Declarative Debugging for Datalog with Aggregation

Raimund Dachselt[2], Lukas Gerlach[1], Philipp Hanisch[1], Alex Ivliev[1], Markus Krötzsch[1], Maximilian Marx[1] and Julián Méndez[2]

[1]*Knowledge-Based Systems Group, TU Dresden, Germany*
[2]*Interactive Media Lab Dresden, TU Dresden, Germany*

Abstract

We propose *summary proof trees* as a way for explaining derivations in Datalog programs with aggregation. These combine structurally similar parts of proof trees into a single, easier to understand structure. We show how to query for such summaries, discuss the implementation in our rule engine *Nemo*, and empirically establish the feasibility of the method. We also briefly introduce a graphical interface for editing these proof queries and visualising the results.

1. Introduction

Rule languages are a versatile approach to data manipulation and transformation, where especially Datalog and its extensions have found many successful applications [1]. Indeed, such rules are a highly declarative way to specify expressive recursive views, which can then be evaluated by scalable modern rule engines [2, 3, 4, 5, 6].

Datalog as such is based on a relational data model [7]. Each rule essentially consists of a select-project-join query (the *body*), the answers of which are added to a specified result table (the *head*). Rules can be mutually recursive: to ensure termination, duplicates are eagerly eliminated during evaluation (*set semantics*). In recent years, Datalog has been used for graph transformation [8, 9, 10, 11], as it is a small step from arbitrary relations to sets of graph edges. Modern Datalog systems that can handle graph data natively include RDFox [2] and Nemo [5], which both support RDF and SPARQL [12]. Here we avoid the technical details of these standards, since simple forms of graph-like data will suffice to illustrate our contribution.

Example 1. Consider a transformation that turns a family tree (given by binary parent (*par*) and significant other (*so*) predicates) into a kinship graph (over binary predicate *kin*).

$$\mathit{kin}(x, y) \leftarrow \mathit{par}(x, y) \tag{1}$$

$$\mathit{kin}(x, y) \leftarrow \mathit{kin}(y, x) \tag{2}$$

$$\mathit{kin}(x, y) \leftarrow \mathit{kin}(x, z), \mathit{par}(z, y) \tag{3}$$

$$\mathit{kin}(x, y) \leftarrow \mathit{par}(x, z), \mathit{so}(z, z'), \mathit{par}(y, z') \tag{4}$$

Rule (1) states that any parent y of x is kin to x, where kinship is symmetric (2) and transitive (3). Rule (4) extends kinship to (step-)siblings. A (graph-like) input database is given by the following facts $\{\mathit{par}(a, b), \mathit{par}(b, c), \mathit{so}(c, d), \mathit{so}(d, c), \mathit{par}(e, d), \mathit{par}(f, d), \mathit{par}(g, f)\}$. We can apply (4) by mapping variables $x \mapsto b$, $z \mapsto c$, $z' \mapsto d$, and $y \mapsto e$, which lets us infer $\mathit{kin}(b, e)$.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2026 Joint Conference (March 24-27, 2026), Tampere, Finland
raimund.dachselt@tu-dresden.de (R. Dachselt); lukas.gerlach@tu-dresden.de (L. Gerlach); philipp.hanisch1@tu-dresden.de (P. Hanisch); alex.ivliev@tu-dresden.de (A. Ivliev); markus.kroetzsch@tu-dresden.de (M. Krötzsch); maximilian.marx@tu-dresden.de (M. Marx); julian.mendez2@tu-dresden.de (J. Méndez)
0000-0002-2176-876X (R. Dachselt); 0000-0003-4566-0224 (L. Gerlach); 0000-0003-3115-7492 (P. Hanisch); 0000-0002-1604-6308 (A. Ivliev); 0000-0002-9172-2601 (M. Krötzsch); 0000-0003-1479-0341 (M. Marx); 0000-0003-1029-7656 (J. Méndez)

Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

As rule systems have matured to support large graphs (e.g., >100M edges on a laptop, >8B edges on a server [5]), an important open challenge remains *explainability* of results. In spite of their declarative nature, recursive rules can be difficult to understand and debug. A classical explanation approach for Datalog are *proof trees* (a.k.a. *derivation trees*), where inferred facts have exactly the premises of the applied rule as children [13, 14]. However, such trees can only explain one fact at a time, and even this is only supported by few systems [4, 5]. Recent works that aim at obtaining *all* possible derivations for a fact (i.e., its *provenance*) still focus on single facts [15, 16, 17, 18]. Even more problematic, modern systems support *aggregation* (e.g., COUNT, SUM, MIN, MAX), such that even a single inference may rely on thousands of premises. Existing approaches do not account for how to represent aggregation within proof trees, and how to practically handle the relevant scale.

In this paper, we therefore develop a new approach for explaining results of Datalog with (stratified) aggregates, and a prototype implementation that illustrates its feasibility. After clarifying preliminaries on Datalog with aggregates (Section 2), the starting point of our work is an extended notion of proof tree, or rather proof graph, which includes witnesses for all elements in sets over which aggregates have been computed (Section 3). This proof graph might be overwhelmingly huge, so we develop ways to select and summarise relevant content (Section 4). The main idea is to combine many individual proofs into a single tree structure that shows their common structure and has sets of inferences at each node. We define two types of *queries* for summarising proof graphs: *type 1 queries* start from a set of inferences and find their maximal common proof structures, and *type 2 queries* start from a given proof structure and find inferences that were produced in this way.

These queries form the basis of a prototype implementation for interactive proof graph exploration, which we develop by extending the Nemo graph rule engine [5]. Our algorithmic approach is explained in Section 5 and evaluated for (proof) query answering performance in Section 6. We finish with a brief overview of how this backend functionality is then used in an interactive user interface (Section 7), which can also be explored online at https://tools.iccl.inf.tu-dresden.de/nemo/tgd-2026.

All of our tools are free and open source (see https://github.com/knowsys/nemo, https://github.com/knowsys/nemo-web, and https://github.com/imldresden/nev).

2. Preliminaries

We start by introducing the syntax and semantics of the extension of Datalog that we consider in this work. A more gentle introduction can be found in recent literature [1].

Syntax We consider countably infinite, mutually disjoint sets of *predicates* $\mathbf{P}$, *constants* $\mathbf{C}$, and *variables* $\mathbf{V}$, where $\mathbf{C}$ contains all integer numbers (and possibly more).[1] A *term* t is a constant or a variable, i.e., $t \in \mathbf{C} \cup \mathbf{V}$. Every predicate symbol $p \in \mathbf{P}$ has an *arity* $\mathrm{ar}(p) \in \mathbb{N}$ (possibly 0).

An *atom* has the form $p(t_1, \dots, t_\ell)$ for predicate $p \in \mathbf{P}$ and terms $t_1, \dots, t_\ell$ with $\ell = \mathrm{ar}(p)$. List of terms are denoted in bold, e.g., $\boldsymbol{t} = t_1, \dots, t_{|\boldsymbol{t}|}$, so we may write the previous atom as $p(\boldsymbol{t})$. Similarly, $\boldsymbol{x}$, $\boldsymbol{y}$, etc. denote lists of variables. We treat lists as sets when order is irrelevant. A logical expression is *ground* if it does not contain variables.

Our syntax for aggregation is inspired by ASP engines like Clingo [19]. We consider the set of *aggregation functions* $\mathbf{F} = \{\mathsf{COUNT}, \mathsf{MAX}, \mathsf{MIN}, \mathsf{SUM}\}$ – further aggregates can easily be added without affecting our results. An *aggregation atom* is an expression of the form

$$r = f\{\boldsymbol{t} : S_1, \dots, S_\ell\}, \tag{5}$$

where $r \in \mathbf{V}$ is the *result variable*, $\boldsymbol{t} \subseteq \mathbf{C} \cup \mathbf{V}$ a list of terms, $f \in \mathbf{F}$ an aggregation function, and $S_1, \dots, S_\ell$ a list of atoms, known as *side conditions*. The part in $\{\dots\}$ is the *aggregation set expression*. A *rule* ϱ is of the form

$$H \leftarrow B_1, \dots, B_n, A_1, \dots, A_m, \tag{6}$$

where $n, m \geq 0$, H and $B_1, \dots, B_n$ are atoms, and $A_1, \dots, A_m$ are aggregation atoms. The *outer variables* of ϱ are all variables that occur in an atom B_i or as result variable in an aggregation atom A_j. H is the *head* and $B_1, \dots, B_n, A_1, \dots, A_m$ the *body* of ϱ. We further require: (a) every variable in H is an outer variable; (b) result variables of aggregation atoms do not occur in aggregation set expressions; (c) for aggregation atoms of the form (5), every variable in $\boldsymbol{t}$ occurs in the side conditions or is an outer variable. Negation will later be introduced as syntactic sugar.

Example 2. We extend Example 1 with the following rule that finds parents with multiple children (*mC*):

$$\begin{aligned} &mC(x) \leftarrow par(y, x), \\ &\qquad c = \mathsf{COUNT}\{z : par(z, x)\}, c \geq 2 \end{aligned} \tag{7}$$

Here, the aggregation atom $c = \mathsf{COUNT}\{z : par(z, x)\}$ counts the number of children z for each parent x, and stores it in the result variable c. The set of outer variables is $\{x, y, c\}$.

A *program* P is a finite set of rules that is *stratified* in the following sense. There is a mapping $L : \mathbf{P} \to \mathbb{N}$ such that, for every rule $\varrho \in P$ of the form (6) with $H = p(\boldsymbol{t})$

1. $L(p) \geq L(q)$ for every atom $q(\boldsymbol{s}) \in \{B_1, \dots, B_n\}$,
2. $L(p) > L(q)$ for every atom $q(\boldsymbol{s}) \in \{S_1, \dots, S_\ell\}$ occurring in the side condition of some $A_1, \dots, A_m$.

Any such L partitions P into disjoint *strata*, where the *stratum* for $i \in \mathbb{N}$ is $P_i = \{p(\boldsymbol{t}) \leftarrow \mathcal{B} \in P \mid L(p) = i\}$.

[1] We adopt a *weak typing* approach where constants of different datatypes are part of a joint domain, as is typical for schema-less graph languages like RDF.

Semantics Variable-free (ground) atoms are also called *facts*, and a *database* is a finite set of facts. Programs are evaluated over databases to obtain newly inferred facts as output.[2] Aggregation functions are applied to sets of tuples. Let $S \subseteq \mathbf{C}^k$ be a set of k-tuples of constants. We associate the following partial function $\bigcup_{k \geq 1} \mathbf{C}^k \to \mathbf{C}$ with each aggregation function:

$$\mathsf{MIN}(S) = \min\{n \mid \langle n, t_2, \dots, t_k\rangle \in S, n \in \mathbb{N}\} \tag{8}$$
$$\mathsf{MAX}(S) = \max\{n \mid \langle n, t_2, \dots, t_k\rangle \in S, n \in \mathbb{N}\} \tag{9}$$
$$\mathsf{SUM}(S) = \textstyle\sum_{\langle n, t_2, \dots, t_k\rangle \in S, n \in \mathbb{N}} n \tag{10}$$
$$\mathsf{COUNT}(S) = |S| \tag{11}$$

Lines (8)–(10) consider only those tuples in S that have a natural number as their first component. We let $\min \emptyset$ and $\max \emptyset$ be undefined, so $\mathsf{MIN}(S)$ and $\mathsf{MAX}(S)$ might also be undefined: rules will not be applicable in cases where this occurs. The $\sum$ in (10) iterates over all tuples, so distinct tuples with the same number n in their first component will lead to n being added multiple times. This is important since Datalog semantics is based on sets (not multisets): if we would project to a set that contains only the numbers to add, duplicates would be eliminated, and each value could only be considered once. The sum over the empty collection is 0, so $\mathsf{SUM}(S)$ is always defined.

An *assignment* μ for a rule ϱ maps variables x in ϱ to constants $\mu(x) \in \mathbf{C}$. For a list of terms $\boldsymbol{t}$, let $\mu(\langle t_1, \dots, t_k\rangle) = \langle \mu(t_1), \dots, \mu(t_k)\rangle$ where $\mu(t) = t$ if μ does not contain t in its domain. We extend this notation to (aggregation) atoms. Given an atom $p(\boldsymbol{t})$, we set $\mu(p(\boldsymbol{t})) = p(\mu(\boldsymbol{t}))$. Let A be an aggregation atom of form (5). Then we define $\mu(A)$ as $\mu(r) = f\{\mu(\boldsymbol{t}) : \mu(S_1), \dots, \mu(S_\ell)\}$. We will also use $\mu(\mathcal{E})$ for an aggregation set expression $\mathcal{E}$ in a similar way.

A rule ϱ is *applicable* over a given database $\mathcal{D}$ if it has a *match*, defined next. Given an assignment μ for ϱ and an aggregation atom $r = f\{\boldsymbol{t} : S_1, \dots, S_\ell\}$ in ϱ, we write $\{\boldsymbol{t} : S_1, \dots, S_\ell\}^\mu$ for the set of all tuples of the form $\mu'(\boldsymbol{t})$ where the μ' is an extension of μ to all variables in $S_1, \dots, S_\ell$ such that $\mu'(S_i) \in \mathcal{D}$ for all $i \in \{1, \dots, \ell\}$. In other words, $\mathcal{E}^\mu$ is the set to which the aggregation set expression $\mathcal{E}$ evaluates under the given bindings for outer variables. Let μ be an assignment for ϱ whose domain is exactly the set of outer variables of ϱ. Then μ is a *match* for ϱ over $\mathcal{D}$ if, for every:

1. body atom B of ϱ, $\mu(B) \in \mathcal{D}$, and
2. aggregation atom $r = f\{\mathcal{E}\}$ of ϱ, $\mu(r) = f(\mathcal{E}^\mu)$.

Note that condition 2 requires f to be defined for $\mathcal{E}^\mu$.

A rule ϱ with head H is *applicable* to $\mathcal{D}$ if it has a match μ over $\mathcal{D}$ and $\mu(H) \notin \mathcal{D}$. The application of ϱ under μ then results in $\mathcal{D} \cup \{\mu(H)\}$.

A program P is evaluated based on some stratification L. We assume without loss of generality that the non-empty strata of L are exactly $P_1, \dots, P_\ell$. Let $\mathcal{D}_0^\infty = \mathcal{D}$. For $i = 0, \dots, \ell - 1$, let $\mathcal{D}_{i+1}^\infty$ be the database resulting in an arbitrary maximal sequence of applications of rules in P_{i+1} starting from $\mathcal{D}_i^\infty$. The *result* $P(\mathcal{D})$ of P over $\mathcal{D}$ is $P(\mathcal{D}) = \mathcal{D}_\ell^\infty$. It is a standard fact in logic programming (and easy to verify) that (a) every $\mathcal{D}_i^\infty$ is finite, (b) $\mathcal{D}_{i+1}^\infty$ does not depend on the order of rule applications, and (c) $P(\mathcal{D})$ does not depend on the chosen stratification.

[2] In some contexts, it is useful to designate *input* and *output* predicates that define the intended "interface" of a program [1], but this is not relevant to our present work.

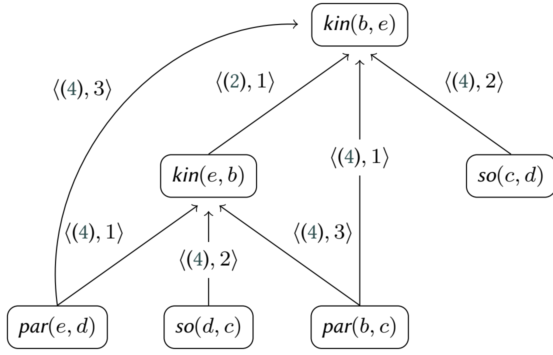

Figure 1: Derivation graph for Example 1

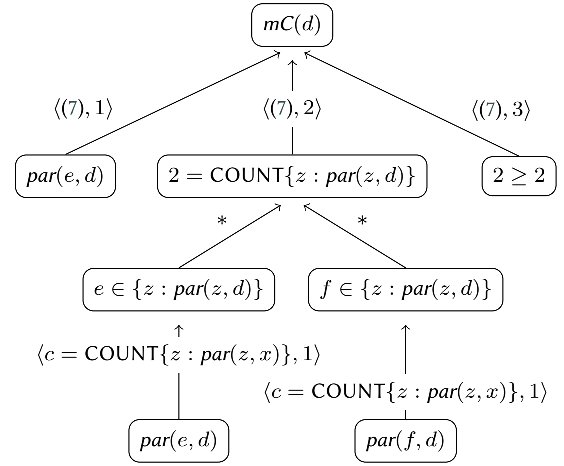

Figure 2: Proof graph for Example 2

Negation and Builtins It is not hard to extend our syntax and semantics to support *builtin* predicates for which defining facts are assumed given without being specified in the database. For example, a unary predicate $b_{\cdot=0}$ can be defined by the single fact $b_{\cdot=0}(0)$ (we would usually prefer infix notation, as in $x = 0$). There can also be infinite builtins, such as the binary $<$; in Datalog extensions, variables that occur with such infinite builtins are typically required to also occur in regular (finite) body atoms [1].

Using just a single finite builtin $\cdot = 0$, we can express (stratified) negation. Indeed, a negated atom $\neg p(\boldsymbol{t})$ can be rewritten into two atoms $x = \mathsf{COUNT}\{\boldsymbol{t} : p(\boldsymbol{t})\}$ and $x = 0$, where x is a fresh variable. We allow $\boldsymbol{t}$ to contain variables that are not used anywhere else: they are "inner" variables in the side condition, interpreted existentially. This is also how rule engines like Nemo and Clingo interpret such variables in negated atoms.

3. Proof Graphs

Every fact that is produced by a program P over a database $\mathcal{D}$ has at least one *proof*, which we will now define more concretely. The structure of proofs contains valuable information for explaining results, debugging programs, and certifying correctness [20]. While *proof trees* (or *derivation trees*) are common in logic programming, we must extend them here to cover aggregation. Moreover, we now consider directed acyclic graphs instead of mere trees, leading to a compact representation of proofs for all inferred facts.

To clarify the role of facts as preconditions of specific rule applications, we refer to specific body atoms in rules: a *body atom position* is a pair $\langle \varrho, i\rangle$ where ϱ is a rule as in (6) containing n atoms and m aggregation atoms, and $1 \leq i \leq m + n$. Likewise, for an aggregation atom A of form (5) with ℓ side conditions, a *side condition atom position* is a pair $\langle A, j\rangle$ with $1 \leq j \leq \ell$. The set of all atom positions (of either kind) of a program P is denoted $\mathsf{APos}(P)$. To denote information about how aggregation atoms were evaluated, we introduce *evaluated aggregation atoms* of the form $c = f\{\boldsymbol{t} : S_1, \ldots, S_\ell\}$ where $c \in \mathbf{C}$ and all other pieces are as in (5). The general shape of the proof structures we consider is as follows:

Definition 1. A *derivation graph* $G = \langle V, E, \lambda\rangle$ for program P is a directed acyclic graph with the components:

- vertex set V, where each $v \in V$ is either a ground atom $p(\boldsymbol{c})$, an evaluated aggregation atom $c = f\{\mathcal{E}\}$, or an expression $\boldsymbol{c} \in \{\mathcal{E}\}$ with $\boldsymbol{c}$ a list of constants and $\mathcal{E}$ an aggregation set expression,
- edge set $E \subseteq V \times V$, and
- edge labelling function $\lambda : E \to \mathsf{APos}(P) \cup \{*\}$.

Intuitively, each node of the derivation graph represents a derivation result, and its children represent the data that it was derived from. For regular ground atoms, these will be the instantiated body atoms of the relevant rule application (edges labelled by body atom position); for evaluated aggregation atoms, they will be expressions $\boldsymbol{c} \in \{\mathcal{E}\}$ for all elements of the evaluated aggregation set expression (edges labelled by $*$); and for expressions $\boldsymbol{c} \in \{\mathcal{E}\}$, they will be the instantiated atoms of the relevant side conditions (edges labelled by side condition atom position). Figure 1 and Figure 2 show examples of derivation graphs for Example 1 and Example 2, respectively. Note that $\mathit{kin}(b, e)$ has two different proofs in Figure 1 (either by applying rule (2) or rule (4)). Here, we are primarily interested in proofs corresponding to the concrete rule applications that occurred during program evaluation, motivating our next definition.

Definition 2. Consider a rule ϱ that was applied to a database $\mathcal{D}$ using a match μ. The *proof graph* for this rule application is the derivation graph G that contains exactly the following labelled edges (and associated vertices):

(A) if A is the jth (aggregation or regular) body atom of ϱ, and H is the head of ϱ, then G contains the edge

$$\mu(A) \xrightarrow{\langle \varrho, j\rangle} \mu(H),$$

where $\mu(A)$ denotes the expression obtained from A by replacing each (necessarily outer) variable x by $\mu(x)$.

(B) if A is an aggregation atom of form (5) where $\mathcal{E}$ is its aggregation set expression, then, for every $\boldsymbol{c} \in \mathcal{E}^\mu$, G contains the edge

$$\boldsymbol{c} \in \{\mu(\mathcal{E})\} \xrightarrow{*} \mu(A).$$

Moreover, if μ' is the extension of μ that was used to establish $\boldsymbol{c} \in \mathcal{E}^\mu$,[3] then G contains the edges

$$\mu'(S_p) \xrightarrow{\langle A, p\rangle} \boldsymbol{c} \in \{\mu(\mathcal{E})\} \quad \text{for each } p \in \{1, \ldots, \ell\}.$$

[3] Like for rule applications, we assume that every tuple considered in an aggregation was found in exactly one way, even if other ways might exist. This assumption is needed to obtain a unique proof.

Definition 3. The *proof graph* for a sequence of rule applications of a program P starting from a database $\mathcal{D}$ is the union of the proof graphs of all individual rule applications in the sequence.

Since rule application sequences are restricted to use previously derived facts, and since stratification ensures that any aggregation atom is only evaluated over side conditions of lower strata, we immediately get the following.

Proposition 1. *The proof graph of any sequence of rule applications is a derivation graph, and in particular it is acyclic.*

Example 3. Recall the rule for finding parents with multiple children from Example 2:

$$mC(x) \leftarrow par(y,x), \quad c = \mathsf{COUNT}\{z : par(z,x)\}, c \geq 2 \tag{7}$$

On the database from Example 1, consider the assignment μ with $x \mapsto d$, $y \mapsto e$, and $c \mapsto 2$. We find extensions μ' with $\mu'(z) = e$ and μ'' with $\mu''(z) = f$ of μ that witness $\{d,e\} \subseteq \{z : par(z,x)\}^{\mu}$. Indeed, these are the only two extensions, and thus μ is a match for rule (7), resulting in $mC(d)$. Figure 2 shows the corresponding proof graph.

4. Querying for Proof Summaries

In this section, we introduce conceptual means for users to select and summarise parts of proof graphs, which will form the basic backend operations in our implementation.

A proof graph for a maximal chain of rule applications offers a complete account of all derivations in $P(\mathcal{D})$, but it is very hard to inspect for real-world graph databases with millions of edges. Indeed, even a proof graph with a few hundred nodes can be hard to navigate or display. For structural simplification, we can "unravel" the directed acyclic graph into a tree if we recursively replace vertices with several outgoing edges by multiple nodes that each have one outgoing (parent) edge. After this operation, the proof graph is a disjoint union of trees, one for each fact in $P(\mathcal{D})$ that was not used in any rule application, and we might well allow users to select one such tree or a subtree of it (starting at a fact of interest). This is the classical "proof tree" view, extended with aggregates.

This classical view, however, neglects the set-based nature of rule-based computation and hides any regularities that emerge from this. Indeed, thousands of facts might be derived in exactly the same way, merely starting from different inputs, but viewing a thousand trees individually to compare them is not feasible. We therefore propose a *summary proof tree* that combines many (unravelled) proof trees with a similar structure. Nodes in such trees then are labelled by sets of vertices that occur in similar positions in the overall proof graph.

Definition 4. Let $G = \langle V, E, \lambda \rangle$ be a proof graph for a program P. A *summary proof tree* $\langle N, T, \sigma, \theta \rangle$ for G is given by a set N of nodes, a set $T \subseteq N \times N$ of edges that form a tree, a node labelling function $\sigma : N \to 2^V$, and an edge labelling function $\theta : T \to \mathsf{APos}(P) \cup \{*\}$, where we require that all edges that lead to the same parent node have mutually distinct edge labels.

The requirement of distinct edge labels is different from proof graphs, where evaluated aggregate atoms may have many children with the same label $*$. Merging these children into one set helps us to combine proof trees, as defined next. Figure 3 depicts the corresponding summary proof tree for the proof graph in Figure 2.

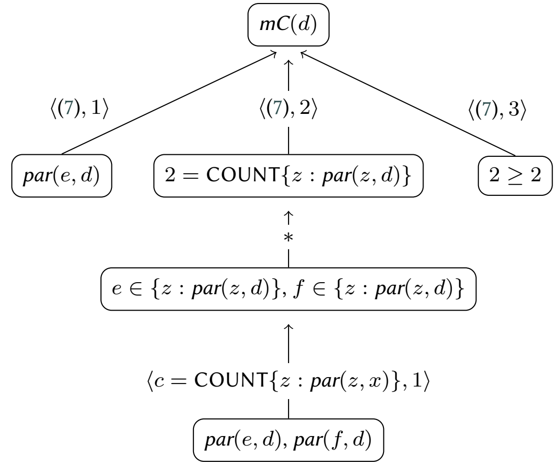

Figure 3: Summary proof tree for Example 2

Definition 5. Let $G = \langle V, E, \lambda \rangle$ be a proof graph. For a vertex $v \in V$, let $\text{in-labels}(v) = \{\lambda(\langle w, v\rangle) \mid \langle w, v \rangle \in E\}$ and $\text{children}(v, L) = \{w \mid \langle w, v\rangle \in E,\ \lambda(\langle w, v\rangle) = L\}$ for a label L.

Given $F \subseteq V$, the *summary proof tree for* F, denoted $\mathcal{S}(F)$, is defined recursively as follows:

- The root of $\mathcal{S}(F)$ is a node n with label $\sigma(n) = F$.
- If $\text{in-labels}(v)$ is the same set for all $v \in F$, then, for every $L \in \text{in-labels}(v)$, there is an edge $m_L \xrightarrow{L} n$ such that m_L is the root of a summary proof tree for $\bigcup_{w \in F} \text{children}(w, L)$.

Intuitively, the summary proof tree determines the maximal common upper portion of the proof trees for the elements of F, labelling each node by the set of all vertices that the individual proof trees use in this position. For sets of ground facts F, the requirement of uniform $\text{in-labels}(v)$ is satisfied exactly if the same rule was applied to derive all facts in F (for sets of vertices from aggregation atoms or aggregated tuples, the requirement always holds). If no aggregates are used, the summary proof tree of a singleton set $\{p(\boldsymbol{c})\}$ corresponds to the classical proof tree for $p(\boldsymbol{c})$. If present, however, the $*$-children of aggregate atoms are combined into a single node with a set of labels, whose children correspond to the aggregate's side conditions, which in turn might have been derived in more than one way.

Definition 5 provides a powerful way to discover similarities in proof structures for many facts. Conversely, it is also useful to start from a proof structure of interest and find all the facts that were derived in a similar way.

Definition 6. Let $G = \langle V, E, \lambda \rangle$ be a proof graph. A *summary tree query* $Q = \langle N, T, \sigma, \theta \rangle$ has the same form as a summary proof tree (Definition 4), but with an additional possible value $\top$ ("no restriction") for node labels.

Let $v \in V$ be a vertex and $\mathcal{S}(\{v\}) = \langle N_v, T_v, \sigma_v, \theta_v \rangle$ its summary proof tree. Then v *structurally matches* Q if there is a mapping $\iota : N \to N_v$ such that (a) the root of Q is mapped to the root of $\mathcal{S}(\{v\})$; (b) if $\langle n, m \rangle \in T$, then $\theta(\langle n, m\rangle) = \theta_v(\langle \iota(n), \iota(m)\rangle)$; and (c) for every $n \in N$, either $\sigma(n) = \top$ or $\sigma_v(\iota(n)) \subseteq \sigma(n)$.

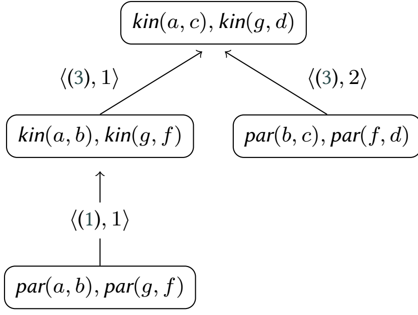

Figure 4: Summary proof tree for $F = \{kin(a, c), kin(g, d)\}$

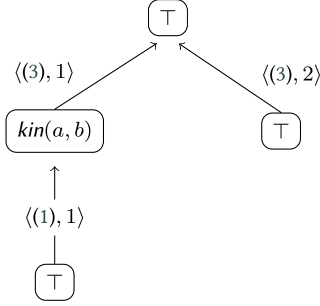

Figure 5: Summary tree query

Let F be the set of all vertices of G that structurally match Q. The *result of* Q over proof graph G, denoted $\mathcal{R}(Q)$, is obtained from the summary proof tree $\mathcal{S}(F)$ by deleting all nodes for which there is no corresponding node in Q.

In other words, $\mathcal{R}(Q)$ always has the same tree structure and edge labels as Q, but possibly different node labels, which are subsets of the upper bounds given in Q.

Example 4. Consider the program from Example 1. The summary proof tree $\mathcal{S}(F)$ for $F = \{kin(a, c), kin(g, d)\}$ is shown in Figure 4. Starting with $F = \{kin(a, c), kin(e, d)\}$ instead would result in a tree containing only the root node with label F, since $kin(a, c)$ is derived from rule (3) and $kin(e, d)$ from rule (1).

Now consider the summary tree query Q that is shown in Figure 5 with the same structure. The left child of the root is labelled by $kin(a, b)$, so only $kin(a, c)$ structurally matches the query, but not $kin(g, d)$. Therefore, the answer $\mathcal{R}(Q)$ only contains facts relevant for the derivation of $kin(a, c)$.

We refer to the queries of Definition 5 (based on vertex set F) as *type 1 queries* (*T1*), whereas those of Definition 6 (based on summary tree query Q) are *type 2 queries* (*T2*). Together, they provide the two main ways for users to interact with proofs in our approach. It is important to note that answers for both query types are based on the concrete matches used during program evaluation, rather than all possible derivations of a fact. As a result, a query might have no answer even though other sequences of rule applications could have produced one.

Type 1 queries can be answered in a straightforward way by considering the summary proof trees for each fact in F and merging them until they diverge. In the following, we therefore focus on the implementation of type 2 queries.

5. Answering Summary Tree Queries

In this section, we show how to answer summary tree queries using Datalog programs. We also discuss how to extend this to language features provided by modern rule engines, and describe our own implementation in Nemo [5].

Modern engines usually implement a semi-naive evaluation strategy for Datalog that applies a given rule to all matches without a corresponding head fact in a single step. We split a predicate p into predicates p^i, grouping all p-facts by the step i in which they were derived. We also assume auxiliary tables $F^i_{\rho,A}$ for each aggregation atom A of the form (5) occurring in rule ρ that, for step i, contain all tuples $\langle \mathbf{x}, c \rangle$, where $\mathbf{x}$ are the outer variables appearing in side conditions of A and c is the value of the result variable.

5.1. Implementation in Datalog

Conceptually, we can divide our approach for answering summary tree queries into two phases. First, we traverse the query bottom-up, computing at each node all facts satisfying the constraints imposed by its subtree. This may introduce facts not relevant to the final answer. In the second phase, we traverse the query top-down to eliminate such facts.

For the summary tree query $Q = \langle N, T, \sigma, \theta \rangle$, we use, for each node $v \in N$ and derivation step i, predicates T^i_v (containing all facts satisfying the constraints imposed by the subtree rooted at v), A^i_v (containing, for each fact in T^i_v, an assignment of the outer variables witnessing the derivation for this fact) and R_v (for the resulting node label of v). We write $T^{<i}_v$ for $\bigcup_{j<i} T^j_v$ (and similarly abbreviate $F^{<i}_{\rho,A}$). For node $v \in V$, we set N^p_v to be all p-facts in $\sigma(v)$ if $\sigma(v) \neq \top$, and to be the set of all derived p-facts otherwise.

For the bottom-up phase, we first consider node v with in-labels(v) only containing atom positions for rule $\rho = H \leftarrow B_1, \ldots, B_n, A_1, \ldots, A_m$ and child vertices w_i for each body atom B_i ($1 \leq i \leq n$). Let ℓ be a leaf node with associated predicate p_ℓ. We use the following rules, where $s(\mathbf{x}, \mathbf{y})$ specifies some arbitrary, but fixed map from $\mathbf{x}$ (outer variables occurring in the rule head) to $\mathbf{y}$ (outer variables not occurring in the rule head). We write $H_1, H_2 \leftarrow \mathcal{B}$ as an abbreviation for the two rules $H_1 \leftarrow \mathcal{B}$ and $H_2 \leftarrow \mathcal{B}$.

$$T^i_\ell(\mathbf{x}) \leftarrow p^i_\ell(\mathbf{x}), N_\ell(\mathbf{x}) \tag{12}$$

$$\begin{aligned} A^i_v(\mathbf{x}, \mathbf{y}), T^i_v(\mathbf{x}) \leftarrow\ & H^i(\mathbf{x}), N_v(\mathbf{x}), s(\mathbf{x}, \mathbf{y}), \\ & T^{<i}_{w_1}(\mathbf{x}, \mathbf{y}), \ldots, T^{<i}_{w_n}(\mathbf{x}, \mathbf{y}) \\ & F^{<i}_{\rho,A_1}(\mathbf{x}, \mathbf{y}, c), \ldots, F^{<i}_{\rho,A_m}(\mathbf{x}, \mathbf{y}, c) \end{aligned} \tag{13}$$

For aggregation, we consider vertices v with an outgoing edge labelled by $\langle \rho, i \rangle$ referring to aggregation atom A. Let w be the unique child with $w \xrightarrow{*} v$ and $w_1, \ldots, w_n$ be the child nodes of w. We add the following rules:

$$A^i_v(\mathbf{x}, c), T^i_v(\mathbf{x}, c) \leftarrow F^i_{\rho,A}(\mathbf{x}, c) \tag{14}$$

$$\begin{aligned} A^i_w(\mathbf{x}, \mathbf{y}), T^i_w(\mathbf{x}) \leftarrow\ & F^i_{\rho,A}(\mathbf{x}, c), s(\mathbf{x} \cup \mathbf{t}, \mathbf{y}) \\ & T^{<i}_{w_1}(\mathbf{x}, \mathbf{y}), \ldots, T^{<i}_{w_n}(\mathbf{x}, \mathbf{y}) \end{aligned} \tag{15}$$

The rules for the top-down phase are as follows, where d is the total number of derivation steps and r is the root of T:

$$R_r(\mathbf{x}) \leftarrow T^{<d+1}_r(\mathbf{x}) \tag{16}$$

$$R_w(\mathbf{x}) \leftarrow R_v(\mathbf{x}), A^{<d+1}_v(\mathbf{x}, \mathbf{y}), T^{<d+1}_w(\mathbf{x}, \mathbf{y}) \tag{17}$$

Table 1
Summary of our evaluation, showing for each rule set (from left to right): the amount of rules and number of derived facts, the average (and maximal) node count per query and the number of queries, the average (and maximal) number of answers per query, the average (and maximal) runtime in milliseconds for answering the queries, percentages of runtimes under 1s and 5s, the amount of time in milliseconds for the baseline implementation to complete tracing and matching

Rule Set	Rules	Derivations	Query size		Queries	Answers		Time (in ms)		$\leq 1\,\mathrm{s}$	$\leq 5\,\mathrm{s}$	Baseline (in ms)	
			mean	max		mean	max	mean	max			Tracing	Matching
Galen	12	1,881,946	16.2	(43)	1000	55K	(1.2M)	1,713	(7,212)	58%	91%	t.o.	t.o.
Doctors-1M	5	792,500	2.4	(3)	5	317K	(1.1M)	661	(2,361)	80%	100%	38,822	2,845
ONT-256	529	5,673,985	2.1	(3)	636	20K	(40K)	17	(81)	100%	100%	t.o.	t.o.
LUBM-100	136	22,317,699	3.4	(5)	134	520K	(5.8M)	173	(1,729)	98%	100%	t.o.	t.o.
Deep-100	1100	20,262	3.2	(10)	1000	0.1K	(0.8K)	1	(2)	100%	100%	496	125
Deep-200	1200	982,501	4.0	(10)	1000	9K	(122K)	12	(129)	100%	100%	t.o.	t.o.

5.2. Implementation in Nemo

Our implementation in Nemo uses the database library *nemo-physical*, a translation of the above rules into relational algebra, and a built-in selection operator for $s(\mathbf{x}, \mathbf{y})$.

We extend the theoretical approach to all language features used in Nemo, including existential rules and arithmetic operations. The latter are evaluated in the bottom-up phase as they would be during rule applications. Existential rules extend Datalog with existentially quantified variables in rule heads. Nemo uses the restricted chase [21] to evaluate such rules. The algorithm already supports such rules. To avoid a Cartesian product in rule (13) if all variables in the head atom are existentially quantified, we replace rule (13) by the following rule, where p is the predicate of the head atom:

$$A_v^i(\mathbf{x}), T_v^i(\mathbf{x}) \leftarrow H^i(\mathbf{x}), N_v^p(\mathbf{x}) \tag{18}$$

In some cases, we can reuse tables computed during program execution. Node v in a summary tree query $Q = (N, T, \sigma, \theta)$ is *simple* if $\sigma(v) = \top$ and v is (a) a leaf node, or (b) an interior node such that in-labels(v) contains only body atom positions for rule ρ with head atom H that does not occur as the head atom of another rule, and all children of v are also simple. If v is simple and in-labels(v) only has body atom positions for rule ρ such that all outer variables of ρ occur in the head of ρ, we have $A_v^i(\mathbf{x}) = T_v^i(\mathbf{x}) = H^i(\mathbf{x})$.

6. Performance Evaluation

This section presents an evaluation of our approach. We demonstrate that (a) the proposed implementation outperforms a naive solution based on Nemo's existing tracing functionality, and (b) the proposed implementation achieves performance sufficient for real-time applications.

6.1. Experimental Setup

We base our performance evaluation on the benchmarks used by Ivliev et al. [5] that are publicly available.[4] We replace LUBM-01k with LUBM-100 due to memory restrictions. The first section of Table 1 lists all the considered benchmarks together with the number of rules and the number of derived facts for each rule set. Galen is a Datalog implementation of EL-reasoning run on a medical ontology [22], while the rest are commonly used benchmarks for existential rules [21].

To generate the queries, we compute summary proof trees for a randomly chosen subset of all derived facts and construct the queries by omitting the fact restrictions. We limit the experiments to 1,000 queries per rule set. The second section of Table 1 shows the average node count of the generated queries and the amount of queries per rule set. With 16 nodes on average, Galen has large queries, while the other data sets average query sizes between 2 and 4.

We further compare our approach with a naive baseline implementation that uses the existing tracing implementation of Nemo. To answer proof queries, we first compute the proof trees for all derived facts. Note that Nemo optimises the computation of multiple traces by reusing previously computed results. We then match each proof tree to the provided query.

The runtime measurements for each experiment are repeated three times. Between each run, the state of the database was reset. All experiments were conducted on consumer hardware using a machine with an AMD Ryzen 5900X CPU and 32 GB of RAM. The individual measurements are provided in the supplementary material.[5] For the baseline experiments, we set a timeout (t.o.) of 10 minutes.

6.2. Results and Discussion

The third section of Table 1 shows the performance of our implementation, including the average number of answers per query, the average runtime in milliseconds, along with the percentage of queries answered in less than 1s and less than 5s, which we classify as "fast" and "acceptable" from a user perspective, respectively. The last section displays the average runtimes for the baseline implementation in milliseconds, divided into the time for computing the proof trees for all facts (Tracing) and the time for matching each proof tree against the query (Matching). Comparing our approach with the baseline, we notice a significant improvement. For Doctors-1M and Deep-100, our implementation achieves a speedup of approximately two orders of magnitude. For the remaining cases, the baseline implementation reaches the 10-minute timeout, whereas our implementation completes in under 5 seconds for the majority of instances. The baseline's performance scales with the number of derived facts, making it infeasible for most of the rule sets evaluated here.

We now discuss the differences in the runtimes for our implementation. For the existential rules benchmarks, all queries, except one in Doctors-1m and three in LUBM-100,

[4]https://github.com/knowsys/nemo-examples/tree/main/evaluations/kr2024

[5]https://github.com/knowsys/nemo-examples/tree/main/evaluations/tgd2026

Nemo Graph Rule Engine   Examples   Help

```
1   par(a, b). par(b, c). sp(c, d).
2   par(e, d). par(f, d).
3
4   kin(?x, ?y) :- par(?x, ?y).
5   kin(?x, ?y) :- par(?x, ?z),
6       sp(?z, ?z2), par(?y, ?z2).
7   kin(?x, ?y) :- kin(?y, ?x).
8   kin(?x, ?y) :- kin(?x, ?z), kin(?z, ?y).
9   parCount(#count(?y), ?x) :- par(?y, ?x).
10  mC(?x) :- parCount(?c, ?x), 2 <= ?c.
```

Figure 6: Nemo Code for Example 2

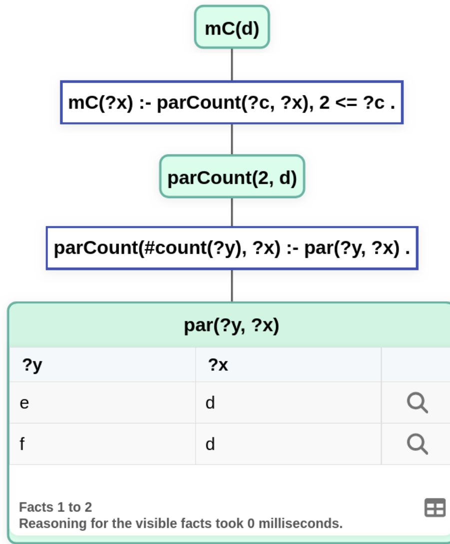

Figure 7: Nev Trace corresponding to Figure 2

can be evaluated in under 1s. For GALEN, 58% of requests can be answered in under 1s, and 91% of requests in under 5s. We attribute the differences in performance to two main factors. First, facts derived in the GALEN benchmarks usually have large proof trees, which naturally increases the time required to answer the corresponding proof queries. Second, within the existential rules benchmarks, the derived facts are approximately evenly distributed across the possible proof query shapes (see supplementary material). This makes the queries more selective, so that fewer facts need to be considered during computation. Meanwhile, in GALEN, we observe an uneven distribution, such that most inferences may be recomputed during the bottom-up phase.

Overall, we conclude that our prototype implementation is capable of answering complex queries for large-scale reasoning tasks efficiently in the majority of the cases, while improving significantly over the baseline implementation.

7. Query Visualisation and Editing

The web version of Nemo [5] integrates a visualisation tool called *nev* (*Nemo Explain Visualizer*[6], inspired by *pev2* [23]) that supports analysis and creation of *T1* and *T2* queries.

T1 queries function as *nev*'s entry point for building a query. The resulting summary proof tree can be manipulated by, e.g., adding or removing rules to issue *T2* queries, which updates the facts in each tree node accordingly.

A *T1* query initialising *nev* can be issued by clicking the magnifying glass next to a row in one of Nemo's result tables. Then, a *T2* query with the same tree shape can be issued with the "unrestrict" button. Also, one can manipulate the current tree shape by, e.g., adding or removing rules. Any such operation will trigger a summary tree query and update the node's contents accordingly. Thus, arbitrary tree queries can be constructed.

In Figure 6, we present Example 2 in Nemo. Since Nemo's syntax for aggregates is different from what we introduce in our theoretical considerations, we are forced to introduce an auxiliary rule for the aggregate storing the number of children for each parent on a new relation *parCount*. We can then trace the fact $mC(d)$ in *nev* as can be seen in Figure 7. This shows a tree resembling Figure 3.

8. Conclusions

We present *summary proof trees* as a means to explain derivations (i.e., traces) of Datalog programs with aggregates. To navigate these potentially large structures, we also introduce *summary tree queries*, which overlay multiple partial derivations to allow the inspection of all derivations with a common shape. We enhance the reasoning engine Nemo with the capability of answering such queries and evaluate the feasibility of the approach using standard benchmarks. Furthermore, we show *nev* as a visualization tool for summary proof trees and as an interactive builder for summary tree queries.

An important direction for future work is the explanation of the absence of facts, which is a common use case for debugging Datalog programs. There exist semi-automated methods that explain missing facts by guiding users to select relevant rules and then to continue from the resulting partial instantiation of the rule [18]. Adapting such techniques to our setting raises several challenges. For one, aggregation introduces additional forms of failure (e.g., empty aggregation, or the aggreagation result fails to meet a condition). Second, it would be interesting to investigate how summary proof trees can be used to explain missing facts.

Finally, we plan to conduct a user study to validate the utility of having such explanatory features for the end user.

Acknowledgments

This work is funded by Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy: EXC 2050/2, 390696704 – "Centre for Tactile Internet" (CeTI); by DFG grant 389792660 as part of TRR 248 – CPEC; by Bundesministerium für Bildung und Forschung (BMBF) and Saxon State Ministry for Science, Culture and Tourism (SMWK) in Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI, SCADS22B); and by BMBF and German Academic Exchange Service (DAAD) in project 57616814 (SECAI, School of Embedded and Composite AI).

Declaration on Generative AI

The authors have not employed any Generative AI tools.

[6] https://tools.iccl.inf.tu-dresden.de/nemo/tgd-2026

References

[1] M. Krötzsch, Modern Datalog: Concepts, methods, applications, in: A. Artale, M. Bienvenu, Y. I. García, F. Murlak (Eds.), Joint Proc. of the 20th and 21st Reasoning Web Summer Schools (RW'24 & RW'25), volume 138 of *OASIcs*, Dagstuhl Publishing, 2025. doi:10.4230/OASIcs.RW.2024/2025.7.

[2] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, RDFox: A highly-scalable RDF store, in: M. A. et al. (Ed.), Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II, volume 9367 of *LNCS*, Springer, 2015, pp. 3–20. doi:10.1007/978-3-319-25010-6_1.

[3] J. Urbani, C. Jacobs, M. Krötzsch, Column-oriented Datalog materialization for large knowledge graphs, in: D. Schuurmans, M. P. Wellman (Eds.), Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16), AAAI Press, 2016, pp. 258–264. doi:10.1609/aaai.v30i1.9993.

[4] H. Jordan, B. Scholz, P. Subotic, Soufflé: On synthesis of program analyzers, in: S. Chaudhuri, A. Farzan (Eds.), Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II, volume 9780 of *LNCS*, Springer, 2016, pp. 422–430. doi:10.1007/978-3-319-41540-6_23.

[5] A. Ivliev, L. Gerlach, S. Meusel, J. Steinberg, M. Krötzsch, Nemo: Your friendly and versatile rule reasoning toolkit, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24), IJCAI Organization, 2024, pp. 743–754. doi:10.24963/kr.2024/70.

[6] Log25, Logica project webpage, Logica Devs, Accessed: 2025-09-04. https://logica-web.github.io/.

[7] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1994.

[8] J. F. Sequeda, On the semantics of R2RML and its relationship with the direct mapping, in: E. Blomqvist, T. Groza (Eds.), Proc. ISWC 2013 Posters & Demonstrations Track (ISWC'13), volume 1035 of *CEUR WS Proceedings*, CEUR-WS, 2013, pp. 193–196. URL: https://ceur-ws.org/Vol-1035/iswc2013_poster_4.pdf.

[9] E. S. Skvortsov, Y. Xia, S. Bowers, B. Ludäscher, Logica-TGD: Transforming graph databases logically, in: M. Boehm, K. Daudjee (Eds.), Proc. Workshops of the EDBT/ICDT 2025 Joint Conf., volume 3946 of *CEUR WS Proceedings*, CEUR-WS, 2025. URL: https://ceur-ws.org/Vol-3946/TGD-4.pdf.

[10] A. Elhalawati, J. V. den Bussche, A. Dimou, A declarative formalization of R2RML using Datalog and its efficient execution, in: A. Margara, T. Kliegr, O. Savkovic, S. Ahmetaj, R. Tommasini, L. Bellomarini, E. Kharlamov, I. G. Ciuciu, D. Roman, G. Konstantinidis, E. Sallinger, A. Soylu (Eds.), Companion Proc. 9th Int. Joint Conf. on Rules and Reasoning (RuleML+RR'25), volume 4083 of *CEUR WS Proceedings*, CEUR-WS, 2025. URL: https://ceur-ws.org/Vol-4083/paper63.pdf.

[11] A. Elhalawati, A. Dimou, O. Hartig, D. Hernández, Flexible RML-based mapping of property graphs to RDF, in: M. Boehm, K. Daudjee (Eds.), Proc. Workshops of the EDBT/ICDT 2025 Joint Conf., volume 3946 of *CEUR WS Proceedings*, CEUR-WS, 2025. URL: https://ceur-ws.org/Vol-3946/TGD-2.pdf.

[12] A. Ivliev, M. Krötzsch, M. Marx, SPARQLing datalog for rule-based reasoning over large knowledge graphs, in: M. Acosta, M. van Erp, S. Rudolph, O. Hartig, B. Spahiu, A. Rula, D. Garijo, F. Osborne (Eds.), Proc. 23rd European Semantic Web Conf. (ESWC'26), LNCS, Springer, 2026. To appear.

[13] C. A. Wieland, Two explanation facilities for the deductive database management system DeDEx, in: H. Kangassalo (Ed.), Proc. 9th Int. Conf. Entity-Relationship Approach (ER'09), ER Institute, 1990, pp. 189–203.

[14] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, D. Srivastava, Explaining program execution in deductive systems, in: S. Ceri, K. Tanaka, S. Tsur (Eds.), Proc. 3rd Int. Conf. Deductive and Object-Oriented Databases (DOOD'93), volume 760 of *LNCS*, Springer, 1993, pp. 101–119. doi:10.1007/3-540-57530-8_7.

[15] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez, A theoretical framework for the declarative debugging of Datalog programs, in: K. Schewe, B. Thalheim (Eds.), Proc. 3rd Int. Workshop on Semantics in Data and Knowledge Bases (SDKB'08), volume 4925 of *LNCS*, Springer, 2008, pp. 143–159. doi:10.1007/978-3-540-88594-8_8.

[16] S. Köhler, B. Ludäscher, Y. Smaragdakis, Declarative datalog debugging for mere mortals, in: P. Barceló, R. Pichler (Eds.), Proc. 2nd Int. Workshop on Datalog in Academia and Industry (Datalog 2.0'12), volume 7494 of *LNCS*, Springer, 2012, pp. 111–122. doi:10.1007/978-3-642-32925-8_12.

[17] A. Elhalawati, M. Krötzsch, S. Mennicke, An existential rule framework for computing why-provenance on-demand for datalog, in: G. Governatori, A. Turhan (Eds.), Proc. 2nd Int. Joint Conf. on Rules and Reasoning (RuleML+RR'22), volume 13752 of *LNCS*, Springer, 2022, pp. 146–163. doi:10.1007/978-3-031-21541-4_10.

[18] D. Zhao, P. Subotić, B. Scholz, Debugging large-scale Datalog: A scalable provenance evaluation strategy, ACM Trans. Program. Lang. Syst. 42 (2020). doi:10.1145/3379446.

[19] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[20] J. Tantow, L. Gerlach, S. Mennicke, M. Krötzsch, Verifying datalog reasoning with Lean, in: Y. Forster, C. Keller (Eds.), Proc. 16th Int. Conf. Interactive Theorem Proving (ITP'25), volume 352 of *LIPIcs*, Dagstuhl Publishing, 2025. doi:10.4230/LIPICS.ITP.2025.36.

[21] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, E. Tsamoura, Benchmarking the chase, in: Proc. 36th Symp. on Principles of Database Systems (PODS'17), ACM, 2017, pp. 37–52. doi:10.1145/3034786.3034796.

[22] Y. Kazakov, M. Krötzsch, F. Simančík, The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies, J. of Automated Reasoning 53 (2013) 1–61. doi:10.1007/S10817-013-9296-3.

[23] Dalibo, Pev2, https://github.com/dalibo/pev2, 2025. Accessed: 2025-07-01.