in the cases of analysis problems, these state-of-the-art techniques do not scale.

In this article, we introduce a novel bottom-up Datalog evaluation strategy for debugging: Our provenance evaluation strategy relies on a new provenance lattice that includes proof annotations and a new fixed-point semantics for semi-naïve evaluation. A debugging query mechanism allows arbitrary provenance queries, constructing partial proof trees of tuples with minimal height. We integrate our technique into Soufflé, a Datalog engine that synthesizes C++ code, and achieve high performance by using specialized parallel data structures. Experiments are conducted with Doop/DaCapo, producing proof annotations for tens of millions of output tuples. We show that our method has a runtime overhead of 1.31× on average while being more flexible than existing state-of-the-art techniques.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; Software testing and debugging;

Additional Key Words and Phrases: Static analysis, datalog, provenance

ACM Reference format:
David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 7 (April 2020), 35 pages.
https://doi.org/10.1145/3379446

1 INTRODUCTION

Datalog and other logic specification languages [Aref et al. 2015; Hoder et al. 2011; Jordan et al. 2016; Madsen et al. 2016] have seen a rise in popularity in recent years, being widely used to solve real-world problems including program analysis [Allen et al. 2015; Jordan et al. 2016], declarative networking [Huang et al. 2011; Zhou et al. 2010], security analysis [Ou et al. 2005], and business

This research was supported partially by the Australian Government through the ARC Discovery Project funding scheme (DP180104030) and a research donation from AWS.
Authors' addresses: D. Zhao and B. Scholz, School of Computer Science, University of Sydney; emails: dzha3983@uni.sydney.edu.au, bernhard.scholz@sydney.edu.au; P. Subotić, University College London; email: Pavle.Subotic.15@ucl.ac.uk.

0164-0925/2020/04-ART7 $15.00
https://doi.org/10.1145/3379446

applications [Aref et al. 2015]. Logic programming provides declarative semantics for computation, resulting in succinct program representations and rapid-prototyping capabilities for scientific and industrial applications. Rather than prescribing the computational steps imperatively, logic specifications define the intended result declaratively and thus can express computations concisely. For example, logic programming has gained traction in the area of program analysis due to its flexibility in building custom program analyzers [Jordan et al. 2016], points-to analyses for Java programs [Bravenboer and Smaragdakis 2009], and security analysis for smart contracts [Grech et al. 2018, 2019].

Despite the numerous advantages, the declarative semantics of Datalog poses a debugging challenge. Strategies employed in debugging imperative programs such as inspecting variables at given points in the program execution do not translate to declarative programming. Logic specifications lack the notions of state and state transitions. Instead, they have relations that contain tuples. These relations may be input relations, such as those describing the instance of an analysis, intermediate relations, or output relations, such as those containing the results of an analysis. Relations can only be viewed in full, without any explanation of the origin or derivation of data, after the completion of a complicated evaluation strategy. Thus, the Datalog user will find the results alone of logic evaluation inconclusive for debugging purposes.

When debugging Datalog specifications, there are two main scenarios: (1) an unexpected output tuple appears or (2) an expected output tuple does not appear. These may occur as a result of a fault in the input data, and/or a fault in the logic rules. Both scenarios call for mechanisms to explain how an output tuple is derived, or why the tuple cannot be derived from the input tuples. The standard mechanism for these explanations is a *proof tree*. In the case of explaining the existence of an unexpected tuple, a proof tree describes formally the sequence of rule applications involved in generating the tuple. However, a failed proof tree, where at least one part of the proof tree does not hold, may explain why an expected tuple cannot be derived in the logic specification. These proof trees can be seen as a form of *data provenance* witness, that is, an explanation vehicle for the origins of data [Buneman et al. 2001; Cheney et al. 2009].

In the presence of complex Datalog specifications and large datasets, Datalog debugging becomes an even bigger challenge. While recent developments in Datalog evaluation engines, such as Soufflé [Jordan et al. 2016], have enabled the effective evaluation of complex Datalog specifications with large data using scalable bottom-up evaluation strategies [Ramakrishnan and Sudarshan 1991; Ullman 1989], unlike top-down evaluation, bottom-up evaluation does not have an explicit notion of a proof tree in its evaluation. Therefore, to facilitate debugging in bottom-up evaluation, state-of-the-art [Deutch et al. 2015; Köhler et al. 2012; Lee et al. 2017] techniques have been developed that rewrite the Datalog specification with provenance information. Using these techniques, Datalog users follow a *debugging cycle* that allows them to find anomalies in the input relations and/or the logic rules. In such setups, the typical debugging cycle comprises the phases of (1) defining an investigation query, (2) evaluating the logic specification to produce provenance witness, (3) investigating the faults based on the provenance information, and (4) fixing the faults. For complex Datalog specifications, the need for re-evaluation for each investigation is impractical. For example, Doop [Bravenboer and Smaragdakis 2009] with a highly precise analysis setting may take multiple days to evaluate for medium to large-sized Java programs. Although state-of-the-art approaches scale for the database querying use cases, such approaches are not practical for industrial scale static analysis problems.

A further difficulty in developing debugging support for Datalog is providing understandable provenance witnesses. Use cases such as program analysis tend to produce proof trees of very large height. For example, investigations on medium sized program analyses in Doop have minimal height proof trees of over 200 nodes. Therefore, a careful balance must be struck between enough

information and readability in the debugging witnesses. Our approach limits information overload and handles large proof trees by allowing the user to interactively explore relevant *fragments* of the proof trees.

In this article, we present a novel debugging approach that targets Datalog programs with characteristics of those found in static program analysis. Our approach scales to large dataset and ruleset sizes and provides succinct and interactively navigable provenance information.

The first aspect of our technique is a novel Datalog provenance evaluation strategy that augments the intensional database (IDB) with *Proof Annotations* and hence allows fast proof tree exploration for *all* debugging queries, *without the need for re-evaluation*. The exploration uses the proof annotations to construct proof trees for tuples *lazily*, i.e., a debugging query for a tuple produces the rule and the subproofs of the rule. The subproofs when expanded in consecutive debugging queries, will produce a *minimal* height proof tree for the given tuple. Our system also supports *non-existence* explanations of a tuple. In this case, proof annotations are not helpful, since they cannot describe non-existent tuples. Thus, we adapt an approach from [Lee et al. 2018] to provide a *user-guided* procedure for explaining the non-existence of tuples.

We implement the provenance evaluation strategy in the synthesis framework of Soufflé [Jordan et al. 2016] to produce *specialized* data structures and an interactive debugging query system for each logic specification. Our approach is tightly integrated into the Soufflé engine and achieves higher performance than existing provenance approaches when more than one provenance query is run. We demonstrate the feasibility of our technique through the complex Java points-to framework, Doop, running the Java DaCapo benchmark suite, which produces tens of millions of output tuples. We demonstrate that the initial implementation of our novel provenance method incurs a runtime overhead of 1.31×, and memory consumption overhead of 1.76× on average.

Our contributions in this work are as follows:

- a provenance evaluation strategy for Datalog specifications: defining a new evaluation domain based on a provenance lattice that extends the standard Datalog subset lattice with proof annotations, and leveraging parallel bottom-up evaluation to give minimal height proof trees,
- a provenance query system for constructing minimal height proof trees utilizing proof annotations, allowing effective bug investigation with a minimum number of user interactions,
- an efficient and scalable integration of the proof tree generator system into Soufflé, using specialized data structures for storing proof annotations, and
- large-scale experiments using the Doop program analysis framework with DaCapo benchmarks with tens of millions of tuples, measuring on average 1.31× overheads for runtime and 1.76× overheads for memory.

The article is organized as follows: In Section 2, we motivate our provenance method and describe its use in a real-world program analysis use case. In Section 3, we detail the theoretical basis of our method with regards to the provenance evaluation strategy along with the provenance queries to construct proof trees for tuples. We also demonstrate the minimality properties and present the practical solution that results from this theory. In Section 4, we detail the implementation of our system in Soufflé. In Section 5, we present experiments that show the feasibility of our provenance system. In Section 6, we outline related work, and we conclude in Section 7.

2 MOTIVATION AND PROBLEM STATEMENT

Real-world Datalog specifications for applications such as program analysis can often contain up to hundreds of mutually recursive rules. With such complex applications, bugs are a common occurrence during the development cycle of a Datalog specification. Buggy Datalog code may

```
l1: a = new O();
l2: b = a;

l3: c = new P();
l4: d = new P();

l5: c.f = a;
l6: e = d.f;
l7: b = c.f;
l8: a = b;
```

(a) Input Program

```
new(a, l1).
assign(b, a).

new(c, l3).
new(d, l4).

store(c, f, a).
load(e, d, f).
load(b, c, f).
assign(a, b).
```

(b) EDB Tuples

```
r1: vpt(Var, Obj) :- new(Var, Obj).
r2: vpt(Var, Obj) :- assign(Var, Var2),
                     vpt(Var2, Obj).
r3: vpt(Var, Obj) :- load(Var, Y, F),
                     store(P, F, Q),
                     vpt(Q, Obj),
                     vpt(P, Obj2),
                     vpt(Y, Obj2).
r4: alias(Var1, Var2) :- vpt(Var1, Obj),
                         vpt(Var2, Obj),
                         Var1 != Var2.
```

(c) Datalog Points-to Analysis

Fig. 1. Program analysis datalog setup.

manifest itself in two main ways: (1) It produces an unexpected output tuple or (2) it fails to produce an expected output tuple.

A common approach to characterize the evaluation of a Datalog specification is through *proof trees*. A proof tree for a tuple describes the derivation of that tuple from input tuples and rules. During the debugging cycle, the presence of any unexpected tuples can be explained by producing a valid proof tree, where all nodes of the proof tree hold. However, a failed proof tree, where at least one part of the proof tree fails to hold, provides valuable insight into why a tuple is not produced. Therefore, both valid and failed proof trees are critical for investigation into anomalies.

Note that potentially there could be an infinite number of valid proof trees for the explanation of any given tuple. However, Datalog developers desire concise proof trees such that that the faulty behavior of the logic specification is revealed quickly. In this section, we describe how proof trees can be used to debug a Datalog specification and an overview of our method for generating minimal proof trees for output tuples.

2.1 Use Case: Program Analysis

2.1.1 Points-To Analysis. We illustrate the utility of debugging the presence of unexpected tuples via proof trees through a program analysis use case. Figure 1 illustrates a points-to analysis implemented in Datalog. The points-to analysis resembles a field-sensitive but flow-insensitive analysis [Sridharan et al. 2005]. The input relations (also known as EDB) of the logic specification are the relations *new*, *assign*, *load*, and *store* express the input program in relational form. The relation *new* represent the object-creation sites of the input program, the relation *assign* the assignments, and relations *load*/*store* the read and write accesses of objects via a field. Figure 1(a) shows an input program encoded in the form of input relations in Figure 1(b). The graph in Figure 2 represents the input relations. The nodes represent either object-creation sites or variables. The edges are object-creation sites, assignments, and load/store instructions. The graph shows a clear separation between objects of l_4 with objects l_1 and l_3. The goal is to compute the var-points-to set in the form of the output relation *vpt*. The Datalog rules computing the var-points-to set are given in Figure 1(c). The first rule makes the variable `Var` point to object `Obj`, where `Obj` is the line number of the object-creation site as an abstraction for all possible objects that could be created by this object-creation site. The second rule shows the transfer of the var-points-to set from source `Var2` of the assignment to its destination `Var`. The third rule transfers the var-points-to set from the source of a store instruction `Q` to the destination of a load instruction `Var`. The transfer is conditional depending on whether field `F` of the load and store instructions match and whether the instance variable `Y`

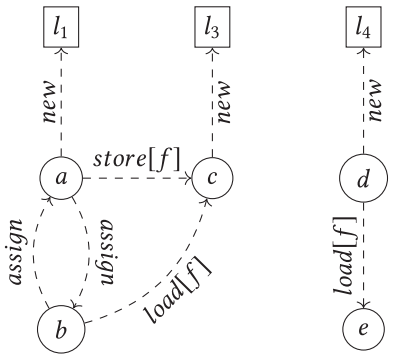


Fig. 2. Points-to input diagram.

$$\dfrac{\dfrac{new(a,l_1)}{vpt(a,l_1)}\,r_1 \qquad \dfrac{assign(b,a) \qquad \dfrac{new(a,l_1)}{vpt(a,l_1)}\,r_1}{vpt(b,l_1)}\,r_2 \qquad a \neq b}{alias(a,b)}\,r_4$$

Fig. 3. Full proof tree for $alias(a, b)$.

$$\dfrac{assign(a,b) \qquad \dfrac{assign(b,a) \qquad \dfrac{\cdots}{vpt(b,l_1)}}{vpt(a,l_1)}\,r_2}{vpt(b,l_1)}\,r_2$$

Fig. 4. Infinitely many derivations for $vpt(b, l_1)$, resulting from the circular assignment in lines l_2 and l_8 in the input program.

of the load and the instance variable P of the store instruction may point to the same object (Var2). The last rule expresses the alias relation between variables Var1 and Var2, i.e., two variables alias if they share at least one object-creation site Obj in their var-points-to sets. The relations vpt and alias are the output of the analysis and are called the IDB of the Datalog specification.

2.1.2 Minimal Height Proof Trees. The analysis example in Figure 1 computes the output relation *alias* that captures the alias information of two variables. A user may investigate *why* a tuple (a, b) exists in the output relation *alias*, i.e., how the analysis derives $alias(a, b)$ from the input data via the rules. Intuitively, this information is contained in the points-to input diagram (cf. Figure 2) showing that variables a and b may reach the same object. However, it is not an explanation, as a proof tree would be, for the tuple $alias(a, b)$ as shown in Figure 3. The proof tree shows that $alias(a, b)$ is derived by rule r_4 using the facts $vpt(a, l_1)$ and $vpt(b, l_1)$. This outcome is expected, since it tells us that a and b point to the same object (l_1 in this case), and thus they may alias.

The importance of minimality of proof tree height is shown in Figure 4, which depicts the proof tree resulting from the assignment in line l_2 in the input program. In the input program, there is a circular assignment in lines l_2 and l_8 caused by the flow-insensitivity of the input program, and thus the tuple $vpt(b, l_1)$ could be derived in an arbitrary number of rule applications, as shown in Figure 4.

Thus, even for this small example, there are infinitely many valid proof trees for the tuple $alias(a, b)$. A provenance system ought to produce the most concise proof tree so that an end user can understand the derivation of a tuple with the least effort.

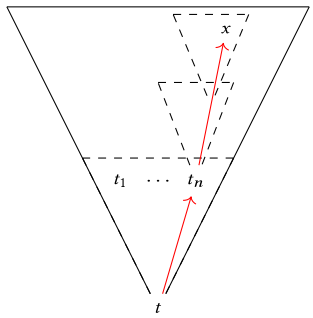


Fig. 5. Interactive exploration of fragments of a proof tree for t.

2.1.3 Proof Tree Fragments for Debugging. Suppose a Datalog user discovers an unexpected tuple in the output, which indicates that a fault exists somewhere in the logic specification. The aim is to investigate the root cause of this fault. Since proof trees provide explanations for the existence of a tuple, the proof tree of an unexpected tuple will help identify the fault in the logic specification.

An example fault could be if rule r_3 was altered as follows:

```
r3: vpt(Var, Obj) :-  load(Var, Y, F),
                      store(P, F, Q),
                      vpt(Q, Obj),
                      vpt(P, Obj1),
                      vpt(Y, Obj2).
```

Note the condition that objects `P` and `Y` must alias now no longer holds. A minor typo may have introduced this fault, and as a consequence of this typo, the analysis produces the extra tuple (a, e) in relation *alias*. This additional tuple becomes a *symptom* of the fault. To diagnose this fault, the proof tree of tuple $alias(a, e)$ highlights the root cause of the fault.

However, in practice, a full proof tree may be too large to provide a meaningful explanation even if it is of minimal height, and as experiments in Section 5 show, proof trees for real-world program analyses (e.g., Doop) can exceed heights of 200. Thus a Datalog user may want to explore only relevant *fragments* of it interactively. A fragment of a proof tree is a partial subtree, which consists of some number of levels. For instance, we may construct fragments comprising of 2 levels to explore only parts of the proof tree that are relevant.

We illustrate the exploration of fragments of the proof tree in Figure 5. In the figure, tuple t denotes the symptom of the fault, i.e., t is an unexpected tuple in the output. The aim is to explore the proof tree for t to find the root cause for this fault. In our example, the user follows the scent of the fault by expanding proof tree fragments that show anomalies. This process produces a path of exploration in the proof tree. The path of exploration discovers the root cause of the fault efficiently, without constructing and displaying the full proof tree of an output tuple.

Concretely, we may wish to explain the tuple $alias(a, e)$. Figure 6 illustrates the exploration of an explanation for $alias(a, e)$ by generating proof tree fragments of 2 levels at a time. The user generates the first fragment and decides that $vpt(e, l_1)$ is the most relevant explanation for the fault, and continues down this path. As a result, the root cause (for example, the erroneous rule r_3) is discovered after two fragments. This interaction mechanism also justifies the choice to minimize the height of proof trees. By doing this, we minimize the number of user interactions (i.e., proof tree fragments) required to discover the root cause for an anomaly.

$$\frac{load(e,d,f) \qquad store(c,f,a) \qquad vpt(a,l_1) \qquad vpt(c,l_3) \qquad vpt(d,l_4)}{vpt(e,l_1)}\, r_3$$

$$\frac{vpt(a,l_1) \qquad vpt(e,l_1) \qquad a \neq e}{\boldsymbol{alias(a,e)}}\, r_4$$

Fig. 6. Exploring the proof of $alias(a, e)$ to find the erroneous rule r_3.

2.1.4 Debugging Non-Existent Tuples. However, the *non-existence* of a tuple may also indicate a fault in the Datalog specification. Moreover, when the Datalog specification includes negations, the non-existence of an expected tuple may lead to the existence of an unexpected tuple and vice versa. In the running example, the input data may be missing the tuple $assign(b, a)$. In this case, the analysis would not consider all cases of assignment in the source program. In our example, the tuple $vpt(b, l_1)$ would not be produced with the altered input data. However, a developer would expect that the assignment in l_2 would cause the tuple to exist, and thus may wish to examine the reason for the tuple's non-existence.

Logically, the non-existence of a tuple results from there being *no valid proof tree* for that tuple. Therefore, to "explain" the non-existence of a tuple, we must show that every attempt to construct a proof tree eventually fails. However, since this is an infinite search space, automated techniques are not tractable. Therefore, we adapt a semi-automated approach from [Lee et al. 2018], using algorithmic debugging ideas [Caballero et al. 2017] to ask queries of the user to aid the construction of a single failed proof tree. Such a failed proof tree may provide valuable insight into the non-existence of the tuple. Further details for debugging non-existent tuples are presented in Section 3.4.

2.2 Proof Trees and Problem Statement

We use standard terminology for Datalog, taken from Abiteboul et al. [1995]. A Datalog specification P consists of a set of *rules*, of the form $R_0(X_0)$:- $R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$. Each $R_i(X_i)$ is a *predicate*, consisting of a *relation name* R_i and an argument X_i consisting of the correct number of variables and constants. The term $\psi(X_1, \ldots, X_n)$ denotes a conjunction of constraints on the variables in the rule. These constraints may include, for example, arithmetic constraints (such as less than), or negation of a predicate. A predicate can be *instantiated* to form a *tuple* where each variable is mapped to a constant. An instantiated rule is a rule with each predicate replaced by its instantiation such that the variable mappings are consistent between predicates and the constraints are satisfied. An *instance* I is a set of tuples, and we denote the input instance to be EDB.

Given a Datalog specification P, an input instance EDB of P, and a tuple t produced by P, we want to find a *proof tree* of minimal height for t. We define a proof tree as follows:

Definition 2.1 (Proof Tree). Let P be a Datalog specification, and let EDB be an input instance. A *proof tree* τ_t for a tuple t computed by P is a labeled tree where (1) each vertex is labeled with a tuple, (2) each leaf is labeled with an input tuple in EDB, (3) the root is labeled with t, and (4) for a vertex labeled with t_0, there is a valid instantiation t_0 :- $t_1, \ldots, t_n$ of a rule ρ in P such that the direct children of t_0 are labeled with $t_1, \ldots, t_n$. Moreover, the vertex is associated with ρ.

A proof tree for t can be viewed as an *explanation* for the existence of t, by showing how it is derived from other tuples using the rules in the Datalog specification.

To formalize the problem statement, we need to characterize proof trees of minimal height. Note that the set of proof trees for a Datalog specification could be constructed inductively by the height of the trees. We denote τ_t to be a proof tree for tuple t, and T^k to be the set of proof trees of height

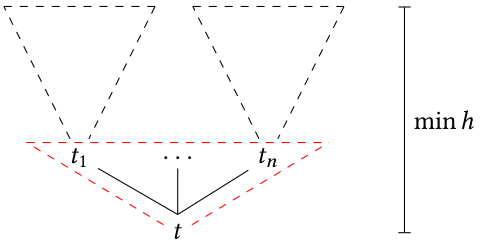


Fig. 7. One level of a proof tree of minimal height for t.

at most k. This construction leads to a convenient description of what it means for a proof tree to be of minimal height.

Definition 2.2. We define the set of all proof trees inductively. Let $T^0 = \{\tau_t \mid t \in EDB\}$ be the set of proof trees for tuples in the input instance. Then, define T^k in terms of T^{k-1}: $T^k = \{\tau_t \mid t \coloneq t_1, \ldots, t_n$ is a valid instantiation and $\forall t_i : \exists \tau_{t_i} \in T^{k-1}\}$ Then, $T = \bigcup_{i \geq 0} T^i$ is the set of all proof trees produced by the specification P.

Note that each T^k consists of proof trees of height at most k, since if $t \coloneq t_1, \ldots, t_n$ is an instantiation of a rule, then the height of the proof tree for t is equal to the maximum height of the proof trees for $t_1, \ldots, t_n$ plus 1. By defining the set of proof trees inductively, a proof tree of minimal height for a given tuple t has height given by

$$\min \left\{ k \geq 0 \mid \exists \tau_t \in T^k \right\}.$$

Intuitively, this means that a proof tree for a tuple t is of minimal height if there does not exist another valid proof tree with a smaller height. We emphasize that a valid proof tree must exist, since we have assumed that tuple is in the IDB of the Datalog specification and therefore its existence can be proved. Based on this inductive construction of proof trees, we reduce the problem of generating a fragment of a proof tree into the following incremental search problem.

Problem statement. Let P be a Datalog specification, and I be the instance computed by P. Given a tuple $t \in I$, find the tuples $t_1, \ldots, t_n$ such that $t \coloneq t_1, \ldots, t_n$ is a valid instantiation of a rule in P leading to a minimal height proof tree.

The problem statement is illustrated in Figure 7, where tuples $t_1, \ldots, t_n$ form the direct children of t in a minimal height proof tree. We also can denote $t_1, \ldots, t_n$ to be a *configuration* of the body of the corresponding rule. If this problem statement can be solved, then such a solution can be applied recursively to construct the subtrees rooted at each t_i, which would then form valid proof trees for those tuples. Thus, this recursive construction solves the original problem of constructing a fragment of the proof tree of minimal height. Once a certain number of levels have been constructed, or if the only remaining leaves are in the EDB (characterized by having a proof tree consisting of only a single node), then the fragment is complete.

3 A NEW PROVENANCE METHOD

A simple, partial solution to the problem might be to evaluate the Datalog specification using a standard evaluation strategy and then generate a proof tree by brute-force searching for a matching configuration for the body of a rule. However, this is an unfeasible approach for real-world problems where the resulting instance may contain millions of tuples, and there are also no guarantees that the proof trees produced in this manner are of minimal height. Alternatively, the Datalog evaluation strategy could be augmented to store minimal height proof trees as part of the computation; however, this would quickly run out of memory on even moderately sized instances.

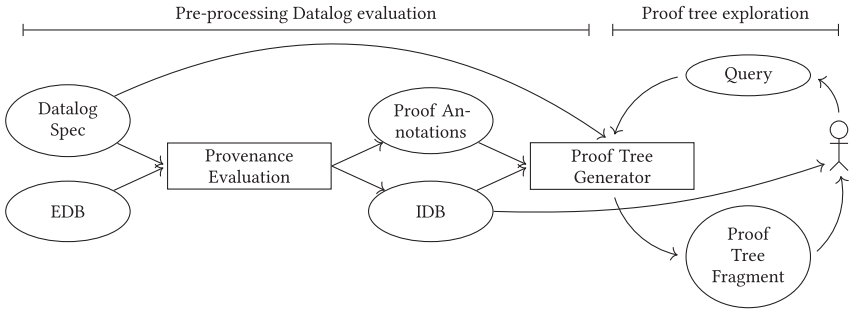


Fig. 8. Synthesized Proof Tree Generator system.

Moreover, the two main evaluation strategies for Datalog, *bottom-up* and *top-down* are unsuitable for solving this problem on their own. Bottom-up evaluation is an efficient method for generating tuples but does not store any information related to proof trees. However, top-down evaluation does compute proof trees as part of its execution, but there are no guarantees for minimality of height. Additionally, to prove the existence of a particular tuple requires proving the existence of every intermediate tuple up to the input tuples, and thus the problem of generating only fragments of proof trees cannot be solved by top-down. Thus, we present a hybrid solution for generating proof trees, consisting of a provenance evaluation strategy based on bottom-up evaluation, plus a debugging query mechanism to construct proof trees.

We summarize the system in Figure 8. The Datalog specification and input tuples (EDB) are the input into the system. The provenance Datalog evaluation generates a set of tuples (IDB) alongside proof annotations for these tuples. For each tuple, the annotation stores two numbers: one referring to the rule generating that tuple, and one referring to the height of a minimal height proof tree for that tuple. Using these annotated tuples as input, the interactive proof tree generator system allows the user to query for a proof tree fragment for any tuple in the IDB.

The proof tree generator is at the core of the interactive exploration of proofs for tuples. A user queries for a fragment of a proof tree, e.g., *two levels of a proof tree for vpt(b, l1)*, and the system returns the corresponding result. This system can answer any number of queries, and the user can query for *any* fragment of the proof tree for *any* tuple. As previously mentioned, this allows the user to interactively explore the proof for a tuple and find a meaningful explanation for a tuple.

The provenance evaluation strategy resembles a pre-computation step for debugging. The evaluation is performed only *once*, but the IDB with proof annotations can subsequently answer *any* debugging query using the same IDB resulting from evaluation. The ability to answer any debugging query without re-evaluation is an advantage over other selective provenance systems [Deutch et al. 2015; Lee et al. 2017], where the query is given prior to evaluation, which is then instrumented based on the query, and thus evaluation must be performed for each different query.

3.1 Standard Bottom-Up Evaluation

The basis of our approach is the standard bottom-up evaluation strategy for Datalog specification [Abiteboul et al. 1995]. The computational domain of standard bottom-up evaluation is the subset lattice consisting of sets of tuples, denoted *instances I*. The *naïve* algorithm for evaluation is based on the immediate consequence operator, Γ_P, which generates new tuples by applying rules in the Datalog specification to tuples in the current instance,

$$\Gamma_P(I) = I \cup \{t \mid t :\!\!- t_1, \ldots, t_n \text{ is a valid instantiation of a rule in } P \text{ with each } t_i \in I\}.$$

The result of Datalog evaluation is attained when Γ_P reaches a fixpoint, i.e., when $\Gamma_P(I) = I$. Note that this evaluation appears closely related to the inductive construction of proof trees and indeed the set of tuples represented by T^i is equal to the set of tuples generated by the ith application of Γ_P.

However, this naïve evaluation will repeat computations, since a tuple computed in some iteration will then be recomputed in every subsequent iteration. Therefore, the standard implementation of bottom-up evaluation in real systems such as in [Jordan et al. 2016] and [Whaley et al. 2005] is *semi-naïve*. Semi-naïve evaluation contains two main optimizations over naïve bottom-up evaluation:

(1) **Precedence graph optimization:** The Datalog specification is split into *strata*. First, a precedence graph of relations is computed, then each strongly connected component of the precedence graph forms a stratum. Each stratum is evaluated in a bottom-up fashion as a separate fixpoint computation in order based on the topological order of SCCs. The input to a particular stratum is the output of the previous strata in the precedence graph.
(2) **New knowledge optimization:** Within a single stratum, the evaluation is optimized in each iteration by considering the new tuples generated in the previous iteration. A new tuple is generated in the current iteration only if it directly depends on tuples generated in the previous iteration, therefore avoiding the recomputation of tuples already computed in prior iterations. We describe this process in further detail in Section 4.1.

With these two optimizations, semi-naïve performs less repeated computations than the naïve algorithm; however, our method for generating proof trees must now be tailored to semi-naïve evaluation.

Another essential extension of Datalog is negation, and the standard semantics for negated Datalog is *stratified negation* [Abiteboul et al. 1995; Greco and Molinaro 2015]. A negated predicate is denoted with a ! symbol, for example `!vpt(Var, Obj)` denotes the negation of `vpt(Var, Obj)`. Semantically, a negated predicate evaluates to true if no matching tuples exist in the instance. With stratified negation semantics, a negated predicate is only allowed if the contained variables exist in positive predicates elsewhere in the body of the rule (a condition also known as *groundedness*) and if the corresponding relation does not appear in a cycle in the precedence graph. During evaluation, the stratification of the precedence graph is carried out in a way such that the negated relations can be treated as input into a stratum, and a negated predicate is treated as a constraint, which holds if no corresponding tuple exists in the input instance.

3.2 Provenance Evaluation Strategy

These standard bottom-up evaluation semantics are extended to compute a minimal height proof tree for each tuple. Our extended semantics store *proof annotations* alongside the original tuples. In particular, for each tuple, the annotations are the *height* of the minimal height proof tree, and a number denoting the *rule* that generated the tuple. By using this extra information, we can efficiently generate minimal height proof trees to answer provenance queries (see Section 3.3).

In the context of semi-naïve evaluation, and in particular the precedence graph optimization, we describe the provenance evaluation strategy here for a single fixpoint computation (i.e., a single stratum). The resulting correctness properties translate directly to the evaluation of the full Datalog specification, since correctness holds for every stratum in the evaluation.

The rule number annotation is easily computed during bottom-up evaluation. With bottom-up evaluation, a new tuple t is generated if there is a rule $\rho_k : R(X) \text{ :- } R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ and a set of tuples $t_1, \ldots, t_n$ such that $t \text{ :- } t_1, \ldots, t_n$ forms a valid instantiation of the above rule. If this is the case, then the rule firing of ρ_k generates t, and thus the identifier

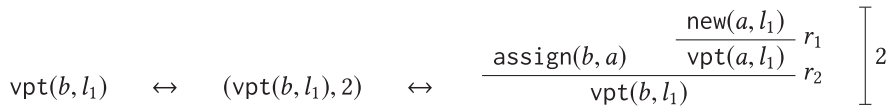


Fig. 9. Connecting a tuple to a proof tree via a height annotation.

$\rho(t) = k$ is stored as the rule number annotation for t. In this way, for each tuple, we track which rule is fired to generate that tuple.

However, the height annotations are more involved and relate closely to the semantics of bottom-up evaluation. Thus, we must develop a formalism for the height annotations, to ensure that it correctly computes the height of the minimal height proof tree for each tuple. To formalize tuples with height annotations, we define a *provenance lattice* as our domain of computation, which extends the standard subset lattice with proof annotations. An element of the provenance lattice is a provenance instance.

Definition 3.1 (Provenance Instance). A *provenance instance* is an instance of tuples I along with a function

$$h : I \to \mathbb{N}$$

which provides a height annotation of each tuple in the instance. We denote a provenance instance to be the pair (I, h).

The aim of these height annotations is to connect a tuple to its proof tree, as depicted in Figure 9. The middle value is a tuple along with its height annotation, which is an example of an augmented tuple in a provenance instance. The corresponding proof tree on the right has height matching this annotation.

Similar to the subset lattice of standard bottom-up evaluation, the domain of provenance evaluation should also form a lattice, in our case, based on the subset lattice of standard bottom-up evaluation, but with elements being provenance instances rather than instances. We denote this to be the *provenance lattice* $\mathcal{L}$, where elements are provenance instances. The ordering $\sqsubseteq$ of elements in the lattice is defined by:

$$(I_1, h_1) \sqsubseteq (I_2, h_2) \iff I_1 \subseteq I_2 \text{ and } \forall t \in I_1 : h_1(t) \geq h_2(t).$$

Intuitively, this ordering specifies that an augmented instance (I_1, h_1) is "less than" another augmented instance (I_2, h_2) if all tuples in I_1 also appear in I_2, with larger or equal height annotation. In $\mathcal{L}$, the bottom element is the empty instance, and a join between two instances (I_1, h_1) and (I_2, h_2) is the instance $(I_1 \cup I_2, h')$, where

$$h'(t) = \begin{cases} h_1(t) & \text{if } t \in I_1 \setminus I_2 \\ h_2(t) & \text{if } t \in I_2 \setminus I_1 \\ \min(h_1(t), h_2(t)) & \text{if } t \in I_1 \cap I_2 \end{cases}.$$

Under this definition, moving "up" the lattice toward the top element results in augmented instances with more tuples and smaller height annotations. This property guarantees the minimality of these height annotations, since a bottom-up Datalog evaluation is equivalent to applying a monotone function to move "up" a lattice.

The property that $\sqsubseteq$ a valid partial order is essential to demonstrate that standard properties of Datalog evaluation hold.

Lemma 3.2. $\sqsubseteq$ *is a partial order.*

In a similar fashion to the immediate consequence operator Γ_P operating on the subset lattice of Datalog instances, provenance evaluation is achieved with a consequence operator $\mathcal{T}_P$ operating on the provenance lattice. The result of evaluation is reached when $\mathcal{T}_P$ reaches a fixpoint, i.e., when $\mathcal{T}_P((I,h)) = (I,h)$. The main property $\mathcal{T}_P$ is that once a fixpoint has been reached, the proof tree height annotations are *minimal*, and they correspond to the heights of the smallest height proof trees.

The consequence operator $\mathcal{T}_P$ is defined in terms of the Γ_P operator:

Definition 3.3 (Consequence operator). The consequence operator, $\mathcal{T}_P$, generates a new provenance instance:

$$\mathcal{T}_P((I,h)) = (\Gamma_P(I), h'),$$

where h' is defined as follows. For any tuple $t \in \Gamma_P(I)$, let

$$G_t = \{(t_1, \ldots, t_n) \mid t \coloneq t_1, \ldots, t_n \text{ is a valid rule instantiation with each } t_i \in \Gamma_P(I)\}$$

be the set of all configurations of rule bodies generating t. Note this may be empty in the case of EDB tuples. Then,

$$h'(t) = \begin{cases} h(t) & \text{if } G_t = \emptyset \\ \min(h(t), \min_{g \in G_t}\{\max_{t_i \in g}\{h(t_i)\} + 1\}) & \text{otherwise} \end{cases}.$$

The generation of new tuples behaves in the same way as Γ_P. To illustrate the height annotations, consider the rule instantiation $\texttt{vpt}(b, l_1) \coloneq (\texttt{assign}(b, a), 0), (\texttt{vpt}(a, l_1), 1)$, with height annotations written alongside body tuples for convenience. From this rule instantiation, we would generate the tuple $\texttt{vpt}(b, l_1)$ with height annotation $\max(0, 1) + 1 = 2$. However, the instantiation $\texttt{vpt}(b, l_1) \coloneq (\texttt{load}(b, c, f), 0), (\texttt{store}(c, f, a), 0), (\texttt{vpt}(a, l_1), 1), (\texttt{alias}(c, c), 2)$ would also generate $\texttt{vpt}(b, l_1)$, but with height annotation $\max(0, 0, 1, 2) + 1 = 3$. The resulting instance after applying $\mathcal{T}_P$ will contain only the smaller annotation, and thus the resulting provenance tuple is $(\texttt{vpt}(b, l_1), 2)$.

Also, note that this semantics allow for the *update* of the height annotation for a tuple $t \in I$. If $\mathcal{T}_P(I,h) = (\Gamma_P(I), h')$ results in $h'(t) < h(t)$, then the height annotation of t is updated. An update may happen if $\mathcal{T}_P$ generates new tuples that form a valid configuration of a rule body generating t, with lower height annotations than a previous derivation.

We illustrate this definition of provenance evaluation strategy using the running example. As before, we denote (t,h) to be a tuple t with height annotation h. To highlight the importance of updating height annotations, we introduce a pre-processing step to generate the input instance for the points-to analysis. For example, a situation may arise in points-to analysis where a subclass constructer takes a superclass object as a parameter:

```
a2 = new O2(a);
a3 = new O3(a2);
```

In this situation, a store (e.g. `a.f = b;`) may also imply `a2.f = b;` and `a3.f = b;`. For the points-to analysis, a pre-processing step may be required to unroll the *store* and *assign* statements through the class hierarchy:

```
store(P, F, Q) :- instanceof(P, SuperP), store(SuperP, F, Q).
assign(Var1, Var2) :- instanceof(Var2, SuperVar2), assign(Var1, SuperVar2).
```

As a result of the recursive pre-processing step, the input instance to the points-to analysis fixpoint contains tuples with different height annotations. Figure 10 shows the derived `vpt` relation under the fixpoint computation with the provenance evaluation strategy. Importantly, in

Input:

$$\{(\texttt{new}(a, l_1), 0), (\texttt{assign}(b, a), 3), (\texttt{new}(c, l_3), 0), (\texttt{new}(d, l_4), 0),$$
$$(\texttt{store}(c, f, a), 2), (\texttt{load}(e, d, f), 0), (\texttt{load}(b, c, f), 0), (\texttt{assign}(a, b), 2)\}$$

Fixpoint iterations:

$$i_0 : \emptyset$$
$$i_1 : \{(\texttt{vpt}(a, l_1), 1), (\texttt{vpt}(c, l_3), 1), (\texttt{vpt}(d, l_4), 1)\}$$
$$i_2 : \{(\texttt{vpt}(a, l_1), 1), (\texttt{vpt}(c, l_3), 1), (\texttt{vpt}(d, l_4), 1), (\texttt{vpt}(b, l_1), 4)\}$$
$$i_3 : \{(\texttt{vpt}(a, l_1), 1), (\texttt{vpt}(c, l_3), 1), (\texttt{vpt}(d, l_4), 1), (\texttt{vpt}(b, l_1), 3)\}$$
$$i_4 : \{(\texttt{vpt}(a, l_1), 1), (\texttt{vpt}(c, l_3), 1), (\texttt{vpt}(d, l_4), 1), (\texttt{vpt}(b, l_1), 3)\}$$

Fig. 10. IDB relation *vpt* in each iteration of the fixpoint computation for the example Datalog specification.

iteration 3, the height annotation for $\texttt{vpt}(b, l_1)$ is updated as a result of a new derivation using $\texttt{load}(b, c, f), \texttt{store}(c, f, a)$, with lower height annotation than the previous derivation using $\texttt{assign}(b, a)$. This example demonstrates that the height annotations of tuples may be updated after they are initially computed, which is essential to ensure minimality.

It remains to be shown that the provenance evaluation strategy is correct, i.e., that $\mathcal{T}_P$ terminates and results in the same set of tuples as Γ_P. Additionally, we must show that the height annotations resulting from provenance evaluation strategy is minimal.

Lemma 3.4. *$\mathcal{T}_P$ computes the same tuples as Γ_P at fixpoint, i.e.*

(1) $\exists k$ s.t. $\mathcal{T}_P(\mathcal{T}_P^k((I, h))) = \mathcal{T}_P^k(I, h)$, and
(2) $\mathcal{T}_P^k(I, h) = (\Gamma_P^k(I), h^k)$ for some level annotation function h^k

Proof. By definition, $\mathcal{T}_P$ generates tuples in the same fashion as Γ_P. Since Γ_P always reaches a fixpoint, say after l iterations, i.e., $\Gamma_P(\Gamma_P^l(I)) = \Gamma_P^l(I)$, we have

$$\mathcal{T}_P^l((I, h)) = \left(\Gamma_P^l(I), h^l\right).$$

Any further applications of $\mathcal{T}_P$ do not change the set of tuples, since Γ_P has already reached a fixpoint. Thus, after l iterations, $\mathcal{T}_P$ computes the same tuples as Γ_P.

If there exists a $k \geq l$ such that $\mathcal{T}_P$ reaches fixpoint after k iterations, then the theorem is proved. Consider applying $\mathcal{T}_P$ to $(\Gamma_P^l(I), h^l)$. The set of tuples will not change. For any tuple $t \in \Gamma_P^l(I)$, the height annotation can only decrease as a result of applying $\mathcal{T}_P$, since $\mathcal{T}_P$ takes the minimum height over all rule configurations generating t and $h^l(t)$ also must result from such a configuration.

The height annotation is bounded from below by 0, since EDB tuples have non-negative annotations, and each subsequently generated tuple has increasing annotation. Therefore, applying $\mathcal{T}_P$ monotonically decreases the height annotation of t, which is bounded from below, so eventually, a fixpoint must be reached. Since this holds for all tuples in $\Gamma_P^l(I)$, $\mathcal{T}_P$ must reach a fixpoint after $k \geq l$ iterations. □

We have established that the provenance evaluation strategy terminates and computes the same set of tuples as standard bottom-up evaluation. It remains to be shown that the proof height annotations are minimal, i.e., that they reflect the real height of the minimal height proof tree for each tuple, and also that they correspond to real proof trees. The property of minimal height annotations is the major result of this section, since it demonstrates that our method generates proof trees of minimal height.

Theorem 3.5. *Let $\mathcal{T}_P^k((I,h)) = (\Gamma_P^k(I), h^k)$ be the resulting instance at fixpoint of $\mathcal{T}_P$. Then, for any arbitrary tuple $t \in \Gamma_P^k(I)$,*

(1) there does not exist any sequence of tuples $t_1, \ldots, t_n$ such that t :- $t_1, \ldots, t_n$ is a valid instantiation of a rule in P with each $t_i \in \Gamma_P^k(I)$ and $h(t) > \max\{h(t_1), \ldots, h(t_n)\} + 1$, and
(2) there is a valid proof tree for t with height $h^k(t)$.

Proof. The proof for part 1 is by contradiction. Assume such a sequence of tuples $t_1, \ldots, t_n$ exists. Consider applying $\mathcal{T}_P$ to the instance.

$$\mathcal{T}_P\left(\Gamma_P^k(I), h^k\right) = \left(\Gamma_P^k(I), h^{k+1}\right)$$

with $h^{k+1}(t) = \min_{g \in G_t}\{\max_{t_i \in g}\{h^k(t_i)\} + 1\}$ by definition of $\mathcal{T}_P$. The set of tuples does not change, since we assume that a fixpoint of Γ_P has already been reached.

Since the sequence $t_1, \ldots, t_n$ is a valid rule body configuration generating t, it is an element of G_t, and therefore is considered when updating the height annotation of t. Since the height annotation resulting from this sequence is lower than $h^k(t)$, the update will happen, and thus a fixpoint has not yet been reached.

Thus, we have a contradiction, and so such a sequence producing a lower height annotation cannot exist.

The proof for part 2 is by induction on the height annotation of t. Let $h = h^k(t)$ for simplicity.

If $h = 0$, then t is in the EDB. In this case, the proof tree with a single node corresponding to t is a valid proof tree. Otherwise, for $h > 0$, assume the hypothesis is true for all tuples with height annotation less than h. By definition of $\mathcal{T}_P$, there exists a sequence t :- $t_1, \ldots, t_n$ such that

$$h = \max(h^k(t_1), \ldots, h^k(t_n)) + 1.$$

By the assumption, there are valid proof trees for each t_i of height $h^k(t_i)$. We can generate a proof tree as follows:

$$\dfrac{\dfrac{\ldots}{t_1} \quad \ldots \quad \dfrac{\ldots}{t_n}}{t}$$

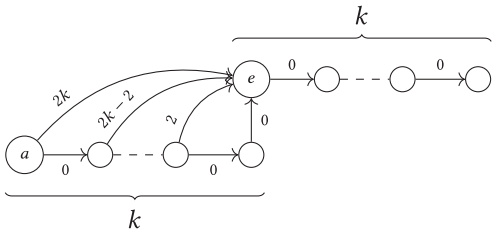
where each $\ldots$ represents the subtree forming a valid proof tree for each t_i. This resulting proof tree has height

$$\max(h^k(t_1), \ldots, h^k(t_n)) + 1,$$

which equals h. This forms a valid proof tree for t of height $h^k(t)$. □

We have shown the correctness and minimal height annotations of the provenance evaluation strategy for a single fixpoint computation. To evaluate a stratified Datalog specification in a semi-naïve fashion, each stratum is evaluated as a separate fixpoint using the provenance evaluation strategy. The correctness of the evaluation of a full Datalog specification follows from the correctness of each fixpoint evaluation.

3.2.1 Constraints and Negation. A constraint in a Datalog specification appears in the body of a rule, and takes the form $X \circ Y$, where X and Y are variables or constants, and $\circ$ is a standard binary relation operator (such as $=$, $<$, etc.). Importantly, every variable that appears in a constraint must be grounded, i.e., they must appear in a positive predicate in the body of that rule. Therefore, when a rule is instantiated, the variables appearing in constraints can also be instantiated with constants, and it can be trivially checked whether the constraint holds true. Explaining that a constraint holds true requires no further expansion in the proof tree, and thus no special consideration must be taken during Datalog evaluation.




```
reach(a, a).
reach(a, N) :- reach (a, M), edge(M, N).
```

Fig. 11. Example Datalog specification demonstrating the upper bound is tight. The label on each edge (x, y) denotes the height annotation $h(\text{edge}(x, y))$. Although edge is an input relation for this fixpoint, the height annotations may be non-zero as a result of some pre-processing stage (see Figure 10 for an example of how this may occur).

A negation also appears in the body of a rule, and takes the form $!R(X)$, where R is a relation name, and X is a sequence of variables or constants. Similarly to a constraint, all variables that appear in a negation must be grounded and an instantiation of the rule also instantiates the variables in the negation. Semantically, a negation holds true if the tuple formed by instantiating X does not exist in the relation R. To explain that a negation holds true requires to enumerate the relation R, showing that the relevant tuple is not contained. However, this is impractical in the presence of large relations, and thus our system displays the instantiation of the negation, asserting that it holds true. Using this approach, no further instrumentation of negations is required during the Datalog evaluation phase. The user may then choose to explain the non-existence of the tuple, using Section 3.4.

3.2.2 Complexity of Provenance Evaluation Strategy. In this section, we discuss the complexity of the provenance evaluation strategy. We characterize this complexity by the number of rule firings during evaluation. With standard bottom-up evaluation, we say that a rule is fired if it generates a new tuple. Therefore, for each tuple generated by the Datalog specification, there is exactly 1 rule firing. However, with the provenance evaluation strategy, a rule is also fired if it results in an update for the height annotation of a tuple. Therefore, we consider the number of updates performed during evaluation of the specification as a characterization of the extra amount of work done by provenance evaluation compared to standard bottom-up evaluation.

Theorem 3.6. *An upper bound for the number of updates performed is* $O(n \times \max h)$*, where* $\max h$ *denotes the maximum attained height annotation for any tuple during evaluation and* n *the number of tuples generated by the specification.*

Proof. To prove this, we need to show two things: (1) that it is a true upper bound and (2) that it is a tight bound.

To prove (1), consider a tuple t attaining a height annotation of $\max h$. Its annotation may only be updated if there is a valid derivation for t with a lower height. In the worst case, in each update, we reduce the annotation by 1, and thus we must perform $\max h$ updates to t. Considering all tuples produced by the specification, we may update all tuples in this way in the worst case, and therefore, we have $O(n \times \max h)$ updates.

To prove (2), we show an example attaining the upper bound, in Figure 11. In this example, the maximum height annotation is $2k$, and the tuple $\texttt{reach}(a, e)$ will be updated k times as new derivations are computed using nodes in the bottom chain. Furthermore, each tuple $\texttt{reach}(a, x)$ corresponding to nodes x in the "leg" must be updated $O(k)$ times as the tuple $\texttt{reach}(a, e)$ is

updated. Since there are k nodes in the leg, each of which is updated $O(k)$ times, we have in total $O(k^2)$ updates, which coincides with the upper bound. Therefore, this upper bound is tight. □

We also note that $\max h$ cannot exceed n, since in each iteration of $\mathcal{T}_P$ where a new tuple is generated, the height annotation of that tuple cannot exceed the maximum height annotation in the previous iteration, plus 1. If no more tuples are generated, then the height annotation for any tuple may not increase. Therefore, by generating a new tuple, we increase $\max h$ by at most 1, and therefore this value is at most the total number of tuples generated.

Therefore, in the worst case, the provenance evaluation strategy may have to do a quadratic amount of extra work compared to standard bottom-up evaluation. However, as real-world examples (see Section 5) show, such instances rarely occur, and scalability is maintained in most real-world cases.

3.3 Proof Tree Construction by Provenance Queries

Given a provenance instance (I, h) computed by the provenance evaluation strategy, and a tuple $t \in I$, the aim is to construct one level of a minimal height proof tree for t. We utilize the height annotations h and rule number annotations that are stored alongside the instance during bottom-up evaluation. We use a top-down approach for proof tree construction, starting from a query tuple and recursively finding tuples that form a valid instantiation of a rule generating the query tuple. Denote $h(t)$ to be the height annotation, and $\rho(t)$ to be the rule corresponding with the rule annotation for t.

The result of this search would be a sequence $t_1, \ldots, t_n$ such that $t \coloneq t_1, \ldots, t_n$ is a valid instantiation of $\rho(t)$ leading to a minimal height proof tree. A pre-requisite is that the provenance instance (I, h) is the result of bottom-up evaluation, and since all possible tuples are computed during this evaluation, we know that each $t_1, \ldots, t_n$ exists in I. Thus, this problem would be solved by searching for tuples in the already computed instance I.

However, we must constrain this search such that the result is part of a proof tree of minimal height, since there may be multiple valid configurations for the body of $\rho(t)$, and some configurations may not lead to minimal height proof trees. These constraints result from the annotations from bottom-up evaluation. From Theorem 3.5, there exists a configuration for the body that leads to a minimal height annotation for the head, and the height annotation for tuple t is generated as

$$h(t) = \max(h(t_1), \ldots, h(t_n)) + 1$$

by the consequence operator. Therefore, a configuration leading to the minimal height proof tree is $t_1, \ldots, t_n$ where $h(t_i) < h(t)$ for each t_i. Note that there may be multiple configurations leading to a proof tree of minimal height, and any of these configurations is a valid result for the problem.

The problem can be phrased as the following goal search. Given a tuple t, and a rule $\rho(t) : R(X) \coloneq R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ generating t, we want to find tuples $t_1, \ldots, t_n \in I$ such that $t \coloneq t_1, \ldots, t_n$ is a valid instantiation of $\rho(t)$, with proof annotations of each t_i satisfying the former constraints,

$$? \coloneq R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n), \text{matches}(t, X_1, \ldots, X_n), h(R_1(X_1)) < h(t), \ldots, h(R_n(X_n)) < h(t).$$

The condition $\text{matches}(t, X_1, \ldots, X_n)$ denotes that for a result $t_1, \ldots, t_n$, the variable mapping from each X_i to t_i is consistent with the variable mapping from X to t. This is related to the problem of unification in Prolog, and in our context is crucial to ensure that the resulting configuration forms a valid instantiation of ρ.

Example: We illustrate this construction using the running example. The query is for the tuple $\texttt{alias}(a, b)$. From the initial bottom-up evaluation, the height annotation is $h(\texttt{alias}(a, b)) = 4$, and the generating rule is $r_4 : \texttt{alias}(\texttt{Var1}, \texttt{Var2}) \coloneq \texttt{vpt}(\texttt{Var1}, \texttt{Obj}), \texttt{vpt}(\texttt{Var2}, \texttt{Obj})$.

```
path2(X, Z) :- edg(X, Y), edg(Y, Z), !edg(X, Z), X != Z.
path2(X, Z) :- edg(X, Y), path2(Y, Z), !edg(X, Z), X != Z.
```

(a) Example program with negation

```
edg(a, b).
edg(b, c).
edg(c, d).
```

(b) EDB Tuples

Fig. 12.

The search is for tuples forming a configuration for the body of r_4, vpt(Var1, Obj), vpt(Var2, Obj) satisfying the constraints

$$\begin{aligned} \texttt{Var1} &= a, \\ \texttt{Var2} &= b, \\ h(\texttt{vpt}(\texttt{Var1}, \texttt{Obj})) &< 4, \\ h(\texttt{vpt}(\texttt{Var2}, \texttt{Obj})) &< 4. \end{aligned}$$

In this example, the first two constraints corresponds with matches$(t, X_1, \ldots, X_n)$, and the last two constraints enforce the conditions for proof height annotations. Therefore, the goal search is

$$\begin{aligned} ?\ \text{:-}\ &\texttt{vpt}(\texttt{Var1}, \texttt{Obj}), \texttt{vpt}(\texttt{Var2}, \texttt{Obj}), \texttt{Var1} \neq \texttt{Var2}, \texttt{Var1} = a, \texttt{Var2} = b, \\ &h(\texttt{vpt}(\texttt{Var1}, \texttt{Obj})) < 4, h(\texttt{vpt}(\texttt{Var2}, \texttt{Obj})) < 4. \end{aligned}$$

In this case, we find the tuples $\texttt{vpt}(a, l_1), \texttt{vpt}(b, l_1)$, which form the next level of the proof tree:

$$\frac{\texttt{vpt}(a, l_1) \qquad \texttt{vpt}(b, l_1) \qquad a \neq b}{\texttt{alias}(a, b)}\, r_4$$

The other constraints and negations in the rule, denoted $\psi(X_1, \ldots, X_n)$ in the goal search, are handled by finding the variable instantiation for $X_1, \ldots, X_n$, and displaying the instantiated constraint/negation as a node in the proof tree. No further proof search is required, as constraints with constants are trivially shown to be true, and negation is proved by asserting that the tuple does not appear in the IDB.

To illustrate how negations and constraints are handled, consider the recursive program in Figure 12 finding all pairs of nodes in a graph with distance at least 2.

The output contains the tuple $\texttt{path2}(a, d)$, and its proof tree would be

$$\frac{\texttt{edg}(a, b) \qquad \dfrac{\texttt{edg}(b, c) \qquad \texttt{edg}(c, d) \qquad !\texttt{edg}(b, d) \qquad b \neq d}{\texttt{path2}(b, d)}\, r_1 \qquad !\texttt{edg}(a, d) \qquad a \neq d}{\texttt{path2}(a, d)}\, r_2$$

It is important to note that the goal search terminates as soon as the first solution is found, which is sufficient for generating a minimal height proof tree. This is in contrast with a standard bottom-up evaluation of a conjunctive query, which finds all possible configurations for the query.

The complexity of the goal search depends highly on the data structures used in the implementation. We assume fully (B-Tree) indexed nested loop joins. Therefore searching for a tuple for a rule with an m nested join, requires $O(\log^m n)$ time. Given a proof tree height of k, we need $O(k \log^m n) \equiv O(\log^m n)$ to traverse a single branch.

3.4 Provenance for Non-Existence of Tuples via User Interaction

The provenance evaluation strategy of the previous section explains the existence of tuples in relations. However, the non-existence of tuples may also indicate faults in either the input relations and/or in the rules.

Therefore, we extend our approach explaining on why a tuple *cannot* be derived, i.e., if the user expects a tuple, but it does not appear in the IDB, then the user may wish to investigate why the tuple is not produced. Alternatively, a user may want to understand why a negated body literal holds in a rule during the debugging process.

A non-existent tuple is characterized by *every* proof tree for the tuple failing to be constructed. The source of failure may be (1) tuples for the construction not being in the EDB/IDB, and/or (2) the constraints of rules not being satisfied. Given the potentially infinite number of failing proof trees, we avoid automatic procedures that represent a serious technical challenge and are not guaranteed to discover a failed proof tree containing the root cause of the fault. In practice, without a formal description of the root cause of the fault, the provenance system cannot decide which failed proof tree is most suitable.[1]

Hence, in our system, we take a pragmatic, semi-automated approach that is inspired by existing work such as [Lee et al. 2017, 2018]. Our system leverages user domain knowledge and allows user interactions to incrementally guide the construction of a single failing proof tree. Each user interaction produces a failing *subproof*, or one level of the proof tree. This failing proof tree provides a succinct representation of valuable information for a Datalog user to discover on why an expected tuple is not being produced by the specification and does not burden the user with too much unnecessary information.

Formally, we define the problem as follows: given a provenance instance (I, h) computed by the provenance evaluation strategy, a tuple $t \notin I$, and a rule $\rho : R(X) \coloneq R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$ with head relation matching t, we aim to find a configuration $t_1, \ldots, t_n$ for the body of ρ, such that either: (1) at least one $t_i \notin I$ or (2) the constraints $\psi(X_1, \ldots, X_n)$ are not satisfied. Such a configuration forms a failing subproof, and recursively constructing subproofs results in a full failed proof tree. Note that it would be impossible to find a configuration where all tuples $t_i \in I$ and constraints $\psi(X_1, \ldots, X_n)$ hold, since the prior assumption is that $t \notin I$. If such an instantiation cannot be found, then the tuple t can be generated by the Datalog specification, and thus $t \in I$.

For showing the non-existence of a tuple, the provenance system supports the Datalog user in constructing the failing proof tree in stages. The debugging query for non-existence has three user interaction steps that are repeated until the root cause of the fault is found. The user interaction steps are as follows:

(1) the user defines a query for the non-existence of a tuple,
(2) the user selects a candidate rule from which the tuple may have been derived,
(3) the user selects candidate variable values of unbound variables in the rule.

The system displays the rule application in the failing proof tree indicating the portions of the rule that fail (i.e., at least one literal / constraint must fail) and the portions of the rule that hold.

The Datalog user can continue the query with the newly found failing literals guiding the system to find the root cause of the fault. This process is semi-automated, since the nature of the fault is known by the Datalog user only.

Example: Consider the example from Figure 1 for which we want to query the non-existence of the tuple $\texttt{vpt}(b, l_4)$. In the first user interaction step, the Datalog user queries for an explanation for the non-existence of the tuple $\texttt{vpt}(b, l_4)$. Then, the Datalog user selects an appropriate rule such as rule r_2.

[1]Proof annotations such as introduced in the previous section can only describe existent tuples in the IDB. It is impossible to consider such annotations for tuples that are *not* produced by the specification.

The system can then produce a partial instantiation for the body of the rule, where variables matching the head are replaced by concrete values from t such as,

$$\texttt{vpt}(b, l_4) \text{ :- } \texttt{assign}(b, \texttt{Var2}), \texttt{vpt}(\texttt{Var2}, l_4).$$

In the last user interaction step, the Datalog user selects instantiations for the remaining free variables in rule r_2. For example, the Datalog user may choose the value d for the free variable Var_2,

$$\texttt{vpt}(b, l_4) \text{ :- } \texttt{assign}(b, d), \texttt{vpt}(d, l_4).$$

Given the instantiated rule, the provenance system will evaluate which portions of the subproof fail and which portions hold. With that information, the Datalog user can continue the exploration of the failing portions to find the root cause of the fault. A simple colour labelling helps the Datalog user to indicate which portions fail and hold, respectively.

$$\frac{\texttt{assign}(b, d)\ \text{X} \qquad \texttt{vpt}(d, l_4)\ \checkmark}{\texttt{vpt}(b, l_4)}\ r_2$$

In the above example, the red color and X denote the non-existence of the tuple $\texttt{assign}(b, d)$ in the IDB, i.e., a failing portion of the proof tree. The blue color with ✓ indicates that $\texttt{vpt}(d, l_4)$ holds.

In summary, our provenance system constructs a single failed subproof to explain the non-existence of a tuple. The construction of the failed subproof is guided by the Datalog user to ensure the answer contains a relevant explanation, given the infinitely many possible failed proof trees. The semi-automatic proof construction approach supports the Datalog user by highlighting which portions of the subproof hold and fail, respectively to guide the exploration.

3.5 Alternative Proof Tree Shapes

Our debugging strategy introduces an *interactive* system to explore fragments of proof trees to pinpoint faults in the Datalog specification. Therefore, we wish to minimize the number of user interactions required to find the fault. For this aim, minimal height proof trees are critical for reducing the number of user interactions in the fault investigation phase. The utility of this approach is backed by several user experiences in industrial-scale applications (see cf. Section 7.1.2 [Subotić et al. 2018]).

While generating proof trees of minimal height is useful for users, in principle our framework is more general and can support a variety of metrics that may be beneficial in future applications. In this section, we outline general properties of proof tree metrics by having the following properties for function h:

(1) The codomain of h must have a partial ordering $\sqsubseteq$, so that an update mechanism can be well defined. It is important that the annotation for a tuple can be updated if the same tuple is generated again with smaller (according to $\sqsubseteq$) annotation. This ensures that the resulting annotations are always minimal, since tuples will continue being updated with smaller annotations until a fixpoint with annotations is reached.

(2) The metric must be *compositional*, i.e., if t is generated by a rule instantiation t :- $t_1, \ldots, t_n$, then $h(t) = f(h(t_1), \ldots, (t_n))$. The importance of this property is twofold. First, it ensures that the values of the annotations can be computed during evaluation of the Datalog specification, by encoding f as a functor in the transformed Datalog specification. For example, a rule may be transformed to be $R(X, f(h_1, \ldots, h_n))$:- $R_1(X_1, h_1), \ldots, R_n(X_n, h_n)$. to compute the value of the annotation.

Second, the compositional property is important for the reconstruction of the proof tree. In the backward search for a body configuration that may produce the head tuple, f may be encoded as a constraint. For example, a backward search may be

$$? :- R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n), \text{matches}(t, X_1, \ldots, X_n), h(t) = f(h(R_1(X_1)), \ldots, h(R_n(X_n))),$$

where the last constraint ensures that the tuples found from the search correctly generate t with matching annotations.

(3) The metric must be *monotone*, i.e., $h(t_i) \sqsubseteq h(t)$ for all $1 \leq i \leq n$, and *bounded*, i.e., there is a minimum value c such that $c \sqsubseteq h(t)$ for any tuple t. This property ensures that the provenance evaluation strategy terminates. Monotonicity ensures that with each rule application, the annotation converges toward the minimum value c, and once it reaches c, then termination must occur.

If a given metric satisfies the above properties, then it can be used instead of proof tree height in our framework. Examples of such metrics could be the size of proof trees by number of nodes or a sequence of k proof tree heights describing the smallest k proof trees for each tuple. One could also combine multiple metrics by a lexicographical ordering, for example producing proof trees of minimal height with a minimum number of nodes.

Given that our framework can be adapted to various proof tree shapes, the provenance system could be adapted for other applications that make use of other metrics. For example, given a program analysis written in Datalog, the origin of a bug alarm can be explained through its provenance. Such ideas, such as *thin slicing* [Sridharan et al. 2007], may also be able to use our provenance framework as a building block, and we leave this integration to future work.

4 IMPLEMENTATION IN SOUFFLÉ

In this section, we describe the implementation of our provenance system in Soufflé [Jordan et al. 2016]. Soufflé[2] is an open-source system that is available under the UPL license and is implemented in C++. Soufflé is a parallel Datalog engine designed for shared memory, multi-core machines, synthesizing highly performant parallel C++ code from Datalog specifications.

Our provenance evaluation strategy and proof tree construction system are tightly integrated into the Soufflé engine.[3] Through this tight integration in Souffé, we are able to achieve high parallel performance for the provenance evaluation and enable a single evaluation phase to answer multiple debugging queries. In contrast, previous approaches [Deutch et al. 2015; Köhler et al. 2012; Lee et al. 2017] implement a Datalog re-writing scheme and simply evaluate the re-written Datalog in an existing engine.

The Soufflé synthesizer performs a series of specialization steps based on Futamura projections [Futamura 1999], which synthesize a C++ program with the same semantics as the Datalog specification. The main specialization step is the compilation of Datalog into an intermediate representation called Relational Algebra Machine (RAM) that has imperative and relational algebra elements to perform simple relational algebra operations to compute fixed-points for semi-naïve evaluation. The RAM representation of a Datalog specification is in turn compiled into C++ code. The resulting C++ code implements a specialized semi-naïve algorithm for the rules in the Datalog specification that have similar performance to a hand-written program [Allen et al. 2015; Jordan et al. 2016]. In the following, we discuss the implementation of semi-naïve evaluation [Abiteboul et al. 1995] in Soufflé and discuss subsequently how the synthesized semi-naïve evaluation is

[2] [sou 2017], https://github.com/souffle-lang/souffle.

[3] Our provenance evaluation strategy is not specific to Soufflé—it can be integrated into any bottom-up evaluation Datalog engine.

```
1  SCAN assign AS t0
2    SEARCH @delta_vpt AS t1 ON INDEX t1.c0=t0.y WHERE (t0.x,t1.c1) NOT IN vpt
3      INSERT (t0.x, t1.c1) INTO @new_vpt
```

Fig. 13. Existence check prior to inserting.

replaced by the provenance evaluation strategy. Semi-naïve evaluation avoids re-computations of tuples by using the fact that new tuples can only be deduced from new tuples in the previous iteration. This is achieved by creating a *new* and *delta* version of each relation. The *new* and *delta* version of a relation store the tuples found in the current iteration and the previous iteration, respectively. For example, with a rule $R_0(X_0)$:- $R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$, each relation R_k is transformed to become a set of relations for each iteration i:

$$R_k^i, new_{R_k}^i, \Delta_{R_k}^i,$$

where R_k^i stores all the tuples for relation R_k that are computed up until iteration i, while $\Delta_{R_k}^i$ stores only the tuples in R_k computed in iteration i, without any tuples computed in previous iterations. The relation $new_{R_k}^i$ is an intermediate relation used to compute the Δ relations. The essential optimization, compared to naïve evaluation, is to realize that in iteration $i+1$, a new tuple is only generated if it directly depends on a tuple generated in iteration i. If this condition does not hold, i.e., if it depends on knowledge generated in prior iterations, then the tuple would also have been generated in a previous iteration, and thus generating it again would be a redundant computation. Thus, a tuple is only generated in iteration $i+1$ if it depends on a Δ^i relation. This constraint is enforced by transforming the original rule to a set of new Datalog rules that perform semi-naïve evaluation:

$$\begin{aligned}
new_{R_0}^{i+1}(X_0) &\text{ :- } \Delta_{R_1}^i(X_1), R_2(X_2), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)\\
&\ldots\\
new_{R_0}^{i+1}(X_0) &\text{ :- } R_1(X_1), \ldots, \Delta_{R_k}^i(X_k), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)\\
&\ldots\\
new_{R_0}^{i+1}(X_0) &\text{ :- } R_1(X_1), \ldots, R_{n-1}(X_{n-1}), \Delta_{R_n}^i(X_n), \psi(X_1, \ldots, X_n).
\end{aligned}$$

Thus, $new_{R_0}^{i+1}$ contains tuples of R_0 that depend directly on tuples generated in iteration i. The relation $\Delta_{R_0}^{i+1}$ is computed as

$$\Delta_{R_0}^{i+1} = new_{R_0}^{i+1} - R_0^i,$$

where the relations are viewed as sets of tuples and $-$ denotes set minus. Thus, $\Delta_{R_0}^{i+1}$ contains only tuples generated in iteration i, and no tuples generated in previous iterations. The relation R_0^{i+1} denotes all tuples generated in iterations $0, \ldots, i+1$, and is computed as the union

$$R_0^{i+1} = R_0^i \cup \Delta_{R_0}^{i+1}.$$

With these auxiliary relations, the final result for the relation R_0 is the final R_0^i once a fix-point is reached, i.e., the result of the Datalog specification has stabilized. Note that Soufflé evaluates the Δ_R^{i+1} relation by computing the tuples without an explicit set minus operation, since an existence check determines whether the tuple already exists in R^i relation before it is inserted into Δ_R^{i+1}. For example, Figure 13 depicts a snippet of a Soufflé RAM program, part of the semi-naïve evaluation of the rule r_2 : `vpt(Var, Obj)` :- `assign(Var, Var2), vpt(Var2, Obj)`. The join is performed via a loop nest iterating over tuples of relations efficiently via indexes [Subotić et al. 2018]. Line 2 computes the new tuples to be added to the Δ_{vpt}^{i+1} relation, where the `NOT IN` operation is an existence check to ensure the generated tuple does not already exist in the vpt^i relation.

4.1 Implementing Provenance Evaluation Strategy

The main challenge of integrating the provenance evaluation strategy is to allow the synthesis to be aware of proof annotations. In particular, the semi-naïve evaluation machinery must be replaced by the provenance evaluation strategy as described in Section 3.2 to handle the proof annotations. Another critical part of this machinery is the synthesis of data structures [Jordan et al. 2019; Subotić et al. 2018] for relations that are specialized for the operations in the program. The synthesized data structures have to be extended for proof annotations as well, enabling an update semantics in Datalog for the annotations.

For the provenance evaluation strategy, we need to amend relations by extra attributes to contain the proof annotations. We utilize the synthesis pipeline of Soufflé by introducing two provenance attributes for each relation. The first attribute represents the rule number of the rule that generated the tuple, and the second attribute represents the proof tree height. These two new attributes are introduced for each relation at the syntactic level in Soufflé. A predicate $R(X)$ is transformed into $R(X, @\text{rule}, @\text{height})$. For the sake of readability in this text, we distinguish between original and provenance tuples, where an original tuple is a provenance tuple without proof annotations. We rewrite all logic rules at the syntactic level to take account of the two provenance attributes constituting the proof annotation for our system, and to compute the value of the annotations. The proof annotation instrumentation is performed as follows where a rule

$$\rho_k : R(X) \text{ :- } R_1(X_1), \ldots, R_n(X_n), \psi(X_1, \ldots, X_n)$$

is transformed into:

$$\begin{aligned}\rho_k : R(X, k, \max(@\text{height}_1, \ldots, @\text{height}_n) + 1) \text{ :-} \\ R_1(X_1, \_, @\text{height}_1), \ldots, R_n(X_n, \_, @\text{height}_n), \psi(X_1, \ldots, X_n).\end{aligned}$$

The transformed provenance rule computes level and height annotations for a new tuple, according to the semantics in Section 3.2. Since the rules are known statically, the rule number annotation k can be assigned a constant value for each rule in the transformed specification. The rule numbers of the body predicates are ignored by using _ in each body predicate, since they do not influence the head predicate.

The transformed provenance rule syntactically represents the computation of proof annotations during rule evaluation. However, the actual execution of provenance rules differs from a standard semi-naïve evaluation as presented in [Abiteboul et al. 1995]. The reason is the update mechanism: A newly discovered provenance tuple may overwrite an existing provenance tuple if they are the same original tuple, but the new tuple has a smaller height annotation.

The provenance evaluation strategy extends the semi-naïve algorithm by updating the rule number and the height annotation of a tuple (as defined by $\mathcal{T}_P$) if the original tuple already exists and the newly generated tuple has smaller height annotation. In other words, if a smaller proof tree could be found in a subsequent iteration for the same tuple, then an update occurs. Otherwise, if the original tuple does not already exist, the provenance tuple is inserted into the relation as is. Thus, the rule computing $\Delta_{R_0}^{i+1}$ in the semi-naïve evaluation is modified to accommodate the possibility of updates, i.e.,

$$\Delta_{R_0}^{i+1} = \left(new_{R_0}^{i+1} - R_0^i\right) \cup \left\{t \in R_0^i \mid h^i(t) > h^{i+1}(t)\right\},$$

where h^i denotes the height annotations in iteration i.

Therefore, with our provenance evaluation strategy, we integrate the possibility of an annotation update into the data structure. During an insertion operation, if the same original tuple is discovered, with a larger height annotation than the current tuple, then an update occurs. Therefore, we wish to call the insert operation if *either* the tuple does not already exist *or* the existing

```
1  SCAN assign AS t0
2    SEARCH @delta_vpt AS t1 ON INDEX t1.x=t0.y
3      IF (t0.x,t1.y,_,(max(t0.@height,t1.@height)+number(1))) PROV NOT IN vpt
4        INSERT (t0.x, t1.y, number(2), (max(t0.@height,t1.@height)+number(1))) INTO @new_vpt
```

Fig. 14. Provenance version of RAM loop nest.

tuple has the larger proof annotation. A specialized existence check is implemented, implicitly implementing the semantics of the provenance Δ_R^{i+1} relation. Similarly to the standard existence check, the special existence check is implemented as part of the data structure that has been specialized for each relation.

The result is the RAM snippet in Figure 14. The obvious differences compared to Figure 13 are in lines 3 and 4, where the level annotation is computed within the loop nest, as part of the insertion. Furthermore, the `PROV NOT IN` operation in line 3 denotes the special provenance existence check, which allows the `INSERT` to proceed if either the tuple does not already exist, or exists with a larger proof height annotation. Thus, this implements the update semantics discussed above.

However, the specializations in the data structures still remain to be discussed. Soufflé employs a highly specialized parallel B-Tree data structure [Jordan et al. 2019], with index orderings for the attributes generated automatically via an optimization problem [Subotić et al. 2018]. The B-Tree employs a special optimistic read/write lock for each node, allowing high throughput for parallel insertion. During an `insert` operation, a thread may obtain a read lease for each node as it checks whether the tuple to be inserted already exists. If an insertion is required, then it checks if the lease has changed, and restarts the whole procedure if it has. Otherwise, it upgrades to a write lease and inserts the tuple into the correct position in the B-Tree. The data structure also takes advantage of Soufflé's Datalog evaluation setting, where a single relation is either read from, or written to, but never at the same time. Therefore, there are no interleaved reads and writes, and so read operations are not synchronized.

With the proof annotations, we modify these specializations so that they can take into account the provenance semantics. The important step is the *update* semantics, and thus we integrate an update mechanism into the `insert` operation, without requiring to delete and then re-insert. The provenance evaluation strategy requires two main modifications to our B-Tree data structure. First, the existence check for insertion should consider only the original tuple, and ignore annotations. This ensures that Datalog set semantics are preserved and that no duplicate original tuples can exist. However, note that we still need to retrieve the full tuple, including its proof annotations. This is important for the proof tree construction, discussed in the next section. To address this concern, we use different lexicographical orderings of indices for the `insert` and `retrieve` operations. The `insert` index order does not include the attributes storing the proof annotations (so that annotations are not considered when checking existence), while the `retrieve` index order does. We also need to ensure that updating an annotation does not change the ordering of tuples according to the index; otherwise subsequent index supported searches will fail. Therefore, the `retrieve` index order requires the annotation attributes to be at the end, as this guarantees that an update to the annotations does not affect tuple ordering.

Second, we must have a mechanism to update existing tuples to implement the update semantics in Section 3.2. To achieve this, we modified the `insert` operation so that it may also update any existing tuple with a smaller proof annotation. This insertion first requires to check if the original tuple exists in the B-Tree. If it does, rather than aborting (as it would with standard Datalog evaluation), then the insertion then checks the annotation. If the height annotation of the existing tuple is larger than the tuple to be inserted, then the annotations of the existing annotation are updated with new values. Note that during an update, a read lease also needs to be validated and

```
> explain alias("a", "b")
                                  new("a", "l1")
                                  -----------(R1)
new("a", "l1")  assign("b", "a") vpt("a", "l1")
-----------(R1) -----------------------------(R2)
vpt("a", "l1")           vpt("b", "l1")          "a" != "b"
-------------------------------------------------------------(R1)
                alias("a", "b")
```

Fig. 15. Explaining the tuple $alias(a, b)$.

```
Enter command > explainnegation vpt("b", "l4")
1: vpt(Var,Obj) :-
   new(Var,Obj).

2: vpt(Var,Obj) :-
   assign(Var,Var2),
   vpt(Var2,Obj).

3: vpt(Var,Obj) :-
   load(Var,Y,F),
   store(P,F,Q),
   alias(P,Y),
   vpt(Q,Obj).

Pick a rule number: 2
Pick a value for Var2: d
assign("b", "d") X  vpt("d", "l4") ✓
-----------------------------------(R2)
            vpt("b","l4")
```

Fig. 16. Explaining the non-existence of the tuple $vpt(''b'',''l4'')$.

upgraded to a write lease. The integration of the update into the `insert` operation avoids any need to delete and re-insert tuples, thus improving the efficiency of the provenance evaluation strategy. These modifications to the B-Tree reflect the desired insertion semantics, and updates are handled directly in the `insert` operation. All other retrieval operations for the B-Tree are not modified, and tuples can be retrieved as normal, including their proof annotations.

4.2 Implementing a Proof Tree Construction User Interface

After the provenance evaluation strategy is completed, the proof tree construction stage is driven by the user. It is critical that this process is also fully parallelized and highly performant.

The user interface is implemented as a command line, where the user can enter queries to explain the existence and non-existence of a tuple. For example, the query `explain alias("a", "b")` results in the proof tree in Figure 15. Explaining the non-existence of a tuple, i.e., the query `explainnegation vpt("b", "l4")`, results in the interaction in Figure 16. The user may also select the size of proof tree fragments to display, i.e., `setdepth 6` instructs the system to construct six levels of the proof tree in the next query. For each debugging query, the system invokes the relevant procedure to construct a proof tree fragment.

It is critical that the proof tree construction procedures are highly performant, since the constructed IDB may be very large, and we may need to search through many tuples to construct a proof tree fragment. Therefore, the proof tree construction procedures must be tightly integrated into the Soufflé system to enable a high-performance, parallel search. We integrate these

```
1  SUBROUTINE vpt_2_subproof
2    SCAN assign AS t0 WHERE t0.x = argument(0) AND t0.@level_number < argument(2)
3      SEARCH vpt AS t1 ON INDEX t1.x=t0.y AND t1.y=argument(1) WHERE t1.@level_number < argument(2)
4        RETURN (t0.x, t0.y, t0.@rule_number, t0.@level_number, t0.y,
5                t1.y, t1.@rule_number, t1.@level_number)
```

Fig. 17. Subroutine for example program.

procedures into the Soufflé RAM, utilizing the existing translation from RAM to parallel C++. Moreover, since the provenance evaluation strategy uses specialized B-Tree data structures, the proof tree construction can also utilize index supported searches to find relevant tuples.

Recall that the proof tree construction is facilitated by searches for subproofs. Therefore, we require a specialized framework in the Soufflé RAM to implement a subproof search. We term this framework a *subroutine* framework. Each subproof search can be implemented as a subroutine, thus integrating with the Soufflé RAM.

To explain the existence of a tuple, a subproof search is required to search the body of a rule for matching body tuples, satisfying the constraint that proof tree height is lower than the current tuple. This backward search for a single rule is implemented as a subroutine. For example, the rule r_2 : `vpt(Var, Obj)` :- `assign(Var, Var2), vpt(Var2, Obj)` is implemented as the subroutine in Figure 17.

Lines 2–5 represent a search through a database that is already constructed by the initial bottom-up evaluation, to find tuples that satisfy the constraints required for the construction of a proof tree fragment. The values of `argument(0)` and `argument(1)` are the values in the head tuple, and `argument(2)` is the height annotation of the head tuple. Therefore, this subproof search is parameterized by the head tuple. The relations of the body atoms, `assign` and `vpt` are searched to find tuples `t0` and `t1` that match the body of the rule. Importantly, the constraints for the level number are encoded in lines 3 and 4, ensuring that the resulting tuples have level number annotations lower than the query tuple. As shown in Section 3.3, being able to apply this operation recursively allows us to generate the full proof tree.

Similarly, to generate a failed subproof to explain the non-existence of a tuple, the search for failing and holding parts of a subproof is implemented as a subroutine. Given an instantiated rule (which is produced via user interaction), a subroutine returns whether each body tuple is in the IDB and whether each constraint is satisfied.

5 EXPERIMENTS

In this section, we conduct experiments with the provenance evaluation strategy and provenance queries implemented in Soufflé (see Section 4). The experiments are conducted for large-scale Datalog specifications. We have the following experimental research claims:

> **Claim-I:** Our provenance evaluation strategy only has a minor impact on the runtime performance and remains scalable for realistic datasets and rulesets.
> **Claim-II:** The provenance queries for exploring proof tree fragments scales to large sizes, allowing efficient interactive exploration of proof trees.
> **Claim-III:** Minimal height proof trees are very large for realistic benchmarks, substantiating the need for interactive proof tree exploration.

Our experiments were performed on a computer with an Intel Xeon Gold 6130 CPU and 192 GB of memory, running Fedora 27. Soufflé executables were generated using GCC 7.3.1.

Table 1. Statistics for Doop Benchmarks

	context-insensitive		1-obj, 1-heap	
Benchmark	# EDB	# IDB	# EDB	# IDB
antlr	8,319,095	21,832,232	8,319,095	24,145,648
bloat	4,468,277	13,104,020	4,468,277	15,417,516
chart	8,743,770	22,975,742	8,743,729	25,289,200
eclipse	4,389,770	13,076,265	4,389,799	15,389,708
fop	8,769,583	22,970,533	8,769,572	25,283,913
hsqldb	9,007,087	24,561,921	9,007,087	26,875,437
jython	5,203,400	17,158,375	5,203,400	19,471,797
luindex	4,396,394	13,415,336	4,396,394	15,728,788
lusearch	4,396,415	13,415,390	4,396,394	15,728,788
pmd	8,388,202	22,853,676	8,388,202	25,167,134
xalan	8,670,980	23,488,951	8,670,966	25,802,385

5.1 Performance of the Provenance Evaluation Strategy

Doop. For the first set of experiments, we use the Doop [Bravenboer and Smaragdakis 2009] points-to analysis framework. We experiment with Doop's *context-insensitive* and *1-object-sensitive, 1-heap* (1-obj, 1-heap) analyses that exhibit different runtime complexities. As inputs for the points-to analyses, we compute the points-to sets for the DaCapo 2006 Java program benchmarks. Each analysis contains approx. 300 relations, 850 rules and produces up to approximately 26 million output tuples on the DaCapo benchmarks (see Table 1).

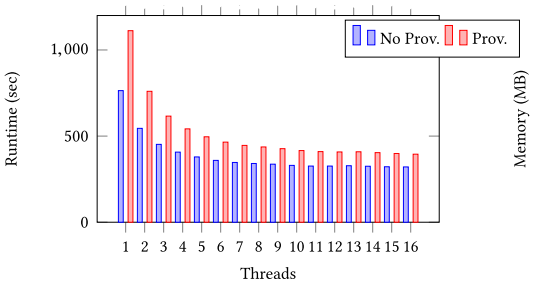
In Table 2, we present the runtime and memory consumption of Soufflé with eight threads, comparing standard Soufflé with our provenance evaluation strategy with proof annotations. We use the DaCapo benchmarks with both the context-insensitive and 1-obj, 1-heap analysis. As expected, Soufflé with proof annotations incurs an overhead during evaluation. This overhead for the provenance evaluation strategy is typically within a factor of 1.3, which is a small overhead to pay for being able to generate minimal proof trees for all possible tuples in the IDB. Hence, we demonstrate the viability of the provenance evaluation strategy for large-scale Datalog specifications, substantiating Claim I. We noticed that the runtime overhead for the context-insensitive analysis was smaller across all benchmarks than that of the 1-obj-1-heap analysis due to cache locality that was more prominent for smaller memory footprints. Note that the overhead for memory consumption is similar to performance overheads, at approximately 1.45×. This overhead results from the storage of extra proof annotations during evaluation.

In contrast, a naïve direct encoding approach (cf. Chapter 5 in [Zhao 2017]), where each tuple is annotated with its full subproof (i.e., direct children in the proof tree), resulted in excessive memory usage (up to 100×) on a simple transitive closure experiment with 2000 tuples, and thus cannot be deployed for large-scale Datalog specifications such as those found in Doop.

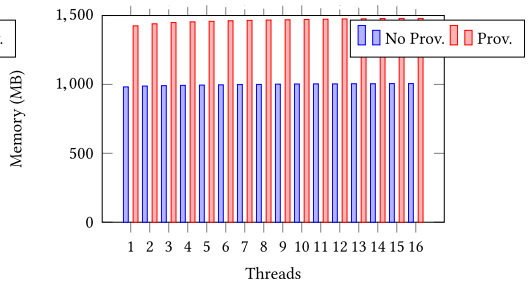
Figure 18(a) and (b) show the total runtime and average memory usage for each of the Doop DaCapo benchmarks with both Doop (context-insensitive and 1-obj-1-heap) analyses, running with multiple threads. The figure demonstrates that the provenance evaluation strategy is scalable, in that the overhead is sustainable with an increasing number of threads. We observe that the overall runtime decreases for provenance and without provenance until 5 threads, and increased thereafter. This is caused by the synchronization of Soufflé's rule evaluation system and is not specific to provenance. It is interesting to note that the runtime overhead is larger with fewer threads, being 1.45× for 1 thread while being 1.23× for 16 threads. Again, this is related to the

Table 2. Runtime and Memory Usage Overheads for Soufflé with and without Proof Annotations with Eight Threads

	Runtime (sec)			Memory (MB)		
Benchmark	No Prov.	Prov.	(×)	No Prov.	Prov.	(×)
context-insensitive						
antlr	9.73	12.29	1.26	595	900	1.51
bloat	9.54	12.25	1.28	596	900	1.51
chart	15.89	19.60	1.23	1,103	1,604	1.45
eclipse	9.64	11.76	1.22	593	898	1.51
fop	15.57	19.48	1.25	1,079	1,579	1.46
hsqldb	16.36	19.73	1.21	1,124	1,642	1.46
jython	11.00	13.62	1.24	731	1,090	1.49
luindex	9.62	12.00	1.25	594	905	1.52
lusearch	9.80	12.23	1.25	593	904	1.52
pmd	15.58	18.90	1.21	1,053	1,542	1.46
xalan	15.59	19.54	1.25	1,091	1,595	1.46
geo-mean			1.24			1.44
1-obj, 1-heap						
antlr	10.84	12.60	1.16	936	1,310	1.40
bloat	15.77	22.00	1.40	732	1,082	1.48
chart	21.84	28.13	1.29	1,242	1,788	1.44
eclipse	15.76	21.00	1.33	729	1,080	1.48
fop	22.21	29.63	1.33	1,216	1,756	1.44
hsqldb	23.01	29.43	1.28	1,256	1,823	1.45
jython	17.54	22.96	1.31	868	1,270	1.46
luindex	15.94	21.55	1.35	730	1,086	1.49
lusearch	15.95	21.25	1.33	731	1,087	1.49
pmd	21.58	28.21	1.31	1,190	1,725	1.45
xalan	22.09	28.68	1.30	1,224	1,773	1.45
geo-mean			1.31			1.46



(a) Total evaluation runtime of Soufflé on all DaCapo benchmarks with each Doop (context-insensitive and 1-obj, 1-heap) analysis with and without provenance



(b) Average evaluation memory usage of Soufflé on all DaCapo benchmarks with each Doop (context-insensitive and 1-obj, 1-heap) analysis with and without provenance

Fig. 18.

Table 3. Statistics for Ddisasm Benchmarks

Benchmark	# EDB	# IDB
cactusADM	845,762	12,758,417
calculix	1,934,084	39,101,856
gamess	9,892,299	204,764,756
gcc	3,968,589	133,994,711
GemsFDTD	445,185	16,483,816
gobmk	4,211,765	34,935,275
gromacs	1,122,734	16,238,070
h264ref	638,837	12,991,980
omnetpp	720,581	18,261,432
perlbench	1,330,330	65,875,051
povray	1,160,773	55,857,237
tonto	4,767,218	285,090,829
wrf	4,690,140	97,986,011

underlying hardware architecture providing caches and memory lanes for each core. With more threads, the memory bandwidth to access the logical relations with proof annotations improves.

The memory usage of the provenance evaluation strategy has a consistent overhead of 1.45×, which aligns with our expectations that there would be a reasonable overhead associated with storing the provenance annotations per tuple. Note that this overhead is constant over any number of threads, since the amount of extra information stored overall does not change with the number of threads.

Ddisasm. For the second set of experiments, we use the Ddisasm [Flores-Montoya and Schulte 2019] disassembler tool. The Ddisasm tool takes as input an executable binary and produces as output an assembly version of that binary. The main part of Ddisasm is a Soufflé specification containing 535 relations and 1,020 Datalog rules. We run Ddisasm with and without provenance annotations, with eight threads. As a benchmark suite, we use a subset of the SPEC CPU 2006 benchmarks, presenting only those with disassembly runtimes longer than 5 s. Each benchmark takes between 400 thousand and 9 million tuples as input and produces between 12 and 285 million tuples as output (see Table 3).

The results in Table 4 demonstrate that the provenance evaluation strategy incurs a runtime and memory overhead. For Ddisasm, the runtime overhead is on average approximately 1.39×, which is an acceptable overhead for generating provenance annotations. However, the memory overhead for Ddisasm is 2.6×, which is considerably higher than for Doop. This is due to extra indices that were automatically generated to cover operations during the proof tree construction stage. In the worst case, a single relation, `instruction`, required three indices for standard Datalog evaluation, but nine indices for the provenance evaluation strategy. This means that tuples in `instruction` are replicated 3× more with provenance, in addition to the overhead of the provenance annotations themselves. Future work optimizing the index generation algorithm in Soufflé will improve the memory overhead for situations where multiple indices are generated due to the provenance evaluation operations.

The main outlier in Table 4 is `tonto`, which exhibits a 2.2× runtime overhead and a 3.37× memory overhead. This particular benchmark generated 76% of its 285M tuples in the `string_part` relation, which had double the indices for the provenance evaluation strategy. As a result of this data replication, along with an increased cache miss rate (approximately double for the provenance

Table 4. Runtime and Memory Usage Overheads for DDISASM on SPEC Benchmarks with and without Provenance Annotations with Eight Threads

	Runtime (sec)			Memory (MB)		
Benchmark	No Prov.	Prov.	(×)	No Prov.	Prov.	(×)
cactusADM	6.76	9.26	1.37	700	1,974	2.82
calculix	16.21	21.65	1.34	1,868	5,025	2.69
gamess	105.46	127.68	1.21	9,636	26,076	2.71
gcc	99.05	104.86	1.06	5,180	12,160	2.35
GemsFDTD	7.22	9.01	1.25	709	1,760	2.48
gobmk	19.39	35.56	1.83	1,450	3,453	2.38
gromacs	7.43	10.54	1.42	894	2,558	2.86
h264ref	5.54	7.85	1.42	644	1,679	2.61
omnetpp	10.4	13.17	1.27	763	1,952	2.56
perlbench	16.06	21.53	1.34	2,454	5,460	2.22
povray	11.57	15.71	1.36	2,071	4,532	2.19
tonto	340.49	749.63	2.20	24,395	82,152	3.37
wrf	58.28	74.2	1.27	4,739	13,085	2.76
geo-mean			1.39			2.60

evaluation strategy, resulting from worse cache coherence from storing provenance annotations), the runtime and memory overheads are larger than for other benchmarks.

While the runtime and memory overhead on DDISASM is higher than on DOOP, the result is still an acceptable price to pay to generate debugging annotations.

Comparison with current approaches. The current state of the art in tracking Datalog provenance is to instrument the specification with a given provenance query. The instrumented Datalog specification can then be evaluated using any Datalog engine. One example of this approach is the top-*k* approach [Deutch et al. 2015, 2018], where Datalog specifications are instrumented based on a provenance query taking the form of a *derivation tree pattern*.

For our experiments, we implemented the instrumentation algorithm presented in Deutch et al. [2018] and evaluated the resulting Datalog using Soufflé, again using DOOP as the test Datalog specification. Since the instrumentation requires a specific derivation tree pattern, we choose one that produces any proof tree for a single tuple from the `VarPointsTo` relation in DOOP. The tuple we choose describes a points-to relationship between two variables in `java.lang.Double` and `java.lang.Long`, which exists in the result for every DaCapo benchmark.

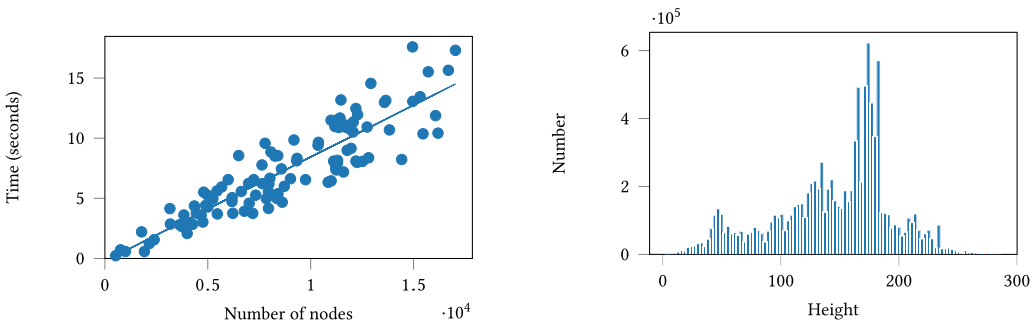
The results in Table 5 showed that during Datalog evaluation time, the difference in both runtime and memory usage is at most 2%. Note that the results differ from the previous section due to using an older version of DOOP, which was better supported by our top-*k* implementation. These results demonstrate that our provenance encoding scheme is at least as scalable as the state-of-the-art in terms of runtime performance. However, the main difference between the two approaches is that we are able to answer *any* provenance query during proof construction time, rather than only having the single proof tree matching the derivation tree pattern. Hence, our provenance evaluation strategy provides no runtime penalty for Datalog evaluation, while having a considerable advantage when exploring the provenance.

5.2 Proof Tree Construction

For the construction of proof trees, the performance of the provenance queries is instrumental. A debugging query constitutes a backward search for a rule (i.e., reverting the computational

Table 5. Runtime and Memory Usage Overheads for Our Provenance Approach Compared to Top-k [Deutch et al. 2018] Using Doop with the DaCapo Benchmarks

	Runtime (sec)			Memory (MB)		
Benchmark	Top-k	Prov.	(×)	Top-K	Prov.	(×)
antlr	19.09	19.02	0.99	1,502	1,502	1.00
bloat	12.46	12.74	1.02	901	902	1.00
chart	19.69	20.12	1.02	1,606	1,605	1.00
eclipse	12.46	12.46	1.00	900	899	1.00
fop	19.73	19.87	1.01	1,579	1,582	1.00
hsqldb	20.47	20.44	1.00	1,647	1,646	1.00
jython	14.11	14.25	1.01	1,092	1,092	1.00
luindex	12.48	12.48	1.00	907	907	1.00
lusearch	12.53	12.47	1.00	905	905	1.00
pmd	19.40	19.42	1.00	1,542	1,543	1.00
xalan	19.63	19.73	1.00	1,596	1,595	1.00
geo-mean			1.01			1.00



(a) Proof Tree Construction Time (b) Proof Tree Heights

Fig. 19. Proof tree construction and statistics.

direction of a rule). The construction of the proof tree is performed level by level. The expansion of a node in the proof tree represents a single debugging query.

In Figure 19(a), we show the time taken to construct proof tree fragments with heights up to 20. We initiate the proof tree construction for randomly sampled output tuples in the Doop DaCapo benchmarks. In the figure, we plot the runtime against the number of nodes in the proof tree fragment. Even for 20 levels, these proof trees contain over 15,000 nodes. Considering that full proof trees may have heights over 200, the corresponding full proof trees would be intractable to compute and understand due to exponential growth. However, this experiment shows that the construction of proof trees is approximately linear in the size of the tree. Therefore, provenance queries can be efficiently computed, and the method will scale well for interactive use. The interactive exploration of proof trees is scalable, with each debugging query on average taking less than 1 ms per node.

5.3 Characteristics of Proof Trees

In the following experiments, we demonstrate the difficulty of proof tree construction for Datalog specifications at large scale. Figure 19(b) shows the distribution of heights of full proof trees for

the DaCapo benchmarks. The proof tree heights can be more than 300. While this may not seem prohibitive, the expected number of nodes in the proof tree is exponential in height. A non-linear regression performed on the sizes of actual proof trees, suggests that the branching factor of proof trees is approximately 1.466 for the DaCapo benchmarks. Therefore, since larger proof trees will have an exponential number of nodes, it is computationally intractable to construct full proof trees for these large specifications. Besides, there is a usability challenge in generating meaningful explanations for the existence of a tuple, which is addressed by the interactive exploration of fragments of a proof tree, with the user exploring only relevant fragments. This is in contrast to a full proof tree, where a user may have to interpret millions of nodes to find an explanation.

6 RELATED WORK

Debugging for logic programming languages has a long history, with work having been done on algorithmic debugging strategies since the 1980s [Drabent and Nadjm-Tehrani 1989; Shapiro 1983]. These works present a framework for the algorithmic debugging method for Prolog programs, where a system asks the user questions about the intended model of the program to find buggy rules. However, they are based on the SLDNF resolution of Prolog that is not truly declarative, and thus the semantics of Datalog differ. Our method aligns practically with the interactive debugging frameworks presented here but applied to the bottom-up evaluation of Datalog and with sophisticated and efficient techniques to generate the debugging information.

Debugging and Provenance. Our method also fits into the established frameworks for provenance in Datalog [Cheney et al. 2009] and debugging for Datalog [Caballero et al. 2008]. The proof trees generated by our method are analogous to the computation graphs presented in [Caballero et al. 2008] and are equally effective for debugging. They can also be seen as an extension of *how*-provenance [Buneman et al. 2001]. Algorithmic debugging [Caballero et al. 2017; Silva 2007] also use notions of provenance (analogous to a *debugging tree* in their terminology) as part of their framework. In this setup, the debugger asks questions of the user, and guides the navigation of a debugging tree based on answers to these questions. *Thin slicing* [Sridharan et al. 2007] has similar ideas of using fragments of a computation graph to answer debugging queries. Our debugging strategy for Datalog fits into these frameworks and could be used as a basis for more sophisticated algorithmic debugging strategies. The main contribution of our work is the novel hybrid approach for generating provenance information.

Provenance in Datalog. Methods for computing provenance in Datalog has been a well-explored field [Arora et al. 1993; Caballero et al. 2015; Deutch et al. 2014, 2015; Köhler et al. 2012; Lee et al. 2017]; however, with the caveat that all these previous approaches store the full provenance information during the evaluation of Datalog. Köhler et al. [2012], for example, stores the whole computation graph as an auxiliary relation during Datalog evaluation, which may be many times larger than the IDB itself in large analysis use cases. Approaches such as in [Deutch et al. 2015] and [Lee et al. 2017] attempt to reduce the impact of provenance storage by only storing information relevant for a particular provenance query, which is given before the instrumentation and evaluation of Datalog. Thus, in these approaches, the Datalog specification needs to be re-evaluated for each different provenance query and therefore extends the investigation phase of the debugging cycle. The closest approach to ours is perhaps [Deutch et al. 2014], where a Boolean circuit representation for provenance is described, as well as an algorithm for generating such a Boolean circuit during Datalog evaluation. However, there is no mechanism for exploring an understandable provenance representation, and no practical implementation of this work. Therefore, our approach is novel by minimizing the storage overhead of provenance information, allowing interactive exploration of proof trees, all while providing an effective integration into an existing semi-naïve Datalog engine.

Other Applications for Datalog Provenance. Debugging Datalog specifications is not the only use case for provenance, with user-guided approaches [Mangal et al. 2015; Raghothaman et al. 2018; Zhang et al. 2014, 2017] for program analysis also relying on tracking the origins of data. In [Raghothaman et al. 2018], Zhang et al. [2017], and Mangal et al. [2015], a user may tag certain static analysis alarms, to increase or decrease their importance in the next analysis cycle. In [Zhang et al. 2014], the analysis system automatically generates an appropriate abstraction, by iteratively trying and refining failing abstractions. All these approaches rely on an annotation framework for Datalog: the user-guided systems require the user to add an annotation representing the importance of an alarm, and the abstraction refinement system requires the system to tag failing analyses with annotations. In any case, our provenance evaluation strategy would fit well into these systems, by providing an annotation framework at the Datalog engine level.

Provenance in Other Areas. Outside of Datalog, provenance has also been a focus of the database community, being useful for understanding the origins of data in large database systems. For instance, *Trio* [Benjelloun et al. 2006; Widom 2005] and *Perm* [Glavic and Alonso 2009; Glavic et al. 2013] are two such systems implementing provenance systems for relational databases. These systems focus on tracking the *lineage* of data in a database, rather than on debugging a query, and thus it is essential that the provenance information is stored directly alongside the data. It is important to note that in the context of databases, Datalog acts as a powerful query language rather then a specification logic. As a result, the Datalog specifications in these use cases typically consist of fewer rules [Liang et al. 2009], that do not exhibit complex patterns found in program analysis benchmarks [Bravenboer and Smaragdakis 2009]. Further study has also been undertaken in querying database provenance, with [Stamatogiannakis et al. 2015] and [Arab et al. 2017] both presenting mechanisms to construct provenance information lazily after the database query is run. Thus, similarly to our approach, these allow the querying of arbitrary parts of provenance information. However, both approaches are applied to database systems, with [Stamatogiannakis et al. 2015] reconstructing provenance information based on tracking file I/O and system calls during query evaluation and [Arab et al. 2017] based on system logs produced by the database system. Therefore, with the highly complex recursive nature of real-world Datalog specifications, similar information may blow up drastically, and experiments of both works only show scalability up to 10,000 tuples. Moreover, these approaches are unsuitable for our setting of in-memory analysis workloads.

Further use cases of provenance includes the incrementally updating Differential Dataflow framework [Chothia et al. 2016], where each dataflow operator is augmented to store a reverse mapping where it maintains the origins of each piece of data. This is analogous to maintaining a full computation graph, which we have aimed to improve on by using more lightweight provenance annotations.

Datalog Extended with Lattice Elements. The idea of extending Datalog with lattice elements supporting *subsumption* is not a new idea [Greco and Zaniolo 1998; Kießling and Güntzer 1994; Madsen et al. 2016]. In the past, these works have allowed the user to define a lattice ordering for tuples, providing subsumption as a way for a "better" tuple to replace "worse" ones. Such ideas allow Datalog to become suitable for greedy algorithms, such as *shortest path*, which is difficult to implement efficiently in standard Datalog without enumerating all possible paths. Our provenance evaluation strategy uses a specialized version of extension with lattice elements (which we call provenance annotations), designed for constructing minimum height proof trees. Furthermore, the utilization of the provenance annotations for proof tree construction is a further extension of the ideas of past work on subsumption.

7 CONCLUSION

In this article we have presented a novel provenance evaluation strategy for Datalog specifications. The provenance evaluation strategy extends tuples in the IDB with proof annotations. With the help of proof annotations, provenance queries can construct minimal proof trees incrementally. Our method has very small overheads at logic evaluation time in comparison to standard top-down evaluation or a naïve provenance methods that encode provenance information explicitly. Hence, our method enables debugging of large-scale logic specifications for the first time. We have implemented our provenance method in a high performance Datalog engine called Soufflé [Jordan et al. 2016], and demonstrated its feasibility through the Doop program analysis framework. We show that the runtime overheads of the provenance evaluation strategy are approximately 1.31×, and the memory overheads are 1.76×.

Our novel approach for generating provenance information minimizing the runtime overhead of Datalog evaluation points to a number of exciting directions for future research. First, while provenance is highly applicable for Datalog debugging, it may also be helpful to explain the results of a Datalog program. For example, integration into a program analysis framework such as Doop could help explain the bug alarms that are found, and improve its utility for program analysis users. Second, techniques from algorithmic debugging could be integrated to further improve the tool as a Datalog debugger, by asking queries of the user to inject additional domain knowledge when exploring the proof tree. Last, the utility of the debugger could be further extended by supporting other modes of failure, such as infinite loops or out of memory errors.

ACKNOWLEDGMENT

We thank Byron Cook and the ARG team at Amazon Web Services.

REFERENCES

2017. souffle-lang/souffle: Soufflé is a variant of Datalog for tool designers crafting analyses in Horn clauses. Soufflé synthesizes a native parallel C++ program from a logic specification. Retrieved October 19, 2017 from http://souffle-lang.github.io/.

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. 2015. *Staged Points-to Analysis for Large Code Bases.* Springer, Berlin, 131–150. DOI: https://doi.org/10.1007/978-3-662-46663-6_7

Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2017. Using reenactment to retroactively capture provenance for transactions. *IEEE Trans. Knowl. Data Eng.* 30, 3 (2017), 599–612.

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15).* ACM, New York, NY, 1371–1382. DOI: https://doi.org/10.1145/2723372.2742796

Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. 1993. Explaining program execution in deductive systems. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases* (1993). 101–119.

Omar Benjelloun, Anish Das Sarma, Chris Hayworth, and Jennifer Widom. 2006. An introduction to ULDBs and the trio system. *IEEE Data Eng. Bull.* 29 (2006), 5–16.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.* 44, 10 (2009), 243–262. DOI: https://doi.org/10.1145/1639949.1640108

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory* (2001). 316–330.

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2008. A theoretical framework for the declarative debugging of datalog programs. In *Proceedings of the International Workshop on Semantics in Data and Knowledge Bases.* Springer, 143–159.

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2015. Debugging of wrong and missing answers for Datalog programs with constraint handling rules. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming.* 55–66. DOI: https://doi.org/10.1145/2790449.2790522

Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A survey of algorithmic debugging. *ACM Comput. Surv.* 50, 4 (2017), 60.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in databases: Why, how, and where. *Found. Trends Databases* 1, 4 (2009), 379–474. DOI: https://doi.org/10.1561/1900000006

Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. Explaining outputs in modern data analytics. *Proc. VLDB Endow.* 9, 12 (2016), 1137–1148.

Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2015. Selective provenance for datalog programs using top-K queries. *Proc. VLDB Endow.* 8, 12 (2015), 1394–1405.

Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2018. Efficient provenance tracking for datalog using top-k queries. *VLDB J.* 27, 2 (01 Apr. 2018), 245–269. DOI: https://doi.org/10.1007/s00778-018-0496-7

Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for datalog provenance. In *ICDT*. 201–212. DOI: https://doi.org/10.5441/002/icdt.2014.22

Lodek Drabent and Simin Nadjm-Tehrani. 1989. Algorithmic debugging with assertions. In *Meta-programming in Logic Programming*. Citeseer.

Antonio Flores-Montoya and Eric Schulte. 2019. Datalog disassembly. *arXiv preprint arXiv:1906.03969* (2019).

Yoshihiko Futamura. 1999. Partial evaluation of computation process—An approach to a compiler-compiler. *High. Order Symbol. Comput.* 12, 4 (Dec. 1999), 381–391.

Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *IEEE Int. Conf. Data Eng.* 174–185.

Boris Glavic, Renée J. Miller, and Gustavo Alonso. 2013. Using SQL for efficient generation and querying of provenance information. In *Lecture Notes in Computer Science*, Volume 8000 (2013), 291–320.

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, declarative decompilation of smart contracts (unpublished).

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'18)*.

Sergio Greco and Cristian Molinaro. 2015. *Datalog and Logic Databases*. Morgan & Claypool.

Sergio Greco and Carlo Zaniolo. 1998. Greedy algorithms in datalog with choice and negation. In *Proceedings of the IJCSLP*. 294–309.

Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ—An efficient engine for fixed points with constraints. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, 457–462.

Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. ACM, 1213–1216. DOI: https://doi.org/10.1145/1989323.1989456

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Proc. Comput. Aid. Verif.* 422–430.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. ACM, New York, NY, 327–339. DOI: https://doi.org/10.1145/3293883.3295719

Werner Kießling and Ulrich Güntzer. 1994. Database reasoning - A deductive framework for solving large and complex problems by means of subsumption. In *Workshop on Information Systems and Artificial Intelligence*. Springer, 118–138.

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative datalog debugging for mere mortals. In *Lecture Notes in Computer Science*, Volume 7494 (2012), 111–122.

Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. 2017. Efficiently computing provenance graphs for queries with negation. *CoRR* abs/1701.05699 (2017). http://arxiv.org/abs/1701.05699.

Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2018. Provenance summaries for answers and non-answers. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1954–1957. DOI: https://doi.org/10.14778/3229863.3236233

Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. 2009. OpenRuleBench: An analysis of the performance of rule engines. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*. 601–610.

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, NY, 194–208. DOI: https://doi.org/10.1145/2908080.2908096

Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 462–473. DOI: https://doi.org/10.1145/2786805.2786851

Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. 2005. MulVAL: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium, Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, 8–8.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 722–735. DOI: https://doi.org/10.1145/3192366.3192417

Raghu Ramakrishnan and S. Sudarshan. 1991. Top-down vs. bottom-up revisited. In *Proceedings of the International Logic Programming Symposium*. MIT Press, 321–336.

Ehud Y. Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA.

Josep Silva. 2007. A comparative study of algorithmic debugging strategies. In *Logic-Based Program Synthesis and Transformation*, Germán Puebla (Ed.). Springer Berlin, 143–159.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 112–122.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 59–76. DOI: https://doi.org/10.1145/1094811.1094817

Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Decoupling provenance capture and analysis from execution. In *Proceedings of the 7th USENIX Workshop on the Theory and Practice of Provenance (TaPP'15)*. USENIX Association.

Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic index selection for large-scale datalog computation. *Proc. VLDB* 12, 2 (2018), 141–153.

Jeffrey D. Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*. ACM, New York, NY, 140–149. DOI: https://doi.org/10.1145/73721.73736

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. *Using Datalog with Binary Decision Diagrams for Program Analysis*. Springer, Berlin, 97–118. DOI: https://doi.org/10.1007/11575467_8

Jennifer Widom. 2005. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*. 262–276.

Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proc. ACM Program. Lang.* 1 (Oct. 2017), Article 57, 30 pages. DOI: https://doi.org/10.1145/3133881

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 239–248. DOI: https://doi.org/10.1145/2594291.2594327

David Zhao. 2017. *Large-Scale Provenance for Souffle*. University of Sydney.

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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010). 615–626.

Received March 2019; revised November 2019; accepted January 2020