Computing the Why-Provenance for Datalog Queries via SAT Solvers

Marco Calautti[1], Ester Livshits[2], Andreas Pieris[2,3], 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
marco.calautti@unimi.it, ester.livshits@ed.ac.uk, apieris@inf.ed.ac.uk, m.schneider@ed.ac.uk

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.

1 Introduction

Datalog has emerged in the 1980s as a logic-based query language from Logic Programming and has been extensively studied since then (Abiteboul, Hull, and Vianu 1995). 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. Among numerous applications, Datalog has been heavily used in the context of ontological query answering. In particular, for several important ontology languages based on description logics and existential rules, ontological query answering can be reduced to the problem of evaluating a Datalog query (see, e.g., (Eiter et al. 2012; Benedikt et al. 2022)), which in turn enables the exploitation of efficient Datalog engines such as Soufflé (Jordan, Scholz, and Subotic 2016), VLog (Urbani, Jacobs, and Krötzsch 2016), RDFox (Nenov et al. 2015), and DLV (Leone et al. 2006), to name a few.

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* (Buneman, Khanna, and Tan 2001). 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 rules forms a tree-like representation of a way for deriving α by starting from D and executing the rules occurring in Σ (Abiteboul, Hull, and Vianu 1995).

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 (Zhao, Subotic, and Scholz 2020; Esparza, Luttenberger, and Schlund 2014). An attempt to change this state of affairs was made in (Elhalawati, Krötzsch, and Mennicke 2022) 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 (Bourgaux et al. 2022). 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. To this end, we introduce the novel class of *unambiguous* proof trees that correspond to conceptually meaningful derivation processes, unlike arbitrary proof trees, which in turn ensures the concep-

tual relevance of the obtained explanations. We then show that unambiguous proof trees, besides their conceptual advantage, also help to exploit off-the-shelf SAT solvers towards an efficient computation of the why-provenance for Datalog queries. Moreover, our SAT-based approach allows us to build the why-provenance of a query answer in an incremental fashion, i.e., one explanation at a time, which is much more useful in practice than the one-shot computation of the whole set as done in (Elhalawati, Krötzsch, and Mennicke 2022). Finally, we experimentally confirm the efficiency of our approach with a dedicated benchmark, and we further show that, in general, it outperforms the machinery proposed in (Elhalawati, Krötzsch, and Mennicke 2022).

The experimental scenarios and the source code are given at https://gitlab.com/aaai24whyprov/datalog-why-provenance.

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

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

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 (Abiteboul, Hull, and Vianu 1995). 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*, denoted $\Sigma(D)$, is

$$\{\alpha \mid \text{ there is a proof tree of } \alpha \text{ w.r.t. } D \text{ and } \Sigma\},$$

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

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

which we denote by $\mathsf{why}(\bar{t}, D, Q)$. ∎

Intuitively speaking, a set of facts $D' \subseteq D$ that belongs to $\mathsf{why}(\bar{t}, D, Q)$ should be understood as a "real" reason why the tuple $\bar{t}$ is an answer to the query Q over the database D, i.e., D' explains why $\bar{t} \in Q(D)$. By "real" we mean that 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 (Bourgaux et al. 2022), 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 (Bourgaux et al. 2022): *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.

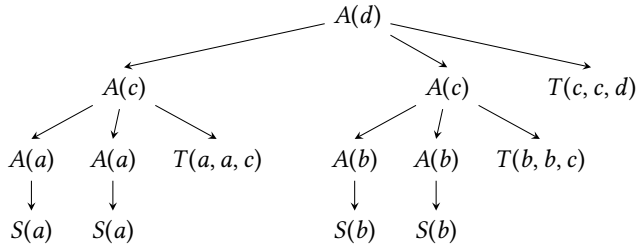
Example 3.1. Consider the Datalog program Σ

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

that encodes the *path accessibility problem* (Cook 1974). 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:



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 $h : V \to V'$ 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 ambiguous 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. 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. The definition of compressed DAG follows.

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 $R_0(\bar{x}_0) \coloneq R_1(\bar{x}_1), \ldots, R_m(\bar{x}_m) \in \Sigma$ and a function $h : \bigcup_{i \in [m]} \bar{x}_i \to \mathbf{C}$ 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.

Constructing the Boolean Formula

We start by introducing the so-called *downward closure* of a fact w.r.t. a database and a Datalog program, taken from (Elhalawati, Krötzsch, and Mennicke 2022), that will play a key role in the construction of the Boolean formula.

Downward Closure. Roughly, the downward closure of a fact α w.r.t. a database D and a Datalog program Σ is a hypergraph that encodes all possible compressed DAGs of α w.r.t. D and Σ. For our purposes, a *(directed) hypergraph* $\mathcal{H}$ is a pair (V, E), where V is the set of its *nodes* and E is the set of its *hyperedges*, i.e., pairs of the form (u, T), where $u \in V$ and T is a non-empty subset of V. For $u, v \in V$, we say that u *reaches* v *in* $\mathcal{H}$ if either $u = v$, or there exists a sequence of hyperedges $(u_1, T_1), \ldots, (u_n, T_n)$ with $u_1 = u$, $v \in T_n$, and $u_i \in T_{i-1}$ for each $i \in \{2, \ldots, n\}$. For a node u, we write $\mathcal{H}_{\downarrow u}$ for the hypergraph (V', E') obtained from $\mathcal{H}$ where V' contains u and all nodes reachable from u, and E' collects all the hyperedges mentioning only nodes of V'.

For a Datalog program Σ and a database D over $\mathsf{edb}(\Sigma)$, the *graph of rule instances (GRI)* of D and Σ is the hypergraph $\mathsf{gri}(D, \Sigma) = (V, E)$, with $V \subseteq \mathsf{base}(D, \Sigma)$, such that

1. $D \subseteq V$, and
2. if there exists a rule $R_0(\bar{x}_0) \coloneq R_1(\bar{x}_1), \ldots, R_n(\bar{x}_n)$ in Σ and a function $h : \bigcup_{i \in [n]} \bar{x}_i \to \mathbf{C}$ such that $\alpha_i = R_i(h(\bar{x}_i)) \in V$, for $i \in [n]$, then $\alpha_0 = R_0(h(\bar{x}_0)) \in V$ and $(\alpha_0, \{\alpha_1, \ldots, \alpha_n\}) \in E$.

Intuitively, $\mathsf{gri}(D, \Sigma)$ encodes all compressed DAGs of all facts α w.r.t. D and Σ. Since we are interested in finding only compressed DAGs of a specific fact α, we do not need to consider $\mathsf{gri}(D, \Sigma)$ in its entirety, but only the subhypergraph of $\mathsf{gri}(D, \Sigma)$ containing α and all nodes reachable from it. This leads to the notion of downward closure.

Definition 4.3. (**Downward Closure**) Consider a Datalog program Σ, a database D over $\mathsf{edb}(\Sigma)$, and a fact α occurring in $\mathsf{gri}(D, \Sigma)$. The *downward closure of α w.r.t. D and Σ*, denoted $\mathsf{down}(\alpha, D, \Sigma)$, is the hypergraph $\mathsf{gri}(D, \Sigma)_{\downarrow \alpha}$. ∎

The downward closure simply keeps from $\mathsf{gri}(D, \Sigma)$ the part that is relevant to derive the fact in question. The key property of the downward closure, which is easy to verify, is summarized by the following technical lemma.

Lemma 4.4. *Consider a Datalog program Σ, a database D over $\mathsf{edb}(\Sigma)$, a fact α over $\mathsf{sch}(\Sigma)$, and a compressed DAG $G = (V, E)$ of α w.r.t. D and Σ. Then, for every node $\beta \in V$ with outgoing edges $(\beta, \gamma_1), \ldots, (\beta, \gamma_n)$ in G, we have that $(\beta, \{\gamma_1, \ldots, \gamma_n\})$ is a hyperedge of $\mathsf{down}(\alpha, D, \Sigma)$.*

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. In what follows, assume that $\mathsf{down}(R(\bar{t}), D, Q) = (V, E)$.

The set of variables of $\phi_{(\bar{t}, D, Q)}$ is the union of three disjoint sets V_N, V_H, and V_E of variables. Each variable in V_N corresponds to a node of $\mathsf{down}(R(\bar{t}), D, Q)$, i.e., $V_N = \{x_\alpha \mid \alpha \in V\}$, each variable in V_H corresponds to a hyperedge of $\mathsf{down}(R(\bar{t}), D, Q)$, i.e., $V_H = \{y_e \mid e \in E\}$, and each variable in V_E corresponds to a "standard edge" that can be extracted from a hyperedge of $\mathsf{down}(R(\bar{t}), D, Q)$, i.e., $V_E = \{z_{(\alpha, \beta)} \mid (\alpha, T) \in E \text{ with } \beta \in T\}$.

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 Q, 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

$$\phi_{graph} \;\wedge\; \phi_{root} \;\wedge\; \phi_{proof} \;\wedge\; \phi_{acyclic}.$$

We proceed to discuss each of the subformulas. In the following, we use $A \Rightarrow B$ as a shorthand for $\neg A \vee B$.

The formula 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:

$$\phi_{graph} \;=\; \bigwedge_{z_{(\alpha, \beta)} \in V_E} (z_{(\alpha, \beta)} \Rightarrow x_\alpha) \wedge (z_{(\alpha, \beta)} \Rightarrow x_\beta).$$

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):

$$\phi_{root} \;=\; x_{R(\bar{t})} \;\wedge\; \left(\bigwedge_{z_{(\alpha,R(\bar{t}))} \in V_E} \neg z_{(\alpha,R(\bar{t}))} \right) \;\wedge \bigwedge_{\substack{x_\alpha \in V_N \\ \text{with } \alpha \neq R(\bar{t})}} \left(x_\alpha \Rightarrow \bigvee_{z_{(\beta,\alpha)} \in V_E} z_{(\beta,\alpha)} \right).$$

We now discuss ϕ_{proof}. Roughly, this formula 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, i.e., its children are the ones coming from some hyperedge of $\mathsf{down}(R(\bar{t}), D, Q)$ and no other nodes are its children (this is needed to guarantee that G is a *compressed* DAG). This is achieved with two subformulas. The first part is in charge of choosing some hyperedge (α, T) of $\mathsf{down}(R(\bar{t}), D, Q)$, for each intensional atom α, while the second guarantees that for each selected hyperedge (α, T) (one per intensional atom α), *all and only* the nodes in T are children of α in G:

$$\phi_{proof} \;=\; \bigwedge_{\substack{x_\alpha \in V_N \text{ with} \\ \alpha \text{ intensional}}} \left(x_\alpha \Rightarrow \bigvee_{y_{(\alpha,T)} \in V_H} y_{(\alpha,T)} \right) \wedge \bigwedge_{\substack{y_e \in V_H \\ \text{with } e=(\alpha,T)}} \left(\bigwedge_{z_{(\alpha,\beta)} \in V_E} y_e \Rightarrow \ell_{e,\beta} \right),$$

where, for a hyperedge $e = (\alpha, T)$, $\ell_{e,\beta}$ denotes $z_{(\alpha,\beta)}$ if $\beta \in T$, and $\neg z_{(\alpha,\beta)}$ otherwise. Let us clarify that, although we are interested in choosing *exactly one* hyperedge (α, T) for each intensional atom α, the above formula uses a disjunction rather than an exclusive or. This is not a problem since a truth assignment that makes two variables $y_{(\alpha,T_1)}$ and $y_{(\alpha,T_2)}$ true is not a satisfying assignment. This is because the second conjunct of ϕ_{proof}, e.g., requires that the variables $z_{(\alpha,\beta)}$ with $\beta \in T_1$ are true, while all others must be false. Hence, since $T_1 \neq T_2$, when considering the hyperedge (α, T_2), this subformula will not be satisfied.

We finally discuss $\phi_{acyclic}$, which is in charge of checking that G, namely the graph whose edges correspond to the true variables in V_E, is acyclic. Checking the acyclicity of a graph via a Boolean formula is a well-studied problem in the SAT literature with a plethora of encodings. For our construction, we employ an encoding based on *vertex elimination* (Rankooh and Rintanen 2022).

This completes the construction of our Boolean formula, which is in CNF, and we can prove it can be constructed in polynomial time in D; the latter relies on the fact that the hypergraph $\mathsf{down}(R(\bar{t}), D, \Sigma)$ can be constructed in polynomial time in D. Now, a truth assignment τ to the variables of $\phi_{(\bar{t},D,Q)}$ naturally gives rise to a database denoted $\mathsf{db}(\tau)$. Formally, for a truth assignment τ from the variables of $\phi_{(\bar{t},D,Q)}$ to $\{0, 1\}$, we write $\mathsf{db}(\tau)$ for the database $\{\alpha \in D \mid x_\alpha \in V_N \text{ and } \tau(x_\alpha) = 1\}$, i.e., the database collecting all facts in D having a corresponding variable in $\phi_{(\bar{t},D,Q)}$ that is true w.r.t. τ. Let $[\![\phi_{(\bar{t},D,Q)}]\!]$ be the family

$$\left\{ \mathsf{db}(\tau) \mid \tau \text{ is a satisfying assignment of } \phi_{(\bar{t},D,Q)} \right\}.$$

We can then show the correctness of our formula:

Proposition 4.5. *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 $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q) = [\![\phi_{(\bar{t},D,Q)}]\!]$.*

5 Why-Provenance via SAT Solvers

Proposition 4.5 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. The goal of this section is to provide an answer to this question.

Implementation Details

Before presenting our experimental results, let us first briefly discuss some interesting aspects of the implementation. In what follows, fix a Datalog query $Q = (\Sigma, R)$, a database D over $\mathsf{edb}(\Sigma)$, and a tuple $\bar{t} \in \mathsf{dom}(D)^{\mathsf{ar}(R)}$.

Constructing the Downward Closure. Recall that the construction of $\phi_{(\bar{t},D,Q)}$ relies on the downward closure of $R(\bar{t})$ w.r.t. D and Σ. It turns out that the hyperedges of the downward closure can be computed by executing a slightly modified Datalog query $Q_\downarrow$ over a slightly modified database $D_\downarrow$. In other words, the answers to $Q_\downarrow$ over $D_\downarrow$ coincide with the hyperedges of the downward closure. To construct the downward closure we can exploit a state-of-the-art Datalog engine, that is, version 2.1.1 of DLV (Adrian et al. 2018). Note that our approach differs form the one in (Elhalawati, Krötzsch, and Mennicke 2022), which uses existential rules. The advantage of using a pure Datalog query to build the downward closure is that we can use the same Datalog engine to both answer and explain queries.

Constructing the Formula. Recall that $\phi_{(\bar{t},D,Q)}$ is expressed as a conjunction of four formulas, where each formula is responsible for a certain task. As it might be expected, the heavy task is to verify that the graph in question is acyclic (performed by the formula $\phi_{acyclic}$). Checking the acyclicity of a directed graph via a Boolean formula is a well-studied problem in the SAT literature. For our implementation, we employ the technique of *vertex elimination* (Rankooh and Rintanen 2022). The advantage of this approach is that the number of Boolean variables needed to encode $\phi_{acyclic}$ is generally orders of magnitude smaller than other standard encodings, such as plain transitive closure. In paricular, the number of variables is $O(n \cdot \delta)$, where n is the number of nodes of the graph, and δ is the so-called *elimination width* of the graph, which, intuitively speaking, is related to how connected the graph is.

Incrementally Constructing the Why-Provenance. Recall that we are interested in the incremental computation of the why-provenance, which is more useful in practice than computing the whole set at once. To this end, we need a way

Scenario	Databases	#Rules
TClosure	D_{bitcoin}(235K), D_{facebook}(88.2K)	2
Doctors-i, $i \in [7]$	D_1(100K)	6
Galen	D_1(26.5K), D_2(30.5K), D_3(67K), D_4(82K)	14
Andersen	D_1(68K), D_2(340K), D_3(680K), D_4(3.4M), D_5(6.8M)	2
CSDA	D_{httpd}(10M), $D_{\mathsf{postgresql}}$(34.8M), D_{linux}(44M)	2

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

to enumerate all the members of the why-provenance without repetitions. This is achieved by adapting a standard technique from the SAT literature for enumerating the satisfying assignments of a Boolean formula, called *blocking clause*. We initially collect in a set S all the facts of D occurring in the downward closure of $R(\bar{t})$ w.r.t. D and Σ. Then, after asking the SAT solver for an arbitrary satisfying assignment τ of $\phi_{(\bar{t},D,Q)}$, we output the database $\mathsf{db}(\tau)$, and then construct the "blocking" clause $\vee_{\alpha \in S} \ell_\alpha$, where $\ell_\alpha = \neg x_\alpha$ if $\alpha \in \mathsf{db}(\tau)$, and $\ell_\alpha = x_\alpha$ otherwise. We then add this clause to the formula, 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.

Experimental Evaluation

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 considered 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 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 downward closure of $R(\bar{t}^i_{s,D})$ w.r.t. D and Σ by first computing the adapted query $Q_\downarrow$ and database $D_\downarrow$ via a Python 3 implementation and then using DLV for the actual computation of the downward closure. Then, 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 (see, e.g., (Audemard and Simon 2018)), 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. We used Python 3.11.2, and the C++ code has been compiled with g++ 12.2.1, using the -O3 optimization flag.

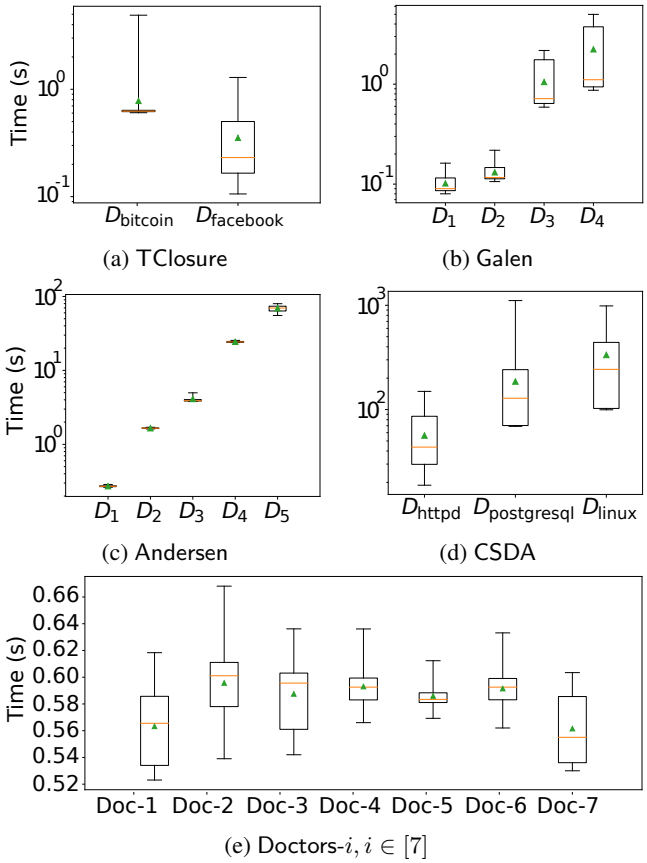
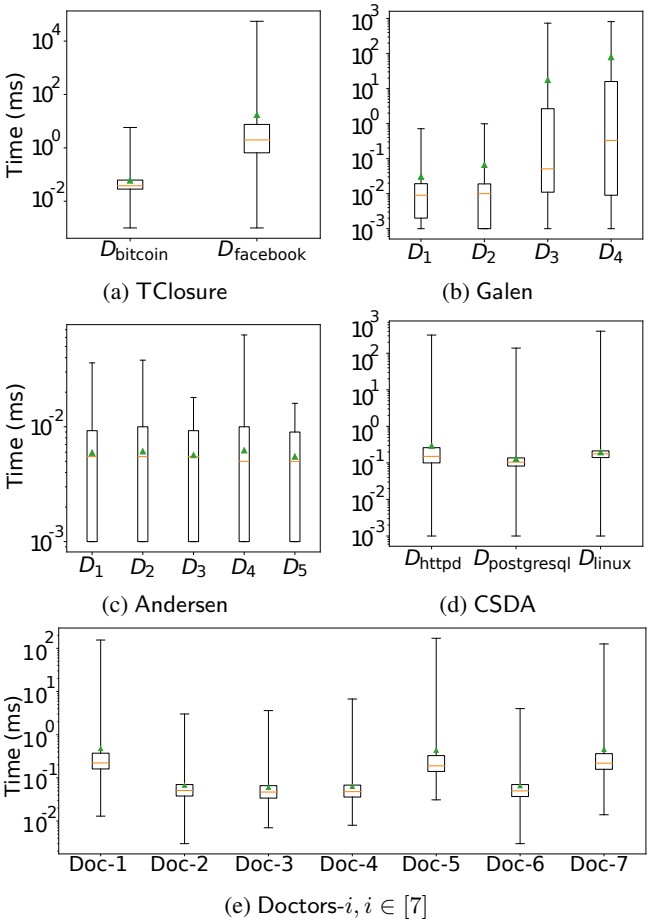


Figure 1: Building the downward closure and the formula.

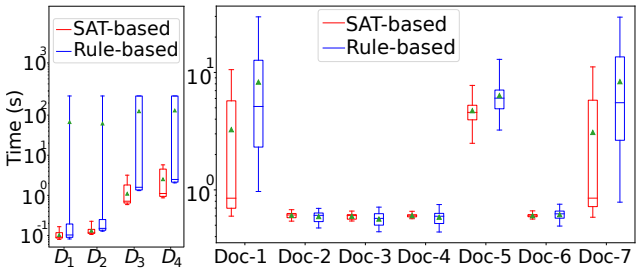
Experimental Results. Concerning the construction of the downward closure and the formula, each plot in Figure 1 corresponds to one of our scenarios, where, for each database of the scenario, we report a box-plot collecting all the runtimes relative to the 100 tuples associated to that database. An exception to this are the Doctors scenarios, which, for the sake of presentation, have been grouped in a single plot, where the database is implicitly the only one available (i.e., D_1). In each box-plot, the bottom and the top borders represent the first and third quartile, i.e., the runtime under which 25% and 75% of all the runtimes occur, respectively, and the orange line represents the median runtime. Moreover, the bottom and the top whisker represent the minimum and maximum runtime, respectively, while the triangle denotes the average runtime. All times are expressed in seconds, and we use logarithmic or linear scale, depending on the scenario.

We can see that 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 downward closure. Considering Andersen, for the databases having size up to half a million facts, the total time is in the order of some seconds, whereas for the very large databases (3.4M and 6.8M facts), the total time is between half a minute and a minute. This is quite encouraging considering the complexity of the query, the large size of the databases, and the limited power



(a) TClosure (b) Galen (c) Andersen (d) CSDA (e) Doctors-$i, i \in [7]$

Figure 2: Incremental computation of the why-provenance.



(a) Galen (b) Doctors-$i, i \in [7]$

Figure 3: Comparison between SAT-based and rule-based.

of our machine. The CSDA scenario is the most demanding one due to the extremely large databases. Note, however, that it would be similarly demanding even for query answering.

For the incremental computation of the why-provenance, we give in Figure 2 the times required to build an explanation, that is, the time between the current member of the why-provenance and the next one (the delay). Each plot is associated with a scenario where, for each database of the scenario, a box-plot collects the delays of constructing the members of the why-provenance for all the 100 randomly chosen tuples for that database; for each tuple, we let the execution stop at 10K explanations or at a 5 minutes timeout. The box-plots have the same meaning as in Figure 1. Note that the Doctors-based scenarios are grouped in one plot. Most of the delays are below 1 millisecond, with most of the medians in the order of microseconds. Therefore, once we have the formula in place, incrementally computing the members of the why-provenance is extremely fast.

Comparative Evaluation. We conclude by comparing our approach with the one of (Elhalawati, Krötzsch, and Mennicke 2022), which, to the best of our knowledge, is the only one in the literature for constructing the why-provenance of a given tuple.[1] We point out that for a query Q, a database D, and a tuple $\bar{t} \in Q(D)$, the technique of (Elhalawati, Krötzsch, and Mennicke 2022), 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, unlike our approach that incrementally constructs $\mathsf{why}_{\mathsf{UN}}(\bar{t}, D, Q)$ via a SAT solver. 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. Hence, a comparison will help us to clarify whether unambiguous proof trees provide any computational advantage.

We use VLog 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 (Elhalawati, Krötzsch, and Mennicke 2022), and thus, we have access to the rewritten set of existential rules; the authors did not release the tools needed to perform the rewriting. The comparison is shown in Figure 3 where we consider, for each scenario consisting of Q and $\mathcal{D}$, database $D \in \mathcal{D}$, and tuple $\bar{t}$ from the 100 randomly selected ones in $Q(D)$, 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, and we use box-plots as usual.

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. Interestingly, the time for building the downward closure in both approaches is comparable. Thus, the performance difference is indeed due to the combination of relying on unambiguous proof trees and exploiting a SAT solver.

6 Future Steps

From our analysis, it is clear that our future efforts should focus on improving the construction of the downward closure. It will be also interesting to understand whether our SAT-based approach can be used for more expressive query languages such as ontology-mediated queries.

[1] Note that (Elhalawati, Krötzsch, and Mennicke 2022) proposes an alternative approach based on systems of equations. However, it is outperformed or on par with the one based on existential rules.

Acknowledgements

We thank the anonymous referees for their feedback. 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

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

Adrian, W. T.; Alviano, M.; Calimeri, F.; Cuteri, B.; Dodaro, C.; Faber, W.; Fuscà, D.; Leone, N.; Manna, M.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2018. The ASP System DLV: Advancements and Applications. *Künstliche Intell.*, 32(2-3): 177–179.

Audemard, G.; and Simon, L. 2018. On the Glucose SAT Solver. *Int. J. Artif. Intell. Tools*, 27(1): 1840001:1–1840001:25.

Benedikt, M.; Buron, M.; Germano, S.; Kappelmann, K.; and Motik, B. 2022. Rewriting the Infinite Chase. *PVLDB*, 15(11): 3045–3057.

Bourgaux, C.; Bourhis, P.; Peterfreund, L.; and Thomazo, M. 2022. Revisiting Semiring Provenance for Datalog. In *KR*.

Buneman, P.; Khanna, S.; and Tan, W. C. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*, 316–330.

Cook, S. A. 1974. An Observation on Time-Storage Trade Off. *J. Comput. Syst. Sci.*, 9(3): 308–316.

Eiter, T.; Ortiz, M.; Simkus, M.; Tran, T.; and Xiao, G. 2012. Query Rewriting for Horn-SHIQ Plus Rules. In *AAAI*.

Elhalawati, A.; Krötzsch, M.; and Mennicke, S. 2022. An Existential Rule Framework for Computing Why-Provenance On-Demand for Datalog. In *RuleML+RR*.

Esparza, J.; Luttenberger, M.; and Schlund, M. 2014. FPsolve: A Generic Solver for Fixpoint Equations over Semirings. In *CIAA*, 1–15.

Jordan, H.; Scholz, B.; and Subotic, P. 2016. Soufflé: On Synthesis of Program Analyzers. In *CAV*, 422–430.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3): 499–562.

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. RDFox: A Highly-Scalable RDF Store. In *ISWC*, 3–20.

Rankooh, M. F.; and Rintanen, J. 2022. Propositional Encodings of Acyclicity and Reachability by Using Vertex Elimination. In *AAAI*, 5861–5868.

Urbani, J.; Jacobs, C.; and Krötzsch, M. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs. In *AAAI*, 258–264.

Zhao, D.; Subotic, P.; and Scholz, B. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.*, 42(2): 7:1–7:35.