Computing the Why-Provenance for Datalog Queries via SAT Solvers[*]

(Discussion Paper)

Marco Calautti[1,**], Ester Livshits[2], Andreas Pieris[2,3] and Markus Schneider[2]

[1]*Department of Computer Science, University of Milan*
[2]*School of Informatics, University of Edinburgh*
[3]*Department of Computer Science, University of Cyprus*

Abstract

Explaining an answer to a Datalog query is an essential task towards Explainable AI, especially nowadays where Datalog plays a critical role in the development of ontology-based applications. A well-established approach for explaining a query answer is the so-called why-provenance, which essentially collects all the subsets of the input database that can be used to obtain that answer via some derivation process, typically represented as a proof tree. It is well known, however, that computing the why-provenance for Datalog queries is computationally expensive, and thus, very few attempts can be found in the literature. The goal of this work is to demonstrate how off-the-shelf SAT solvers can be exploited towards an efficient computation of the why-provenance for Datalog queries. Interestingly, our SAT-based approach allows us to build the why-provenance in an incremental fashion, that is, one explanation at a time, which is much more useful in a practical context than the one-shot computation of the whole set of explanations as done by existing approaches.

Keywords

Datalog queries, explainability, why-provenance, computational complexity

1. Introduction

Datalog has emerged in the 1980s as a logic-based query language from Logic Programming and has been extensively studied since then [2]. The name Datalog reflects the intention of devising a counterpart of Prolog for data processing. It essentially extends the language of unions of conjunctive queries, which corresponds to the select-project-join-union fragment of relational algebra, with the important feature of recursion, much needed to express some natural queries.

As for any other query language, explaining why a result to a Datalog query is obtained is crucial towards transparent data-intensive applications. A well-established approach for providing such explanations to query answers is the so-called *why-provenance* [3]. Its essence is to collect all the subsets of the input database that as a whole can be used to derive a certain answer. More precisely, for Datalog queries, the why-provenance of an answer tuple $\bar{t}$ is obtained by considering all the possible proof trees T of the fact $\mathrm{Ans}(\bar{t})$, with Ans being the answer predicate of the Datalog query in question, and then collecting all the database facts that label the leaves of T. Recall that a proof tree of a fact α w.r.t. a database D and a set Σ of Datalog

SEBD 2024: 32nd Symposium on Advanced Database Systems, June 23-26, 2024, Villasimius, Sardinia, Italy

[*] This is a short version of the paper [1]

[**] Corresponding author.

marco.calautti@unimi.it (M. Calautti); ester.livshits@ed.ac.uk (E. Livshits); apieris@inf.ed.ac.uk (A. Pieris); m.schneider@ed.ac.uk (M. Schneider)

rules forms a tree-like representation of a way for deriving α by starting from D and executing the rules occurring in Σ [2].

Despite its wide acceptance, why-provenance for Datalog queries comes with two weaknesses: it is computationally very expensive, and it may provide counterintuitive explanations. The first weakness is manifested by the fact that, although why-provenance for Datalog queries has been around for decades, only a couple of works have considered implementing it for recursive queries [4, 5]. An attempt to change this state of affairs was made in [6] by focusing on the more practical setting of computing the why-provenance of a given query answer (a.k.a. *on-demand* why-provenance), instead of computing the why-provenance for all the query answers. Concerning the second weakness, it has been observed that there are proof trees that correspond to unnatural derivation processes, e.g., derivations where an atom is derived from itself [7]. Now, an explanation witnessed via such an unnatural proof tree, might be classified as a counterintuitive one as it does not correspond to an intuitive derivation process that can be extracted from the proof tree; this is further discussed in Section 3.

The main goal of this work is to tackle the two weaknesses of why-provenance for Datalog queries discussed above. In particular, we place our work in the more practical setting of on-demand why-provenance, and target an efficient implementation that provides conceptually meaningful explanations for the given query answer.

2. Preliminaries

We consider the disjoint countably infinite sets $\mathbf{C}$ and $\mathbf{V}$ of *constants* and *variables*, respectively. We may refer to constants and variables as *terms*. For brevity, given an integer $n > 0$, we may write $[n]$ for the set of integers $\{1, \ldots, n\}$.

Relational Databases. A *schema* $\mathbf{S}$ is a finite set of relation names (or predicates) with associated arity. We write R/n to say that R has arity $n \geq 0$; we may write $\mathsf{ar}(R)$ for n. A *(relational) atom* α over $\mathbf{S}$ is an expression of the form $R(\bar{t})$, where $R/n \in \mathbf{S}$ and $\bar{t}$ is an n-tuple of terms. By abuse of notation, we may treat tuples as the *set* of their elements. A *fact* is an atom that mentions only constants. A *database* over $\mathbf{S}$ is a finite set of facts over $\mathbf{S}$. The *active domain* of a database D, denoted $\mathsf{dom}(D)$, is the set of constants in D.

Syntax and Semantics of Datalog Programs. A *(Datalog) rule* σ over a schema $\mathbf{S}$ is an expression of the form

$$R_0(\bar{x}_0) \coloneq R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$$

for $n \geq 1$, where $R_i(\bar{x}_i)$ is a (constant-free) relational atom over $\mathbf{S}$ for $i \in \{0, \ldots, n\}$, and each variable in $\bar{x}_0$ occurs in $\bar{x}_k$ for some $k \in [n]$. We refer to $R_0(\bar{x}_0)$ as the *head* of σ, denoted $\mathsf{head}(\sigma)$, and to the expression that appears on the right of the :– symbol as the *body* of σ, denoted $\mathsf{body}(\sigma)$, which we may treat as the set of its atoms.

A *Datalog program* over a schema $\mathbf{S}$ is defined as a finite set Σ of Datalog rules over $\mathbf{S}$. A predicate R occurring in Σ is called *extensional* if there is no rule in Σ having R in its head, and *intensional* if there exists at least one rule in Σ with R in its head. The *extensional (database) schema* of Σ, denoted $\mathsf{edb}(\Sigma)$, is the set of all extensional predicates in Σ, while the *intensional schema* of Σ, denoted $\mathsf{idb}(\Sigma)$, is the set of all intensional predicates in Σ. Note that, by definition,

$\mathsf{edb}(\Sigma) \cap \mathsf{idb}(\Sigma) = \emptyset$. The *schema* of Σ, denoted $\mathsf{sch}(\Sigma)$, is the set $\mathsf{edb}(\Sigma) \cup \mathsf{idb}(\Sigma)$, which is in general a subset of $\mathbf{S}$ since some predicates of $\mathbf{S}$ may not appear in Σ.

An elegant property of Datalog programs is that they have three equivalent semantics: model-theoretic, fixpoint, and proof-theoretic [2]. We recall the proof-theoretic semantics of Datalog programs since it is closer to the notion of why-provenance. To this end, we need the key notion of proof tree of a fact. For a database D and a Datalog program Σ, let $\mathsf{base}(D, \Sigma) = \{R(\bar{t}) \mid R \in \mathsf{sch}(\Sigma) \text{ and } \bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}\}$, the facts that can be formed using predicates of $\mathsf{sch}(\Sigma)$ and constants of $\mathsf{dom}(D)$.

Definition 2.1. (**Proof Tree**) Consider a Datalog program Σ, a database D over $\mathsf{edb}(\Sigma)$, and a fact α over $\mathsf{sch}(\Sigma)$. A *proof tree of* α *w.r.t.* D *and* Σ is a finite labeled rooted tree $T = (V, E, \lambda)$, with $\lambda : V \to \mathsf{base}(D, \Sigma)$, such that:

1. If $v \in V$ is the root, then $\lambda(v) = \alpha$.
2. If $v \in V$ is a leaf, then $\lambda(v) \in D$.
3. If $v \in V$ is a node with $n \geq 1$ children $u_1, \ldots, u_n$, then there is a rule $R_0(\bar{x}_0) \text{ :- } R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n) \in \Sigma$ and a function $h : \bigcup_{i \in [n]} \bar{x}_i \to \mathbf{C}$ such that $\lambda(v) = R_0(h(\bar{x}_0))$, and $\lambda(u_i) = R_i(h(\bar{x}_i))$ for each $i \in [n]$. ■

Essentially, a proof tree of a fact α w.r.t. D and Σ indicates that we can derive α starting from D end executing the rules of Σ. Now, given a Datalog program Σ and a database D over $\mathsf{sch}(\Sigma)$, the *semantics of* Σ *on* D is $\Sigma(D) = \{\alpha \mid \text{ there is a proof tree of } \alpha \text{ w.r.t. } D \text{ and } \Sigma\}$. That is, the set of facts that can be proven using D and Σ.

Datalog Queries. It is now straightforward to recall the syntax and the semantics of Datalog queries. A *Datalog query* is a pair $Q = (\Sigma, R)$, where Σ is a Datalog program and R a predicate of $\mathsf{idb}(\Sigma)$. Now, for a database D over $\mathsf{edb}(\Sigma)$, the *answer* to Q over D is defined as the set

$$Q(D) \ = \ \left\{\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)} \mid R(\bar{t}) \in \Sigma(D)\right\},$$

that is, the set of tuples $\bar{t}$ of $\mathsf{dom}(D)^{\mathsf{ar}(R)}$ such that the fact $R(\bar{t})$ can be derived using D and Σ.

Why-Provenance for Datalog Queries. As discussed in the introduction, why-provenance is a standard way of explaining query results. It essentially collects all the subsets of the database (without unnecessary atoms) that allow to prove (or derive) a query result. We now formalize this simple idea.

Given a proof tree $T = (V, E, \lambda)$ (of some fact w.r.t. some database and Datalog program), the *support* of T is the set $\mathsf{support}(T) = \{\lambda(v) \mid v \in V \text{ is a leaf of } T\}$, which is essentially the set of facts that label the leaves of the proof tree T. Note that $\mathsf{support}(T)$ is a subset of the underlying database since, by definition, the leaves of a proof tree are labeled with database atoms. The formal definition of why-provenance for Datalog queries follows.

Definition 2.2. (**Why-Provenance for Datalog**) Consider a Datalog query $Q = (\Sigma, R)$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}$. The *why-provenance of* $\bar{t}$ *w.r.t.* D *and* Q is defined as the family of sets of facts

$$\mathsf{why}(\bar{t}, D, Q) = \{\mathsf{support}(T) \mid T \text{ is a proof tree of } R(\bar{t}) \text{ w.r.t. } D \text{ and } \Sigma\}. \quad ■$$

Intuitively speaking, a set of facts $D' \subseteq D$ that belongs to $\mathsf{why}(\bar{t}, D, Q)$ should be understood as a reason why the tuple $\bar{t}$ is an answer to the query Q over the database D, i.e., D' explains why $\bar{t} \in Q(D)$. In particular, *all* the facts of D' are used in order to derive $\bar{t}$ as an answer.

3. Unambiguous Proof Trees

The standard notion of why-provenance defined above relies on arbitrary proof trees. However, as discussed in [7], there are proof trees that are counterintuitive. For example, such a proof tree is one where a fact is derived from itself, that is, it contains two nodes labeled with the same fact and one is a descendant of the other. Now, a member of $\mathsf{why}(\bar{t}, D, Q)$, witnessed via such an unnatural proof tree, might be classified as a counterintuitive explanation of $\bar{t}$ as it does not correspond to a natural derivation process that can be extracted from the proof tree. Therefore, we need refined classes of proof trees that overcome the conceptual limitations of arbitrary proof trees. Some refined classes of proof trees have been recently discussed in [7]: *non-recursive proof trees*, *minimal-depth proof trees*, and *hereditary minimal-depth proof trees*. Roughly speaking, a non-recursive proof tree is a proof tree that does not contain two nodes labeled with the same fact and one is a descendant of the other, a minimal-depth proof tree is a proof tree that has the minimum depth among all the proof trees of the same fact, and a hereditary minimal-depth proof tree is minimizing the depth of each of its subtrees. Although non-recursive and (hereditary) minimal-depth proof trees form well-justified notions that deserve our attention, there are still proof trees from those classes that can be classified as counterintuitive. In fact, there are proof trees that are non-recursive and (hereditary) minimal-depth, but they are ambiguous in the way some facts are derived. Here is a simple example that illustrates this phenomenon.

Example 3.1. Consider the Datalog program Σ

$$\begin{array}{lcl} A(x) & \text{:-} & S(x) \\ A(x) & \text{:-} & A(y), A(z), T(y,z,x) \end{array}$$

that encodes the *path accessibility problem* [8]. The predicate S represents source nodes, A represents nodes that are accessible from the source nodes, and T represents accessibility conditions, that is, $T(y, z, x)$ means that if both y and z are accessible from the source nodes, then so is x. We further consider the database

$$D = \{S(a), S(b), T(a,a,c), T(b,b,c), T(c,c,d)\}.$$

The following is a proof tree of the fact $A(d)$ w.r.t. D and Σ that is non-recursive and (hereditary) minimal-depth, but it suffers from the ambiguity issue described above:

$$\begin{tikzcd}
 & & & & A(d) \arrow[dlll] \arrow[dr] \arrow[drrr] & & & \\
 & A(c) \arrow[dl] \arrow[d] \arrow[dr] & & & & A(c) \arrow[dl] \arrow[d] \arrow[dr] & & T(c,c,d) \\
A(a) \arrow[d] & A(a) \arrow[d] & T(a,a,c) & & A(b) \arrow[d] & A(b) \arrow[d] & T(b,b,c) & \\
S(a) & S(a) & & & S(b) & S(b) & &
\end{tikzcd}$$

Indeed, there are two nodes labeled with the fact $A(c)$, but their subtrees differ, and thus, it is ambiguous how $A(c)$ is derived. Hence, the database D, which belongs to the why-provenance of (d) w.r.t. D and Q due to the above proof tree, might be classified as a counterintuitive explanation since it does not correspond to an intuitive derivation process where each fact is derived once due to a unique reason. Indeed, the intuitive explanations that one expects are the following: d is accessible from a via c (i.e., the subset $\{S(a), T(a,a,c), T(c,c,d)\}$ of D), or d is accessible from b via c (i.e., the subset $\{S(b), T(b,b,c), T(c,c,d)\}$ of D). □

This leads to the class of unambiguous proof trees, where all occurrences of a fact must be proved via the same derivation. Two rooted trees $T = (V, E, \lambda)$ and $T' = (V', E', \lambda')$ are *isomorphic*, denoted $T \approx T'$, if there exists a bijection $h : V \rightarrow V'$ such that, for each $v \in V$, $\lambda(v) = \lambda'(h(v))$, and for each $u, v \in V$, $(u, v) \in E$ iff $(h(u), h(v)) \in E'$. Let $T[v]$ be the subtree of the proof tree T rooted at node v. The formal definition of unambiguous proof trees follows.

Definition 3.2. (Unambiguous Proof Tree) Consider a Datalog program Σ, a database D over $\mathsf{edb}(\Sigma)$, and a fact α over $\mathsf{sch}(\Sigma)$. An *unambiguous proof tree of* α *w.r.t.* D *and* Σ is a proof tree $T = (V, E, \lambda)$ of α w.r.t. D and Σ such that, for all $v, u \in V$, $\lambda(v) = \lambda(u)$ implies $T[v] \approx T[u]$. ■

Why-provenance relative to unambiguous proof trees is defined in the obvious way: for a Datalog query $Q = (\Sigma, R)$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}$, the *why-provenance of* $\bar{t}$ *w.r.t.* D *and* Q *relative to unambiguous proof trees* is the family of sets of facts

$$\{\mathsf{support}(T) \mid T \text{ is an unambiguous proof tree of } R(\bar{t}) \text{ w.r.t. } D \text{ and } \Sigma\},$$

denoted $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$. The main concern of this work is to efficiently compute the why-provenance of a tuple relative to unambiguous proof trees. To this end, we are going to exploit off-the-shelf SAT solvers and report encouraging results; this is the subject of the next two sections. To the best of our knowledge, this is the first time that SAT solvers are used for computing the why-provenance. Let us stress that focusing on unambiguous proof trees, apart from their conceptual advantage discussed above, was crucial towards our encouraging results as it is unclear how a SAT-based implementation can be made practical for arbitrary proof trees.

4. From Why-Provenance to SAT

In this section, we show that the why-provenance of a tuple relative to unambiguous proof trees can be extracted from the satisfying truth assignments of a Boolean formula.

Compactly Representing Unambiguous Proof Trees. The construction of the Boolean formula relies on a characterization of the existence of an unambiguous proof tree of a fact α w.r.t. D and Σ via the existence of a so-called *compressed directed acyclic graph (DAG)* of α w.r.t. D and Σ, which, intuitively speaking, is a compact representation of an unambiguous proof tree of α w.r.t. D and Σ; this is needed, as an unambiguous proof tree can be exponentially large in the size of D. We proceed to formalize the notion of compressed DAG and give the characterization in question. Recall that a DAG G is *rooted* if it has exactly one node, the *root*, with no incoming edges. A node of G is a *leaf* if it has no outgoing edges.

Definition 4.1. **(Compressed DAG)** Consider a Datalog program Σ, a database D over $\mathsf{edb}(\Sigma)$, and a fact α over $\mathsf{sch}(\Sigma)$. A *compressed DAG* of α w.r.t. D and Σ is a rooted DAG $G = (V, E)$, with $V \subseteq \mathsf{base}(D, \Sigma)$, such that:

1. The root of G is α.
2. If $\beta \in V$ is a leaf node, then $\beta \in D$.
3. If $\beta \in V$ has $n \geq 1$ outgoing edges $(\beta, \gamma_1), \ldots, (\beta, \gamma_n)$, then there is a rule $R_0(\bar{x}_0)$:- $R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m) \in \Sigma$ and a function $h : \bigcup_{i \in [m]} \bar{x}_i \rightarrow \mathbf{C}$ such that $\beta = R_0(h(\bar{x}_0))$ and $\{\gamma_i\}_{i \in [n]} = \{R_i(h(\bar{x}_i)) \mid i \in [m]\}$. ■

For a compressed DAG $G = (V, E)$, the *support* of G is defined analogously to the support of a proof tree, that is, $\mathsf{support}(G) = \{v \in V \mid v \text{ is a leaf of } G\}$. The desired characterization follows.

Proposition 4.2. *For a Datalog program Σ, a database D over* $\mathsf{edb}(\Sigma)$*, a fact α over* $\mathsf{sch}(\Sigma)$*, and a database $D' \subseteq D$, the following are equivalent:*

1. *There exists an unambiguous proof tree T of α w.r.t. D and Σ such that* $\mathsf{support}(T) = D'$.
2. *There exists a compressed DAG G of α w.r.t. D and Σ such that* $\mathsf{support}(G) = D'$.

Note that the above characterization relies on the fact that we focus on unambiguous proof trees. More precisely, unambiguity allows us to use a single node in the compressed DAG as a representative for all the (possibly exponentially many) nodes in the proof tree labelled with the same fact.

The Boolean Formula. Fix a Datalog query $Q = (\Sigma, R)$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}$. We construct in polynomial time in D a Boolean formula $\phi_{(\bar{t},D,Q)}$ such that the why-provenance of $\bar{t}$ w.r.t. D and Q relative to unambiguous proof trees can be computed from the truth assignments that make $\phi_{(\bar{t},D,Q)}$ true.

Among the Boolean variables of $\phi_{(\bar{t},D,Q)}$, we also have the disjoint sets of variables V_N and V_E, where each variable in V_N corresponds to a possible node of some compressed DAG of $R(\bar{t})$ w.r.t. D and Σ, while each variable in V_E corresponds to a possible edge between two nodes of some compressed DAG of $R(\bar{t})$ w.r.t. D and Σ. The key idea is that the variables of V_N and V_E that become true via a satisfying truth assignment of $\phi_{(\bar{t},D,Q)}$, induce the nodes and the edges of a compressed DAG G for $R(\bar{t})$ w.r.t. D and Σ, which, by Proposition 4.2, implies that $\mathsf{support}(G) \in \mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$. The Boolean formula $\phi_{(\bar{t},D,Q)}$ is a conjunction of the form

$$\phi_{graph} \ \wedge \ \phi_{root} \ \wedge \ \phi_{proof} \ \wedge \ \phi_{acyclic},$$

where ϕ_{graph} is in charge of guaranteeing consistency between the truth assignments of the variables in V_N and the variables in V_E, i.e., if an edge between two nodes is part of G, then the two nodes must belong to G as well. The formula ϕ_{root} guarantees that the atom $R(\bar{t})$ is a node of G, is the root of G, and no other node v of G can be the root (i.e., v must have at least one incoming edge). The formula ϕ_{proof} is in charge of guaranteeing that, whenever an intensional atom α is a node of G, then it must have the correct children in G. Finally, $\phi_{acyclic}$ is in charge of checking that G, namely the graph whose edges correspond to the true variables in V_E, is acyclic. We can prove the following crucial result about the above Boolean formula:

Scenario	Databases	Recursive	#Rules
TClosure [9, 10]	D_{bitcoin}(235K), D_{facebook}(88.2K)	✓	2
Doctors [6]	D_1(100K)	✗	6
Galen [6]	D_1(26.5K), D_2(30.5K), D_3(67K), D_4(82K)	✓	14
Andersen [11]	D_1(68K), D_2(340K), D_3(680K), D_4(3.4M), D_5(6.8M)	✓	2
CSDA [11]	D_{httpd}(10M), $D_{\text{postgresql}}$(34.8M), D_{linux}(44M)	✓	2

Table 1
Experimental scenarios. For each database D, the number of tuples occurring in D is in parenthesis.

Proposition 4.3. *Consider a Datalog query $Q = (\Sigma, R)$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}$. It holds that:*

- *$\phi_{(\bar{t},D,Q)}$ is in CNF and can be constructed in polynomial time w.r.t. D.*
- *There is a one-to-one correspondence between the members of $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$ and the satisfying assignments of $\phi_{(\bar{t},D,Q)}$.*
- *Each member of $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$ can be recovered from a satisfying assignment of $\phi_{(\bar{t},D,Q)}$ in polynomial time w.r.t. the size of $\phi_{(\bar{t},D,Q)}$.*

5. Implementation and Experimental Evaluation

Proposition 4.3 provides a way for computing the why-provenance of a tuple relative to unambiguous proof trees via off-the-shelf SAT solvers. But how does this machinery behave when applied in a practical context? In particular, we are interested in the incremental computation of the why-provenance by enumerating its members, which is more useful in practice than computing the whole set at once.

This is achieved by adapting a standard technique from the SAT literature for enumerating the satisfying assignments of a Boolean formula, called *blocking clause*. After asking the SAT solver for an arbitrary satisfying assignment τ of $\phi_{(\bar{t},D,Q)}$, we output the member of $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$, denoted $\mathsf{db}(\tau)$, corresponding to τ, and then construct a "blocking" clause $C_{\mathsf{db}(\tau)}$ which expresses that no other satisfying assignment τ' should give rise to the same member of the why-provenance. This will exclude the previously computed explanations from the computation. We keep adding such blocking clauses each time we get a new member of the why-provenance until the formula is unsatisfiable.

We now proceed to experimentally evaluate the SAT-based approach discussed above. To this end, we consider a variety of scenarios from the Datalog literature consisting of a query $Q = (\Sigma, R)$ and a family of databases $\mathcal{D}$ over $\mathsf{edb}(\Sigma)$. All the scenarios are summarized in Table 1.

Experimental Setup. For each scenario s consisting of the query $Q = (\Sigma, R)$ and the family of databases $\mathcal{D}$, and for each $D \in \mathcal{D}$, we have computed $Q(D)$ using DLV [12], version 2.1.1, and selected 100 tuples $\bar{t}^1_{s,D}, \ldots, \bar{t}^{100}_{s,D}$ from $Q(D)$ uniformly at random. Then, for each $i \in [100]$, we constructed the Boolean formula $\phi_{(\bar{t}^i_{s,D},D,Q)}$ via a C++ implementation. Finally, we ran the state-of-the-art SAT solver Glucose [13], version 4.2.1, with the above formula as input, to enumerate the members of $\mathsf{why}_{\mathsf{UN}}(\bar{t}^i_{s,D}, D, Q)$. All the experiments have been conducted on a laptop with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, and 32GB of RAM, running Fedora Linux 37.

Experimental Results and Take-Home Messages. Concerning the construction of the formula, in most of the scenarios, the runtime is in the order of some seconds, and we have observed that almost all the time is spent for computing the intermediate data structures, needed to build the formula. Considering the more demanding scenarios Andersen and CSDA, the total time is in the order of some minutes. Note, however, that it would be similarly demanding even for query answering.

For the incremental computation of the why-provenance, we considered the time between the current member of the why-provenance and the next (the delay). All delays are in the order of milliseconds. Hence, once the formula is built, incrementally computing the members of the why-provenance is extremely fast.

We also compared our SAT-based approach with the one of [6], which is the only one in the literature for constructing the why-provenance of a given tuple. We point out that for a query Q, a database D, and a tuple $\bar{t} \in Q(D)$, the technique of [6], which we call *rule-based*, constructs the set $\mathsf{why}(\bar{t}, D, Q)$ via a rewriting of Q into a set of existential rules. Since $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q) \subseteq \mathsf{why}(\bar{t}, D, Q)$, one may think that computing $\mathsf{why}(\bar{t}, D, Q)$ is more demanding. However, computing $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$ requires checking for unambiguity. We use VLog [14], version 0.9.0, for the rule-based implementation, and we perform the comparison over the Galen and Doctors-based scenarios as these are the only ones considered in [6], and thus, we have access to the rewritten set of existential rules. For the comparison, since the approach of [6] does not support incremental computation, we considered the *end-to-end runtime* for constructing $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$ and $\mathsf{why}(\bar{t}, D, Q)$ using the SAT-based and the rule-based implementation, respectively; we set a 5 minutes timeout for both approaches.

We observed that our SAT-based implementation consistently outperforms the rule-based one. In particular, for the Galen scenario, in most cases, the rule-based implementation does not finish in less than 5 minutes, with the worst case occurring with the largest database, where 41 out of the 100 runs time out.

6. Future Steps

From our analysis, it is clear that our future efforts should focus on improving the construction of the Boolean formula. Since proof trees describe a *finite* reasoning process, it will be interesting to understand how they can be adapted to other rule-based formalisms with finite reasoning, in order to apply our SAT-based approach, such as logic programs with finite stable models [15, 16, 17], and ontology-mediated queries that guarantee the termination of the chase (e.g., see [18, 19, 20, 21, 22, 23]). Finally, it would be interesting to see how to adapt our approach to explain answers obtained over uncertain data, such as inconsistent databases (e.g., see [24, 25, 26, 27, 28]).

Acknowledgments

This work was funded by the European Union - Next Generation EU under the MUR PRIN-PNRR grant P2022KHTX7 “DISTORT”, and by the EPSRC grant EP/S003800/1 “EQUID”.

References

[1] M. Calautti, E. Livshits, A. Pieris, M. Schneider, Computing the why-provenance for datalog queries via SAT solvers, in: AAAI, 2024, pp. 10459–10466.

[2] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[3] P. Buneman, S. Khanna, W. C. Tan, Why and where: A characterization of data provenance, in: ICDT, 2001, pp. 316–330.

[4] D. Zhao, P. Subotic, B. Scholz, Debugging large-scale datalog: A scalable provenance evaluation strategy, ACM Trans. Program. Lang. Syst. 42 (2020) 7:1–7:35.

[5] J. Esparza, M. Luttenberger, M. Schlund, Fpsolve: A generic solver for fixpoint equations over semirings, in: CIAA, 2014, pp. 1–15.

[6] A. Elhalawati, M. Krötzsch, S. Mennicke, An existential rule framework for computing why-provenance on-demand for datalog, in: RuleML+RR, 2022.

[7] C. Bourgaux, P. Bourhis, L. Peterfreund, M. Thomazo, Revisiting semiring provenance for datalog, in: KR, 2022.

[8] S. A. Cook, An observation on time-storage trade off, J. Comput. Syst. Sci. 9 (1974) 308–316.

[9] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, C. E. Leiserson, Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics, CoRR abs/1908.02591 (2019). URL: http://arxiv.org/abs/1908.02591.

[10] J. McAuley, J. Leskovec, Learning to discover social circles in ego networks, in: NIPS, 2012, p. 539–547.

[11] Z. Fan, S. Mallireddy, P. Koutris, Towards better understanding of the performance and design of datalog systems, in: Datalog 2.0, 2022, pp. 166–180.

[12] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (2006) 499–562.

[13] G. Audemard, L. Simon, On the glucose SAT solver, Int. J. Artif. Intell. Tools 27 (2018) 1840001:1–1840001:25.

[14] J. Urbani, C. Jacobs, M. Krötzsch, Column-oriented datalog materialization for large knowledge graphs, in: AAAI, 2016, pp. 258–264.

[15] M. Calautti, S. Greco, F. Spezzano, I. Trubitsyna, Checking termination of bottom-up evaluation of logic programs with function symbols, TPLP 15 (2015) 854–889.

[16] M. Calautti, S. Greco, I. Trubitsyna, Detecting decidable classes of finitely ground logic programs with function symbols, in: PPDP, 2013, pp. 239–250.

[17] M. Calautti, S. Greco, C. Molinaro, I. Trubitsyna, Logic program termination analysis using atom sizes, in: IJCAI, 2015, pp. 2833–2839.

[18] T. Gogacz, J. Marcinkowski, A. Pieris, Uniform restricted chase termination, SIAM J. Comput. 52 (2023) 641–683.

[19] M. Calautti, A. Pieris, Semi-oblivious chase termination: The sticky case, ToCS 65 (2021) 84–121.

[20] M. Calautti, G. Gottlob, A. Pieris, Non-uniformly terminating chase: Size and complexity, in: PODS, 2022, pp. 369–378.

[21] M. Krötzsch, M. Marx, S. Rudolph, The power of the terminating chase (invited talk), in: ICDT, volume 127, 2019, pp. 3:1–3:17.

[22] M. Calautti, M. Milani, A. Pieris, Semi-oblivious chase termination for linear existential

rules: An experimental study, VLDB 16 (2023) 2858–2870.
[23] M. Calautti, G. Gottlob, A. Pieris, Chase termination for guarded existential rules, in: PODS, 2015, pp. 91–103.
[24] M. Arenas, L. E. Bertossi, J. Chomicki, Consistent query answers in inconsistent databases, in: PODS, 1999, pp. 68–79.
[25] M. Calautti, M. Console, A. Pieris, Counting database repairs under primary keys revisited, in: PODS, 2019, pp. 104–118.
[26] M. Calautti, S. Greco, C. Molinaro, I. Trubitsyna, Preference-based inconsistency-tolerant query answering under existential rules, AI 312 (2022) 103772.
[27] M. Calautti, L. Caroprese, S. Greco, C. Molinaro, I. Trubitsyna, E. Zumpano, Existential active integrity constraints, ESWA 168 (2021) 114297.
[28] M. Calautti, M. Console, A. Pieris, Benchmarking approximate consistent query answering, in: PODS, 2021, pp. 233–246.