An Existential Rule Framework for Computing Why-Provenance On-Demand for Datalog

Ali Elhalawati[0000-0003-1457-0031], Markus Krötzsch[0000-0002-9172-2601], and Stephan Mennicke[0000-0002-3293-2940]

Knowledge-Based Systems Group, TU Dresden, Dresden, Germany
{fistname.lastname}@tu-dresden.de

Abstract. Why-provenance — explaining why a query result is obtained — is an essential asset for reaching the goal of *Explainable AI*. For instance, recursive (Datalog) queries may show unexpected derivations due to complex entanglement of database atoms inside recursive rule applications. Provenance, and why-provenance in particular, helps debugging rule sets to eventually obtain the desired set of rules. There are three kinds of approaches to computing why-provenance for Datalog in the literature: (1) the complete ones, (2) the approximate ones, and (3) the theoretical ones. What all these approaches have in common is that they aim at computing provenance for all IDB atoms, while only a few atoms might be requested to be explained. We contribute an on-demand approach: After deriving all entailed facts of a Datalog program, we allow for querying for the provenance of particular IDB atoms and the structures involved in deriving provenance are computed only then. Our framework is based on terminating existential rules, recording the different rule applications. We present two implementations of the framework, one based on the semiring solver FPsolve, the other one based Datalog(S), a recent extension of Datalog by set terms. We perform experiments on benchmark rule sets using both implementations and discuss feasibility of provenance on-demand.

Keywords: Datalog provenance · why-provenance · Datalog(S)

1 Introduction

Explainability and justification of data that stems from complex, even recursive, processes have attracted both, the community of knowledge representation as well as the database community. The reason is that recursive programs tend to get complicated, which makes it hard for users to debug, audit, establish trust in, or even query the data. Data provenance is one particular way of achieving these aspects and has a long-standing tradition in the field of relational databases and non-recursive queries [5,8]. For recursive queries, provenance for Datalog has been studied and found to be expressible with quite similar tools as the ones underlying provenance for non-recursive queries [13,10]. Provenance describes how annotations on database tuples or facts have been used to obtain a query result.

In this work, we explore the task of providing explainability for Datalog programs by *why-provenance* [5]. Why-provenance considers the witnesses of a derivation, which

are sets of database atoms that can be used to achieve the derivation through reasoning. Previous works [12,14,3] contributed to computing why-provenance for databases, which are practical but restricted to non-recursive programs (e.g., SQL queries). Only a few works actually considered the implementation of why-provenance for Datalog. Most of it has been either on the theoretical side [9,10], bound to certain underapproximations of why-provenance [20], restricted to non-recursive Datalog programs [17], or require an expensive transformation of the instantiated rules in a Datalog program to a system of equations [11].

Since providing why-provenance for all the data derived from a database via a Datalog program is an expensive task (the set of all witnesses is generally exponential in the size of the database), we explore computing why-provenance for Datalog programs in a more practical framework. Instead of computing provenance for all atoms, we propose and implement an on-demand approach. Our contributions are summarized as follows:

- We introduce a novel on-demand approach to why-provenance computation restricting its computation to a given goal atom. This approach is a purely rule-based one (Sect. 3).
- Upon the structures we introduce for the on-demand approach, we show how to create a system of equations (on-demand) whose solutions, interpreted over the so-called why-semiring, are identical with the why-provenance of the goal atom (Sect. 4). This representation can be presented to a semiring solver, like FPsolve [11], in a serialized form.
- Based on the same structures used before, we provide a novel approach to computing why-provenance based on Datalog(S) rules (Sect. 5). Datalog(S) is a recent extension of Datalog having set terms as first-class citizens [7].
- We experimented with both of the aforementioned realizations and get a first insight on their runtime behavior (Sect. 6).

Beyond the main part of the paper (sections 3–6), we provide our basic notions in Sect. 2, related work in Sect. 7, and our conclusions and future work in Sect. 8.

2 Preliminaries

In this section, we introduce our notation for databases, Datalog, and (why-)provenance. Therefore, we assume a fixed first-order vocabulary $\mathbf{C}$ of constants, $\mathbf{V}$ of variables ($\mathbf{C} \cap \mathbf{V} = \emptyset$), and $\mathbf{P}$ of predicate names. Each predicate name $p \in \mathbf{P}$ has an arity $ar(p) \in \mathbb{N}$. A list of *terms* $t_1, \ldots, t_n$ ($t_i \in \mathbf{C} \cup \mathbf{V}$) is often abbreviated by $\vec{t}$ and has length $|\vec{t}| = n$. For convenience, we treat lists of terms $\vec{t}$ as sets when order is irrelevant. We call an expression $p(\vec{t})$ an *atom* if $p \in \mathbf{P}$, $\vec{t} \subseteq \mathbf{C} \cup \mathbf{V}$, and $|\vec{t}| = ar(p)$. An atom is *ground* if each of its terms is a constant and a finite set $\mathcal{D}$ of ground atoms is called a *database*.

A *Datalog rule* is a first-order formula of the form

$$\rho\colon \forall \vec{x}.p_1(\vec{t_1}) \wedge \ldots \wedge p_m(\vec{t_m}) \rightarrow q(\vec{u}), \tag{1}$$

where $p_1(\vec{t_1}), \ldots, p_m(\vec{t_m})$, and $q(\vec{u})$ are atoms using only variables in $\vec{x}$, such that each variable in $q(\vec{u})$ further occurs in some atom $p_i(\vec{t_i})$ (*safety*). We call $q(\vec{u})$ the *head* of ρ

(denoted $\mathsf{head}(\rho)$) and $p_1(\vec{t_1}) \wedge \ldots \wedge p_m(\vec{t_m})$ the *body* of ρ (denoted $\mathsf{body}(\rho)$). We may treat conjunctions as sets, e.g., to write $p_1(\vec{t_1}) \in \mathsf{body}(\rho)$, and tacitly omit the universal quantifiers for brevity. A variable-free rule is a *rule instance* (i.e., in which all atoms are ground atoms). A set of Datalog rules Σ is referred to as a *Datalog program*. A predicate p that occurs only in rule bodies of Σ is an *EDB predicate* (extensional database predicate); all other predicates are *IDB predicates* (intensional database predicates). Datalog programs Σ are evaluated over databases that use only EDB predicates of Σ. The distinction in EDB and IDB predicates is necessary for a construction in Sect. 3.

Datalog Semantics The semantics of a Datalog program Σ over a database $\mathcal{D}$ can equivalently be defined via least models, least fixpoints of a consequence operator, or proof trees [1]. A *ground substitution* θ is a mapping from variables to constants. For a database $\mathcal{D}$ and a rule ρ, a *match* is a ground substitution θ with $\mathsf{body}(\rho)\theta \subseteq \mathcal{D}$. It is *satisfied* in $\mathcal{D}$ if $\mathsf{head}(\rho)\theta \subseteq \mathcal{D}$, and unsatisfied otherwise. A rule is satisfied if all of its matches are. Models and entailment (of ground facts) for Datalog is defined as usual. We will denote the *least model* of program Σ and database $\mathcal{D}$ as $\Sigma(\mathcal{D})$. An *immediate consequence operator* T_Σ can be defined on databases as $T_\Sigma(\mathcal{D}) = \{\mathsf{head}(\rho)\theta \mid \rho \in \Sigma,\ \theta \text{ is a match for } \rho \text{ on } \mathcal{D}\}$. The iterative application of T_Σ starting from the initial database converges to $\Sigma(\mathcal{D})$.

Example 1. Consider the database $\mathcal{D} = \{e(1,2), e(2,2), e(2,3), e(4,3)\}$ which may be interpreted as a directed graph $(\{1,2,3,4\}, \{(1,2), (2,2), (2,3), (4,3)\})$. Furthermore, let $\Sigma = \{\rho_1, \rho_2\}$ with

$$\begin{aligned} \rho_1 &: \quad e(x,y) \rightarrow t(x,y) \\ \rho_2 &: \quad e(x,z) \wedge t(z,y) \rightarrow t(x,y) \end{aligned}$$

Applying T_Σ iteratively to $\mathcal{D}$ yields additional t-atoms. As a result, $\Sigma(\mathcal{D}) = \mathcal{D} \cup \{t(1,2), t(1,3), t(2,2), t(2,3), t(4,3)\}$. For instance, $t(1,2)$ can be obtained by applying ρ_1 for $\theta_1 = \{x \mapsto 1, y \mapsto 2\}$. Another possibility to derive the same fact is by ρ_2 for match $\theta_2 = \{x \mapsto 1, y \mapsto 2, z \mapsto 2\}$. Of course, this application requires us to have also derived $t(2,2)$, which can be done by ρ_1 and $\theta_3 = \{x \mapsto 2, y \mapsto 2\}$.

Graph of Rule Instances One can describe the derivation steps of T_Σ in a suitable hypergraph. A *(vertex-labeled, directed) hypergraph* $\mathcal{H}$ has the form $(\mathcal{V}, \mathcal{E}, \mathsf{tip}, \mathsf{tail}, \lambda)$, where $\mathcal{V}$ is a finite set of vertices, $\mathcal{E} \subseteq \mathbf{2}^{\mathcal{V}}$ is a set of hyperedges, with each $e \in \mathcal{E}$ having $\mathsf{tip}(e) \in \mathcal{V}$ and $\mathsf{tail}(e) \subseteq \mathcal{V}$, and $\lambda : \mathcal{V} \rightarrow \mathcal{L}$ is a vertex labeling function for some set of labels $\mathcal{L}$. Note that we only allow a single tip per edge but many tails. $\mathcal{H}$ is a *hypertree* if the directed graph with edges $\{t \rightarrow \mathsf{tip}(e) \mid t \in \mathsf{tail}(e), e \in \mathcal{E}\}$ is a tree (with edges pointing from children to parents); we write $\mathit{leaves}(\mathcal{H})$ for its leaves. The *graph of rule instances* for a Datalog program Σ and database $\mathcal{D}$ is the hypergraph $\mathit{GRI}(\Sigma, \mathcal{D}) := (\Sigma(\mathcal{D}), \mathcal{E}, \mathsf{tip}, \mathsf{tail}, \lambda)$, where $\mathcal{E} = \{(\rho, \theta) \mid \rho \in \Sigma, \theta \textit{ a satisfied match for } \rho \textit{ over } \Sigma(\mathcal{D})\}$ with $\mathsf{tip}(\rho, \theta) = \mathsf{head}(\rho)\theta$ and $\mathsf{tail}(\rho, \theta) = \mathsf{body}(\rho)\theta$. The labeling λ is simply the identity function (non-identity labelings will be needed below).

Example 2. Reconsider Σ and $\mathcal{D}$ from Example 1. Each atom $A \in \Sigma(\mathcal{D})$ is a node in the graph of rule instances $\mathit{GRI}(\Sigma, D) = (\Sigma(D), E, \mathsf{tip}, \mathsf{tail}, \mathit{id}_{\Sigma(\mathcal{D})})$. The hyperedges $(\mathcal{E})$ are determined by the rules and matches producing $\Sigma(\mathcal{D})$. For instance, $e = (\rho_1, \theta) \in \mathcal{E}$ with

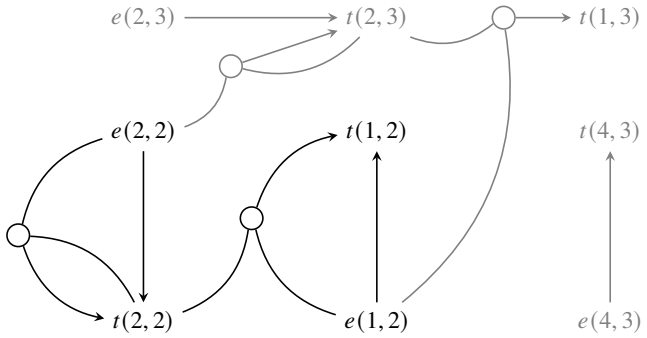

Fig. 1: The Graph of Rule Instances of Σ and $\mathcal{D}$

$\mathsf{tip}(e) = t(1,2)$ and $\mathsf{tail}(e) = \{e(1,2)\}$ for applying ρ_1 for match $\theta = \{x \mapsto 1, y \mapsto 2\}$. Fig. 1 shows a depiction of the full graph, including all the edges. Arrow tips point to the result of applying function tip to the respective hyperedge. If an edge has a single tail-node, just like e above, we depict it as a simple directed edge. Edges having more than one tail-node use a small circle to join the tail-nodes via undirected edges. The distinction between the gray part and the rest of the graph will be explained in Sect. 3.

Proof Trees An alternative approach to the semantics of Datalog is the proof-theoretic one, justifying inferred atoms $A \in \Sigma(\mathcal{D})$ in terms of so-called proof trees, a necessary prerequisite for provenance. A proof tree for A is a hypertree $T = (V, E, \mathsf{tip}, \mathsf{tail}, \lambda)$ with root node v such that $\lambda(v) = A$, $v \in \mathit{leaves}(T)$ implies $\lambda(v) \in \mathcal{D}$, and for each non-leaf node $v \in V \setminus \mathit{leaves}(T)$, there is exactly one edge $e \in E$ with $\mathsf{tip}(e) = v$ with a rule $\rho \in \Sigma$ and satisfied match θ in $\Sigma(\mathcal{D})$, such that $\lambda(\mathsf{tip}(e)) = \mathsf{head}(\rho)\theta$ and $\{B \mid B \in \mathsf{body}(\rho)\theta\} = \{\lambda(t) \mid t \in \mathsf{tail}(e)\}$. Such a proof tree explains one possible derivation of atom A, of which there may be infinitely many (due to recursion). Note that each atom $A \in \mathcal{D}$ has only a single proof tree, which has a single vertex and no edges. By $\mathbb{T}(A, \Sigma, \mathcal{D})$ (or just $\mathbb{T}(A)$ if Σ and $\mathcal{D}$ are clear from the context) we denote the set of all proof trees of $A \in \Sigma(\mathcal{D})$. There is a strong correspondence between $\mathit{GRI}(\Sigma, \mathcal{D})$ and the union of all the proof trees for all atoms $A \in \Sigma(\mathcal{D})$:

Proposition 1. *Let Σ be a Datalog program and $\mathcal{D}$ a database. Then*

$$\mathit{GRI}(\Sigma, D) = (\Sigma(D), E, \mathit{tip}, \mathit{tail}, \mathit{id}_{\Sigma(\mathcal{D})}),$$

where for every $A \in \Sigma(\mathcal{D})$, proof tree $(V, E, \mathit{tip}_T, \mathit{tail}_T, \lambda) \in \mathbb{T}(A)$, and $e \in E$, there is an $e' \in \mathcal{E}$ such that $\lambda(\mathit{tip}_T(e)) = \mathit{tip}(e')$ and $\lambda(\mathit{tail}_T(e)) = \mathit{tail}(e')$.

Thus, the graph of rule instances *captures* the set of all proof trees of the atoms in $\Sigma(\mathcal{D})$ by finite means. If a hypergraph $\mathcal{H}$ captures a (possibly infinite) set of hypergraphs $\mathbb{H}$ in the sense of Prop. 1, we denote this fact by $\mathcal{H} \succ \mathbb{H}$.

Example 3. In Fig. 2, we depict three different proof trees deriving $t(1,2)$ from Σ and $\mathcal{D}$ (cf. Example 1). While tree (a) represents the derivation by rule ρ_1, as discussed earlier,

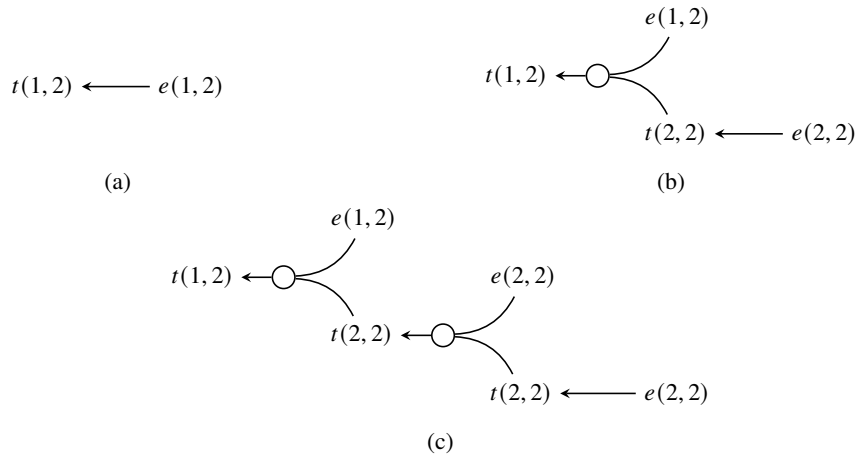

Fig. 2: A Selection of Proof Trees for $t(1, 2)$

trees (b) and (c) use one or two rule applications of ρ_2, prepended by a rule application of ρ_1. Note how in Fig. 2 (c), there are several nodes having the same label. Observe that the unfolding step performed between Fig. 2 (b) and (c) can be performed arbitrarily often, yielding an infinite set of proof trees for $t(1, 2)$.

Why-Provenance for Datalog Provenance is about additional information, usually by means of annotations attached to the atoms of a database. Therefore, we assume a dedicated set K of annotations (e.g., sources, multiplicities, or costs) and provide for each atom $A \in \mathcal{D}$ an annotation $\alpha(A) \in K$. A *K-annotated database* is, thus, a pair $(\mathcal{D}, \alpha)$ such that $\mathcal{D}$ is a database and $\alpha : \mathcal{D} \to K$. To model tuples having no annotations we would assume special symbols like $\bot$ or $\emptyset$ to be part of K.

Let Σ be a Datalog program and $(\mathcal{D}, \alpha)$ a K-annotated database. Why-provenance takes the annotations of database atoms and provides a set of witnesses (sets of database annotations) from all those database atoms that can be used to infer a given atom $A \in \Sigma(\mathcal{D})$. In particular, the annotations of the leaf nodes of a single proof tree for $A \in \Sigma(\mathcal{D})$, as a set, form a witness in the why-provenance of A. For atom $A \in \Sigma(\mathcal{D})$ and a proof tree $T \in \mathbb{T}(A)$, we call the set $\{\alpha(\lambda(v)) \mid v \in \mathit{leaves}(T)\}$ a *witness for A*, denoted by $\alpha(T)$. *Why-provenance of A (w.r.t. Σ and $\mathcal{D}$)* $\mathbf{Why}(A, \Sigma, \mathcal{D})$ (or $\mathbf{Why}(A)$ for short) is the set of all witnesses for A, meaning $\mathbf{Why}(A, \Sigma, \mathcal{D}) := \{\alpha(T) \mid T \in \mathbb{T}(A)\}$. For every $A \in \Sigma(\mathcal{D})$, $\mathbf{Why}(A, \Sigma, \mathcal{D}) \subseteq \mathbf{2}^{\mathbf{2}^{\mathcal{D}}}$ and since $\mathcal{D}$ is finite, the set of all witnesses for $A \in \Sigma(\mathcal{D})$ is also finite, despite the fact that $\mathbb{T}(A, \Sigma, \mathcal{D})$ may be infinite.

3 Rule-Based Provenance On-Demand

Throughout this section, we assume the fixed Datalog program Σ and database $\mathcal{D}$. We have seen that the graph of rule instances $\mathit{GRI}(\Sigma, \mathcal{D})$ captures all proof trees of all atoms $A \in \Sigma(\mathcal{D})$. However, this complete representation of the rule instances is exponential

and, therefore, quite costly to always compute with the set $\Sigma(\mathcal{D})$, especially when we are considering a debugging scenario, in which the provenance of only a few atoms is ever queried. In this section, we present how we avoid the full construction of the GRI, but may still access the necessary parts of it, upon the query for why-provenance of a particular atom $A \in \Sigma(\mathcal{D})$. The key insight to our on-demand approach is that the *downward closure* of $A \in \Sigma(\mathcal{D})$ w.r.t. $\mathit{GRI}(\Sigma, \mathcal{D})$ captures all proof trees $T \in \mathbb{T}(A)$. For hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}, \mathsf{tip}, \mathsf{tail}, \lambda)$ and node $v \in \mathcal{V}$, we denote by $v^{\downarrow}$ the hypergraph $(\mathcal{W}, \mathcal{F}, \mathsf{tip}_{|\mathcal{F}}, \mathsf{tail}_{|\mathcal{F}}, \lambda_{|\mathcal{W}})$, such that $\mathcal{W}$ and $\mathcal{F}$ are the smallest sets satisfying (1) $v \in \mathcal{W}$ and (2) if $u \in \mathcal{W}$ and there is an edge $E \in \mathcal{E}$ with $\mathsf{tip}(E) = u$, then $E \in \mathcal{F}$ and $\mathsf{tail}(E) \subseteq \mathcal{V}$. Note, $\mathcal{W} \subseteq \mathcal{V}$ and $\mathcal{F} \subseteq \mathcal{E}$, justifying the domain restrictions $\mathsf{tip}_{|\mathcal{F}}$, $\mathsf{tail}_{|\mathcal{F}}$, and $\lambda_{|\mathcal{W}}$. Such downward closures are sufficiently representing proof trees in the sense of Prop. 1:

Proposition 2. *For $\mathit{GRI}(\Sigma, \mathcal{D}) = (\Sigma(\mathcal{D}), \mathcal{E}, \mathit{tip}, \mathit{tail}, \lambda)$ and $A \in \Sigma(\mathcal{D})$, we have $A^{\downarrow} \succ \mathbb{T}(A)$.*

Example 4. We stick with atom $t(1, 2) \in \Sigma(\mathcal{D})$ from Example 1, whose downward-closure is the black part of the GRI depicted in Fig. 1. Proof trees in Fig. 2 (a) and (b) directly embed into the relevant part of the GRI. Note that there is no proof tree of $t(1, 2)$ relating to the grayed-out parts of the GRI (cf. Fig. 1).

To achieve an *on-demand-driven* provenance computation we therefore perform two steps: First, we *record* the nodes of $\mathit{GRI}(\Sigma, \mathcal{D})$ (i.e., $\mathcal{V}$), during the derivation of $\Sigma(\mathcal{D})$. This is a necessary prerequisite to prepare for subsequent provenance computations. As we cannot use atoms as terms, we associate each atom $A \in \Sigma(\mathcal{D})$ with a term v_A that may be seen as an identifier for A. Second, when asked for the why-provenance of atom A, we only construct the downward closure $A^{\downarrow}$ by resorting to v_A. We subsequently show how we do both steps via rule-based reasoning using existential quantification in the head of Datalog rules only for the recording step.

For each predicate p occurring in Σ or $\mathcal{D}$, let p^+ be a fresh predicate name with $ar(p^+) = ar(p) + 1$. For each atom $A = p(\vec{t})$, we introduce the aforementioned identifier v_A for A in the last component of its p^+-copy (i.e., $A = p(\vec{t})$ entails the atom $p^+(\vec{t}, v_A)$). We obtain such a copy by using the following rule for predicate p:

$$p(x_1, \ldots, x_{ar(p)}) \rightarrow \exists v.\ p^+(x_1, \ldots, x_{ar(p)}, v). \tag{2}$$

Existential quantification in rule heads leaves the realm of pure Datalog, but is here meant in a safe sense. State-of-the-art rule reasoners, like VLog [6], implement the standard chase which ensures that each atom is copied only once and, therefore, also associated to exactly one (new) identifier v_A. Any reasoner that guarantees this uniqueness constraint may be used to obtain the same results. For each predicate p occurring in Σ, we need to consider a respective rule (2). Let Σ^+ denote the set of all such rules. Note, Σ^+ does not depend on $\mathcal{D}$ because atoms with predicate names not occurring in Σ have no effect on the additional atoms derived in $\Sigma(\mathcal{D})$.

Upon the query for why-provenance of an atom $p(t_1, \ldots, t_n) \in \Sigma(\mathcal{D})$, we may now resort to $p^+(t_1, \ldots, t_n, v) \in (\Sigma \cup \Sigma^+)(\mathcal{D})$ as a starting point, where v indicates the base node from which the downward closure shall be computed. If we add the following rule

to our overall rule set, we trigger the computation of the downward closure from v by the derived fact $G(v)$:

$$\forall x.\ p^+(t_1, \ldots, t_n, x) \rightarrow G(x) \tag{3}$$

We assume G to be a fresh predicate. For an atom A, let us denote its triggering rule (3) by $trig(A)$. For the construction of the downward closure, we match rule instances on atoms produced by rules in Σ^+ (e.g., p^+) and connect the associated nodes accordingly. To cope with the different numbers of atoms per rule, we use auxiliary fresh predicates $\mathcal{E}_1, \ldots, \mathcal{E}_k$, such that $ar(\mathcal{E}_i) = i$ $(1 \leq i \leq k)$. We pick a high enough $k \in \mathbb{N}$ (e.g., the maximum number of atoms in a rule of Σ). Then for each rule $\rho = p_1(\vec{t_1}) \wedge \ldots \wedge p_l(\vec{t_l}) \rightarrow q(\vec{u}) \in \Sigma$, we create the rule

$$G(v_0) \wedge q^+(\vec{u}, v_0) \wedge p_1^+(\vec{t_1}, v_1) \wedge \ldots \wedge p_l^+(\vec{t_l}, v_l) \rightarrow \mathcal{E}_{l+1}(v_0, v_1, \ldots, v_l) \wedge \bigwedge_{i=1}^{l} G(v_i), \tag{4}$$

where $v_0, v_1, \ldots, v_l$ is a list of variables distinct from those used in ρ. Note, having more than one head atom in a Datalog rule, as in (4), is just a shorthand for as many single-head rules as there are atoms in the head. Requiring the derivation of atoms $G(v_1), \ldots, G(v_l)$ makes sure that the downward closure is triggered recursively. Let $\Sigma^\diamond$ be the set of rules like (4) for each $\rho \in \Sigma$. Additionally, add to $\Sigma^\diamond$ one rule

$$q^+(\vec{t}, v) \rightarrow \mathcal{E}_1(v) \tag{5}$$

for each EDB predicate p of Σ. $\mathcal{E}_1$-atoms will later be used as the base case for provenance computation. For an atom $A \in \Sigma(\mathcal{D})$, adding $trig(A)$ to $\Sigma \cup \Sigma^+ \cup \Sigma^\diamond$ suffices to fully describe the downward closure of A (via v_A) as follows:

Proposition 3. *Let $A \in \Sigma(\mathcal{D})$ and $\mathcal{M} = (\Sigma \cup \Sigma^+ \cup \Sigma^\diamond \cup \{trig(A)\})(\mathcal{D})$, Then E with $\mathsf{tip}(E) = v_{B0}$ and $\mathsf{tail}(E) = \{v_{B1}, \ldots, v_{Bn}\}$ is an edge of $A^\downarrow$ if, and only if, there is a set $\{i_1, i_2, \ldots, i_n\} = \mathsf{tail}(E)$, such that $\mathcal{E}_{n+1}(v_{B0}, v_{i_1}, \ldots, v_{i_n}) \in \mathcal{M}$.*

Throughout the next two sections, we incorporate these preliminary constructions in two solutions for obtaining why-provenance of atoms $A \in \Sigma(\mathcal{D})$. The first is based on a representation of why-provenance as the solutions to a system of equations [13] specific to $\Sigma(\mathcal{D})$. The second is a purely rule-based approach, incorporating the recent extension of Datalog by set primitives, called Datalog(S) [7].

4 Realization as Solutions to Systems of Equations

Green et al. [13] have shown that provenance for Datalog (over K-annotated databases) can be generalized to solutions of a *system of equations*, specific to the Datalog program Σ and the K-annotated database $(\mathcal{D}, \alpha)$, interpreted over certain semirings. In the equations we use a generalized join operator $\otimes$ (for combining the leaf nodes of a single proof tree) and $\oplus$ as the generalized union (for combining alternative proof trees). Furthermore, a system of equations uses variables from a set $\mathbb{V}$, such that for each atom $A \in \Sigma(\mathcal{D})$, $\mathbb{V}(A) \in \mathbb{V}$ and

$$\mathbb{V}(A) = \bigoplus_{T \in \mathbb{T}(A, \Sigma, \mathcal{D})} \left(\bigotimes_{B \in \alpha(T)} \mathbb{V}(B) \right) \tag{6}$$

is the characteristic equation for A (w.r.t. Σ and $\mathcal{D}$). The system of equations for Σ and $\mathcal{D}$ is then the pair $(\mathbb{V}, \mathbb{E})$, such that $\mathbb{E}$ contains the characteristic equation (6) for each $A \in \Sigma(\mathcal{D})$. $(\mathbb{V}, \mathbb{E})$ is interpreted over semirings that provide different granularities of provenance information. For Datalog provenance, it is of utmost importance that the semiring at hand is ω-continuous, guaranteeing that infinite sums, like the ones introduced by (6) (recall that the set $\mathbb{T}(A, \Sigma, \mathcal{D})$ is generally infinite), have well-defined solutions. Fortunately, the semiring for why-provenance, $\mathbf{Why}(\mathbb{V}, K) = (\mathbf{2}^{\mathbf{2}^K}, \cup, \Cup, \emptyset, \{\emptyset\})$ enjoys this property. Note, for sets A and B of subsets of K, $A \Cup B := \{a \cup b \mid a \in A, b \in B\}$. An *assignment* over $\mathbf{Why}(\mathbb{V}, K)$ is a function $\beta : \mathbb{V} \to \mathbf{2}^{\mathbf{2}^K}$. An assignment β is *valid for* $(\mathbb{V}, \mathbb{E})$ if, and only if, (a) $\beta(\mathbb{V}(A)) = \alpha(A)$ for each $A \in \mathcal{D}$, and (b) for each equation of the form (6),

$$\beta(\mathbb{V}(A)) = \bigcup_{T \in \mathbb{T}(A,\Sigma,\mathcal{D})} (\beta(\mathbb{V}(B_1)) \Cup \ldots \Cup \beta(\mathbb{V}(B_l))) \,, \tag{7}$$

where for each $T \in \mathbb{T}(A, \Sigma, \mathcal{D})$, $\alpha(T) = \{B_1, \ldots, B_l\}$ ($l \in \mathbb{N}$). Thus, operator $\oplus$ is evaluated via union ($\cup$) and $\otimes$ via cross-union ($\Cup$).

The general infinity of equations like (6) is impractical for providing actual tool support. Fortunately, an alternative system of equations makes use of the fixpoint approach to solving equations over semirings, encompassed by the finite representation of all proof trees, the graph of rule instances $GRI(\Sigma, \mathcal{D}) = (\Sigma(\mathcal{D}), \mathcal{E}, \mathsf{tip}, \mathsf{tail}, \lambda)$. For the finite system of equations, we use the set $\Sigma(\mathcal{D})$ as the set of variables. To make the distinction between $\Sigma(\mathcal{D})$ and its system of equation clear, we will denote the variable of $A \in \Sigma(\mathcal{D})$ by its identifier v_A (cf. Sect. 3). As set of equations $\mathbb{E}$, we have the following equation for each atom $A \in \Sigma(\mathcal{D})$:

$$v_A = \bigoplus_{E \in \mathcal{E}, \mathsf{tip}(E) = v_A} \left(\bigotimes_{w \in \mathsf{tail}(E)} w \right) \tag{8}$$

Green et al. [13] showed that systems of equations created from (6) are equivalent in their solutions with systems of equations using only finite equations of the form (8) (one for each $v_A \in \mathcal{V}$).

Based on (8), we can also obtain a system of equations on-demand for node v_A by pursuing the additional rules we added in the last section. Upon why-provenance query for $A \in \Sigma(\mathcal{D})$, let $\Sigma^\star = \Sigma \cup \Sigma^+ \cup \Sigma^\circ \cup \{trig(A)\}$ and $\mathcal{M} = \Sigma^\star(\mathcal{D})$. Then we derive the system of equations on-demand for A by querying for the atoms of relations $\mathcal{E}_1, \ldots, \mathcal{E}_k$. Of course, we will use the node identifiers as variables and define it by $V := \{v \mid p^+(\vec{t}, v) \in \mathcal{M}\}$ (due to Σ^+). For each variable $v \in V$, let $\mathcal{E}(v) := \{\{v_1, \ldots, v_l\} \mid 1 \leq l \leq k \wedge \mathcal{E}_{l+1}(v, v_1, \ldots, v_l) \in \mathcal{M}\}$. Both sets can be constructed by querying for the respective atoms in $\mathcal{M}$. The *system of equations for A on-demand* is $(V, \mathbb{E})$ where $\mathbb{E}$ is the set containing

$$v = \bigoplus_{E \in \mathcal{E}(v)} \left(\bigotimes_{w \in E} w \right) \tag{9}$$

for each $v \in V$. This is the system of equations we present a semiring solver like FPsolve [11]. Interpreted over the why-semiring, the valid assignments for v_A refer to the provenance of A.

5 Realization with Datalog(S)

Our second solution to why-provenance on-demand is a pure rule-based approach. Let us assume a Datalog program Σ, a database $\mathcal{D}$, an atom $A \in \Sigma(\mathcal{D})$ we want to know why-provenance for, and the rule set $\Sigma^\star$ containing Σ, Σ^+, Σ°, and $\{trig(A)\}$ (cf. Sect. 3). We now associate with each node identifier v (due to rules in Σ^+) sets of node identifiers that each represents a witness for the atom represented by A. Sets are not part of the terms in Datalog, but the recent extension of Datalog with sets [7] does include special set terms. The language facilitating basic set operations during reasoning is called Datalog(S) and will be briefly introduced in the first part of this section. Later on, we give a fixed set of Datalog(S) rules $\Sigma^k_{\mathbf{Why}}$ for Σ, capable of traversing the downward closure produced by $\Sigma^\star$ and collecting witnesses in the above-mentioned sense.

5.1 Datalog with Sets

Datalog(S) is a recent extension of Datalog that introduces a new term set, the set variables $\mathbf{V_S}$, which can be used in a Datalog(S) program. Furthermore, the set terms that can be used are defined inductively: (1a) $V \in \mathbf{V_S}$ is a set term, (1b) $\mathbf{0}$ is a set term representing the empty set in Datalog(S), (1c) for each term $t \in \mathbf{C} \cup \mathbf{V}$, $\{t\}$ is a set term, and (2) if T_1 and T_2 are set terms, then $(T_1 \bigcup T_2)$ is a set term. We often drop the parenthesis in unions. A Datalog(S) rule has the form

$$\forall \vec{x}, \vec{X}.\ \varphi[\vec{x}, \vec{X}] \rightarrow q(\vec{u}), \tag{10}$$

where $\vec{x} \subseteq \mathbf{V}$, $\vec{X} \subseteq \mathbf{V_S}$, $\varphi \cup \{q(\vec{u})\}$ is a set of atoms (potentially) with set terms, such that each variable in $\vec{u}$ is an element of $\vec{x} \cup \vec{X}$. Instances (and models) are variable-free sets of atoms or set atoms (i.e., atoms containing set terms). Since we use Datalog(S) only for the computation of provenance, we can assume databases to be set-atom-free ground instances. Substitutions must match set variables with set terms and variables in $\mathbf{V}$ with non-set terms, and matching rules are defined accordingly. The procedures evaluating a set of Datalog(S) rules, also called *Datalog(S) programs*, are similar to the ones for Datalog. Carral et al. provide a reasoning approach based on an encoding of Datalog(S) programs in existential rules [7]. We subsequently give a Datalog(S) program that traverse the downward-closure we constructed in Sect. 3.

5.2 Collecting Sets from Downward Closures

Let *prov* be a fresh binary predicate (i.e., it does not occur in $\Sigma^\star$). The rules collecting the witnesses as set terms produce *prov*-atoms with a node identifier in the first position and a set term (a witness) in the second. Recall that the model $\Sigma^\star(\mathcal{D})$ contains facts for the predicates $\mathcal{E}_1, \ldots, \mathcal{E}_k$ (for some $k \in \mathbb{N}$ determined by the rules in Σ). Furthermore, all

database node identifiers v_A (with $A \in \mathcal{D}$) are represented by atoms $\mathcal{E}_1(v_A) \in \Sigma^\star(\mathcal{D})$. It is going to be these database node identifiers that will be carried along in the witnesses of atoms $B \in \Sigma(\mathcal{D})$. Therefore, we construct a set of Datalog(S) rules $\Sigma^k_{\mathbf{Why}}$ in which each rule introduces new *prov*-atoms as witness information. In the base case, rule

$$\mathcal{E}_1(x) \to \mathit{prov}(x, \{x\}) \tag{11}$$

belongs to $\Sigma^k_{\mathbf{Why}}$. Upon evaluation, the provenance of each database atom A is represented by the atom $\mathit{prov}(v_A, \{v_A\})$. For each $j \in \{1, \ldots, k-1\}$, the set $\Sigma^k_{\mathbf{Why}}$ contains the following rule:

$$\mathcal{E}_{j+1}(x_0, x_1, \ldots, x_j) \wedge \bigwedge_{i=1}^{j} \mathit{prov}(x_i, X_i) \to \mathit{prov}(x_0, X_1 \bigcup \cdots \bigcup X_j) \tag{12}$$

Since every edge of the downward closure effectively represents a rule instance, the union of all witnesses belonging to the body of rule instance is a witness for the head.

6 Implementation and Experimental Results

We implemented our on-demand approach (cf. Sect. 3) using VLog [6] and realized why-provenance computation with the approaches described in sections 4 and 5. We chose FPsolve [11] as the semiring solver since it offers a built-in implementation of the why-semiring. We emulated Datalog(S) by means of terminating existential rules, again using VLog, as described by Carral et al. [7]. For a given Datalog program Σ, we have run a script translating the program into $\Sigma^\star$ (cf. Sect. 3). VLog does not only perform the initial Datalog reasoning step, but is also crucial in the construction of the system of equations for FPsolve (Sect. 4) and for performing the necessary reasoning steps of our Datalog(S)-based process (Sect. 5).

As input, we used the Doctors scenario from the *Chasebench* [4] and a Datalog implementation of the EL ontology Galen[1] using the ELK reasoning calculus [15]. In particular, we rewrote the Doctors scenario into Datalog (originally, it contains existential rules), which we obtained by replacing all occurrences of existentially quantified variables uniformly by fresh constants. The Doctors scenario provides seven queries (`q1`–`q7`) alongside the dataset 100k, which we subsequently identify by the names of the queries. As for the EL ontology Galen, we queried for the `rdfs:subClassOf`-property. As input data for Galen we obtained 10%, 15%, 25%, 40%, and 50% random samples (subsequently referred to as `g10`,`g15`,...,`g50`). We are well aware that such a random sampling may destroy a lot of the complicated reasoning in Galen. These initial experiments still allow for a glimpse on how the two on-demand provenance approaches handle increasing sizes of datasets, performance-wise.

Experimental Workflows As input we get a Datalog program Σ and a database $\mathcal{D}$. Furthermore, why-provenance of (a randomly chosen) atom $A \in \Sigma(\mathcal{D})$ must be provided. We describe the tools and steps involved in the computation of $\mathbf{Why}(A)$.

[1] Galen is an ontologies found in the Oxford Library

Common Preparation: We have created the rule set $\Sigma^\star$ from Σ using python scripts. Then VLog reasoned for $\Sigma^\star(\mathcal{D})$ having created a node identifier v_B for each $B \in \Sigma(\mathcal{D})$ and the downward-closure $v_A^\downarrow$.

FPsolve Process: When we ask FPsolve for $\mathbf{Why}(A)$, we query $\Sigma^\star(\mathcal{D})$ as described in Sect. 4 and produce an input file f for FPsolve that represents a serialization of the downward-closure $v_A^\downarrow$. Then we call FPsolve on input f to produce the output of why-provenance (as a string). For this process, we could separate the times for (a) the common preparation step above, (b) the writing of f (which requires querying $\Sigma^\star(\mathcal{D})$), and (c) the time for FPsolve to solve the given system of equations.

Datalog(S) Process: VLog does not feature incremental reasoning, which means that we had to simulate the materialization for $\mathbf{Why}(A)$ in *one shot*: We measure the time taken by VLog to produce the materialization $(\Sigma^\star \cup \Sigma^k_{\mathbf{Why}})(\mathcal{D})$. The differences between the runtimes of the preparation step ($\Sigma^\star$ only) and the full materialization is inconclusive since different rule sets have unpredicatably different runtime behavior. Even if the one rule set is a superset of another, VLog may take less time for the materialization because additional rules may mean that there are shortcuts that were previously impossible.

As one why-provenance query appears insufficient – we could have just been lucky with the random choice of the atom $A \in \Sigma(\mathcal{D})$ – we have picked 50 atoms from $\Sigma(\mathcal{D})$ at random and repeated the processes for each of them separately. Therefore, we show aggregations of the runtime behavior. We have set a timeout of 200 seconds for each run. All experiments have been performed on a machine with an Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz processor, 378 GB of RAM, 1 TB of storage, and Debian GNU/Linux 9.13. Pointers to the datasets we have used in the experiments and the resulting runtimes we obtained are available on GitHub.

6.1 Feasibility of the On-Demand Approach to Why-Provenance

There are at least three user scenarios to consider for the question of feasibility of provenance on-demand: (1) the *provenance power user*, who constantly queries for the provenance of any atom that has been derived, (2) the *provenance non-user*, who never poses a query for provenance, and (3) the *Datalog debugger*, who looks at the materialization, finds a small number atoms (to be buggy) and tries to fix the Datalog program according to the why-provenance of the previously identified atoms. Then, user (3) starts over with a fresh Datalog program for which provenance has to be computed anew. As an alternative approach to provenance on-demand, we consider computing why-provenance of all derived atoms alongside the derivation of $\Sigma(\mathcal{D})$. This approach may be quite satisfactory for user scenario (1) since, once provenance is computed by either FPsolve or our Datalog(S) solution, retrieving provenance information is simply reduced to query answering over a database. However, performing this exponential step alongside the computation of $\Sigma(\mathcal{D})$ may be a lot more time-consuming than just creating linearly many node identifiers. In fact, the blank and gray bars in Fig. 3 show median reasoning times for $\Sigma(\mathcal{D})$ (blank bar) and those for $\Sigma^\star(\mathcal{D})$ (gray bar), respectively. Especially for user scenario (2), the time difference between the different modes may be considered pleasant because even if there is no provenance query at all, reasoning times

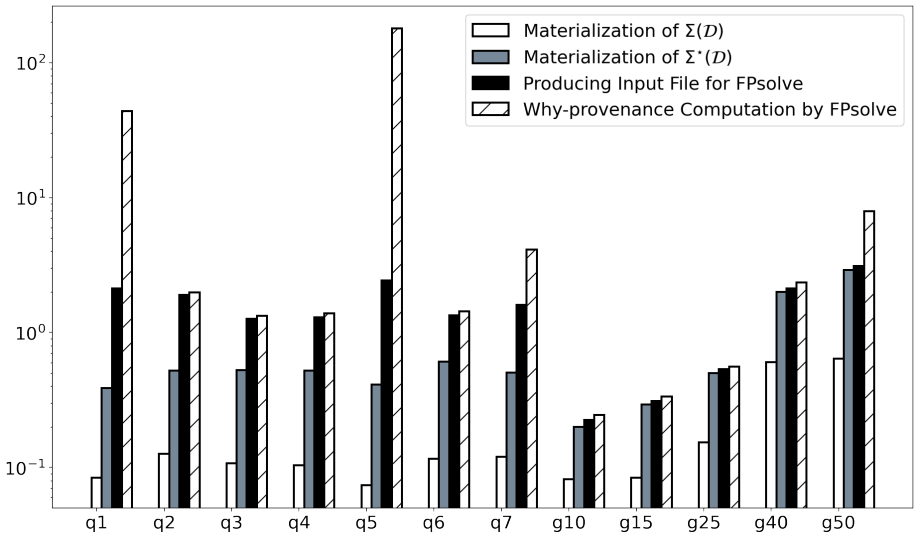

Fig. 3: Median Runtimes of Different Phases of the FPsolve Process

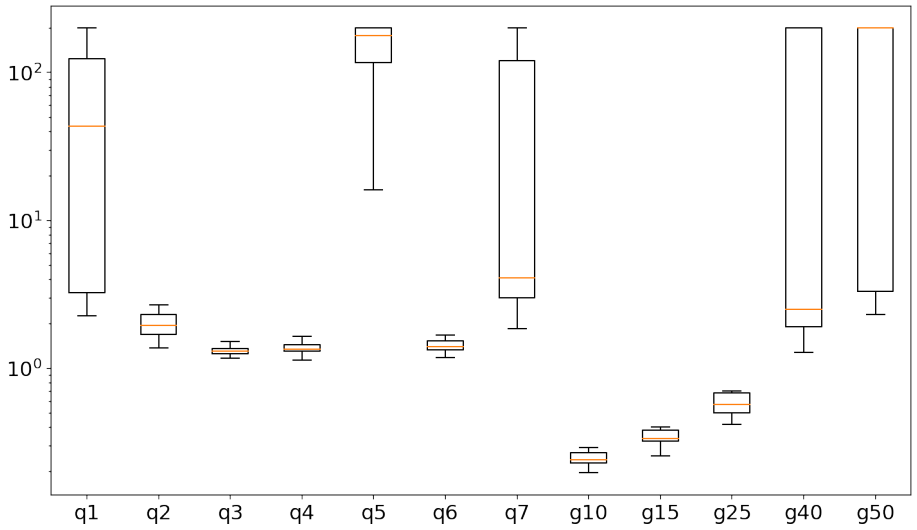

Fig. 4: Summary of the Runtime Results in Seconds for the FPsolve Process

for $\Sigma^\star(\mathcal{D})$ are well below 1 second or close to the time for reasoning with just Σ. Our on-demand process has especially been designed for user scenario (3) and needs to be integrated with, for instance, Datalog synthesis processes like the one by Raghothaman et al. [18]. Here we can only tell from our experience with initial experiments with FPsolve: when trying to compute all provenance for all atoms, FPsolve took hours or even days to finish. By glimpsing on the runtime reports of the next subsection, we see that our Datalog(S) approach would work equally bad. A final evaluation comparing the on-demand approach with the "all provenance at once" approach is left for future work.

6.2 Performance of Why-Provenance Computation

For this experiment we report about the different time measurements we explained earlier for the two approaches. Note that the preparation times (common to both processes) have been reported in the previous subsection (cf. gray bars in Fig. 3)

The FPsolve Process Additionally to the preprocessing step, producing the downward-closure, we can report on the median times for producing the input file for FPsolve (black bars in Fig. 3), which includes i/o operations to hard disk. As we can see, this part of the process takes some time but in most cases is superseded by FPsolve solving the system of equations (while solving also always includes reading the file we produced). In Fig. 4, we see a summary of the overall runtimes of the fifty runs for each query/scenario. The boxplots reflect on the ranges of runtimes (rectangles), the media runtime (orange line), and runtime deviations (whiskers). While most of the queries have somewhat stable runtime results, queries `q1`, `q5`, and `q7` seem to have rather diverse ones. In our Galen experiments, we can see that from `g40` (the 40% sample of Galen) on, we produce timeouts. At `g50`, almost all runs did not finish within the time bound. One reason may be that the file sizes (reflecting on the size of the system of equations) grows by orders of magnitude from one Galen sample to a bigger one.

Looking for reasons of why FPsolve sometimes has bad runtimes, we observed that the computed downward closures are significantly larger for queries `q1`, `q5`, `q7` than for the other ones in Doctors (up to one order magnitude). The size of the downward closure correlates with the produced system of equations that is given as input to FPsolve. It appears as if reading (from disk) and solving bigger systems of equations is not the ideal use case for a tool like FPsolve. One way resolving the issue is to create a service pipeline that does not have the intermediate file representation that also has to be read from hard disk.

The Datalog(S) Process Fig. 5 reports on the overall runtime of our Datalog(S) process (interpretation is as for Fig. 4). On one hand, we observe more diversity in the smaller Galen samples than observed for FPsolve. On the other hand, we obtain even more stable runtimes for the queries in the Doctors scenario. By comparing figures 4 and 5, we see that FPsolve is faster than Datalog(S) in Galen experiments by fine margins. However, Datalog(S) was faster than FPsolve in the queries of the Doctor scenario, sometimes with significant differences as, for instance, in `q1`, `q5` and `q7`. Our Datalog(S) solution has the advantage that we use a single tool, here VLog, that has been made for handling large amounts of data. Even the big structures produced for computing witnesses as

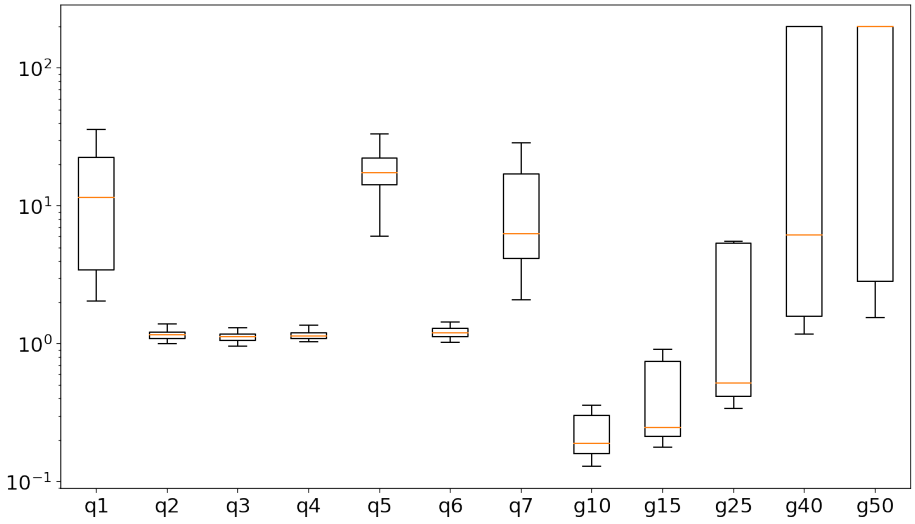

Fig. 5: Summary of Runtime Results in Seconds for the Datalog(S) Process

sets can be handled in acceptable time. Especially the results in the Doctors scenario came as a surprise, given that the implementation of Datalog(S) we use is prone to introducing redundancy by means of having a single set represented several (but finitely many) times [7].

Whether it really pays off that we do not have to leave the reasoner for another (more specialized) tool is still an open question. A reasoner, particularly designed to evaluate Datalog(S) programs, may show a lot more improvement over the FPsolve process than we see with our implementation.

7 Related Work

We are not aware of any tool that computes why-provenance for Datalog programs besides FPsolve. However, several provenance notions as well as implementation ideas for Datalog provenance have been studied. GProM [3] and its extension PUG [17] provide why- and why-not-provenance for non-recursive Datalog programs with negations. These tools are also based on graph representations of the derivation process. Since they only consider non-recursive Datalog programs, they do not have to deal with cycles. Having only a graph depiction for a Datalog program with a huge database can be confusing and hard to trace, as edges and nodes in the graph can get overwhelming. In [20], Zhao et al. represent provenance in the form of *Proof Trees* that show the rule applications required to reach a particular derivation. The output of their provenance is the minimal proof tree, from which only a single witness can be derived. Similar ideas have been used as part of a synthesis process of Datalog programs [18]. However, presenting only a single witness does not cover the original notion of why-provenance, as we tackle it here. Deutch et al. [10] introduced provenance in the form of a finite

circuit structure which is another representation of the proof trees view and the GRI. One result of this work has been the *absorptive semiring* for provenance, being incomparable to why-provenance. Köhler et al. [16] produce the graph of rule instances for Datalog programs using a rewriting with Skolem functions which is very similar to how we construct the downward closure by existential rules. They use this graph to provide the lineage of an atom A, being the union of all witnesses of A, an easier task than computing why-provenance which can completely be implemented in Datalog. Recently, an interesting approach has been investigated by Ramusat et al. [19] who view provenance computation as an operation of iterating over a graph to obtain the shortest path. They consider graph databases but we can imagine this work to also be generalizable to provenance for Datalog.

8 Conclusions

We presented an on-demand approach to why-provenance for recursive queries by means of Datalog programs. As a preprocessing step, we annotate each and every atom by a node identifier (linear in $\Sigma(\mathcal{D})$). We also showed how to compute the necessary information for computing (why-)provenance (a form of sub-hypergraph of the so-called graph of rule instances). These steps are purely rule-based and enable for subsequent why-provenance computation, given an atom A we want to know provenance of. We presented two realizations, one based on a semiring solver and one based on Datalog(S) reasoning. The latter solution builds on a recent extension of Datalog regarding set terms as first-class citizens. Our initial experiments show that there is room for improvement, for which more extensive experiments will be necessary. An interesting way to go is to exploit the recent characterization of certain provenances as (generalized) shortest paths [19]. Of course, this will leave the realm of a rule-based approach. We also believe that the graph of rule instances, as we compute it, can be used by Al-Rabbaa et al. [2] to give "good proofs" for Horn description logics with a translation into Datalog. The key task here will be to translate "good proofs" as obtained from traversing the GRI of the Datalog program into "good proofs" of the underlying description logics.

Acknowledgments This work is partly supported by the German Research Foundation (DFG) in project number 389792660 (TRR 248, Center for Perspicuous Systems), by the Federal Ministry of Education and Research (BMBF) in project number 13GW0552B (KIMEDS, KI-assistierte Zertifizierung medizinischer Software) and in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), by BMBF and German Academic Exchange Service (DAAD) in project 57616814 (SECAI, School of Embedded Composite AI), as well as by the Center for Advancing Electronics Dresden (cfaed).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Al-Rabbaa, C., Borgwardt, S., Koopmann, P., Kovtunova, A.: Explaining Ontology-Mediated Query Answers using Proofs over Universal Models. In: Proc. 6th Int. Joint Conf. on Rules and Reasoning (RuleML+RR 2022). Springer (2022)

3. Arab, B.S., Feng, S., Glavic, B., Lee, S., Niu, X., Zeng, Q.: GProM - A Swiss Army Knife for Your Provenance Needs. Proc. IEEE Data Eng. Bull. **41**(1), 51–62 (2018)
4. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Sallinger, E., Van den Bussche, J., Geerts, F. (eds.) Proc. 36th Symposium on Principles of Database Systems (PODS'17). pp. 37–52. ACM (2017)
5. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) Proc. 8th Int. Conf. on Database Theory (ICDT'01). LNCS, vol. 1973, pp. 316–330. Springer (2001)
6. Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., Urbani, J.: VLog: A rule engine for knowledge graphs. In: Ghidini et al., C. (ed.) Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II). LNCS, vol. 11779, pp. 19–35. Springer (2019)
7. Carral, D., Dragoste, I., Krötzsch, M., Lewe, C.: Chasing Sets: How to Use Existential Rules for Expressive Reasoning. In: Kraus, S. (ed.) Proc. 28th Int. Joint Conf. on Artificial Intelligence (IJCAI'19). pp. 1624–1631. ijcai.org (2019)
8. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in Databases: Why, How, and Where. J. of Found. Trends Databases **1**(4), 379–474 (2009)
9. Damásio, C.V., Analyti, A., Antoniou, G.: Justifications for Logic Programming. In: Cabalar, P., Son, T.C. (eds.) Proc. 12th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013). pp. 530–542. Springer (2013)
10. Deutch, D., Milo, T., Roy, S., Tannen, V.: Circuits for Datalog Provenance. In: Schweikardt, N., Christophides, V., Leroy, V. (eds.) Proc. 17th Int. Conf. on Database Theory (ICDT 2014). pp. 201–212. OpenProceedings.org (2014)
11. Esparza, J., Luttenberger, M., Schlund, M.: FPsolve: A Generic Solver for Fixpoint Equations over Semirings. In: Holzer, M., Kutrib, M. (eds.) Proc. 19th Int. Conf. on Implementation and Application of Automata (CIAA 2014). LNCS, vol. 8587, pp. 1–15. Springer (2014)
12. Glavic, B., Miller, R.J., Alonso, G.: Using SQL for Efficient Generation and Querying of Provenance Information. In: Tannen, V., Wong, L., Libkin, L., Fan, W., Tan, W., Fourman, M.P. (eds.) In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman, pp. 291–320. Springer (2013)
13. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proc. 26th Symposium on Principles of Database Systems (ACM SIGACT-SIGMOD-SIGART 2007). pp. 31–40 (2007)
14. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proc. Int. Conf. on Management of Data (SIGMOD 2010). pp. 951–962. ACM (2010)
15. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. J. of Automated Reasoning **53**, 1–61 (2013)
16. Köhler, S., Ludäscher, B., Smaragdakis, Y.: Declarative Datalog Debugging for Mere Mortals. In: Barceló, P., Pichler, R. (eds.) Proc. 2nd Int. Workshop on Datalog in Academia and Industry (Datalog 2.0 2012). LNCS, vol. 7494, pp. 111–122. Springer (2012)
17. Lee, S., Ludäscher, B., Glavic, B.: PUG: a framework and practical implementation for why and why-not provenance. Proc. VLDB **28**(1), 47–71 (2019)
18. Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., Scholz, B.: Provenance-guided synthesis of Datalog programs. Proc. ACM Program. Lang. **4**(POPL), 62:1–62:27 (2020)
19. Ramusat, Y., Maniu, S., Senellart, P.: Provenance-Based Algorithms for Rich Queries over Graph Databases. In: Velegrakis, Y., Zeinalipour-Yazti, D., Chrysanthis, P.K., Guerra, F. (eds.) Proc. 24th Int. Conf. on Extending Database Technology (EDBT 2021). pp. 73–84. OpenProceedings.org (2021)
20. Zhao, D., Subotic, P., Scholz, B.: Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. Proc. ACM Trans. Program. Lang. Syst. **42**(2), 7:1–7:35 (2020)