Selective Provenance for Datalog Programs Using Top-K Queries

Daniel Deutch  
Tel Aviv University  
danielde@post.tau.ac.il

Amir Gilad  
Tel Aviv University  
amirgilad@post.tau.ac.il

Yuval Moskovitch  
Tel Aviv University  
moskovitch1@post.tau.ac.il

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, and here we focus on a detailed form of provenance (how-provenance), defining it 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 based on (1) instrumenting the datalog program so that, upon evaluation, it generates only relevant provenance, and (2) efficient top-k (relevant) provenance generation, combined with bottom-up datalog evaluation. The algorithm 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.

1. INTRODUCTION

Many real-life applications rely on an underlying database in their operation. In different domains, such as declarative networking [40], social networks [49], and information extraction [23], it has recently been proposed to use datalog for the modeling of such applications.

Consider, for example, AMIE [23], 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 [53]) 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.

Datalog programs capturing the logic of real-life applications are typically quite complex, with many, possibly recursive, rules and an underlying large-scale database. 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. For instance, AMIE rules are highly complex and include many instances of recursion and mutual recursion. 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. When viewing a derived fact, it is thus essential to also view an explanation of the process of its derivation.

A conceptual question in this respect is what constitutes a “good” explanation. An approach advocated by previous work is to define provenance by looking at derivations of facts, and distinguishing between alternative and joint use of facts in such derivations. In the context of datalog programs, a notion of explanations that follows this approach is based on derivation trees [29]. 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. Derivation trees are particularly appealing as explanations, since not only they 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 fact provenance.

Defining provenance as the set of 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, and may be prohibitively large even in absence of recursion. 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 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. A simple use of the patterns is to limit provenance tracking to particular facts of interest; but the language is rich enough to also allow specifying complex features of derivations that are of interest. For instance, in the AMIE example, by viewing derivations that involve integration of data from different sources (e.g. ontologies), one may gain insight into the usefulness of the integration or reliability of obtained facts. From a different perspective, if one ontology is less trustworthy than the other, the application owner may wish to see explanations based only on the more reliable source; etc.

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 weight of a tree (our solution supports a rich class of aggregation functions).

Novel algorithm 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 (or alternatively grounding of the datalog program with respect to the database), and then querying the provenance, is a solution that fails to scale. On the other hand, discarding partial derivations “on-the-fly” is also challenging, since their inclusion in the answer set may depend on consequent derivation steps (as well as on other derivations which may or may not be ranked higher). Our solution then consists of two main steps:

1. Static (i.e. independent of the underlying database) “instrumentation” of the datalog program \( P \) with respect to the seLPQL query (in fact, its tree pattern component \( p \)). We introduce a precise definition of the output of this instrumentation (see proposition 4.2), which is a new datalog program \( P_p \) that “guide” provenance tracking based on \( p \). Intuitively, \( P_p \) satisfies that for every database \( D \), the derivation trees induced by \( P \) and \( D \) are also induced (up to renaming of relations) by \( P_p \) and \( D \), and crucially the trees that follow the pattern \( p \) are exactly those that involve particularly marked relations. The fact that a program satisfying this property (for every database) can be effectively computed is non-trivial, and a major challenge here is that \( P \) may involve recursion. Our novel solution is based on encoding, using datalog rules, a “require/guarantee” relation for satisfaction of parts of the tree pattern. 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.

2. Bottom-up evaluation of \( P_p \) w.r.t. an underlying database \( D \) while generating a compact representation of the top-k relevant trees (of \( P \)) for each (relevant) output tuple. Here again, our solution consists of two steps. The first involves computing the top-1 tree side-by-side with bottom-up datalog evaluation. The basic idea here is somewhat inspired by prior work on computing the best derivation weight for Context Free Grammars (see section 7), but requires significant efforts to (1) account for datalog and (2) generate a compact representation of the tree itself (whose size may be prohibitively large) rather than just its weight. We further design a novel algorithm for computing the top-k derivation trees, by exploring modifications of the top-1 tree. Challenges in the design of this algorithm include, among others, (1) the avoidance of generating multiple trees that are the same up to renaming (i.e. correspond to a single tree of \( P \)); and (2) avoiding costly materialization of trees.

The final step of the algorithm is then the materialization of only the top-k trees based on the compact representation.

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.

2. PRELIMINARIES

We provide necessary preliminaries on datalog and the provenance of output data computed by datalog programs.

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. A datalog program is a finite set of datalog rules. A datalog rule is an expression of the form \( R_1(u_1) : -R_2(u_2)...R_n(u_n) \) where \( R_i \)'s are relation names, and \( u_1,...,u_n \) are sets of variables with appropriate arities. \( R_1(u_1) \) is called the rule’s head, and \( R_2(u_2)...R_n(u_n) \) is called the rule’s body. Every variable occurring in \( u_1 \) must occur in at least one of \( u_2,...,u_n \). We make the distinction between extensional (ebd) and intensional (idb) facts and relations. A datalog program is then a mapping from ebd 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 \( D \cup \{A\} \) 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 2.1.** AMIE [23] is a system for the automatic inference of rules, by identifying “patterns” in a Knowledge Base (KB). Rules of AMIE form a datalog program and are then evaluated with respect to a database instance (which is a KB) to compute an idb instance (which is an extended KB). We consider the program inferred by AMIE based on patterns in YAGO. Among many others, the idb instance includes a binary relation dealsWith, including information on international trade relations (an edb “copy” of this relation appears as well, with rules for copying its content that are omitted for brevity). The program includes the following rule, intuitively specifying that dealsWith is a symmetric relation (ignore for now the numbers in parentheses).

$$r_1(0.8) \text{dealsWith}(a, b) \iff \text{dealsWith}(b, a)$$

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

$$r_2(0.5) \text{dealsWith}(a, b) \iff \text{imports}(a, c), \text{exports}(b, c)$$

$$r_3(0.7) \text{dealsWith}(a, b) \iff \text{dealsWith}(a, f), \text{dealsWith}(f, b)$$

The rules $r_1, r_2, r_3$ form a datalog program (which is a strict subset of the actual program obtained by AMIE).

### 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). Given a datalog program $P$ and a database $D$, we denote by $\text{trees}(P, D, t)$ the set of all possible derivation trees for $t \in P(D)$, and define $\text{trees}(P, D) = \bigcup_{t \in P(D)} \text{trees}(P, D, t)$.

**Example 2.2.** Three derivation trees for the fact $t = \text{dealsWith}(\text{Cuba}, \text{France})$, based on the program given in Example 2.1 and the example database given in Table 1, are presented in Figure 1. 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).

### 3. QUERYING DATALOG PROVENANCE

We introduce a query language for derivation trees, based on two facets: (1) boolean criteria describing derivations of interest, and (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, used to query derivations.

**Definition 3.1.** 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 $\neg, \lor$ and $\land$ can also be applied to tree patterns (with the expected semantics).

**Example 3.2.** Several tree patterns are presented in Figure 2. The pattern $p_1$ specifies interest in all derivations of facts of the form $\text{dealsWith}(\text{Cuba}, *)$ (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 *\_DBPedia() to match all relations from YAGO and DBPedia resp.; then $p_3$ selects derivations of facts $\text{dealsWith}(\text{Cuba}, *)$ that are based on integrated data

---

This requires a slight change of the definition of patterns, which is easy to support, to allow * in relation names.
from both sources, and \( p_4 \) selects derivations that use facts from YAGO but no fact from DBPedia. Both patterns shed light on the usefulness of integration in this context.

We next define the semantics of derivation tree patterns, i.e. the notion of matching a given derivation tree.

**Definition 3.3.** Given a derivation tree \( \tau \) and a derivation tree pattern \( p \), a match of \( p \) in \( \tau \) is a mapping \( h \) from the nodes of \( p \) to nodes of \( \tau \), and from the regular (transitive) edges of \( p \) to edges (resp. paths) of \( \tau \) such that (1) the root of \( p \) is mapped to the root of \( \tau \), (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 \( \tau \). (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 \( \tau \) from \( h(m) \) to \( h(n) \) and (6) for any two edges \( e_1 \) and \( e_2 \) in \( p \), their corresponding edge/path in \( \tau \) are disjoint.

**Definition 3.4.** 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 \( p(\text{trees}(P,D)) \).

**Example 3.5.** Consider the datalog program \( P \) given in Example 2.1, the database instance given in Table 1 and the tree pattern \( p_2 \) in Figure 2b. The set \( p_2(P,D) \) includes infinitely many derivation trees, including in particular \( \tau_2 \) and \( \tau_3 \) shown in Figure 1.

### 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 \((M,+,0,\leq)\) 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 3.6.** A weight-aware datalog instance is a triple \((P,D,w)\) where \( w \) maps rules in \( P \) and tuples in \( D \) to elements of an ordered monoid \((M,+,0,\leq)\). The monoid operation is referred to as the aggregation function.

**Example 3.7.** We demonstrate multiple choices of monoid and the corresponding applications.

**Derivation size** To rank derivation trees by their size we may use the monoid \((\mathbb{Z},+,0,\leq)\), 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 \([0,1],[0,0,\cdot]\). 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 \([0,1],[1,1,\leq]\) monoid.

**Access control** Consider the case where each fact/rule is associated with a different access control credential, e.g. one of \( A = \{ \text{Top secret (T)}, \text{Secret (S)}, \text{Confidential (C)}, \text{Unclassified (U)} \} \). 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.

**Note.** There is an intuitive correspondence between our weighted model and that of semiring provenance [29]. In the model of [29], two abstract operations are used: multiplication (in a semiring) is used to combine provenance of facts participating in a single derivation, and semiring addition is used to combine provenance of multiple derivations of the same fact. Our aggregation function \(+\) of the monoid) thus intuitively has the same role as semiring multiplication, while our counterpart of the semiring addition is fixed to be the top-k operation (which allows us to support a concise presentation of results). Indeed, our above examples of aggregation functions were all shown to be useful as provenance (but the problem of computing top-k derivation trees based on such function was not previously considered). We
further note that one could consider a probabilistic interpretation of weights (this requires some care to guarantee that the semantics indeed induces a probability distribution), but (as also noted in the context of semiring provenance), a probabilistic semantics would not allow to capture ranking by arbitrary monoids such as e.g. the “derivation minimal confidence” or “access control” examples above.

In the following sections we propose a two-step algorithm for solving TOP-K, as explained in the Introduction and depicted in Figure 3. The algorithm will serve as proof for the following theorem.

**Theorem 3.12.** For any Program $P$, pattern $p$ and database $D$, we can compute the top-k derivation trees for each fact 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), double exponential in the pattern width (which is typically even smaller), and linear in the output size.

**4. PROGRAM INSTRUMENTATION**

We now present the first step of the algorithm for solving TOP-K, which is instrumenting the program with respect to the 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 $R^v_1$. Intuitively, derivations for facts in $R^v$ must match the sub-pattern rooted by $v$; derivations for $R^v_1$ must include a sub-tree that matches the sub-pattern 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).

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

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

```
input : Weighted Program $P$ and a pattern $p$
output : “Instrumented” Program $P'$_

1 foreach pattern node $v \in p$ do
2    let $v_0, \ldots, v_n$ be the immediate children of $v$;
3    foreach rule $[R(x_0, ..., x_m) : -\beta] \in P$ 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 $[R'(y_0, ..., y_m) : -A(\beta)]$ to $P'$;
8            else
9                foreach $\beta' \in ex(A(\beta), \{v_0, ..., v_n\})$ do
10                   Add $[R'(y_0, ..., y_m) : -\beta']$ to $P'$;
11        if $v$ is a transitive child then
12            foreach $\beta' \in tr-ex(\beta, v)$ do
13                Add $[R'(x_0, ..., x_m) : -\beta']$ to $P'$;
14            foreach rule $[R'(y_0, ..., y_m) : -\beta]$ for transitive $v$ do
15                Add $[R'(y_0, ..., y_m) : -\beta]$ to $P'$;
16            HandleDDB();
17    Clean failed rules in $P'$;
18 return the union of rules in $P$ and $P'$;
```

Algorithm 1 then generates a new program, instrumented by the 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 sub-trees rooted in the children of $v$. To this end we define the set of “expansions” $ex(\text{atoms}, \{v_0, ..., v_n\})$ as follows. Consider all one-to-one (but not necessarily onto) functions $f$ that map the set $\{v_0, ..., v_n\}$ to the set $\text{atoms} = \{a_0, ..., a_l\}$. Each such function defines a new set of atoms obtained from $\text{atoms}$ by replacing atom $a_i = R(x_0, ..., x_m)$ by $R^v(x_0, ..., x_m)$ if $f(v_j) = a_i$ and $v_j$ is not a transitive child, or by $R^v_1(x_0, ..., x_m)$ if $v_j$ is a transitive child (atoms to which no node is mapped remain intact). We then define $ex(\text{atoms}, \{v_0, ..., v_n\})$ as the set of all atoms sets obtained for some choice of function $f$. In line 10 the algorithm generates a rule for each set in 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 sub-pattern (see Example 4.1).

The algorithm thus far deals with satisfaction of the sub-tree rooted at $v$, by designing rules that propagate the satis-
\[ \begin{align*}
[\rho_1^v] & \text{dealsWith}^{v_0}(\text{Cuba, b}) :- \text{dealsWith}^{v_1}(\text{b, Cuba}) \\
[\rho_2^v] & \text{exports}^{v_2}(\text{Cuba, tobacco}) :- \text{exports}(\text{Cuba, tobacco})
\end{align*} \]

Figure 4: Two rules of the instrumented program

fication of the sub-trees 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 \( R^v(...) \) we will have a copy of the same rule for \( R^0(...) \), lines 14-15), or through (indirect) satisfaction by an atom in the body. For the latter, we define \( tr = \text{ex}(\text{atoms}, v) \) as the set of all atoms sets obtained from \( \text{atoms} \) by replacing a single atom \( R(x_0, ..., x_m) \) in \( \text{atoms} \) by \( R^v(x_0, ..., x_m) \) (and keeping the other atoms intact), and add the corresponding rules (line 13). Then the function \( \text{HandleEDB} \) adds rules for nodes that locally match edb facts, copying matching facts into the new relations \( T^0(...) \) and \( T^v(...) \). The final step of the algorithm is "cleanup" (line 17), removing unreachable rules.

Example 4.1. Consider the program \( P \) given in Examples 2.1, and the tree pattern shown in Figure 2b, where \( v_0 \) is the root node in \( p_2 \) and \( v_1 \) is the leaf. Two out of the rules of the output program are shown in Figure 4, and we next illustrate their generation process. Since all rules in \( P \) locally match \( v_0 \) through the assignment \( A = \{ a \leftarrow \text{Cuba}, b \leftarrow \ast \} \), \( v_0 \) is not a leaf and \( \{\text{dealsWith}^{v_0}(\text{b, Cuba})\} \) is the only \( \beta' \) obtained for rule \( r_1 \) and \( \text{ex}(A, \text{dealsWith}(\text{b, a}), v_1) \), we have that in line 10 the algorithm adds the rule \( r_1' \). Intuitively, derivations for facts in \( \text{dealsWith}^{v_0}(...) \) must match the sub-pattern rooted by \( v_0 \). Then derivations for facts in \( \text{dealsWith}^{v_1}(...) \) must include a sub-tree that matches the sub-pattern rooted by \( v_1 \), and generated rules for \( \{\text{dealsWith}^{v_0}(b, \text{Cuba})\} \) enforce that (since a \( \text{dealsWith} \) atom cannot satisfy \( v_1 \)) one of the atoms in the body of a used rule will be derived in a way eventually satisfying \( v_1 \). Rule \( r_2' \) is added by \( \text{HandleEDB} \) since \( \text{exports}(a, b) \) locally matches \( v_1 \).

The instrumented program satisfies the following fundamental property. Given an atom \( R(\ldots) \), \( R^v(\ldots) \) or \( R^0(\ldots) \) we define its \( \text{origin} \) to be \( R(\ldots) \), i.e. the atom obtained by deleting the annotation \( v \) or \( v' \) (if exists). For a derivation tree \( \tau \) we define \( \text{origin}(\tau) \) as the tree obtained from \( \tau \) by replacing each atom by its origin and pruning branches added due to the function \( \text{HandleEDB} \) ("copying" edb facts).

We now have:

**Proposition 4.2.** Let \( P_0 \) 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:

\[ \text{trees}(P, D) = \bigcup_{\tau \in \text{trees}(P_0, D)} \text{origin}(\tau) \]  

(1)

\[ p(P, D) = \bigcup_{\tau = R^0(\ldots)} \bigcup_{\tau \in \text{trees}(P_0, D, \tau)} \text{origin}(\tau) \]  

(2)

\[ w(\text{origin}(\tau)) = w(\tau) \quad \forall \tau \in \text{trees}(P, D) \]  

(3)

We refer to \( v \) and \( v' \) in \( R^v(\ldots) \) and \( R^v(\ldots) \) as annotations. Intuitively, the first part of the proposition means that for every database, \( P_0 \) 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.

**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 \( v \in p \) it 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 (which is small in practical cases) is due to the need to consider all "expansions". Furthermore, we next show that a polynomial dependency on the program and pattern is impossible to achieve (proof deferred to the online version [22]).

**Proposition 4.3 (Lower Bound).** There is a class of patterns \( \{p_1, \ldots\} \) and a class of programs \( \{P_1, \ldots\} \), such that \( w(p_0) = O(n) \), \( |P_0| = O(n) \) and there is no program \( P' \) of size polynomial in \( n \) that satisfies the three conditions of Proposition 4.2 with respect to \( P_0, p_0 \).

**Optimizations.** The bound we have established on the instrumented program size is only a worst case one, realized when e.g. all pattern nodes match all program rules. To improve performance in practice, we employ further optimizations that simplify the instrumented program (thus reducing the time of the top-k computation that follows). A particular such optimization relates to the existence of constants in facts that label the pattern nodes. Recall that the algorithm assigns corresponding constants in generated rules. When these constants appear in labels of non-transitive nodes (i.e. nodes not connected to their parents through transitive edges), our optimization then "propagates" the assignment of constants in a bottom-up manner, thus generating rules that are more specific to assignments that will eventually lead to valid derivations. Additionally, for non-transitive leaves labeled by edb facts, the algorithm leads to generation of rules that simply copy the content of an edb relation to an idb relation; our optimization avoids this redundant step.

**4.2 Boolean combinations of patterns**

Algorithm 1 allows intersection of a single pattern with a program. We next explain how to use (modifications of) the algorithm to account for 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 size of the pattern (the exponent is multiplication of the individual size of patterns, in the case of conjunction).

**Negation.** The algorithm for intersecting a negation of a pattern is similar to Algorithm 1 with some modifications, as follows. We use relation names \( R^v \) and \( R^{-v} \) for every relation name \( R \) in the program and for every pattern node \( v \). Derivations for \( R^v \) should not match the sub-pat- roled by \( v \) and derivations for \( R^{-v} \) should not
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.

### Algorithm 2: Top-k

```plaintext
input : Weighted Datalog Program $P$, Database $D$
output : Top-$k$ tree for facts of the form $R^{v_0}(\ldots)$
1 Init $DTable$ with $(t, 0, \text{null})$ 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 $[\arg \max_{Cand} \text{Top}(t, DTable, P)]$ to $DTable$;
5 return the entries of all $e \in DTable$ s.t. the fact $t$ of $e$ is of the form $R^{v_0}(\ldots)$;
```

#### Example 5.1
Consider the two rules given in Example 4.1, and the database $D$ shown in Table 1. Algorithm 2 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 $t_3 = \text{exports}^{s_1}(\text{Cuba}, \text{tobacco})$ can be derived with weight 1 using the edb fact $t_1 = \text{exports(Cuba,tobacco)}$ and the rule denoted $r_2^*$ in Example 4.1. Other facts can be derived in the first iteration but $t_3$ is the fact with maximal weight. The algorithm thus adds $(t_3, 1, \{s_1\})$ to $DTable$, where $s_1$ is a pointer to the entry of $t_1$ in $DTable$. In the next iteration, the algorithm can derive the fact $t_4 = \text{dealsWith}^{s_1}(\text{France}, \text{Cuba})$ using $t_3$ and the edb fact $t_2 = \text{imports}(\text{France}, \text{tobacco})$ 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 $(t_4, 0.5, \{s_2, s_3\})$ to $DTable$. After $t_4$ is added to $DTable$, the fact $t_5 = \text{dealsWith}^{s_0}(\text{Cuba}, \text{France})$ can be derived with overall weight of 0.5 · 0.8 = 0.4, and $(t_5, 0.4, \{s_4\})$ is added to $DTable$.

#### 5.2 Top-k

The algorithm for Top-k computes the top-$i$ derivations for each fact $t \in P^+_i(D)$ in a bottom-up manner for $2 \leq i \leq k$. For each $i$ it essentially repeats the procedure of Algorithm 2, but starting with $DTable$ consisting of the top-$(i-1)$ trees, i.e. $r_j^*$ for all $t \in P^+_i(D)$ and $j < i$. A subtlety is that different trees in $P^+_i(D)$ 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 $\tau_i$ for a fact $t$ is a top-$i$ candidate, if one of the following holds: (i) $\tau_i$ uses at least one “new” fact that was added in the $i$'th iteration or (ii) the last derivation step in $\tau_i$ is different from the last derivation step in $\tau_j^*$ for all $1 \leq j < i$, such that $\text{origin}(\tau_i) \neq \text{origin}(\tau_j^*)$. Given the top-$(i-1)$ derivation trees, to compute the $i$'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 $\tau_i$ with maximal weight (out of candidates computed for all facts) and add it to a new entry $t^*$ 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

---

**Disjunction and Conjunction.** Disjunction of patterns may be performed by repeatedly intersecting the original program with each of the disjuncts (in arbitrary order), accumulating the obtained rules into a single program. As for conjunction, we again perform repeated intersection with the conjuncts, but this time use the output of each intersection step as the input for the next step.

### 5. FINDING TOP-K DERIVATION TREES

The second step of the algorithm is finding top-$k$ derivation trees that conform to the 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 $k > 1$.

#### 5.1 Top-1

Algorithm 2 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).
in time that is polynomial in $DTable$ compactly represented trees do not have the same origin: each $i$ 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 $t \in P\phi(D)$ where $P\phi$ is the instrumented program given the program $P$ and pattern $\phi$. The number of facts $t \in P\phi(D)$ is at most $|D||P\phi| = |D||(P|^{w(p)})$, 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.

5.3 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 [18]. 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 $\tau$ for a fact $t \in P\phi(D)$, corresponds to an instantiation of a derivation rule $r$ in $P\phi$. Given a node $u \in \tau$, a modification of $u$ in $\tau$ is using a different instantiation to derive $u$, i.e. using different derivation rule $r' \in P\phi$ or a different assignment to the variables in $r$ s.t. for the obtained tree $\tau'$ it holds that $\text{origin}(\tau') \neq \text{origin}(\tau)$. We say that two modifications are different if for their results $\tau_1$ and $\tau_2$ satisfy $\text{origin}(\tau_1) \neq \text{origin}(\tau_2)$.

Given a derivation tree $\tau$, we denote by $\tau_{u,r,\sigma}$ the derivation tree obtained by modifying $u$ in $\tau$ using $r$ and $\sigma$. 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 $i$th best derivation tree can be obtained by a modification of any 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 $\tau$ 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 $\tau$, however it appears only once in $DTable$. Thus, modifying the entry of $t$ in $DTable$ would result in modifying the sub-trees 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 $\tau$ to $t$ (including $t$) for each modification of $t$'s sub-tree.

6. IMPLEMENTATION AND EXPERIMENTS

We have implemented our algorithms in a system prototype called sel1P (for “selective provenance”, demonstrated in [15]). The system is implemented in JAVA and its architecture is depicted in Figure 3: the user feeds the system with a datalog program and a sel1PQL query, and the instrumented program is computed and fed, along with an input database, to the TOP-K component. This component is implemented by modifying and extending IRIS [31], a JAVA-based system for in-memory datalog evaluation. Users may then choose a tuple of interest from the output DB and view a visualization of the top-$k$ qualifying explanations (according to the pattern) for the chosen tuple.

We have conducted experiments to examine the scalability and usefulness of the approach, in various settings. We next describe the dedicated benchmark (including both synthetic and real data) developed for the experiments, and then the experimental results.

6.1 Evaluation Benchmark

We have used the following datasets, each with multiple sel1PQL queries, and for increasingly large output databases. The weights in the reported results are all elements of the monoid $\langle 0, 1, \cdot, 1, < \rangle$; we have experimented with all other monoids given in Example 3.7, 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 [31]. The program consists of 8 rules and generates up to 4.26M tuples; weights have been randomly assigned in the range $[0, 1]$.

2. AMIE We have used a recursive datalog program consisting of rules mined by AMIE [23], automatically translated into datalog syntax, with weights assigned by AMIE and reflecting rule confidence. The underlying input database is that of YAGO [53]. The program consists of 23 rules (many of which involve recursion and mutual recursion) for Information Extraction that generate up to 1.2M tuples.

3. Explain We have used a variant of the recursive datalog program described in [3], as a use-case for the “explain” system, see discussion of related work (arithmetic operations were treated through dedicated relations, and aggregation was omitted). 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.
Baseline algorithms. 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 tested two “extreme” choices: (1) standard, semi-naive evaluation with no provenance tracking, using IRIS implementation; and (2) compact representation of full provenance, based on the notion of equations systems from [29], where for each idb fact there is an equation representing its dependency on other idb facts and on edb facts, with additional optimizations that allow for “sharing” of identical parts between different equations.

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

6.2 Experimental Results

Figure 5 presents the execution time of standard semi-naive evaluation and of selective provenance tracking for the four datasets and for different seqPQL 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 Figure 5a, the results for the IRIS dataset are presented for 4 different patterns: \( p_1 \) 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 semi-naive evaluation was very reasonable: 38% overhead w.r.t. the evaluation time of semi-naive even for the complex six-node pattern and only 3% - 21% for the other patterns. The absolute execution time is also reasonable: 56-65 seconds for the different patterns and for output database of over 4.2M tuples (note that for this output size, the execution time of standard semi-naive evaluation is already 53 seconds In contrast, generation of full provenance was infeasible (in terms of memory consumption) beyond output database of 1.6M tuples, taking over 5 minutes 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 semi-naive 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. 5 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 Figure 5b. 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 seconds (1.3 times slower than semi-naive) for \( p_i^* \). For \( p_i^* \) and the same number of output tuples it took 62 seconds (less than 2 times slower than semi-naive), 44.6 seconds (1.3 times slower than semi-naive) for \( p_i^* \) and 105 seconds (3.2 times slower than semi-naive) for \( p_i^* \). 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. semi-naive in average) for \( p_1 \), less than 2 minutes (3.5 times slower than semi-naive) for \( p_2 \), 2.6 minutes (4.8 times slower) for \( p_3 \), and less than 2 and 2.9 minutes (3.6 and 5.4 times slower) for \( p_4 \) and \( p_5 \) respectively.

In Figure 5c 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. We observe a non-negligible but reasonable overhead with respect to semi-naive 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 seconds (and 56% overhead with respect to semi-naive in average), 33 seconds for \( p_{11} \) (1.8 times slower than semi-naive in average), 74 seconds for \( p_{12} \) (4 times slower) and 82 seconds for \( p_{13} \) (4.5 times slower than semi-naive). Here full provenance tracking was extremely costly, requiring over 6.5 hours for output database size of 1.7M tuples.

Figure 5d 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. The computation time for 1.16M output tuples was less than 3.2 minutes (7% overhead w.r.t semi-naive) for \( p_{14} \), 3.3 minutes (10% overhead w.r.t semi-naive) for \( p_{15} \) and 4.4 minutes (85% overhead w.r.t the evaluation time of semi-naive) for \( p_{16} \). Full provenance tracking has required over 2 hours 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 Figure 6 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).

Discussion. Recall that the algorithm consists of two steps: program instrumentation and top-K evaluation. The instrumentation step is extremely fast (less than 1 second 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 con-
stants rather than wildcards) lead to smaller programs and
good performance, while more complex patterns can lead
to meeting the lower bound of Prop. 4.2, and consequently
to a greater overhead (yet, unlike full provenance tracking,
execution time was still feasible even for the complex pro-
grams and patterns we have considered). We note that the
optimizations outlined in section 4 have indeed improved the
algorithm’s performance by as much as 50%, due to reducing
the number of rules.

7. RELATED WORK

We next overview multiple lines of related work.

Data provenance models. Data provenance has been studied
for different data transformation languages, from rela-
tional algebra to Nested Relational Calculus, with different
provenance models (see e.g. [6, 29, 28, 24, 35, 11, 55, 7, 19]) and applications [54, 42, 50, 41, 26], and with different
means for efficient storage (e.g. [5, 9, 46, 19]). In particular,
semiring-based provenance for datalog has been studied in
[29], and a compact way to store it, for some class of semir-
ings, was proposed in [17]. However, no notion of selective
provenance was proposed in this work. Consequently; (1)
the resulting structure is very complex and difficult to un-
derstand (it is not geared towards presentation, thus there
is no support of ranking or selection criteria), and (2) as
we have experimentally showed, tracking full datalog prove-
nance 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 ap-
proaches: one that tracks full provenance and then allows
the user to query it (as in [34, 32]), 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 [27, 25], 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) the presence of recursion (we support recur-
sive datalog rather than SQL), (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. For
instance, in [3] the authors present an explanation facility
called “explain” for CORAL, a deductive database system.
It allows users to specify subsets of rules as different “mod-
ules”, and then to set provenance tracking “on” or “off” for
each module. For the chosen modules, the system main-
ains a record of all instantiations of rules that have been
generated during the program execution. This is a coun-
terpart of our notion of full provenance, since all derivation
trees may be obtained from this structure. Once full prove-
nance is tracked, one may analyze it (e.g. using further
CORAL queries), or browse through it using a dedicated
Graphical User Interface. In contrast to our work, this line
of work focuses on analyzing the full provenance, and cannot
be used to specify in advance the structure of derivations
that should be tracked (in-advance specification is limited to
the coarse-grain specification of modules). As we have
shown, tracking full provenance is infeasible for large-scale
data and complex programs. Indeed, experiments in [3] are
reported only for a relatively small scale data (up to 30K
rule instantiations, which implies less than 30K tuples in
the output database). Consequently, we focus on static
instrumentation that allows to avoid full provenance tracking.
This then leads to the need for a careful design of a declarative
language (and corresponding algorithms) for specifying
selective provenance tracking, such that the language is rich
eough to express properties and ranking functions of inter-
est, while allowing for feasibility of instrumentation (which
was not addressed in [3]). Indeed, selPQL allows the spec-
fication of expressive queries through the combination of
tree patterns and ranking, while still allowing for efficient
instrumentation. These major distinctions in the problem
setting also naturally imply that our technical development
is novel. The same distinctions apply to other works in this
context, such as the debugging system for the LDL deductive database presented in [51]. A feature that is present in [51] and absent here is the ability to query missing facts, i.e., explore why a fact was not generated. Incorporating such feature (e.g., to find ranked explanations for absence of facts) is an intriguing direction for future work.

Program slicing. In [10, 47] 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. While the high-level idea is similar to ours, and the transformation languages they account for are more expressive, our focus here is on supporting provenance for programs whose output data is large (in contrast, the output size for the programs in the experiments of [10, 47] is much smaller than in our experiments). We have thus chosen datalog as a formalism, leading to our tree-based language for patterns, to our theoretical complexity guarantees (which naturally could not be obtained for arbitrary functional programs), and to our experimental study supporting large-scale output data. Importantly, our ranking mechanism and top-k computation are also absent from this line of work.

Workflow provenance. Different approaches for capturing workflow provenance appear in the literature (e.g., [14, 13, 2, 30, 20, 52, 43]), 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 the different parses of Context Free Grammars (CFGs) has been studied in different lines of work. In [38] the author proposes an algorithm for finding the top-1 weight of a derivation in a weighted CFG; other works have studied the problem of finding top-k parses of a given string (where, unlike in our case, the derivation size is bounded) in a probabilistic context free grammar. In [12] the authors study the problem of querying the space of parse trees of strings for a given probabilistic context free grammar, using an expressive query language, but focus on computing probabilities of results, where the probability is obtained by summation over all possible parse trees satisfying a pattern.

There are technical similarities between datalog and CFGs; but perhaps the most significant conceptual difference is that in datalog there is a separation between the program and the underlying data, which has no counterpart in CFGs. In particular, we have shown that it is essential for the algorithm performance that we avoid grounding the program (which is the equivalent of full provenance generation) and instead instrument it without referring to a particular database instance. These are considerations that are of course absent when working with CFGs. This means that no counterpart of our novel instrumentation algorithm (or of the key Proposition 4.2) appears in these works. Then, the top-k trees computation requires again a novel algorithm and subtle treatment of different cases. We have highlighted some of the novel challenges in this respect in the Introduction.

Probabilistic XML. Different works have studied models and algorithms for representing and querying probabilistic distributions over XML documents (see e.g. [36, 37, 39]). Top-k queries over probabilistic XML was studied in e.g. [44, 8, 39]. A technical similarity is in the use of tree patterns for querying a compactly represented set of trees, each associated with a probability value (the counterpart of weights in our model). However our different motivation of querying datalog provenance is then reflected in many technical differences. First, the separation between the program and the underlying data and the need for instrumentation that is independent of the data (as explained in the Introduction and in the discussion of CFGs above) is also absent from models for probabilistic XML, and leads to novel challenges and a novel instrumentation algorithm, which also significantly affects the further development for top-k computation (due to the need to avoid generation of trees that are duplicates up to “instrumentation annotations”). An additional difference is due to our use of a general weight function rather than probabilities. We further note that beyond the difference in the model, the problem typically considered for probabilistic XML is different than ours. The problem typically studied in these works is that of finding the probability of an “answer” (e.g., a match), or the top-k such answers based on the answers probabilities. The difference is that the probability of an answer is defined as a sum over all possible worlds (e.g., all possible trees in which this match appears), where we are computing a maximum (or top-k) over the possible trees. This is a different problem with different motivation and different techniques for solutions. Furthermore, for most realistic models this problem becomes \#P-hard in general (while ours is PTIME), and various restrictions which are not imposed in our case are required in the context of probability computation to allow for tractability.

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 [48, 33, 45] which may be expressed as a first-order knowledge base with probabilities attached to formulas, and probabilistic datalog where probabilities are attached to rules (e.g., [21, 16]). However, the focus in these lines of work is on the problem of probabilistic inference, i.e., computing the probability of a fact or formula (by summing over all possible worlds in which the fact appears/the formula is satisfied); to our knowledge, no counterparts of our query language or techniques were studied in these contexts. In contrast, the various formulations of probabilistic inference typically lead to very high complexity, with solutions that involve approximation algorithms based on sampling.

8. CONCLUSION
We have presented in this paper selPQL, a top-k query language for datalog provenance, and an efficient algorithm for tracking selective provenance guided by a selPQL query. We have showed that the algorithm incurs polynomial data complexity and have experimentally studied its performance for various datalog programs and selPQL queries. There are many intriguing directions for future work, including further optimizations and incorporating considerations such as diversification and user feedback.
Acknowledgements. This research was supported by the Israeli Science Foundation (grant No. 1636/13) and the Broadcom Foundation and Tel Aviv University Authentication Initiative.

9. REFERENCES


