The VLDB Journal (2018) 27:245–269
https://doi.org/10.1007/s00778-018-0496-7

REGULAR PAPER

Efficient provenance tracking for datalog using top-*k* queries

Daniel Deutch[1] · **Amir Gilad**[1] · **Yuval Moskovitch**[1]

Received: 21 December 2016 / Revised: 15 November 2017 / Accepted: 31 January 2018 / Published online: 22 February 2018

Abstract
Highly expressive declarative languages, such as *datalog*, are now commonly used to model the operational logic of data-intensive applications. The typical complexity of such datalog programs, and the large volume of data that they process, call for result *explanation*. Results may be explained through the tracking and presentation of *data provenance*, defined here as the set of derivation trees of a given fact. While informative, the size of such *full* provenance information is typically too large and complex (even when compactly represented) to allow displaying it to the user. To this end, we propose a novel top-*k* query language for querying datalog provenance, supporting selection criteria based on tree patterns and ranking based on the rules and database facts used in derivation. We propose an efficient novel algorithm that computes in polynomial data complexity a compact representation of the top-*k* trees which may be explicitly constructed in linear time with respect to their size. We further experimentally study the algorithm performance, showing its scalability even for complex datalog programs where full provenance tracking is infeasible.

Keywords Provenance · Datalog · Top-K

1 Introduction

Many real-life applications rely on an underlying database in their operation. In different domains, such as Declarative Networking [44], Social Networks [54], and Information Extraction [24], it has recently been proposed to use *datalog* for the modeling of such applications.

Consider, for example, AMIE [24], a system for mining logical rules from Knowledge Bases (KBs), based on observed correlations in the data. After being mined, rules are then treated as a datalog program (technically, a syntax of Inductive Logic Programming is used there) which may be evaluated with respect to a KB of facts (e.g., YAGO [58]) that, in turn, were directly extracted from sources such as Wikipedia. This allows addressing incompleteness of KBs, gradually deriving additional new facts and introducing them to the KB. The datalog program depicted in Fig. 1 is composed of rules automatically inferred by AMIE.

Datalog programs capturing the logic of real-life applications are typically quite complex, with many, possibly recursive, rules and an underlying large-scale database. For instance, AMIE rules are highly complex and include many instances of recursion and mutual recursion (see again Fig. 1). Furthermore, since AMIE rules are automatically mined, there is an inherent uncertainty with respect to their validity. Indeed, many rules mined in such a way are not universally valid, but are nevertheless very useful (and used in practice), since they contribute to a higher recall of facts.

In such complex systems, accompanying derived facts with *provenance* information, i.e., an explanation of the ways they were derived, is of great importance. Such provenance information may provide valuable insight into the system's behavior and output data, useful both for the application developers and their users.

Example 1 The binary relation $dealsWith$ includes information on international trade relations. For instance, AMIE has "learned" the following rule, intuitively specifying that $dealsWith$ is a symmetric relation (ignore for now the numbers in parentheses).

$r_1(0.8)$ `dealsWith(a, b):- dealsWith(b, a)`

✉ Amir Gilad
amirgilad@mail.tau.ac.il

Daniel Deutch
danielde@post.tau.ac.il

Yuval Moskovitch
moskovitch1@post.tau.ac.il

[1] Tel Aviv University, Tel Aviv, Israel

```
dealsWith(a, b) :- imports(a, c), exports(b, c)
dealsWith(a, b) :- dealsWith(b, a)
dealsWith(a, b) :- dealsWith(a, f), dealsWith(f, b)
hasChild(a, b) :- isMarriedTo(e, a), hasChild(e, b)
hasChild(a, b) :- isMarriedTo(a, f), hasChild(f, b)
isMarriedTo(a, b) :- isMarriedTo(b, a)
isMarriedTo(a, b) :- hasChild(a, c), hasChild(b, c)
influences(a, b) :- influences(a, f),
                    influences(f, b)
isCitizenOf(a, b) :- wasBornIn(a, f),
                     isLocatedIn(f, b)
diedIn(a, b) :- wasBornIn(a, b)
dealsWith(a, b) :- exports(a, f), exports(b, f)
dealsWith(a, b) :- imports(a, f), imports(b, f)
directed(a, b) :- created(a, b)
influences(a, b) :- influences(a, f),
                    influences(b, f)
isPoliticianOf(a, b) :- diedIn(a, f),
                        isLocatedIn(f, b)
isPoliticianOf(a, b) :- livesIn(a, f),
                        isLocatedIn(f, b)
isInterestedIn(a, b) :- influences(a, f),
                        isInterestedIn(f, b)
worksAt(a, b) :- graduatedFrom(a, b)
influences(a, b) :- influences(e, a),
                    influences(e, b)
isInterestedIn(a, b) :- isInterestedIn(e, b),
                        influences(e, a)
produced(a, b) :- created(a, b)
isPoliticianOf(a, b) :- wasBornIn(a, f),
                        isLocatedIn(f, b)
```

Fig. 1 AMIE program

Many other rules with the $dealsWith$ relation occurring in their head were mined by AMIE, including some additional rules whose validity is questionable: ($imports$ and $exports$ are additional binary relations)

$r_2(0.5)$ `dealsWith(a, b):- imports(a, c), exports(b, c)`
$r_3(0.7)$ `dealsWith(a, b):- dealsWith(a, f), dealsWith(f, b)`

In this example, when viewing a concrete derived "dealsWith" fact, it is thus highly useful to see an explanation for it, including in particular which rules were used for its derivation: intuitively, we may trust facts derived via the first rule but not via the second, unless for a concrete fact the derivation using the latter appears to "make sense" (e.g., if the derivation involves a rare product).

A conceptual question in this respect is what constitutes a "good" explanation. There are many different models defining such explanations through different notions of provenance. The models greatly vary in the level of detail that they capture: for instance, provenance may be defined as the set of input tuples contributing to a tuple derivation (the lineage model of [6]); the boolean combination thereof [33]; or their combination using an algebraic structure as in [30]. Different models are useful for different applications. In this work, we capture explanations for datalog through the notion of derivation trees. A derivation tree of an intensional fact t, defined with respect to a datalog program and an extensional database, completely specifies the rules instantiations and intermediate facts jointly used in the gradual process of deriving t.

Table 1 Database

	exports	
t_1:	Country	Product
	France	wine
	Cuba	tobacco
	Cuba	coffee beans
	imports	
t_2:	Country	Product
	Cuba	wine
	Mexico	wine
	Mexico	tobacco
	France	tobacco
	dealsWith	
	Country_a	Country_b
	Mexico	France

Example 2 As a simple example, consider the datalog program consisting of the rules r_1, r_2, r_3 from Example 1 and the instance presented in Table 1. Figure 2 depicts derivation trees for the fact $dealsWith(Cuba, France)$.

Derivation trees are particularly appealing as explanations, since unlike boolean provenance or the lineage model, they do not only include the facts and rules that support a given fact, but they also describe *how* they support it, providing insight on the structure of inference. A single fact may have multiple derivation trees (alternative derivations), and the set of all such trees (each serving as "alternative explanation") is the provenance of that fact. Defining provenance as the set of all possible derivation trees leads to a challenge: the number of possible derivation trees for a given program and database may be extremely large and even infinite in presence of recursion in the program. *This is the main challenge that we aim to address in this work.*

We next outline our approach and main contributions in addressing this problem, as well as the challenges that arise in this context.

Novel query language for datalog provenance. We observe that while full provenance tracking for datalog may be costly or even infeasible, it is often the case that only parts of the provenance are of interest for analysis purposes. To this end, we develop a query language called `selPQL` that allows analysts to specify which derivation trees are of interest to

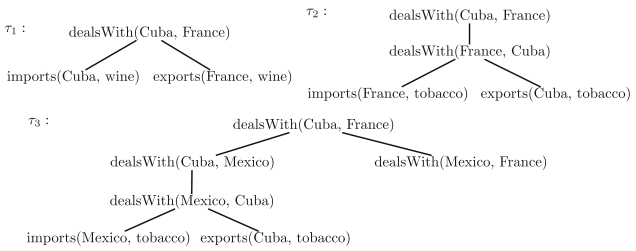


Fig. 2 Derivation Trees

them. A `selPQL` query includes a *derivation tree pattern*, used to specify the structure of derivation trees that are of interest. The labels of nodes in the derivation tree pattern correspond to facts (possibly with wildcards replacing constants), and edges may be regular or "transitive", matching edges or paths in derivation trees, respectively.

Example 3 Analysts may be interested in explanations for a particular $dealsWith$ fact (due of its importance), in all $dealsWith$ facts involving a particular country, or in explanations for all $dealsWith$ facts (rather than all relations). The two latter cases may be captured by patterns that involve wildcards. Beyond specifying facts of interest, analysts may be interested in specific features of their derivations, e.g., viewing derivations that involve integration of data from different sources or ones that rely on particular sources.

Importantly, and since the number of qualifying derivation trees may still be very large (and in general even infinite), we support the retrieval of *a ranked list of top-k qualifying trees* for each fact of interest. To this end, we allow analysts to assign *weights* to the different facts and rules. These weights are aggregated to form the weights of trees.

Example 4 The weight function may be uniform across all rules and tuples, in which case *concise* derivations are preferable to long ones. As another example, AMIE associates confidence values with rules; these may be aggregated to form the confidence in a given derivation.

Novel algorithms for selective provenance tracking. We then turn to the problem of efficient provenance tracking for datalog, guided by a `selPQL` query. We observe (and experimentally prove) that materializing full provenance, i.e., a compact representation of the possible trees, and then querying the provenance, is a solution that fails to scale. Our solution then consists of two main steps. The first is to instrument the datalog program P with respect to the tree pattern p of the `selPQL` query. We introduce a precise definition of the output of this instrumentation (see Proposition 1), which is a new datalog program P_p that "guides" provenance tracking based on p. Namely, for each pair of (relation of P, part of p) we design a novel relation name and corresponding rules whose body relations together "guarantee" satisfaction of the pattern part. Then, we show a bottom-up evaluation algorithm for P_p w.r.t. a database D that generates compact representation of the top-k qualifying trees. This is done by first computing the top-1 tree side-by-side with bottom-up datalog evaluation. We then further design novel algorithms for computing the top-k derivation trees, by exploring modifications of the top-1 tree. We show heuristic solutions as well as a solution that supports diversification of retrieved trees.

Complexity analysis and experimental study. We analyze the performance of our evaluation algorithm from a theoretical perspective, showing that the complexity of computing a compact representation of selected derivation trees is *polynomial in the input database size*, with the exponent depending on the size of the datalog program and the `selPQL` query; the enumeration of trees from this compact representation is then *linear in the output size (size of top-k trees)*. We have further implemented our solution, and have experimented with different highly complex and recursive programs. Our experimental results indicate the effectiveness of our solution even for complex programs and large-scale data where full provenance tracking is infeasible.

Note This paper significantly extends [16,17]. Specifically, our main novel contributions here are as follows:

1. We present full proofs for all theoretical results.
2. We introduce many new examples throughout the paper (e.g., Examples 12–21, 23).
3. We provide an extensive and complete study (Sect. 5) of the case of boolean combinations of patterns. We provide a new algorithm (Algorithm 2) with a correctness proof (Proposition 3), and new examples (Examples 15–20)
4. We introduce a novel algorithm for computing *diverse* top-k trees (in Sect. 6.4).
5. We present new experiments concerning the tracking of full provenance, diversification, and boolean combinations of patterns, as well as full details including the patterns and programs used.

2 Preliminaries

We provide an overview of datalog and its provenance using the notations in Table 2.

2.1 Datalog

We assume that the reader is familiar with standard datalog concepts [1]. Here we review the terminology and illustrate it with an example.

Table 2 Notations Table

P	Datalog program
r	Datalog rule
β	Body of a rule
D	Database
t	Fact
$P(D)$	Intensional Database
R	idb relation
T	edb relation
τ	Derivation tree
$trees(P, D, t)$	Derivation trees of t with respect to P,D
$trees(P, D)$	All derivation trees with respect to P,D
p	Pattern
v	Pattern node
v^0	Root of pattern p
$p(P, D)$	Derivation trees in $trees(P, D)$ matching p
P_p	Instrumented program (P w.r.t. p)
R^v, $R^{v'}$	Annotated relation

Definition 1 A *datalog program* is a finite set of datalog *rules*. A datalog rule is an expression of the form:

$$R_1(\mathbf{u_1}) :- R_2(\mathbf{u_2}) \ldots R_n(\mathbf{u_n}), x_i \neq x_j$$

where R_i's are relation names, and $\mathbf{u_1}, \ldots, \mathbf{u_n}$ are sets of variables with appropriate arities. $R_1(\mathbf{u_1})$ is called the rule's *head*, and $R_2(u_2) \ldots R_n(u_n)$ is called the rule's body. Every variable occurring in $\mathbf{u_1}$ must occur in at least one of $\mathbf{u_2}, \ldots, \mathbf{u_n}$. x_i and x_j are variables or constants occurring in $\mathbf{u_1}, \ldots, \mathbf{u_n}$ and any assignment to the variables must satisfy the disequalities constrains.

We make the distinction between extensional (edb) and intensional (idb) facts and relations. An extensional relation is a relation occurring only in the body of the rules. An intensional relation is a relation occurring in the head of some rule. A datalog program is then a mapping from edb instances to idb instances, whose semantics may be defined via the notion of the *consequence operator*. First, the *immediate* consequence operator induced by a program P maps a database instance D to an instance $D \bigcup \{A\}$ if there exists an *instantiation* of some rule in P (i.e., a consistent replacement of variables occurring in the rule with constants) such that the body of the instantiated rule includes only atoms in D and the head of the instantiated rule is A. Then the consequence operator is defined as the *transitive closure* of the immediate consequence operator, i.e., the fixpoint of the repeated application of the immediate consequence operator. Finally, given a database D and a program P we use $P(D)$ to denote the restriction to idb relations of the database instance obtained by applying to D the consequence operator induced by P.

Example 5 Reconsider the datalog program depicted in Fig. 1. Among many others, the idb instance includes the binary relation $dealsWith$ (an edb "copy" of this relation appears as well, with a rule to copy its contents that is omitted for simplification) and the binary edb relations $imports$ and $exports$.

The rules r_1, r_2, r_3 from Example 1 form a datalog program whose evaluation (with respect to the instance presented in Table 1; the presented table $dealsWith$ is its edb copy) follows the immediate consequence operator until convergence. For instance, using rule r_2 we may assign $Cuba$, $France$, $wine$ to a, b, c, respectively, obtaining the new idb fact $dealsWith(Cuba, France)$. Then using rule r_1 we obtain the idb fact $dealsWith(France, Cuba)$, etc., until no new fact may be added in such a way.

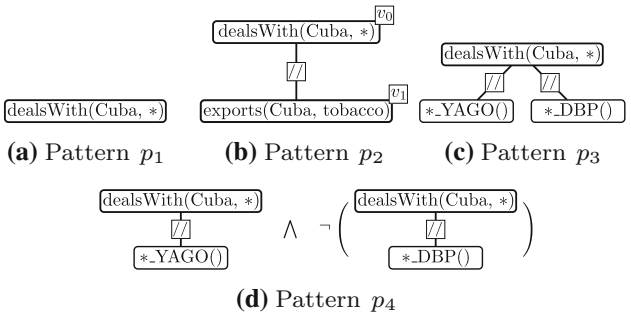
2.2 Datalog provenance

It is common to characterize the process of datalog evaluation through the notion of *derivation trees*. A *derivation tree* of a fact t with respect to a datalog program and a database instance D is a *finite* tree whose nodes are labeled by facts. The root is labeled by t, leaves are labeled by edb facts from D, and internal nodes by idb facts. The tree structure is dictated by the consequence operator of the program: the labels set of the children of node n corresponds to an instantiation of the body of some rule r, such that the label of n is the corresponding instantiation of r's head (we refer to this as an *occurrence* of r in the tree). Disequalities are not included in the derivation tree, even if they appear in r. Given a datalog program P and a database D, we denote by $trees(P, D, t)$ the set of all possible derivation trees for $t \in P(D)$, and define $trees(P, D) = \bigcup_{t \in P(D)} trees(P, D, t)$.

A single derivation tree is quite simple to understand and is even natural to visualize. However, there may be *infinitely* many (and exponentially many in the absence of recursion in P) possible derivation trees of a given fact, and so it is infeasible to materialize $trees(P, D)$.

Example 6 Three derivation trees for the fact $t = dealsWith(Cuba, France)$, based on the program given in Example 1 and the database given in Table 1, are presented in Fig. 2. For instance, τ_2 corresponds to the derivation that uses the edb facts t_1 and t_2 and the rule r_2 to derive the idb fact $dealsWith(France, Cuba)$, and then, use r_1 to derive t.

Already in the small-scale demonstrated example there are *infinitely many* derivation trees for t (due to the presence of recursion in rules); for the full program and database, many trees are substantially different in nature (based on different rules and/or rules instantiated and combined in different ways).



(a) Pattern p_1 **(b)** Pattern p_2 **(c)** Pattern p_3

(d) Pattern p_4

Fig. 3 Tree Pattern Examples

3 Querying datalog provenance

We introduce a query language called `selPQL` for derivation trees, based on two facets: (1) *boolean criteria* describing derivations of interest, (2) a *ranking function* for derivations.

3.1 Derivation tree patterns

Recalling our definition of provenance as a *possibly infinite set of trees*, we next introduce the notion of *derivation tree patterns*.

Definition 2 A *derivation tree pattern* is a node-labeled tree. Labels are either wildcards (*), or edb/idb facts, in which wildcards may appear instead of some constants. Edges may be marked as *regular* (/) or *transitive* (//), and in the latter case may be matched to a path of any length.

The boolean operators $\neg$, $\vee$ and $\wedge$ can be applied to tree patterns. Intuitively, given the tree pattern p_1 and p_2, $\neg p_1$ is used to specify that we are interested in trees that do not match (see semantics of matching below) p_1, $p_1 \vee p_2$ (and $p_1 \wedge p_2$) are used to specify that we are interested in trees that match p_1 or (resp. and) p_2.

Example 7 Several tree patterns are presented in Fig. 3. The pattern p_1 specifies interest in all derivations of facts of the form $dealsWith(Cuba, *)$ (any constant may replace the wildcard). The other patterns further query the *structure* of derivation. Specifically, p_2 specifies that the analyst is interested in derivations of such facts that are (directly or indirectly) based on the fact that Cuba exports tobacco. The patterns p_3 and p_4 are relevant when (omitted) rules integrate two ontologies (YAGO and DBpedia). We use ∗_YAGO() and ∗_DBP()[1] to match all relations from YAGO and DBpedia resp.; then p_3 selects derivations of facts $dealsWith(Cuba, *)$ that are based on integrated data from *both* sources, and p_4 selects derivations that use facts from YAGO but *no* fact from DBpedia.

[1] This requires a slight change of the definition of patterns, which is easy to support, to allow * in relation names.

We next define the semantics of derivation tree patterns, in the spirit of XML query languages with some technical differences (see below).

Definition 3 Given a derivation tree τ and a derivation tree pattern p, a *match of p in τ* is a mapping h from the nodes of p to nodes of τ, and from the regular (transitive) edges of p to edges (resp. paths) of τ such that (1) the root of p is mapped to the root of τ, (2) a node labeled by a label l which does not contain wildcards, is mapped to a node labeled by l, (3) a node labeled by a label l which includes wildcards is mapped to a node labeled by l', where l' may be obtained from l by replacing wildcards by constants, (4) a node labeled by a wildcard can be mapped to any node in τ. (5) If n, m are nodes of p and e is the directed (transitive) edge from m to n, then $h(e)$ is an edge (path) in τ from $h(m)$ to $h(n)$ and (6) for any two edges e_1 and e_2 in p, their corresponding edge/path in τ are disjoint.

We next define the semantics of a pattern with respect to a datalog instance.

Definition 4 Given a (possibly infinite) set S of derivation trees and a derivation tree pattern p, we define $p(S)$ ("the result of evaluating p over S") to be the (possibly infinite) subset S' consisting of the trees in S for which there exists a match of p. Given a pattern p, a datalog program P and an extensional database D, we use $p(P, D)$ as a shorthand for $p(trees(P, D))$.

Example 8 Consider the datalog program P given in Example 1, the database instance given in Table 1 and the tree pattern p_2 in Fig. 3b. The set $p_2(P, D)$ includes infinitely many derivation trees, including in particular τ_2 and τ_3 shown in Fig. 2.

The boolean operators $\neg$, $\vee$ and $\wedge$ can also be applied to tree patterns, with the expected semantics, i.e., $\neg p_1$ matches every tree where there is no match of p_1, and $p_1 \vee p_2$ ($p_1 \wedge p_2$) matches trees that match p_1 or (resp. and) p_2. For instance, the pattern p_4 in Fig. 3d specifies that we wish to view derivations that are based solely on YAGO and do not use DBpedia fact.

3.2 Ranking derivations

Even when restricting attention to derivation trees that match the pattern, their number may be too large or even infinite, as exemplified above. We thus propose to *rank* different derivations based on the rules and facts used in them. We allow associating weights with the input database *facts* as well as the individual *rules*, and aggregating these weights. Different choices of weights and aggregation functions may be used, capturing different interpretations. We support a general class of such functions via the notion of an *ordered monoid*, which

is a structure $(M, +, 0, <)$ such that M is a set of elements, $+$ is a binary operation which we require to be commutative, associative, and monotone non-increasing in each argument, i.e., $x + y \leq \min(x, y)$ (with respect to the structure's order), 0 is the neutral value with respect to $+$, and $<$ is a total order on M.

Definition 5 A *weight-aware datalog instance* is a triple (P, D, w) where w, the weight function, maps rules in P as well as tuples in D to elements of an ordered monoid $(M, +, 0, <)$. The monoid operation is referred to as the *aggregation function*.

Example 9 We demonstrate multiple choices of monoid and the corresponding applications.

Derivation size To rank derivation trees by their size we may use the monoid $(\mathbb{Z}^-, +, 0, <)$, and set the weight of every rule to be -1; then the weight of a derivation tree is the negative of its size.

Derivation (total) confidence Another way to rank derivations is to associate *confidence* values with rules. In AMIE, such confidence values reflect the rules' support in underlying data. Here we use the monoid $([0, 1], \cdot, 1, <)$. *This is the example that will be used in the sequel; rules' weights are specified next to them and facts weights are all 1.*

Derivation minimal confidence One could alternatively impose a preference relation on trees based on the confidence in their "weakest" rule/fact (so that top trees are those whose least trusted component is best trusted among all trees). This can be captured by the $([0, 1], \min, 1, <)$ monoid.

Access control Consider the case where each fact/rule is associated with a different access control credential, e.g., one of $A = \{$Top secret $(\mathbb{T})$, Secret $(\mathbb{S})$, Confidential $(\mathbb{C})$, Unclassified $(\mathbb{U})\}$. We may rank trees based on their overall credential (typically defined as the maximum credential of fact/rule used), so that non-secret trees are preferable as explanations. Here we use $(A, \min, \mathbb{U}, <)$, where $\mathbb{T} < \mathbb{S} < \mathbb{C} < \mathbb{U}$.

We may then define the weight of a derivation tree as the result of aggregating the weights of facts and derivation rules used in the tree.

Definition 6 The weight of a derivation tree τ with respect to a weight-aware datalog instance, denoted, abusing notation, as $w(\tau)$, is defined as $\sum_r w(r) + \sum_t w(t)$ where the sums (performed in the weights monoid) range over all rules and tuples occurrences in τ.

Example 10 Setting the weight of every rule to be -1 and using the monoid $(\mathbb{Z}^-, +, 0, <)$, the weights of the trees in Fig. 2 are $w(\tau_1) = -1$, $w(\tau_2) = -2$ and $w(\tau_3) = -3$.

Using the weight function w defined by the confidence value associated with rules (appearing next to them, in brackets) and aggregating via multiplication, the weights of exemplified trees (Fig. 2) are $w(\tau_1) = 0.5$, $w(\tau_2) = 0.5 \cdot 0.8 = 0.4$ and $w(\tau_3) = 0.7 \cdot 0.8 \cdot 0.5 = 0.28$.

Last, we may define top-k problem.

Definition 7 Given a pattern p, a weight-aware datalog instance (P, D, w) and a natural number k, we use $top\text{-}k(p, P, D, w)$ to denote the set containing for each fact t in $P(D)$ the k derivation trees of t that are of highest weight (ties are decided arbitrarily) out of those in $p(P, D)$. We use *TOP-K* to denote the problem of finding $top\text{-}k(p, P, D, w)$ given the above input.

Example 11 In general, there are infinitely many finite derivation trees for the fact $dealsWith(Cuba, France)$ (due to the recursive rule r_1), as well as infinite derivations which we algorithmically avoid generating (see Sect. 6). The top-2 results w.r.t. the pattern given in Fig. 3b are τ_2 and τ_3 in Fig. 2 with weights of 0.4 and 0.28, respectively. Note that τ_1 does not match the pattern.

Note that if the database is large then it is unreasonable (and typically unneeded) to specify a weight for every individual tuple; instead, tuples can have a default weight of 1, which may be augmented by adding custom rules to the datalog program, that copy tuples of interest (e.g., all tuples of a particular relation or any other selection criteria) to an auxiliary relation, and assigning the weights to these custom rules.

Specifying queries in `selPQL`. Some users may lack sufficient understanding of the structure of the program and the content of the database, creating a bootstrapping problem in writing `selPQL` queries. A possible use case is to first specify and evaluate a "general" pattern, which, e.g., only restricts the attention to particular output tuples. Then, after browsing through relevant explanations, the user may refine her patterns accordingly.

In the following sections, we propose a two-step algorithm for solving *TOP-K*, as explained in the Introduction and depicted in Fig. 4. The algorithm will serve as proof for the following theorem.

Theorem 1 *For any Program P, pattern p and database D, we can compute the top-k derivation trees for each fact*

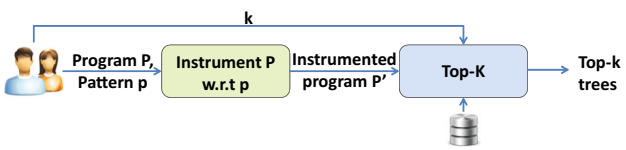


Fig. 4 High-level Framework

matching the root of p in $O(k^3 \cdot |D|^{O(|P|^{w(p)})} + |out|)$ time where $w(p)$ is the pattern width (i.e., the maximal number of children of a node in p) and $|out|$ is the output size.

The worst case time complexity is polynomial in the database size with exponential dependency on the program size (which is typically much smaller), and double exponential in the pattern width (which is typically even smaller), and linear in the output size. We note that the output size (even the size of a single derivation tree) may be exponential in the Database size (though in practice top-k trees are typically small); the linear dependency on the output size is of course optimal in this respect.

4 Program instrumentation

We now present the first step of the algorithm for solving *TOP-K*, which is instrumenting the program with respect to a `selPQL` pattern. We first present an algorithm for a single pattern instrumentation, and then generalize it to Boolean combinations of patterns.

4.1 A single pattern

We first define relation names for the output program, and then its rules.

New relation names We say that a pattern node v is a *transitive child* if it is connected with a transitive edge to its parent. For every relation name R occurring in the program and for every pattern node v we introduce a relation name R^v. If v is a transitive child we further introduce a relation name R^{v^t}. Intuitively, derivations for facts in R^v must match the subpattern rooted by v; derivations for R^{v^t} must include a subtree that matches the subpattern rooted by v. These will be enforced by the generated rules, as follows.

New rules We start with some notations. Let v be a pattern node, let $v_0, \ldots, v_n$ be the immediate children of v. Given an atom (in the program) $atom$, we say that it *locally matches* v if the label of v is $atom$, or the label of v may be obtained from $atom$ through an assignment A mapping variables of $atom$ to constants or wildcards (if such assignment exists, it is unique). We further augment A so that a variable x mapped to a wildcard, is now mapped to itself (Intuitively, this is the required transformation to the atom so that *a match with the pattern node is guaranteed*).

Example 12 The atom $dealsWith(a, b)$ locally matches the pattern node v_0 of the tree pattern shown in Fig. 3b through the assignment $A = \{a \leftarrow Cuba, b \leftarrow *\}$, but doesn't locally match v_1.

Overloading notation, we will then use $A(\beta)$, where β is a rule body, i.e., a set of atoms, to denote the set of atoms obtained by applying A to all atoms in β.

Algorithm 1: Instrumentation w.r.t. tree pattern

input : Weighted Program P and a pattern p
output: "Instrumented" Program P_p

```
1  foreach pattern node v ∈ p do
2      Let v_0, ..., v_n be the immediate children of v;
3      foreach rule [R(x_0, ..., x_m) : −β] in P do
4          if R(x_0, ..., x_m) locally matches v through partial
           assignment A then
5              Let (y_0, ..., y_m) := A(x_0, ..., x_m);
6              if v is a leaf then
7                  Add [R^v(y_0, ..., y_m) : −A(β)] to P_p;
8              else
9                  foreach β' ∈ ex(A(β), {v_0, ..., v_n}) do
10                     Add [R^v(y_0, ..., y_m) : −β'] to P_p;
11         if v is a transitive child then
12             foreach β' ∈ tr − ex(β, v) do
13                 Add [R^{v^t}(x_0, ..., x_m) : −β'] to P_p;
14     foreach rule [R^v(y_0, ..., y_m) : −β] for transitive v do
15         Add [R^{v^t}(y_0, ..., y_m) : −β] to P_p;
16     HandleEDB ();
17 Clean failed rules in P_p ;
18 return the union of rules in P and P_p;
```

Algorithm 1 then generates a new program, instrumented by the `selPQL` pattern, as follows. For brevity we do not specify the weight of the new rules: they are each simply assigned the weight of the rule from which they originated, or 0 (neutral value of the monoid) if there is no such rule. The algorithm traverses the pattern in a top–down fashion, and for every pattern node v it looks for rules in the program whose head locally matches v (lines 3–4). For each such rule it generates a new rule as follows: if v is a leaf (lines 6–7), then intuitively this "branch" of the pattern is guaranteed to be matched and we add rules which are simply the "specializations" of the original rule, meaning that we apply to their body the same assignment used in the match.

Otherwise (lines 8–10), we need derivations of atoms in the body of the rule to satisfy the subtrees rooted in the children of v. To this end we define the set of "expansions" $ex(atoms, \{v_0, \ldots, v_n\})$ as follows. Consider all one-to-one (but not necessarily onto) functions f that map the set $\{v_0, \ldots, v_n\}$ to the set $atoms = \{a_0, \ldots, a_k\}$. Each such function defines a new set of atoms obtained from $atoms$ by replacing atom $a_i = R(x_0, \ldots, x_m)$ by $R^{v_j}(x_0, \ldots, x_m)$ if $f(v_j) = a_i$ and v_j is not a transitive child, or by $R^{v_j^t}(x_0, \ldots, x_m)$ if v_j is a transitive child (atoms to which no node is mapped remain intact). We then define

$ex(atoms, \{v_0, \ldots, v_n\})$ as the set of all atoms sets obtained for some choice of function f.

Example 13 Consider the set of atoms in the body of r_3, and the pattern given in Fig. 3b with the transitive node v_1. The set $ex(\{dealsWith(a, f), dealsWith(f, b)\}, \{v_1\})$ consists of $\{dealsWith(a, f)^{v_1^t}, dealsWith(f, b)\}$ and $\{dealsWith(a, f), dealsWith(f, b)^{v_1^t}\}$

In line 10 the algorithm generates a rule for each set out of these sets of atoms. Intuitively, each such rule corresponds to alternative "assignment of tasks" to atoms in the body, where a "task" is to satisfy a subpattern.

The algorithm thus far deals with satisfaction of the subtree rooted at v, by designing rules that propagate the satisfaction of the subtrees rooted at the children of v to atoms in the bodies of relevant rules. However, if the current pattern node v is *transitive* (lines 11–13), then more rules are needed, to account for the possibility of the derivation satisfying the tree rooted at v only in an indirect fashion. A possibly indirect satisfaction is either through a direct satisfaction (and thus for every rule for $R^v(\ldots)$ we will have a copy of the same rule for $R^{v^t}(\ldots)$, lines 14–15), or through (indirect) satisfaction by an atom in the body. For the latter, we define $tr - ex(atoms, v)$ as the set of all atoms sets obtained from $atoms$ by replacing a single atom $R(x_0, \ldots, x_m)$ in $atoms$ by $R^{v^t}(x_0, \ldots, x_m)$ (and keeping the other atoms intact), and add the corresponding rules (line 13). Then the function `HandleEDB` adds rules for nodes that locally match edb facts, copying matching facts into the new relations $T^v(\ldots)$ and $T^{v^t}(\ldots)$. The final step of the algorithm is "cleanup" (line 17), removing unreachable rules. These rules have no derivation, i.e., each derivation requires use of at least one idb relation for which there is no rule in P_p (this may be done in a bottom-up fashion). In addition, new rules that are added by the algorithm and are not reachable from the rules added for the root node of the pattern (i.e., rules for $R^{v_0}(\ldots)$) are deleted. This can be done in a top–down fashion. The algorithm returns a new program consisting of the set of newly generated rules *together* with the original rules.

Example 14 Consider the program P consisting of the rules r_1, r_2 and r_3 given in Example 1, and the tree pattern shown in Fig. 3b, where v_0 is the root node in p_2 and v_1 is the leaf.

Since all rules in P locally match v_0 through the assignment $A = \{a \leftarrow Cuba, b \leftarrow *\}$, v_0 is not a leaf and $\{dealsWith^{v_1^t}(b, Cuba)\}$ is the only β' obtained for rule r_1 and $ex(A(dealsWith(b, a)), v_1)$, we have that in line 10 the algorithm adds the rule

$\texttt{dealsWith}^{v_0}\texttt{(Cuba,b):-dealsWith}^{v_1^t}\texttt{(b,Cuba)}$

Similarly, the rules

```
dealsWith^v0(Cuba, b):- imports(Cuba, a), exports^v1^t(b, c)
dealsWith^v0(Cuba, b):- imports^v1^t(Cuba, a), exports(b, c)
dealsWith^v0(Cuba, b):- dealsWith^v1^t(Cuba, f),
                                    dealsWith(f, b)
dealsWith^v0(Cuba, b):- dealsWith(Cuba, f),
                                    dealsWith^v1^t(f, b)
```

are generated using the rules r_2 and r_3

Next, the algorithm adds the rules for v_1. Since, $exports(\ldots)$ is an edb relation, there are no rules in P that locally matches it and no new rules are added in lines 3–11. v_1 is a transitive node and thus in line 13 the following rules are added

```
dealsWith^v1^t(a, b):- dealsWith^v1^t(b, a)
dealsWith^v1^t(a, b):- imports^v1^t(a, c), exports(b, c)
dealsWith^v1^t(a, b):- imports(a, c), exports^v1^t(b, c)
dealsWith^v1^t(a, b):- dealsWith^v1^t(a, f), dealsWith(f, b)
dealsWith^v1^t(a, b):- dealsWith(b, f), dealsWith^v1^t(f, b)
```

Finally, since the edb relation $exports(a, b)$ locally matches v_1 through the assignment $A = \{a \leftarrow Cuba, b \leftarrow tobacco\}$, the function `HandleEDB` adds the rules

```
exports^v1(Cuba, tobacco):- exports(Cuba, tobacco)
exports^v1^t(Cuba, tobacco):- exports(Cuba, tobacco)
```

Intuitively, derivations for facts in $dealsWith^{v_0}(\ldots)$ must match the subpattern rooted by v_0. Then derivations for facts in $dealsWith^{v_1^t}(\ldots)$ must include a subtree that matches the subpattern rooted by v_1, and generated rules for $dealsWith^{v_1^t}(\ldots)$ enforce that (since a $dealsWith$ atom cannot satisfy v_1) one of the atoms in the body of a used rule will be derived in a way eventually satisfying v_1.

The algorithm then performs a "cleanup" of atoms for which there is no derivation, in this example the idb relation $imports^{v_1^t}(\ldots)$ has no rule in P' thus the derivation rules that use it such as

```
dealsWith^v1^t(a, b):- imports^v1^t(a, c), exports(b, c)
```

are deleted. In addition the relation $exports^{v_1}(\ldots)$ is not reachable from the derivation rules for $dealsWith^{v_0}(\ldots)$ and thus the derivation rule added for it is deleted. Finally the output program is:

```
dealsWith(a, b):- dealsWith(b, a)
dealsWith(a, b):- imports(a, c), exports(b, c)
dealsWith(a, b):- dealsWith(a, f), dealsWith(f, b)
dealsWith^v0(Cuba, b):-dealsWith^v1^t(b, Cuba)
dealsWith^v0(Cuba, b):- imports(Cuba, a), exports^v1^t(b, c)
dealsWith^v0(Cuba, b):- dealsWith^v1^t(Cuba, f),
                                    dealsWith(f, b)
dealsWith^v0(Cuba, b):- dealsWith(Cuba, f),
                                    dealsWith^v1^t(f, b)
dealsWith^v1^t(a, b):- dealsWith^v1^t(b, a)
dealsWith^v1^t(a, b):- imports(a, c), exports^v1^t(b, c)
dealsWith^v1^t(a, b):- dealsWith^v1^t(a, f), dealsWith(f, b)
dealsWith^v1^t(a, b):- dealsWith(b, f), dealsWith^v1^t(f, b)
exports^v1^t(Cuba, tobacco):- exports(Cuba, tobacco) [r']
```

The instrumented program satisfies the following fundamental property. Given an atom $R(\ldots)$, $R^v(\ldots)$ or $R^{v^t}(\ldots)$

we define its *origin* to be $R(\ldots)$, i.e., the atom obtained by deleting the annotation v or v^t (if exists). For a derivation tree τ we define $origin(\tau)$ as the tree obtained from τ by replacing each atom by its origin and pruning branches added due to the function `HandleEDB` ("copying" edb facts). We now have:

Proposition 1 *Let P_p be the output of Algorithm 1 for input which is a program P and pattern p with root v^0. For every database D, we have that:*

$$trees(P, D) = \bigcup_{\tau \in trees(P_p, D)} origin(\tau) \quad (1)$$

$$p(P, D) = \bigcup_{t = R^{v^0}(\ldots)} \bigcup_{\tau \in trees(P_p, D, t)} origin(\tau) \quad (2)$$

$$w(origin(\tau)) = w(\tau) \quad \forall \tau \in trees(P_p, D) \quad (3)$$

Proof 1. Since $P \subseteq P_p$, every $\tau \in trees(P, D)$ is also in $trees(P_p, D)$ and it holds that $origin(\tau) = \tau$, thus $trees(P, D) \subseteq \bigcup_{\tau \in trees(P_p, D)} origin(\tau)$.
In addition, recall that every node in a derivation tree $\tau \in trees(P_p, D)$ corresponds to a derivation rule in P_p. From the construction of the new rules in P_p, the set of rules obtained by removing the annotations from relation names in P_p is exactly the set of rules in P (possibly with repetitions), and the rules added by `HandleEDB`. $origin(\tau)$ is obtained by removing the annotation from τ and pruning branches added due to the function `HandleEDB`, thus every node in $origin(\tau)$ corresponds to a derivation rule in P, therefore $trees(P, D) \supseteq \bigcup_{\tau \in trees(P_p, D)} origin(\tau)$, namely

$$trees(P, D) = \bigcup_{\tau \in trees(P_p, D)} origin(\tau)$$

2. Let $p|_v$ be the subpattern of p rooted at v. We prove by induction on the height of the pattern $p|_v$ that for every pattern node v it holds that

$$p|_v(P, D) = \bigcup_{t = R^v(\ldots)} \bigcup_{\tau \in trees(P_p, D, t)} origin(\tau)$$

Base case: v is a leaf. There are two possible cases:

- v locally matches an edb fact $T(\ldots)$. In this case for each $\tau \in p|_v(P, D)$, τ is simply an edb atom, and the function `HandleEDB` adds rules that copy the relevant tuple from the database into the new relation $T^v(\ldots)$ (and $T^{v^t}(\ldots)$). The derivation tree τ of $T^v(\ldots)$ (and $T^{v^t}(\ldots)$, in the case where v is a transitive node) consists of two nods, a root, $T^v(\ldots)$ (or $T^{v^t}(\ldots)$), and a leaf, $T(\ldots)$, and $origin(\tau)$ is simply $T(\ldots)$ in this case.
- v locally matches an idb atom $R(\ldots)$ through partial assignment A. In this case, in line 7 the algorithm adds a new rule for each rule in P if its head locally matches v (i.e., $t = R(\ldots) \in P(D) \Leftrightarrow R^v \in P_p(D)$). The relations in the body of each such rule are not annotated and thus the derivation trees of facts in the body are derivation trees in $trees(P, D)$. Derivation trees τ of $R^v(\ldots)$ consist of one of the rules added in line 7 and the derivation trees of each fact in the rule's body. Therefore, $origin(\tau)$ is the tree obtained by removing the annotation v, and it is a derivation tree in $trees(P, D)$.
For the case where v is transitive, the algorithm adds two types of derivation rules for R^{v^t}, (i) the rules added in line 13 and (ii) in line 15. Recall that (when v is a leaf) $\tau \in p|_{v^t}(P, D) \Leftrightarrow$ (1) the root of τ locally matches v (in this case $\tau \in p|_v(P, D)$) or (2) there exists a node (not the root) in τ that locally matches v. The rules added in line 15 capture case (1) and this case is similar to the case where v is not transitive.
The rules added in line 13 capture case (2). Note that the body of such rules contains exactly one annotated relation name $S^{v^t}(\ldots)$ while the rest are facts in $P(D)$ and thus their derivation trees are in $trees(P, D)$. We can thus show by induction that the proposition holds for $S^{v^t}(\ldots)$. A derivation tree τ that contains type (i) rules must contain a derivation rules of type (ii) (since initially there are no annotated facts in the database). If $S^{v^t}(\ldots)$ is derived using type (ii) rule, then clearly, by removing the annotations we obtain a derivation tree in $trees(P, D)$.

Suppose that the proposition holds for all v s.t. the $p|_v$ is with height $< k$. Let v be a pattern node where $p|_v$ is with height k, with children $v_0, \ldots, v_n$.

- If v is not transitive, then a derivation tree $\tau \in p|_v(P, D) \Leftrightarrow$ the root of τ locally matches v and $\forall v_j \; \exists u$ s.t. u is a child of the root in τ and for the subtree rooted at it τ_j it holds that $\tau_j \in p|_{v_j}(P, D)$. Observe that the last derivation step in any derivation tree $\tau \in p|_v(P, D)$ can be done by a derivation rule r added by the algorithm in line 10. To see this, consider the root of τ. The root must locally match v so the algorithm adds a labeled rule with the partial assignment A that make them locally match. This labeled rule is r. The body of the rule β can consist of both annotated and non-annotated atoms. Derivation trees of atoms that are not annotated are trees in $trees(P, D)$. For annotated relation it holds that $\tau_j \in p|_{v_j}(P, D)$ and since $p|_{v_j}$ are at depth $k - 1$ in the tree by the induction hypothesis it holds that

$$\tau \in p|_{v_j}(P, D) = \bigcup_{t=R^{v_j}(\ldots)} \bigcup_{\tau \in trees(P_p, D, t)} origin(\tau)$$

Therefore, the derivation tree obtained by replacing $R^v(\ldots)$ with $R(\ldots)$ and replacing each subtree τ' rooted at the children of the root of τ with $origin(\tau')$ is $origin(\tau)$ and it holds that $origin(\tau) \in p|_v(P, D)$

3. The weights of the new rules added by the algorithm are assigned the weights of the rules from which they originated, and rules added due to the function `HandleEDB` are added with weight 0 (i.e., the natural with respect to $+$ in the monoid). In addition, the set of edb facts occurring in τ is exactly the set of edb facts occurring in $origin(\tau)$ (due to the construction of the rules added by `HandleEDB`). Therefore, we have:

$$\begin{aligned} w(\tau) &= \sum_{r \in \tau} w(r) + \sum_{t \in \tau} w(t) \\ &= \sum_{r \in origin(\tau)} w(r) + \sum_{t \in origin(\tau)} w(t) \\ &= w(origin(\tau)) \end{aligned}$$

□

We refer to v and v^t in $R^v(\ldots)$ and $R^{v^t}(\ldots)$ as *annotations*. Intuitively, the first part of the proposition means that for every database, P_p defines the same set of trees as P if we ignore the annotations (in particular we generate the same set of facts up to annotations); the second part guarantees that by following the annotations we get exactly the derivation trees that interest us for provenance tracking purposes; and the third part guarantees that the weights are kept. This will be utilized in the next step, where we evaluate the instrumented program while retrieving relevant provenance.

4.2 Complexity and output size

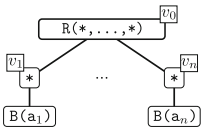
Given a datalog program P of size $|P|$ and a pattern p, the algorithm traverses the pattern, and for each node $v \in p$ iterates over the program rules. Let $w(p)$ be the width of p, i.e., the maximal number of children of a node in p. The maximal number of new rules the algorithm adds is $O(|P|^{w(p)})$. The exponential dependency on the pattern width is due to the need to consider all "expansions". Note that the exponential dependency is on the pattern width, which is expected to be small in practice. Furthermore, we next show that a polynomial dependency on the program and pattern is impossible to achieve.

Proposition 2 *(Lower Bound) There is a class of patterns $\{p_1, \ldots\}$ and a class of programs $\{P_1, \ldots\}$, such that $w(p_n) = O(n)$, $|P_n| = O(n)$ and there is no program IP_n of size polynomial in n that satisfies the three conditions of Proposition 1 with respect to P_n, p_n.*

Proof Consider the following datalog program P_n ($a_1, \ldots, a_n$ are constants):

```
R(x1, x2,..., xn) :- R1(x1), ... Rn(xn)
R1(x) :-B(x)
 ...
Rn(x) :-B(x)
```

and the pattern p_n:

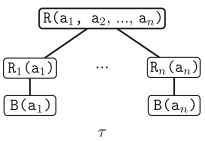


Both the pattern width and program size are polynomial in n (the pattern width $w(p_n)$ is n and the program P_n consists of $n + 1$ rules). We claim that every instrumented program P^i satisfying the conditions of Proposition 1 must include at least $n!$ rules. To observe that this is the case, first note that to satisfy the proposition's condition (1), P^i must include a relation R^{v_0}, with rules that are "copies" of the first rule of P_n (we say that a rule r' is a copy of a rule r if r may be obtained from r' by replacing every relation name in r by its origin. We note that in fact our algorithm generates the following $n!$ "copies":

```
r1:  R^v0(x1, x2, ..., xn) :- R1^v1(x1), ... Rn^vn(xn)
       ...
     R^v0(x1, x2, ..., xn) :- R1^vn(x1), ... Rn^v1(xn)
```

For each $R_i^{v_j}$, there is a rule of the form $R_i^{v_j}(a_j) : -B(a_j)$.

We then observe that P^i must include these rules (up to renaming) as well. First, without loss of generality, assume that P^i includes all of the above rules except for the rule r_1. For the database D that contains the facts `B(ai)` for $1 \leq i \leq n$, the derivation tree $\tau \notin trees(P^i, D)$, although $origin(\tau) \in trees(P_n, D)$, thus violating the proposition's condition (2).



Alternatively, if P^i "groups" two relation names (w.l.o.g. say $R_1^{v_1}$ and $R_1^{v_2}$) together (say using relation name $R_1^{v_{12}}$), and then, e.g., generates the two rules $R_1^{v_{12}}(a_1) : -B(a_1)$ and $R_1^{v_{12}}(a_2) : -B(a_2)$ (and "groups together" the corresponding rules for R) to allow sub-derivations involving R_1 to either use a_1 or a_2 (the "extreme case" would go back to the original program, thus allowing any constants to be

used in conjunction with R_1. Then, we obtain a derivation $\tau_2 \in trees(P^i, D, t)$, for $t = R^{v_0}(\ldots)$, although $origin(\tau_2) \notin p_n(P_n, D)$, where τ_2 follows the same structure of τ having two occurrences of $B(a_2)$ and no occurrence of $B(a_1)$, again violating the equality in the proposition's condition (2). It is then easy to observe that no other alternative program can satisfy the conditions. □

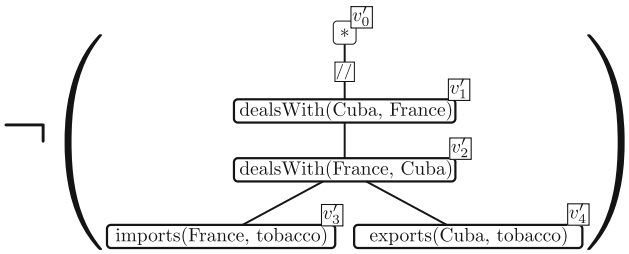


Fig. 5 Negation of a Pattern

5 Boolean combinations of patterns

Algorithm 1 allows intersection of a single pattern with a program. We next explain how to account for `selPQL` queries that involve boolean combinations of patterns, i.e., negation, conjunction, and disjunction. The time complexity and output program size remain polynomial in the size of the original program, with exponential dependency on the width of the pattern (the exponent is multiplication of the individual size of patterns, in the case of conjunction).

5.1 Negation

The algorithm for intersecting a negation of a pattern is similar to Algorithm 1 with some modifications, as follows. We use relation names $R^{\neg v}$ and $R^{\neg v^t}$ for every relation name R in the program and for every pattern node v. Derivations for $R^{\neg v}$ should not match the subpattern rooted by v and derivations for $R^{\neg v^t}$ should not include a descendant that matches the subpattern rooted by v. If the root of the pattern is labeled v_0, derivations of facts in annotated relations $R^{\neg v_0}$ are derivations that satisfy the negated pattern (i.e., does not match the pattern).

We then extend the idea of "expansion set" to define $neg\text{-}ex(atoms, \{v_0, \ldots, v_n\})$ where $atoms$ is a set of atoms and the v_i's are pattern nodes as follows: if $|atoms| \geq n$, then $neg\text{-}ex(\ldots)$ is a set of $n+1$ atoms sets where the i'th set is the set obtain from $atoms$ by replacing every atom $R(x_0, \ldots, x_m)$ by $R^{\neg v_i^t}(x_0, \ldots, x_m)$, if v_i is a transitive child and by $R^{\neg v_i}(x_0, \ldots, x_m)$ otherwise. If $|atoms| < n$ then $neg\text{-}ex(\ldots)$ is the set that contains only the set $atoms$. If the root of the pattern is labeled v_0, we track all rules whose head is labeled by $\neg v_0$.

Example 15 Consider the set of atoms in the body of r_3, and the pattern given in Fig. 5. Intuitively, this pattern matches all derivation trees that does not contain the derivation tree τ_2 shown in Fig. 2 as subtree. The set $neg\text{-}ex(\{dealsWith(Frnace, f), dealsWith(f, Cuba)\}, \{v'_3, v'_4\})$ consist of $\{dealsWith(Frnace, f)^{\neg v'^3}, dealsWith(f, Cuba)^{\neg v'^3}\}$ and $\{dealsWith(Frnace, f)^{\neg v'^4}, dealsWith(f, Cuba)^{\neg v'^4}\}$.

Furthermore, given a pattern node v, and a rule in the program $r = R(x_1, \ldots, x_m)\text{:-}\beta$, we define the set $mis(v, r)$ as follows. If $R(x_1, \ldots, x_m)$ does not locally match v (i.e., v cannot be obtain by any assignment) then $mis(v, r) = \{\beta\}$. If $R(x_1, \ldots, x_m)$ locally matches v through a partial assignment A, then $mis(v, r)$ consist of $\beta, x_i \neq \sigma_i$ for each $\{x_i \leftarrow \sigma_i\} \in A$. Note that $mis(v, r)$ may be empty (e.g., when $R(x_1, \ldots, x_m)$ locally matches v with an empty assignment).

Example 16 Consider the rule r_2 given in Example 1, and the pattern node v'_2 of the negated tree pattern shown in Fig. 5. $dealsWith(a, b)$ locally matches v'_2 through the assignment $\{a \leftarrow France, b \leftarrow Cuba\}$ thus $mis(v'_1, r_2)$ consist of $\{dealsWith(b, a), a \neq France\}$ and $\{dealsWith(b, a), b \neq Cuba\}$.

Let v be a node in the pattern p, we define $Tr(v)$ to be the last transitive node on the path from the root of p to v (including v if it is transitive). In the case where no such node exists $Tr(v) = \bot$.

Example 17 For the negated pattern shown in Fig. 5, $Tr(v'_0) = \bot$ and $Tr(v'_i) = v'_1$ for $1 \leq i \leq 4$.

Finally, given a set of atoms $atoms$ and a node v we define $tr\text{-}neg(atoms, v)$ as the set of atoms obtained from $atoms$ by replacing each atom $R(\ldots)$ in $atoms$ by $R^{\neg v^t}(\ldots)$. Additionally, we define $tr\text{-}neg(atoms, \bot) = atoms$.

Algorithm 2 generates a new program, instrumented by a negated pattern, as follows. We use the notation v^* to denote v^t if v is transitive and v otherwise. Similarly to Algorithm 1, we do not specify the weight of the new rules. The algorithm traverses the pattern in a top–down fashion and starts by generating rules using $mis(\ldots)$ (lines 4–5). Intuitively, for a given node v and rule $r = R(x_1, \ldots, x_m)\text{:-}\beta$, if v is not transitive then for any rule in $R^{\neg v}(\ldots)\text{:-}\beta' \in mis(v, r)$ either $R(\ldots)$ does not locally match v, or it does locally match v through a partial assignment A, and β' contains the disequality $x_i \neq \sigma_i$ for some $\{x_i \leftarrow \sigma_i\} \in A$. In both cases, any derivation of $R^{\neg v}(\ldots)$ cannot match the subpattern rooted by v, and thus we avoid derivations that match it. Recall that derivations for $R^{\neg v^t}$ should not include a descendant that matches the subpattern rooted by v, thus we use $tr\text{-}neg(\ldots)$ to ensure that the derivation tree does not include any descendant that matches the subpattern of the last transitive node

Algorithm 2: Instrumentation w.r.t. negated tree pattern

input : Weighted Program P and a negated pattern p
output: "Instrumented" Program P_p

1 **foreach** *pattern node* $v \in p$ **do**
2 Let $v_0, \ldots, v_n$ be the immediate children of v;
3 **foreach** *rule* $[r = R(x_0, \ldots, x_m) : -\beta]$ *in* P **do**
4 **foreach** $\beta \in mis(v, r)$ **do**
5 Add $[R^{\neg v^*}(\ldots)$:-$tr\text{-}neg(\beta, Tr(v))]$ to P_p;
6 **if** $R(x_0, \ldots, x_m)$ *locally matches* v, *and* v *is not a leaf* **then**
7 Let A be the partial assignment that cause the match and $(y_0, \ldots, y_m) := A(x_0, \ldots, x_m)$;
8 **foreach** $\beta' \in neg\text{-}ex(A(\beta), \{v_0, \ldots, v_n\})$ **do**
9 Add $[R^{\neg v^*}(y_0, \ldots, y_m)$:-$\beta']$ to P_p;
10 `HandleEDBneg` ();
11 Clean failed rules in P_p ;
12 **return** the union of rules in P and P_p;

on the path from the root to v. Intuitively, derivation trees the use the rules for $R^{\neg v^*}(\ldots)$ match the subpattern that includes all the nodes from the root to v and the last derivation step "breaks" the match for the rest of the pattern, but we still need to avoid derivations that match the subpattern rooted at the last transitive node. If there is no such transitive node, then any derivation that uses the rules generated in line 5 does not match the pattern.

Example 18 Reconsider the program P consisting of the rules r_1, r_2 and r_3 given in Example 1, and the negation of the tree pattern shown in Fig. 5. The rules

$\texttt{dealsWith}^{\neg v_2'}\texttt{(a, b)}$:-$\texttt{dealsWith}^{\neg v_1'^t}\texttt{(b, a)}$, $a \neq$ France
$\texttt{dealsWith}^{\neg v_2'}\texttt{(a, b)}$:-$\texttt{dealsWith}^{\neg v_1'^t}\texttt{(b, a)}$, $b \neq$ Cuba

are added in line 5 due to the node v_2' and the rule r_2, and since $Tr(v_2') = v_1'$.

Then, in the case where $R(\ldots)$ locally matches v through a partial assignment A and v is not a leaf (line 6), in addition to the above rules, we further consider derivations that contain the partial assignment A that causes the match. In this case, a derivation that does not match a subpattern rooted by v, either has less than n children in the derivation, where n is the number of the children of v, or the derivation rooted by at least one of $R(\ldots)$'s children does not match one of the children of v, which is captured by the $neg\text{-}ex(\ldots)$ set. Thus we add for each $\beta' \in neg\text{-}ex(A(\beta), \{v_0, \ldots, v_n\})$ the rule $R^{\neg v^*}(y_0, \ldots, y_m) : -\beta'$ in lines 8–9. If v is not a leaf, any derivation that contains the partial assignment A matches the subpattern and thus we do not add rules in this case.

Example 19 $R(a, b)$ locally matches v_2' through the assignment $\{a \leftarrow France, b \leftarrow Cuba\}$, and v_2' is not a leaf, thus the rules

$\texttt{dealsWith}^{\neg v_2'}\texttt{(France, Cuba)}$:-$\texttt{dealsWith}^{\neg v_3'}\texttt{(France, f)}$, $\texttt{dealsWith}^{\neg v_3'}\texttt{(f, Cuba)}$
$\texttt{dealsWith}^{\neg v_2'}\texttt{(France, Cuba)}$:-$\texttt{dealsWith}^{\neg v_4'}\texttt{(France, f)}$, $\texttt{dealsWith}^{\neg v_4'}\texttt{(f, Cuba)}$

are added in line 9 by the algorithm to capture derivations that contain the partial assignment.

Finally, instead of adding rules for edb atoms that locally match v, the function `HandleEDBneg` in line 10 adds the rules $T^{\neg v}(x_0, \ldots, x_m) : -T(x_0, \ldots, x_m)$ and $T^{\neg v^t}(x_0, \ldots, x_m) : -T(x_0, \ldots, x_m)$ for each edb atom $T(x_0, \ldots, x_m)$ that does not locally matches v. For edb atom $T(x_0, \ldots, x_m)$ that locally matches v through a partial assignment A, the function adds the rules $T^{\neg v}(x_0, \ldots, x_m) : -T(x_0, \ldots, x_m), x_i \neq \sigma_i$ and $T^{\neg v^t}(x_0, \ldots, x_m) : -T(x_0, \ldots, x_m), x_i \neq \sigma_i$ for each $x_i \leftarrow \sigma_i$ pair in the assignment A.

Example 20 The edb atom $imports(\ldots)$ does not locally match v_4' and thus the rules

$\texttt{imports}^{\neg v_4'}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$
$\texttt{imports}^{\neg v_4'^t}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$

are added by `HandleEDBneg`. In addition, the function `HandleEDBneg` adds the rules

$\texttt{imports}^{\neg v_3'}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$, $a \neq$ France
$\texttt{imports}^{\neg v_3'^t}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$, $a \neq$ France
$\texttt{imports}^{\neg v_3'}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$, $b \neq$ tobacco
$\texttt{imports}^{\neg v_3'^t}\texttt{(a, b)}$:-$\texttt{imports(a, b)}$, $b \neq$ tobacco

Proposition 3 *Let P_p be the output of Algorithm 2 for input which is a program P and negated pattern p with root v^0. For every database D, we have that:*

$$trees(P, D) = \bigcup_{\tau \in trees(P_p, D)} origin(\tau) \quad (1)$$

$$p(P, D) = \bigcup_{t = R^{\neg v^0}(\ldots)} \bigcup_{\tau \in trees(P_p, D, t)} origin(\tau) \quad (2)$$

$$w(origin(\tau)) = w(\tau) \quad \forall \tau \in trees(P_p, D) \quad (3)$$

Proof 1. Clearly $trees(P, D) \subseteq \cup_{\tau \in trees(P_p, D)} origin(\tau)$ based on the same reasoning of Proposition 1 ($P \subseteq P_p$). For the opposite containment, consider the derivation tree τ formed from a rule generated by the algorithm. Rules generated by the algorithm may differ from the original rules in two ways: (1) an annotated original body and therefore removing the annotation from τ will result in a tree from $trees(P, D)$, (2) adding disequalities to the body which further restrict the trees generated by the rule, so these rules lead to the creation of a subset of derivation trees, and thus if such a rule participated in τ, then surely $origin(\tau) \in trees(P, D)$
2. Let $p|_v$ be the subpattern of p rooted at v. We prove by induction on the height of the pattern $p|_v$ that for every pattern node v it holds that

$$p|_v(P,D) = \bigcup_{t=R^{\neg v}(\ldots)} \bigcup_{\tau \in trees(P_p,D,t)} origin(\tau)$$

Base case: v is a leaf and τ is a derivation tree. There are two possible cases:

- v is not transitive. $\tau \in p|_v(P,D)$ iff one of the following holds: (1) the root of τ does not have the same relation name as v or (2) the root of τ contains different constants than v. Given the pattern p, the algorithm produced a rule with the head relation labeled by $\neg v$ for every rule that does not locally match v. Therefore, in case (1) $\tau \in \bigcup_{t=R^{\neg v}(\ldots)} \bigcup_{\tau \in trees(P_p,D,t)} origin(\tau)$. In case (2), let A be the assignment that makes the root of τ locally match v through the rule r. For every pair $x \leftarrow \sigma_i \in A$ the algorithm adds a labeled rule r' with the same head and body relations but adding the disequality $x \neq \sigma_i$. This way, the labeled rules are never assigned with the same constants that make the root of τ and v locally match. Similarly, if the relation of v is edb, for case (1) we would add all rules of the form $T^{\neg v}(\ldots) :- T(\ldots)$ if v does not have the relation T and for case (2) we also add rules with disequalities.
- v is transitive. Now $\tau \in p|_v(P,D)$ iff (1) or (2) from the previous case holds *and* (3) no node of τ satisfies v. In this case $Tr(v) = v$ and thus in the rules added by the algorithm the atoms in the body are annotated by $\neg v^t$ (using $tr\text{-}neg(\ldots)$ in line 5). Note that since v is a leaf, every rule added by the algorithm where the head is annotated with $\neg v^t$, must be added in line 5 and thus the every atom in the body of such rule must be annotated with $\neg v^t$ as well, or by the function `HandleEDBneg`. Those rules are added only for relation (and assignments) that does not satisfy v, and thus every derivation that use such rule clearly can not satisfy the pattern. Clearly, for every derivation tree τ that use thus rules $origin(\tau) \in trees(P,D)$

Suppose that the proposition holds for all v s.t. $p|_v$ has height $< k$. Let v be a pattern node where $p|_v$ has height k, with children $v_0, \ldots, v_n$.

- v is not transitive. Let $\tau \in p|_v(P,D)$. This means that at least one of the following holds: either the root of τ does not satisfy v or a subtree of τ does not satisfy a subpattern rooted at one of the children of v. In the former case, the proposition can be proven similarly to the base case of the induction. The latter case holds iff (1) the number of children of v in the pattern is greater than the number of children of the root of τ **or** (2) $\exists p_j \; \forall u$ s.t. u is the child of the root of τ and for the subtree rooted at it, τ_j it holds that $\tau_j \in p|_{\neg v_j}(P,D)$. By the induction hypothesis, $\tau_j \in \bigcup_{t=R^{\neg v_j}(\ldots)} \bigcup_{\tau \in trees(P_p,D,t)} origin(\tau)$ for all τ_j who are rooted at the children of v. There are two kinds of rules added by the algorithm. Cases (1) and (2) are captured by $neg - ex(\ldots)$ and thus the appropriate rules are added by the algorithm. Hence, the last derivation step in τ can be done using a labeled rule r because either the head of r does not locally match v or a subtree of τ does not satisfy a subpattern rooted at v. So the last derivation step in τ must by done by a rule r produced by the algorithm. Therefore, the derivation tree obtained by replacing $R^{\neg v}(\ldots)$ with $R(\ldots)$ and replacing each subtree τ' rooted at the children of the root of τ with $origin(\tau')$ is $origin(\tau)$ and it holds that $\tau \in \bigcup_{t=R^{\neg v}(\ldots)} \bigcup_{\tau \in trees(P_p,D,t)} origin(\tau)$.
- The case where v is transitive is similar to the case of a transitive leaf.

3. This is immediate given Proposition 1, as the new rules have the same weights as the original ones. □

5.2 Disjunction and conjunction

Disjunction and conjunction of patterns may be performed by repeatedly applying Algorithm 1. In the following we refer a single (may be negated) tree pattern as a *simple* pattern and pattern that is composed using disjunctions and/or conjunctions of simple patterns as a *combined* pattern.

Given a program P and a combined pattern p, P_p can be computed using the following procedure: If $p = p_1 \vee p_2$, and p_i $(i = 1, 2)$ is a simple pattern, use Algorithm 1 (or Algorithm 2) to intersect P with p_i and obtain P_{p_i}. If p_i is combined recursively apply the same procedure to obtain P_{p_i}, and return $P_2 = P_{p_1} \cup P_{p_2}$. If $p = p_1 \wedge p_2$, first computes P_{p_1} to obtain P_{p_1} (either by using Algorithms 1 or 2 for a simple pattern, or by a recursive call if p_1 is combined), then instrument P_{p_1} with p_2 (using Algorithms 1 or 2) to obtain P_p.

Example 21 Recall the program P composed of r_1, r_2, r_3 shown in Example 1, and consider the combined patten $p = p_2 \wedge p'$ where p_2 is shown in Fig. 3b and p' is the negated pattern depicted in Fig. 5. The procedure for combined patterns instrumentation first computes the program P_{p_2} as shown in Example 14. Then it intersects the program P_{p_2} with the pattern p' using Algorithm 2. The rules in $P_{p_2 p'}$ may consist of annotation from both pattern. For instance, the rules

```
dealsWith^{v0 ¬v'1}(Cuba, France):-
                        dealsWith^{v'1 ¬v'2}(France, Cuba)
dealsWith^{¬v'1}(Cuba, b):-dealsWith^{¬v'2}(b, Cuba),
                                b ≠ France
dealsWith^{v0 ¬v'1}(Cuba, b):-dealsWith^{v'1 ¬v'1}(Cuba, a)
```

are generated by the algorithm using the node v'_1 in p' and the annotated rule

$\texttt{dealsWith}^{v_0}\texttt{(Cuba, b):-dealsWith}^{v'_1}\texttt{(b, Cuba)}$

which is obtained in the first instrumentation and thus is part of P_{p_2}. Derivations of facts with the annotation $v_0 \neg v'_0$ are derivations that match the combined pattern.

6 Finding top-k derivation trees

The second step of the algorithm is finding top-k derivation trees that conform to the `selPQL` query pattern, based on the instrumented program and now also the input database. We next describe the algorithm for top-k; then we will present a heuristic optimization.

The algorithm operates in an iterative manner. We start by explaining the algorithm for finding the top-1 derivation. The generation of the top-1 qualifying tree is done alongside with bottom-up standard (provenance-oblivious) evaluation of the datalog program with respect to the database. We then extend the construction to top-k for $k > 1$.

6.1 Top-1

Algorithm 3 computes the top-1 derivation

in a bottom-up manner. Each entry in the data structure $DTable$ represents the top-1 derivation tree of a fact t, and contains the fact itself, its top-1 derivation weight, and pointers to the entries in the table corresponding to the derivation trees of the "children" of t in the derivation. Starting with a set of all edb facts (with empty trees) in $DTable$ (line 1), in each iteration, the algorithm finds the set of facts that can be derived via facts in $DTable$ using a single application of a rule in P (line 3). For each such candidate we compute its best derivation out of those using facts in $DTable$ and a single rule application (this is done by a procedure called `Top`). The fact for which the maximal (in terms of weight) such derivation is found is added to $DTable$ (Line 4). Finally, the algorithm returns the entries in $DTable$ of facts that match the root node v^0 of the pattern.

Example 22 Consider the program given in Example 14, and the database D shown in Table 1. Algorithm 3 first initializes $DTable$ with the edb atoms from D, each with its weight (in this case all weights are 1). Then, in lines 2–4, the algorithm finds the set of facts that can be derived via the facts in $DTable$. In the first iteration the fact $t_3 = exports^{v'_1}(Cuba, tobacco)$ can be derived with weight 1 using the edb fact $t_1 = exports(Cuba, tobacco)$ and the rule denoted r' in Example 14. Other facts can be derived in the first iteration but t_3 is the fact with maximal weight. The algorithm thus adds $(t_3, 1, \{*t_1\})$ to $DTable$, where $*t_1$ is a pointer to the entry of t_1 in $DTable$. In the next iteration, the algorithm can derive the fact $t_4 = dealsWith^{v'_1}(France, Cuba)$ using t_3 and the edb fact $t_2 = imports(France, tobacco)$ with overall weight of 0.5. When t_4 is selected in Line 4 (other facts may be chosen due to ties), the algorithm adds $(t_4, 0.5, \{*t_2, *t_3\})$ to $DTable$. After t_4 is added to $DTable$, the fact $t_5 = dealsWith^{v_0}(Cuba, France)$ can be derived with overall weight of $0.5 \cdot 0.8 = 0.4$, and $(t_5, 0.4, \{*t_4\})$ is added to $DTable$.

Algorithm 3: Top-1

input : Weighted Datalog Program P, Database D
output: Top-1 tree for facts of the form $R^{v_0}(\ldots)$

1 Init $DTable$ with $(t, 0, null)$ for all $t \in D$;
2 **while** *$DTable$ changes* **do**
3 Let $Cand$ be the set of all facts derived via facts in $DTable$ and are not in it;
4 Add $[\arg\max_{t \in Cand} \texttt{Top}(t, DTable, P)]$ to $DTable$
5 **return** the entries of all $e \in DTable$ s.t. the fact t of e is of the form $R^{v_0}(\ldots)$;

Complexity. The algorithm adds in each iteration a new fact. The number of facts that can be derived is polynomial in the Database size, thus this is an upper bound on the number of iterations. Lines 3 and 4 are both polynomial in the Database size, and therefore the overall time complexity and output size of Algorithm 3 are polynomial in the Database size.

6.2 Top-K

The algorithm for *TOP-K* computes the top-i derivations for each fact $t \in P_p(D)$ in a bottom-up manner for $2 \leq i \leq k$. For each i it essentially repeats the procedure of Algorithm 3, but starting with $DTable$ consisting of the top-$(i-1)$ trees, i.e., τ_t^j for all $t \in P_p(D)$ and $j < i$. A subtlety is that different trees in $P_p(D)$ may have the same origin in $P(D)$, thus computing top-k using the instrumented program should be done carefully in order to avoid generating the same tree (up to annotations) over and over again.

To this end, we say that a derivation tree τ_t for a fact t is a *top-i candidate*, if one of the following holds: (i) τ_t uses at least one "new" fact that was added in the i'th iteration or (ii) the last derivation step in τ_t is different from the last derivation step in τ_t^j for all $1 \leq j < i$, such that $origin(\tau_t) \neq origin(\tau_t^j)$. Given the top-$(i-1)$ derivation trees, to compute i'th best tree for each fact we compute in a

bottom-up manner top-i candidates that can be derived from facts in $DTable$ using a single rule application. Then we select the candidate τ_t with maximal weight (out of candidates computed for all facts) and add it to a new entry t^i in $DTable$. The step of computing the i'th best tree terminates when there are no more new facts to add to $DTable$. To find the top-k derivations we may simply compute the top-i for each $1 \leq i \leq k$. After the k'th iteration $DTable$ contains a compact representation the top-k derivation trees. The enumerate of top-k trees for each fact may then simply be done by pointer chasing.

Complexity. The algorithm for *TOP-K* computes for each $1 \leq i \leq k$ the top-i derivation trees for each fact. For each i, the computation of the top-i trees consists of at most $DTable$ iterations, each polynomial in $DTable$ with exponent $|P|^{w(p)}$. A subtlety is in the verification that two compactly represented trees do not have the same origin: we note that a recursive such comparison may be performed in time that is polynomial in $DTable$ with the exponent depending on the maximal tree width (maximal number of children of a tree node), which in turn depends only on the program size. Next, $DTable$ contains at most k entries for each fact $t \in P_p(D)$ where P_p is the instrumented program given the program P and pattern p. The number of facts $t \in P_p(D)$ is at most $|D|^{|P_p|} = |D|^{(|P|^{w(p)})}$, where $|D|$ is the extensional database size, thus on the i'th step, the size of $DTable$ is bounded by $i \cdot |D|^{(|P|^{w(p)})}$. Therefore, the time complexity of the i'th step is $O(i^2 \cdot |D|^{O(|P|^{w(p)})})$. The complexity of computing the top-k derivation trees is therefore

$$\sum_{i=1}^{k} O(i^2 \cdot |D|^{O(|P|^{w(p)})}) = O(k^3 \cdot |D|^{O(|P|^{w(p)})})$$

Finally, generating the top-k trees from $DTable$ is linear in the output size, and thus the overall complexity of `TOP-K` is $O(k^3 \cdot |D|^{O(|P|^{w(p)})} + |out|)$, where $|out|$ is the output size.

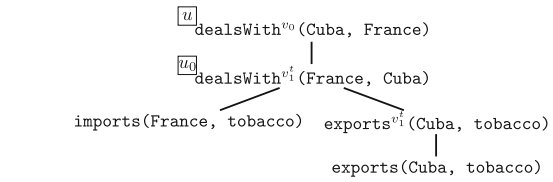
6.3 Alternative heuristic top-k computation

An alternative approach for finding top-k derivations is based on ideas of the algorithm for k shortest paths in a graph [20]. The basic idea is to obtain the i'th best derivation tree of a fact t by modifying one of the top-$(i-1)$ derivation trees of t. Each node u with children $u_0, \ldots, u_m$ in a derivation tree τ for a fact $t \in P_p(D)$, corresponds to an instantiation of a derivation rule r in P_p. Given a node $u \in \tau$, a *modification* of u in τ is using a different instantiation to derive u, i.e., using different derivation rule $r' \in P_p$ or a different assignment to the variables in r s.t. for the obtained tree τ' it holds that $origin(\tau) \neq origin(\tau')$. We say that two modifications are different if for their results τ_1 and τ_2 satisfy $origin(\tau_1) \neq origin(\tau_2)$.

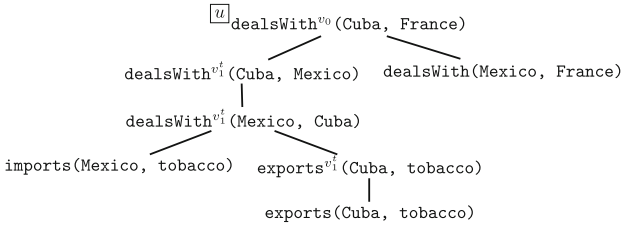
Given a derivation tree τ, we denote by $\tau_{u,r,\sigma}$ the derivation tree obtained by modifying u in τ using r and σ. We define $\delta(u, r, \sigma) = w(\tau) - w(\tau_{u,r,\sigma})$. Intuitively, $\delta(u, r, \sigma)$ is the "cost" of the modification. Note that the i'st best derivation tree can be obtained by a modification of any one of the top-$(i-1)$ trees. Given the top-$(i-1)$ derivation trees for the fact t, the next best derivation can be computed as follows: traverse each one of the top-i trees τ in a top–down fashion, compute the cost of all possible different modifications (without recomputing trees that were already considered; this can be done by tracking the rules and assignment used for each modification), and find the modification of minimal cost. The algorithm for top-k computes, for each output fact, the top-k derivation trees as described above, and terminates when we find top-k derivation or when there are no more modifications to apply on the trees found by the algorithm. Note that the consideration of modifications can be done without materializing the derivation trees, but rather only using $DTable$. A subtlety is that a fact t may have multiple occurrences in a derivation tree τ; however, it appears only once in $DTable$. Thus, modifying the entry of t in $DTable$ would result in modifying the subtrees rooted at all occurrences of t (instead of modifying a subtree rooted at one occurrence of t). To avoid these modifications, we generate a new copy of all the facts in the path from the root of τ to t (including t) for each modification of t's subtree.

Example 23 Reconsider the output program of the algorithm in Example 14. The top-2 derivation trees for the fact $dealsWith(Cuba, France)$ are shown in Fig. 6a, and we next partially illustrate the computation process using the alternative approach. The top-1 derivation tree τ^1 of the fact $dealsWith(Cuba, France)$ is depicted in Fig. 6. The nodes u and u_0 correspond to the derivation rule

[r] `dealsWith`^{v_0}`(Cuba, b):- dealsWith`$^{v_1^t}$`(b, Cuba)`



(a) Top-1



(b) Top-2

Fig. 6 Top-2 Derivation Trees (with annotations)

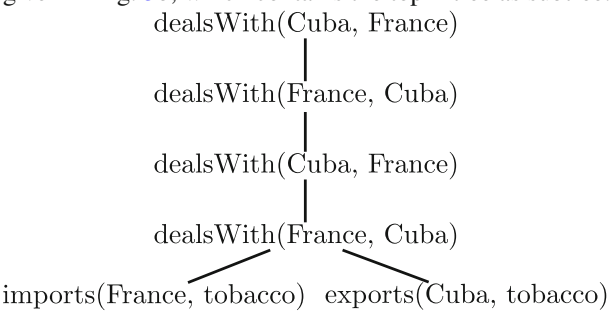
with the assignment $\sigma = \{b \leftarrow France\}$. The weight of τ^1 is 0.4. By replacing r with

$[r']$ dealsWith^{v_0}(Cuba, b):- dealsWith$^{v'_1}$(Cuba, f), dealsWith(f,b)

and the assignment $\sigma' = \{b \leftarrow France, f \leftarrow Mexico\}$, we obtain the top-2 derivation tree $\tau^1_{\delta(u,r',\sigma')} = \tau^2$. The weight of τ^2 is 0.28 and $\delta(u, r', \sigma') = 0.12$. $origin(\tau^1)$ and $origin(\tau^2)$ are shown in Fig. 2 (as τ_2 and τ_3, respectively).

6.4 Diversification

Consider the Datalog program of our running example. The following tree is the top-3 derivation tree w.r.t. the pattern given in Fig. 3b, which contains the top-1 tree as subtree.



Our paradigm may be adapted to support *diversification*, and avoid such derivations, by intersecting the program with the negated top-i result before computing the top-$(i + 1)$ result.

For instance we may intersect the program with a negated pattern consisting of a new root labeled by a wildcard, connected by a transitive edge to a copy of the i'th result (denoted by $gen(\tau_i)$ for the top-i derivation tree τ_i); this will make sure that the i'th tree will not appear as a subtree in the $(i + 1)$ result. This approach is manifested in Algorithm 4 which iteratively computes the top-i trees as follows. It first instruments the program using Algorithm 1 with the pattern to form the new program (line 1). The algorithm then iterates k times to computes the top-i tree, τ_i using Algorithm 3 and the corresponding general pattern $gen(\tau_i)$ (lines 3 – 4). In the final step of the iteration at line 5, it builds a new program by intersecting the negated general pattern $gen(\tau_i)$ with the current program to generate the program computing the top-$(i + 1)$ result.

Example 24 We exemplify Algorithm 4 with the input $k = 2$, the program consisting of the rules r_1, r_2, r_3 taken from the program depicted in Fig. 1 as P_0, the database presented in Fig. 2 as D and the pattern depicted in Fig. 3c as p_0. At the first step, we instrument P_0 with the pattern p_0 to receive the instrumented program P_1 from Example 14 (line 1). We then assign to τ_1 the output of Algorithm 3 with p_1 and D as input (line 3). In this case, τ_1 is the tree depicted in Fig. 6a. We conclude the first iteration by assigning to p_1 the pattern $\neg gen(\tau_1)$ (line 4) and we use the instrumentation algorithm with adaptation for negation (line 5) to compute the instrumented program P_2 partially depicted in Examples 18, 19, 20 and 21. We then employ Algorithm 3 again with the input P_2 and D to receive the tree τ_2 depicted in Fig. 6b and finally we assign p_2 the pattern $\neg gen(\tau_2)$. Concluding the run of the algorithm, in line 6, we return the trees τ_1, τ_2 depicted in Fig. 6.

Algorithm 4: Top-K

input : Integer k, Weighted Datalog Program P_0, Database D, Pattern p_0
output: Top-k tree for facts of the form $R^{v_0}(\ldots)$

1 $P_1 \leftarrow Instrumentation(P_0, p_0)$;
2 **for** $i = 1, \ldots, k$ **do**
3 $\quad \tau_i \leftarrow Top - 1(P_i, D)$;
4 $\quad p_i = \neg gen(origin(\tau_i))$;
5 $\quad P_{i+1} \leftarrow InstrumentationWithNegation(P_i, p_i)$;
6 **return** τ_i for all $i = 1, \ldots, k$;

7 Implementation and optimizations

We have implemented a system prototype called selP (for "selective provenance"), demonstrated in [17]). selP serves both for demonstrating the usefulness of our solution and for conducting the experimental study detailed in the next section. The system is implemented in JAVA and runs on Windows 7. Its architecture is depicted in Fig. 4. The implementation extends and modifies the implementation of IRIS [32], a JAVA-based system for datalog evaluation. We start by describing the general architecture of selP, then detail some important optimizations.

The user feeds the system with a datalog program (as text) and is provided with dedicated interfaces for writing selPQLqueries: weights can be attached to rules in the text (assigning weights to tuples can be simulated through rules, see discussion in Sect. 3.2), and patterns (as defined in Sect. 3.1) may be drawn using a dedicated screen (shown in Fig. 7). Then, the evaluation proceeds by generating the instrumented program using the appropriate algorithm based on the selPQLquery structure (Algorithm 1, Algorithm 2, or their variants described in Sect. 5.2). The instrumented program along with an input database is fed to the *TOP-K* component, that includes an implementation of Algorithm 4 (which involves invocations of Algorithms 3 and 2).

Unsurprisingly, we have observed that the most important factor affecting the system's performance is the complexity of the instrumented program. Specifically, two features of the program were crucial: the number of rules, and the generality of the program in terms of the existence of constants. To improve performance in practice, we employ optimizations

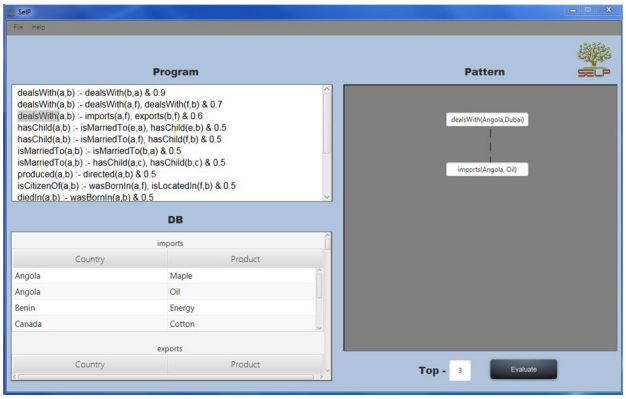


Fig. 7 Input Screen

that simplify the instrumented program (without violating correctness in the sense of Propositions 1 and 3), thereby reducing the time of the top-k computation that follows. We next explain the optimizations; see discussion of the practical effect on the execution time in Sect. 8.

Use of constants. Recall that the instrumentation algorithms may assign constants in generated rules, when such constants appear in the pattern. In this case, a subset of the resulting rules, namely those generated for "direct satisfaction" (see Sect. 4) will include variables that, by definition can only be assigned a constant (for every database). The optimization is then to "propagate" the assignment of constants in a bottom-up manner (i.e., instantiating the appropriate constants in the rules of the parent node, based on the constants in the rules of the child node), thus generating rules that are more specific and reduce the workload in the top-k computation process.

Avoiding redundant rules. Several components of our solutions may generate, in different circumstances, rules that are redundant in the sense that they may be omitted and Propositions 1 and 3 continue to hold. We next detail 4 such cases and their treatment.

First, rules generated for direct satisfaction (either non-transitive parent, or the direct satisfaction rules for the transitive case) of leaves labeled by edb facts are redundant. For a parent node v_p of such a leaf v_l, Algorithm 1 generates rules that contain an atom in the body with the same relation name as in the rules generated for v_l. As a result, every derivation using these rules will also use the atom corresponding to the leaf. We therefore avoid generating those rules.

Second, Algorithm 2 may produce annotated rules whose sole purpose is to copy a relation from the original program (lines 4, 5). These rules can be discarded, and the rules using their annotated head can be changed back to the original relation.

Furthermore, some of the rules generated by Algorithm 2 may be instantiations of other rules where the variables are replaced by specific constants, and these can also be safely omitted.

Finally, recall that Algorithm 4 iteratively intersects a program with boolean combination of patterns. In each iteration, rules that were generated in the preceding iteration may become redundant and can be safely removed. For example, consider executing Algorithm 4 with $k = 3$. In the third iteration of the loop, rules whose head is annotated with $v_0 \neg v_0'$ are redundant since the pattern can only be derived from rules whose head is annotated with $v_0 \neg v_0' \neg v_0''$.

8 Experiments

We next describe the dedicated benchmark (including both synthetic and real data) developed for the experiments, and then the experimental results.

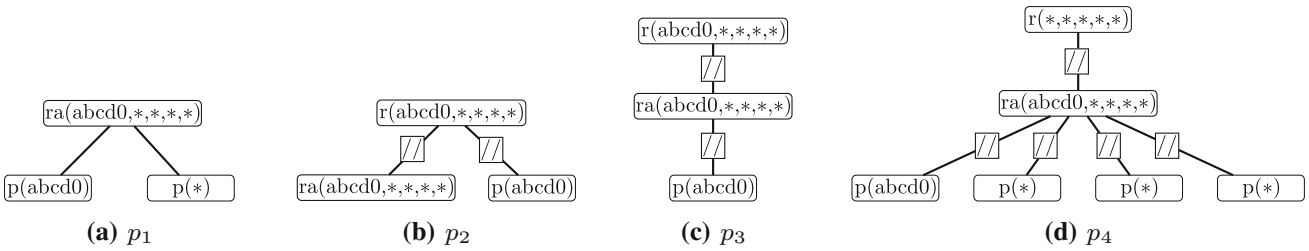
8.1 Evaluation benchmark

We have used the following datasets, each with multiple `selPQL` queries (different number of requested results and different patterns, varying in size and structure), and for increasingly large output databases. The weights in the reported results are all elements of the monoid $([0, 1], \cdot, 1, <)$; we have experimented with all other monoids given in Example 9, but omit the results for them since the observed effect of monoid choice was negligible.

1. **IRIS** We have used the non-recursive datalog program and database of the benchmark used to test IRIS performance in [32]. The program consists of 8 rules and generates up to 4.26M tuples; weights have been randomly assigned in the range [0,1]. The program is:

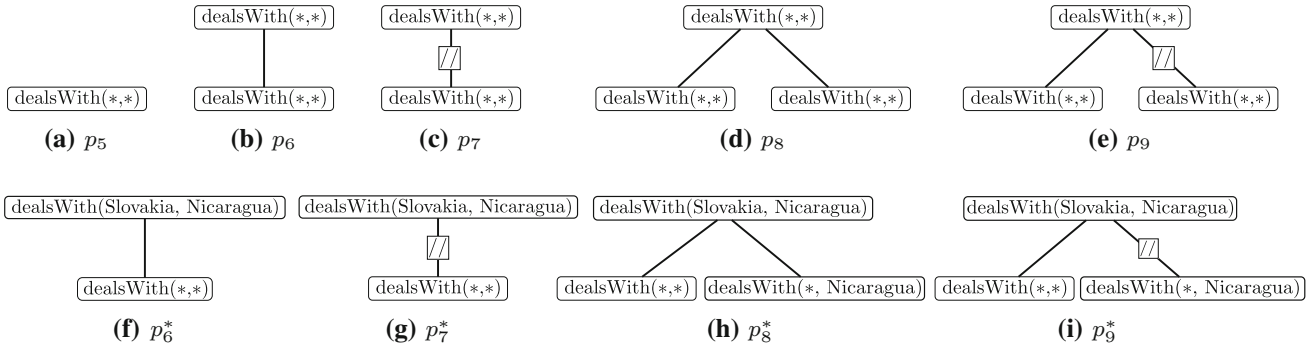
```
ra(A,B,C,D,E) :- p(A),p(B),p(C),p(D),p(E)
rb(A,B,C,D,E) :- p(A),p(B),p(C),p(D),p(E)
r(A,B,C,D,E) :- ra(A,B,C,D,E), rb(A,B,C,D,E)
q(A) :- r(A,B,C,D,E)
q(B) :- r(A,B,C,D,E)
q(C) :- r(A,B,C,D,E)
q(D) :- r(A,B,C,D,E)
q(E) :- r(A,B,C,D,E)
```

2. **AMIE** We have used the recursive datalog program presented in Fig. 1 consisting of rules mined by AMIE [24]. These rules were automatically translated into datalog syntax; the weights were also assigned to the rules by AMIE and reflect the confidence of each of the rules. The underlying input database is that of YAGO [58], and the program generates up to 1.2M tuples.
3. **Explain** We have used a the following variant of the recursive datalog program described in [3], as a use case



(a) p_1 **(b)** p_2 **(c)** p_3 **(d)** p_4

Fig. 8 Example Patterns for IRIS



(a) p_5 **(b)** p_6 **(c)** p_7 **(d)** p_8 **(e)** p_9

(f) p_6^* **(g)** p_7^* **(h)** p_8^* **(i)** p_9^*

Fig. 9 Example Patterns for AMIE

for the "explain" system (aggregation and arithmetics omitted):

```
b_o_m(Part, C) :- subpart_cost(Part, SubPart, C)
subpart_cost(Part, Part, Cost) :-
                    basic_part(Part, Cost)
subpart_cost(Part, Subpart, Cost) :-
           assembly(Part, Subpart, Quantity),
           b_o_m(Subpart, TotalSubcost),
```

The database was randomly populated and gradually growing so that the output size is up to 1.17M tuples, and weights have been randomly assigned in the range [0,1].

4. **Transitive Closure** Last, we have used a recursive datalog program consisting of 3 rules and computing Transitive Closure in an undirected weighted graph. The database was randomly populated to represent undirected fully connected weighted graphs, yielding output sizes of up to 1.7M tuples.

Size of input instances. We have experimented with an increasing database size for each of the datasets. We list below the sizes of the input instances, per dataset.

1. **IRIS**: 11, 12, 13, 14, 15, 16, 17.
2. **AMIE**: 67314, 75318, 85449, 98787, 117661, 142677, 177718, 231916, 338642, 626799.
3. **Explain**: 156, 306, 456, 606, 756, 905, 1051, 1201, 1351, 1504, 1655.
4. **Transitive Closure**: 101, 201, 301, 401, 501, 601, 701, 801, 901, 1001, 1101, 1201, 1301.

Patterns. The `selPQL` patterns used in our experiments are shown in Figs. 8, 9, 10 and 11. For the AMIE, TC and Explain datasets, the program is recursive and so the patterns select out of an infinite set of trees. For IRIS, the selectivity of all 4 patterns is as follows: denoting the size of the relation p by $|p|$, all patterns select $|p|^4$ derivation trees out of $8 \cdot |p|^5$ derivation trees overall.

8.2 Baselines: no provenance tracking and full provenance tracking

To our knowledge, no solution for evaluation of top-k queries (or tree patterns) over datalog provenance has been previously proposed. To nevertheless gain insight on alternatives, we have compared, in terms of incurred time and space, our solution to two possible extremes: (1) standard, seminaive evaluation with no provenance tracking, using IRIS implementation; and (2) compact representation of full provenance. In this respect, recall that our proposed solution is based on the idea of having the user specify `selPQL` patterns before the execution of the datalog program, and instrumenting the program to generate only relevant provenance. An alternative approach, and one that is pursued in different contexts in previous work [34,36], is to track full provenance information, and then visualize and/or allow users to query it. In the context of datalog, previous work [3,19,30] has proposed full provenance tracking methods; specifically [19] has proposed a circuit-based construction to reduce the provenance size. However, we observe that

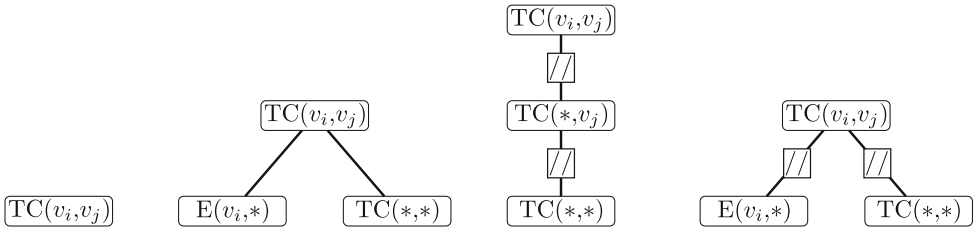


Fig. 10 Example Patterns for TC

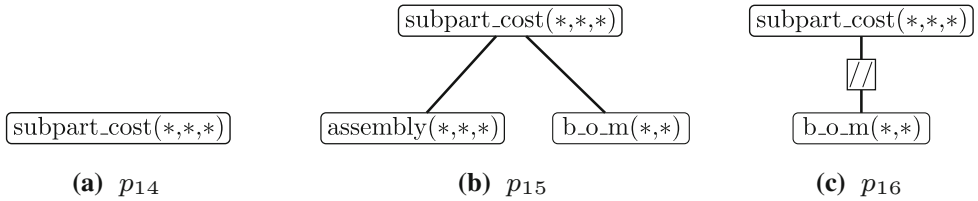


Fig. 11 Example Patterns for Explain

for complex recursive datalog programs, full provenance size grows rapidly with respect to the provenance-oblivious output database size, and so such solutions fail to scale. Specifically, we have implemented the (boolean) circuit-based approach of [19] in an iterative fixpoint algorithm, which is executed, similarly to our solution, along side with standard seminaive evaluation. The overhead of full provenance generation done in this way is exemplified in Fig. 14. In addition, we note that a compact circuit-based representation is highly complex and even when it can be realized, it can only serve as an internal intermediate step toward answering queries over provenance rather than be directly visualized.

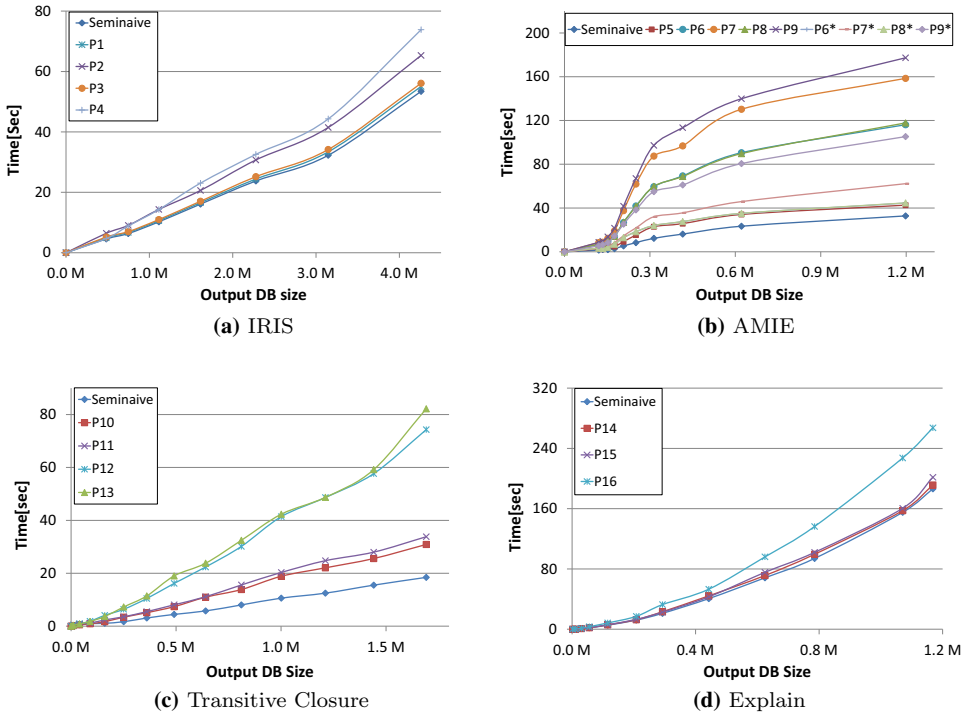
All experiments were executed on Windows 7, 64-bit, with 8GB of RAM and Intel Core Duo i7 2.10 GHz processor.

8.3 Experimental results

Figure 12 presents the execution time of standard seminaive evaluation and of selective provenance tracking for the four datasets and for different `selPQL` queries of interest (fixing $k = 3$ for this experiments set). Full provenance tracking has incurred execution time that is greater by order of magnitude and is thus omitted from the graphs and only described in text.

In Fig. 12a, the results for the IRIS dataset are presented for the 4 different patterns depicted in Fig. 8: (p_1) binary tree pattern with three nodes without transitive edges and (p_2) with two transitive edges, (p_3) three nodes chain pattern with two transitive edges, and (p_4) six-node pattern with three levels and four transitive edges. The pattern width and structure unsurprisingly has a significant effect on the execution time, but the overhead with respect to seminaive evaluation was very reasonable: 38% overhead w.r.t. the evaluation time of seminaive even for the complex six-node pattern and only 3–21% for the other patterns. The absolute execution time is also reasonable: 56–65 s for the different patterns and for output database of over 4.2M tuples (note that for this output size, the execution time of standard seminaive evaluation is already 53 s. In contrast, generation of full provenance was infeasible (in terms of memory consumption) beyond output database of 1.6M tuples, taking over 5 min of computation for this size.

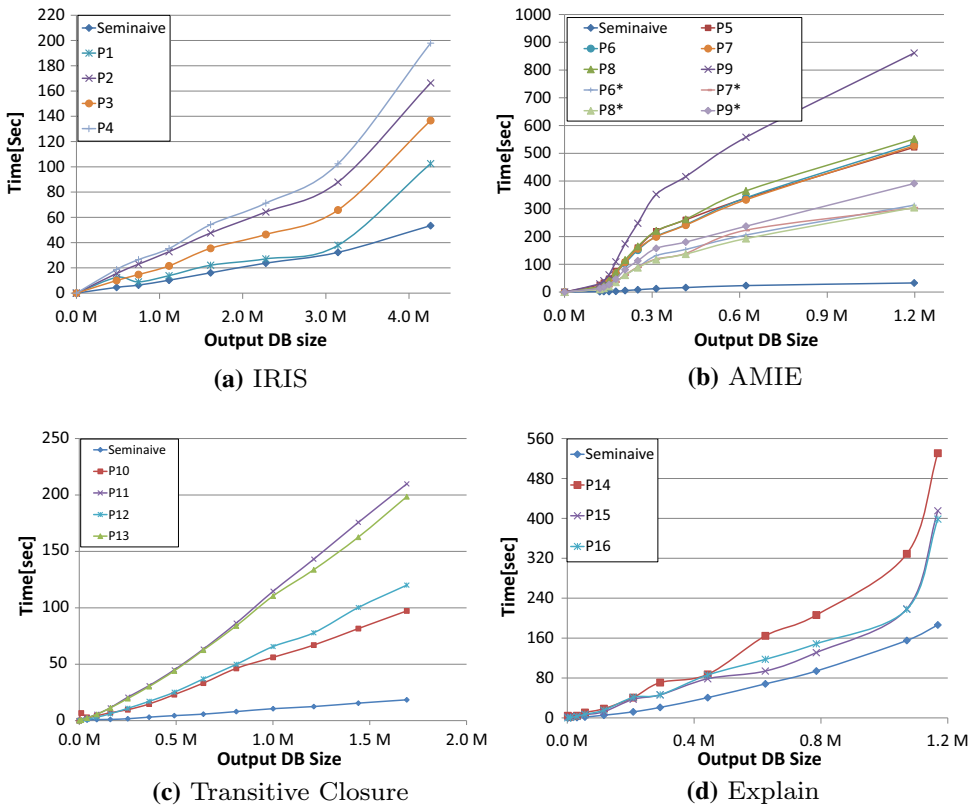
As explained above, the program we have considered for the AMIE dataset is much larger and more complex. Full provenance tracking was completely infeasible in this complex settings, failing due to excessive memory consumption beyond output database of 100K tuples. Of course, the complex structure leads to significantly larger execution time also for seminaive and selective provenance tracking. It also leads to a larger overhead of selective provenance tracking, since instrumentation yields an even larger and more complex program. Still, the performance was reasonable for patterns of the flavor shown as examples throughout the paper. We show results for the AMIE dataset and 9 different representative patterns that are presented in Fig. 9. Five patterns without any constants (only wildcards): (p_5) a single node pattern, (p_6) a 2-node pattern with a regular edge and (p_7) with a transitive edge, (p_8) a binary 3-node pattern with regular edges, and (p_9) with one transitive edge. The other 4 patterns are (p_i^*) for all $6 \leq i \leq 9$, where each (p_i^*) has the same nodes and edges of (p_i), but with half of the wildcards replaced by constants. The results are shown in Fig. 12b. We observe that the "generality" of the pattern, i.e., the part of provenance that it matches, has a significant effect on the performance. For the "specific" patterns p_i^*, the computation time and overhead was very reasonable: the computation time for 1.2M output tuples was only 44.5 s (1.3 times slower than semi-



(a) IRIS **(b)** AMIE

(c) Transitive Closure **(d)** Explain

Fig. 12 Heuristic top-3: time of computation as a function of DB size



(a) IRIS **(b)** AMIE

(c) Transitive Closure **(d)** Explain

Fig. 13 Diverse top-3: Time of computation as a function of DB size

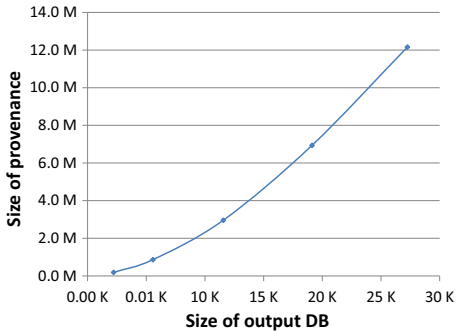


Fig. 14 Full provenance size (AMIE dataset)

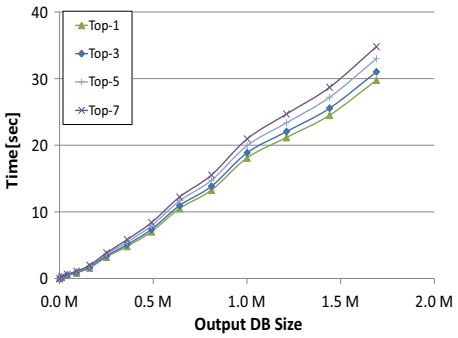


Fig. 15 Varying K (TC dataset)

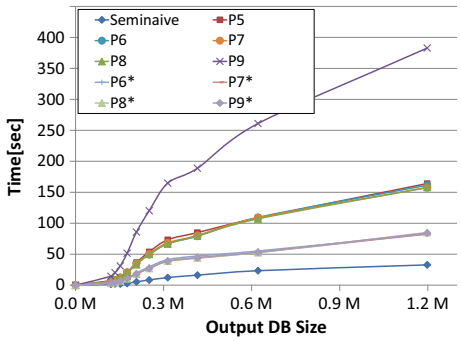


Fig. 16 Boolean combination of patterns (AMIE dataset)

naive) for p_6^*. For p_7^* and the same number of output tuples it took 62 s (less than 2 times slower than seminaive), 44.6 s (1.3 times slower than seminaive) for p_8^* and 105 s (3.2 times slower than seminaive) for p_9^*. The patterns containing only wildcards lead to a larger instrumented program, which furthermore has more eventual matches in the data, and so computation time was greater (but still feasible). The computation time for 1.2M output tuples was less than a minute (and 61% overhead w.r.t. seminaive in average) for p_5, less than 2 min (3.5 times slower than seminaive) for p_6, 2.6 min (4.8 times slower) for p_7, and less than 2 and 2.9 min (3.6 and 5.4 times slower) for p_8 and p_9, respectively.

In Fig. 12c, we present the results for the TC dataset and 4 different patterns: (p_{10}) a single node, (p_{11}) 3-nodes binary tree pattern with regular edges, (p_{12}) 3-nodes chain pattern with 2 transitive edges, and (p_{13}) binary tree pattern with three nodes and 2 transitive edges. All of these patterns are shown in Fig. 10. We observe a non-negligible but reasonable overhead with respect to seminaive evaluation (and the execution time is generally smaller than for the AMIE dataset). The execution time for 1.7M output tuples for p_{10} was 31 s (and 56% overhead with respect to seminaive in average), 33 s for p_{11} (1.8 times slower than seminaive in average), 74 s for p_{12} (4 times slower) and 82 s for p_{13} (4.5 times slower than seminaive). Here full provenance tracking was extremely costly, requiring over 6.5 hours for output database size of 1.7M tuples.

Figure 12d displays the results for the "explain" dataset. We considered 3 different patterns: (p_{14}) a single node, (p_{15}) a 3-nodes binary tree pattern with regular edges and (p_{16}) a 2-node pattern with a transitive edge (see Fig. 11). The computation time for 1.16M output tuples was less than 3.2 min (7% overhead w.r.t seminaive) for p_{14}, 3.3 min (10% overhead w.r.t seminaive) for p_{15} and 4.4 min (85% overhead w.r.t seminaive) for p_{16}. Full provenance tracking has required over 2 h even for an output database size of 115K.

From top-1 to top-k So far we have shown experiments with a fixed value of $k = 3$. In Fig. 15 we demonstrate the effect of varying k, using the TC dataset and fixing the pattern to be p_{10}. The overhead due to increasing k is reasonable, due to our optimization using the heuristic algorithm for *TOP-K* (after top-1 trees were computed): about 6%, 13%, and 21% average overhead for top-3, top-5 and top-7, respectively, with respect to top-1 execution time. Similar overheads were observed for other patterns and for the other datasets. Our optimization was indeed effective in this respect, outperforming the non-optimized version with a significant gain, e.g., average of 64% for $k = 3$, 77% for $k = 5$ and 82% for $k = 7$ (and again the trend was similar for the other patterns and datasets).

*Full provenance tracking*W̃e have highlighted throughout the section the infeasibility of full provenance tracking for the various settings. To illustrate the size overhead, we show in Fig. 14 the size of full provenance as a function of the output size, for the AMIE dataset. Observe the huge overhead of full provenance tracking, leading to failure of the solution beyond 25K output tuples.

Boolean combinations of patterns A näive way to examine Algorithm 2 is to input a negated pattern and program and use the instrumented program as input to Algorithm 3 which will derive the top-k trees (by one of the approaches described in this paper). However, intersecting a program with a negated pattern (especially a selective one, containing constants) results in a complex program that is designed to track almost the full provenance, which we have demon-

strated to be infeasible. Another possibility is to examine its performance over a boolean combination of patterns which involves a conjunction of a regular pattern and a negated pattern of the form $p_1 \wedge \neg p_2$. That is, running Algorithms 1 and 2 on the input program (in that order) and then inputting the instrumented program to Algorithm 3. This approach results in a program focused on deriving trees that match p_1 but do not match p_2. Specifically, we have examined the case where p_2 is a pattern that only contains constants. Figure 16 shows the results for all the patterns in the AMIE dataset. Runtimes for the patterns p_5, p_6, p_7, p_8, p_9, p_6^*, p_7^*, p_8^*, p_9^* are 164, 161.4, 157.4, 157.9, 382.8, 82.3, 82.8, 85.1, 84.2 s, respectively, for 1.2M output tuples. As expected, there is an overhead to the evaluation as opposed to the evaluation of a program which was intersected with a single regular pattern, even when using the heuristic approach to find the top-3 (Fig. 12b).

Performance of Algorithm 4 Figs. 13a–c, and Fig. 13 show the results for Algorithm 4 for the aforementioned programs and patterns with $k = 3$. The instrumentation step is negligible in terms of runtime (for both Algorithms 1 and 2), thus, most of the runtime is spent during the evaluation of the instrumented program. The overhead of the algorithm w.r.t seminaive is higher than in the heuristic approach described above since in the heuristic approach, Algorithm 3 runs once and then the top-1 tree , while during Algorithm 4, Algorithm 3 runs 3 times: first, for the intersected program with the original pattern, P_1; second, for P_2 which is P_1 intersected with the negated derivation tree obtained in the first iteration; and third, for P_3 which is P_2 intersected with the negated derivation tree obtained in the second iteration. The patterns in the IRIS dataset had reasonable performance times of 102.6, 166.4, 136.6, 197.9 s, for patterns p_1, p_2, p_3, p_4, respectively. The results for the AMIE dataset is depicted in Fig. 13. For 1.2M output tuples, the execution times for p_5, p_6, p_7, p_8, p_9 were 521.6, 533.9, 526.9, 551.5, 861.4 s, respectively. Execution times for the more selective patterns p_6^*, p_7^*, p_8^*, p_9^* were unexpectedly faster taking 313, 303.2, 304.1, 391.2 s, respectively, for 1.2M output tuples. For the TC dataset and patterns p_{10}, p_{11}, p_{12}, p_{13} the results are 97.2, 209.8, 120.1, 198.4 s, respectively, for 1.7M output tuples. Finally, patterns p_{14}, p_{15}, p_{16} of the Explain dataset had the following runtimes: 530.7, 415.2, 398.5. Interestingly, the runtimes for p_{15}, p_{16} (three-node and two-node pattern, respectively) were faster than for the pattern p_{14} (a single node pattern).

Effect of optimizations Recall that the algorithm consists of two steps: program instrumentation and top-k evaluation. The instrumentation step is extremely fast (less than 1 s in all experiments), since it is independent of the database. A crucial factor affecting the performance of the top-k step is the complexity of the obtained instrumented program, which in turn is highly dependent on the size and complexity of the pattern and of the original program. As observed in the experiments, "simple" patterns (small, containing constants rather than wildcards) lead to smaller programs and good performance, while more complex patterns can lead to meeting the lower bound of Proposition 2, and consequently to a greater overhead (yet, unlike full provenance tracking, execution time was still feasible even for the complex programs and patterns we have considered). The optimizations outlined in Sect. 7 played a vital role in reducing the top-1 and top-k overheads. For instance, for the AMIE dataset, the computation time has improved by 22% to 50%. Specifically, for p_8, the computation time for 1.2M output tuples has improved by 50% and the time has decreased from 3.9 min to less than 2 min. For p_9, the improvement was more than 45% and the running time has decreased from 5.4 min to less than 3 min. As for TC dataset, the computation time for 1.7M output tuples decreased from 108 to 82 s (24% improvement) for p_{13} and from 48 to 33 s (30% improvement) for p_{11}. Overall, the optimizations outlined in Sect. 7 have indeed improved the algorithm's performance by as much as 50%, by reducing the number of rules, and restricting the generality of the programs.

Effect of patterns. The input cases which we have measured also evaluate the effect of "hardness" of the patterns on the computation. Specifically, the patterns chosen to measure the performance on the AMIE dataset and are shown in Fig. 9 can be divided into two sets. Patterns $p_5 - p_9$ are designed to show the system's performance in extreme situations. This was achieved by using very general (non-selective) patterns that in particular contain only wildcards. On the other hand, the pattern p_i^* has an identical tree structure as the pattern p_i for all $5 \leq i \leq 9$, respectively. However, p_i^* contains more constants than p_i and thus is more specific. The performance of the patterns of the form p_i^* is much better (unfortunately it is not possible to set a precise notion of selectiveness, since the pattern "selects" out of an infinite set of trees; and so we simply experiment with a large set of different patterns). These experiments are depicted in Fig. 12b. For example, the computation time for 1.2M output tuples for the partially instantiated patterns for the AMIE program is 40%–65% better.

9 Related work

Data provenance models Data provenance has been studied for different data transformation languages, from relational algebra to Nested Relational Calculus, with different provenance models (see, e.g., [6,7,11,21,25,29,37,52]) and applications [27,45,46,55,59], and with different means for efficient storage (e.g., [5,9,21,50]). In particular, semiring-based provenance for datalog has been studied in [30], and a

compact way to store it, for some class of semirings, was proposed in [19]. [14] introduces a theoretical model for justifications/explanations for datalog with negation. However, no notion of selective provenance was proposed in these works. As we have experimentally shown, tracking full datalog provenance fails to scale.

Selective provenance for non-recursive queries. There are multiple lines of work on querying data provenance, where the provenance is tracked for non-recursive queries (e.g., relational algebra or SQL). Here there are two approaches: one that tracks full provenance and then allows the user to query it (as in [34,36]), and one that allows on-demand generation of provenance based on user-specified criteria. A prominent line of work in the context of the latter is that of [26,28], where the system (called Perm) supports SQL language extensions to let the user specify what provenance to compute. Three distinct features in our settings are (1) our support of *recursion*, (2) the use of *tree patterns* to query derivations (which is natural for datalog), and (3) the support of ranking of results. These differences lead to novel challenges and consequently required novel modeling and solutions (as explained in the Introduction and in the description of the technical content).

Explanation for deductive systems. There is a wealth of work on explaining executions for deductive DBMSs [3,41,56], and some of them (e.g., [41]) compute explanations by augmenting the program with new rules. However, in contrast to our work, these works focus on tracking full provenance (either of the full program or of a given module as in [3]) and then analyzing it (e.g., using CORAL queries [3]) or visualizing through it using a dedicated interface. As we have shown, tracking full provenance is infeasible for large-scale data and complex programs. For instance, experiments in [3] are reported only for a relatively small-scale data (up to 30K rule instantiations). A feature that is present in [56] and absent here is the ability to query *missing* facts, i.e., explore why a fact was not generated. Incorporating such feature is an intriguing direction for future work. Finally, we note that [42] defines preference relations for datalog which allows to discard subsumed tuples on the fly, thus effecting the derivations. However, the efficient storage or presentation of provenance is not discussed.

Program slicing. In [10,51] the authors study the notion of program slicing for a highly expressive model of functional programs and for Nested Relational Calculus, where the idea is to trace only relevant parts of the execution. Our focus here is on supporting provenance for programs whose *output data* is large. Ranking and top-*k* queries are also absent from this line of work.

Workflow provenance. Different approaches for capturing workflow provenance appear in the literature (e.g., [2,13, 15,22,31,47,57]); however, there the focus is typically on the control flow and the dataflow between process modules, treating the modules themselves and their processing of the data as black boxes. A form of "instrumenting" executions in preparation for querying the provenance is proposed in [4], but again the data is abstracted away, the queries are limited to reachability queries and there is no ranking mechanism.

Context Free Grammars. Analysis of Context Free Grammars (CFGs) has been studied in different lines of work. For instance, in [40] the author proposes an algorithm for finding the top-1 weight of a derivation in a weighted CFG; in [12] the authors study the problem of querying parse trees of a given probabilistic context free grammar. There are technical similarities between datalog and CFGs; but a significant conceptual difference is that in datalog there is a separation between the program and the underlying data, which has no counterpart in CFGs. This means that no counterpart of our novel instrumentation algorithm appears in these works. Then, the top-*k* trees computation requires again a novel algorithm and subtle treatment of different cases.

Probabilistic XML. Multiple works have studied models and query languages for probabilistic XML (see, e.g., [38,39,43]), including top-*k* queries [8,43,48]. A technical similarity is in the use of tree patterns for querying a compactly represented set of trees, each associated with a weight (probability). However, our different motivation of querying datalog provenance is then reflected in many technical differences, including the separation between the program and the underlying data; the use of general weights rather than probabilities; their aggregation (summation over all possible worlds) in the context of probabilistic XML rather than the retrieval of individual top-*k* trees; and the resulting complexity.

Markov Logic Networks and other probabilistic models. The combination of highly expressive logical reasoning and probability has been studied in multiple lines of work. These include Markov Logic Networks [35,49,53] and probabilistic datalog (e.g., [18,23]). However, the focus in these lines of work is on *probabilistic inference*; to our knowledge, no counterparts of our query language or techniques were studied in these contexts.

10 Conclusion

We have presented `selPQL`, a top-*k* query language for datalog provenance, and an efficient algorithm for tracking selective provenance guided by a `selPQL` query. There are many intriguing directions for future work, including further optimizations, additional criteria for ranking trees, different notions of diversification, and the incorporation of user feedback.

Acknowledgements This research has been partially funded by the Israeli Science Foundation (978/17, 1636/13) and the Blavatnik Interdisciplinary Cyber Research Center (TAU ICRC). The contribution of Yuval Moskovitch is part of Ph.D. thesis research conducted at Tel Aviv University.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
2. Ailamaki, A., Ioannidis, Y.E., Livny, M.: Scientific workflow management by database management. In: SSDBM (1998)
3. Arora, T., Ramakrishnan, R., Roth, W.G., Seshadri, P., Srivastava, D.: Explaining program execution in deductive systems. In: DOOD (1993)
4. Bao, Z., Davidson, S.B., Milo, T.: Labeling recursive workflow executions on-the-fly. In: SIGMOD (2011)
5. Bao, Z., Köhler, H., Wang, L., Zhou, X., Sadiq, S.: Efficient provenance storage for relational queries. In: CIKM (2012)
6. Benjelloun, O., Sarma, A., Halevy, A., Theobald, M., Widom, J.: Databases with uncertainty and lineage. VLDB J. **17**, 243 (2008)
7. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. ACM Trans. Database Syst. **33**(4), 1 (2008)
8. Chang, L., Yu, J.X., Qin, L.: Query ranking in probabilistic XML data. In: EDBT (2009)
9. Chapman, A.P., Jagadish, H.V., Ramanan, P.: Efficient provenance storage. In: ACM SIGMOD, SIGMOD '08 (2008)
10. Cheney, J., Ahmed, A., Acar, U.A.: Database queries that explain their work. In: CoRR, abs/1408.1675 (2014)
11. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: why, how, and where. Found. Trends Databases **1**(4), 379 (2009)
12. Cohen, S., Kimelfeld, B.: Querying parse trees of stochastic context-free grammars. In: ICDT (2010)
13. Cohn, D., Hull, R.: Business artifacts: a data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32**(3), 3 (2009)
14. Damásio, C.V., Analyti, A., Antoniou, G.: Justifications for logic programming. In: Logic Programming and Nonmonotonic Reasoning (2013)
15. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD (2008)
16. Deutch, D., Gilad, A., Moskovitch, Y.: Selective provenance for datalog programs using top-k queries. PVLDB **8**(12), 1394 (2015)
17. Deutch, D., Gilad, A., Moskovitch, Y.: selp: selective tracking and presentation of data provenance (demo). In: ICDE (2015)
18. Deutch, D., Koch, C., Milo, T.: On probabilistic fixpoint and markov chain query languages. In: PODS (2010)
19. Deutch, D., Milo, T., Roy, S., Tannen, V.: Circuits for datalog provenance. In: ICDT (2014)
20. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. **28**(2), 652 (1998)
21. Fink, R., Han, L., Olteanu, D.: Aggregation in probabilistic databases via knowledge compilation. PVLDB **5**(5), 490 (2012)
22. Foster, I., Vockler, J., Wilde, M., Zhao, A.: Chimera: a virtual data system for representing, querying, and automating data derivation. In: SSDBM (2002)
23. Fuhr, N.: Probabilistic datalog:a logic for powerful retrieval methods. In: *SIGIR* (1995)
24. Galárraga, L.A., Teflioudi, C., Hose, K., Suchanek, F.M.: Amie: association rule mining under incomplete evidence in ontological knowledge bases. In: WWW (2013)
25. Geerts, F., Poggi, A.: On database query languages for k-relations. J. Appl. Logic **8**(2), 173–185 (2010)
26. Glavic B., Alonso, G.: Perm: processing provenance and data on the same data model through query rewriting. In: ICDE, pp. 174–185 (2009)
27. Glavic, B., Alonso, G., Miller, R.J., Haas, L.M.: TRAMP: understanding the behavior of schema mappings through provenance. PVLDB **3**(1), 1314–1325 (2010)
28. Glavic, B., Miller, R.J., Alonso, G.: Using sql for efficient generation and querying of provenance information. In: In Search of Elegance in the Theory and Practice of Computation. Springer (2013)
29. Glavic, B., Siddique, J., Andritsos, P., Miller, R.J.: Provenance for data mining. In: Tapp (2013)
30. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS (2007)
31. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic Acids Res. **34**, W729 (2006)
32. http://www.iris-reasoner.org
33. Imieliński, T., Lipski Jr., W.: Incomplete information in relational databases. J. ACM **31**(4), 761 (1984)
34. Ives, Z.G., Haeberlen, A., Feng, T., Gatterbauer, W.: Querying provenance for ranking and recommending. In: TaPP (2012)
35. Jha, A.K., Suciu, D.: Probabilistic databases with markoviews. PVLDB **5**(11), 1160 (2012)
36. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: SIGMOD (2010)
37. Kenig, B., Gal, A., Strichman, O.: A new class of lineage expressions over probabilistic databases computable in p-time. In: SUM, pp. 219–232 (2013)
38. Kimelfeld, B., Kosharovsky, Y., Sagiv, Y.: Query evaluation over probabilistic XML. VLDB J. **18**(5), 1117 (2009)
39. Kimelfeld, B., Sagiv, Y.: Matching twigs in probabilistic XML. In: VLDB (2007)
40. Knuth, D.E.: A generalization of Dijkstra's algorithm. Inf. Process. Lett. **6**(1), 1 (1977)
41. Köhler, S., Ludäscher, B., Smaragdakis, Y.: Declarative datalog debugging for mere mortals. In: Datalog in Academia and Industry (2012)
42. Köstler, G., Kießling, W., Thöne, H., Güntzer, U.: Fixpoint iteration with subsumption in deductive databases. J. Intell. Inf. Syst. **4**(2), 123 (1995)
43. Li, J., Liu, C., Zhou, R., Wang, W.: Top-k keyword search over probabilistic XML data. In: ICDE (2011)
44. Loo, B.T. et al.: Declarative networking: language, execution and optimization. In: SIGMOD (2006)
45. Meliou, A., Gatterbauer, W., Suciu, D.: Reverse data management. PVLDB **4**(12), 1490 (2011)
46. Meliou, A., Suciu, D.: Tiresias: the database oracle for how-to queries. In: SIGMOD (2012)
47. Missier, P., Paton, N., Belhajjame, K.: Fine-grained and efficient lineage querying of collection-based workflow provenance. In: EDBT (2010)
48. Ning, B., Liu, C., Yu, J.X.: Efficient processing of top-k twig queries over probabilistic XML data. World Wide Web **16**(3), 299 (2013)
49. Niu, F., Zhang, C., Re, C., Shavlik, J.W.: Deepdive: Web-scale knowledge-base construction using statistical learning and inference. In: VLDS, pp. 25–28 (2012)
50. Olteanu, D., Zavodny, J.: Factorised representations of query results: size bounds and readability. In: ICDT (2012)
51. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional programs that explain their work. In: SIGPLAN (2012)
52. Prov-overview, w3c working group note. http://www.w3.org/TR/prov-overview/ 2013
53. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1–2), 107 (2006)

54. Ronen, R., Shmueli, O.: Automated interaction in social networks with datalog. In: CIKM (2010)
55. Roy, S., Suciu, D.: A formal approach to finding explanations for database queries. In: SIGMOD (2014)
56. Shmueli, O., Tsur, S.: Logical diagnosis of LDL programs. New Gener. Comput. **9**(3/4), 277 (1991)
57. Simhan, Y.L., Plale, B., Gammon, D.: Karma2: provenance management for data-driven workflows. Int. J. Web Serv. Res. **5**(2), 317 (2008)
58. Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: a core of semantic knowledge. In: WWW (2007)
59. Suciu, D., Olteanu, D., Ré, C., Koch, C.: *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2011)