Selective Provenance for Datalog Programs Using Top-K Queries

Daniel Deutch, Amir Gilad, Yuval Moskovitch
Tel Aviv University

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 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 then 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` 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 allows to specify 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, when 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 provide simple means to the analyst to affect the rank of results, by assigning *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 based on 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 relevance to 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, relation names are designed for each pair of (relation of P, part of p), and corresponding rules whose body relations together "guarantee" satisfaction of pattern parts.
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. Our solution here is again in 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, 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; see section 7. We further design a novel algorithm for computing the top-k derivation trees, by exploring modifications of the top-1 tree. Subtleties in doing that efficiently include (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 materialization of trees.

Finally, a final step is 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 we illustrate it with an example.

Definition 2.1. *[1] A* Datalog program *is a finite set of* Datalog rules*. A Datalog rule is an expression of the form:*

$$R_1(\mathbf{u_1}) : -R_2(\mathbf{u_2})...R_n(\mathbf{u_n})$$

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

We make the distinction between extensional (i.e. occurring in the input database) and intensional (i.e. defined by rules) facts and relations (and relation instances), using edb for the former and idb for the latter.

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 (this fixpoint is guaranteed to uniquely exist). Finally, given a database

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
p^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^t}	Annotated relation

Table 1: Notations Table

D and a program P we will use $P(D)$ (termed the "program result") to denote the restriction to idb relations of the database instance obtained by applying to D the consequence operator induced by P.

EXAMPLE 2.2. *AMIE [23] is a system for the automatic inference of* rules*, by identifying "patterns" in the KB. Taking a database perspective, the rules form a* Datalog program *and are evaluated with respect to a database instance (the incomplete KB) to form an idb instance (the completed KB). Among many others, the idb instance includes a binary relation dealsWith (an edb "copy" of this relation appears as well, with a rule to copy its contents that is omitted for simplification), including 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)`

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 edb 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)`

The rules r_1, r_2, r_3 form a Datalog program whose evaluation (with respect to the instance presented in Table 2; 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 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

exports

Country	Product
France	wine
Cuba	tobacco
Cuba	coffee beans

imports

Country	Product
Cuba	wine
Mexico	wine
Mexico	tobacco
France	tobacco

dealsWith

Country$_a$	Country$_b$
Mexico	France

Table 2: Database

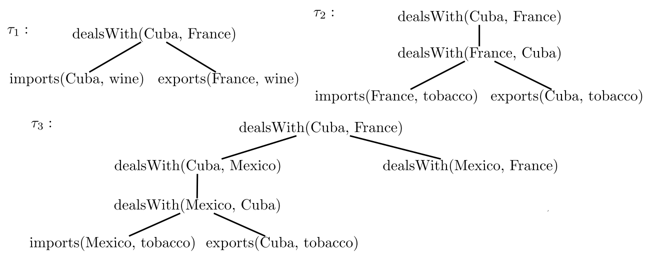


Figure 1: Derivation Trees

is dictated by the consequence operator of the Datalog program: the labels set of the children of node n corresponds to an instantiation (via an assignment) of a 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 $trees(P, D, t)$ 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 absence of recursion in P) possible derivation trees of a given fact, and so it is infeasible to materialize $trees(P, D)$.

EXAMPLE 2.3. *Three derivation trees for the fact $t = dealsWith(Cuba, France)$, based on the program given in Example 2.2 and the example Database given in Table 2, 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).*

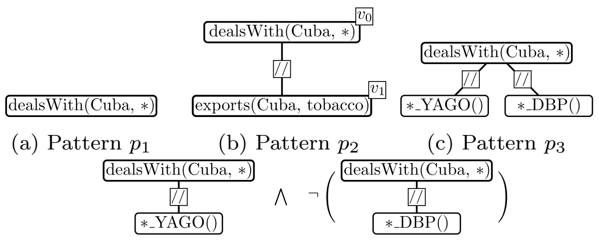
3. QUERYING DATALOG PROVENANCE

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.*

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*



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

(d) Pattern p_4

Figure 2: Tree Pattern Examples

may be marked as regular *(/) or* transitive *(//), and in the latter case may be matched to a path of any length.*

Example 3.2. *Consider a scenario where an analyst is particularly interested in derivations of facts $dealsWith(Cuba, *)$ for some constant replacing the wildcard. All explanations for such facts may be requested using the pattern in Figure 2a. The pattern in Figure 2b queries the structure of derivations, and 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 pattern in Fig. 2c is relevant when (omitted) rules perform* integration *of two ontologies (YAGO and DBPedia); we use $*$_YAGO() and $*$_DBP() [1] to match the set of all relations from YAGO and DBPedia respectively; then the pattern specifies interest in derivations of facts $dealsWith(Cuba, *)$ that are based on integrated data from both sources, in order to determine the usefulness of the integration in this context.*

We next define the semantics of derivation tree patterns, i.e. the notion of matching a given derivation tree; the semantics is in the spirit of XML query languages with some technical differences (see below).

Definition 3.3. *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.*

Note. In the tree pattern semantics, the root of the tree pattern p is mapped to the root of the derivation tree τ to allow specifying interest in provenance of particular facts, or facts belonging to particular relations. A semantics looking for a match of p in a sub-tree of τ can easily be simulated through the use of wildcard-labeled root connected by a transitive edge to p. The constraint on mapped edges/paths to be disjoint is quite natural here and also simplifies the presentation of algorithms; however it may also be relaxed, by considering all possible orderings and specifying them as a disjunction of patterns.

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

Definition 3.4. *Given a (finite or 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 (finite or 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 3.5. *Consider the Datalog program P given in Example 2.2, the Database instance given in Table 2 and the tree pattern p_2 in Figure 2b. The set $p_2(P, D)$ includes infinitely many derivation trees, including in particular τ_2 and τ_3 shown in Figure 1.*

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 Figure 2d 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. To this end, we propose to *rank* different derivations based on the rules and facts used in them. We propose to model ranking of derivations by associating weights with the input database *facts* as well as the individual *rules*, and aggregating these weights.

We consider weights that are elements of an *ordered monoid*, namely 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 3.6. *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 3.7. *To rank derivation trees by their length we use the monoid $(\mathbb{Z}^-, +, 0, <)$, and the weighting function w_1 that maps all rules to weight -1; then the weight of a derivation is the negative of its length. Another way to rank the derivation trees is to use* confidence *values associated with rules. In AMIE such confidence values reflect the rule's support in underlying data. Here we use ordered monoid $([0, 1], \cdot, 1, <)$ and assign e.g. to the three rules in our example, the weights $w_2(r_1) = 0.8$, $w_2(r_2) = 0.5$, $w_2(r_3) = 0.7$ (this is the weight function we will use in subsequent examples). Similarly, fact weights may reflect their confidence (in this simplified example we assign the neutral* 1

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

to all facts). Another alternative is to use min *as aggregate function, which leads to ranking derivations based on their "weakest" rule/fact, etc.*

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 3.8. *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 3.9. *Using the weight function w_2 defined by the confidence value associated with rules in the above example and aggregating via multiplication, the weights of exemplified trees are $w_2(\tau_1) = 0.5$, $w_2(\tau_2) = 0.5 \cdot 0.8 = 0.4$ and $w_2(\tau_3) = 0.7 \cdot 0.8 \cdot 0.5 = 0.28$.*

Last, we may define top-k queries and their results.

Definition 3.10. *Given a pattern p, a weight-aware Datalog instance (P, D, w) and a natural number k, we use $top-k(p, P, D, w)$ to denote the set containing for each fact t in $P(D)$ the k derivation trees of t that are of highest weight*[2] *out of those in $p(P, D)$. We use* TOP-K *to denote the problem of finding $top-k(p, P, D, w)$ given the above input.*

Example 3.11. *The top-2 results for the fact $dealWith(Cuba, France)$ with the weighting function w_2 defined in Example 3.7 and the pattern given in Figure 2b are τ_2 and τ_3 in Figure 1 with weights of* 0.4 *and* 0.28 *respectively. Note that τ_1 does not match the pattern.*

We propose a two step algorithm for solving `TOP-K` as depicted in Figure 3. The first is based on a static analysis of the program (i.e., and crucially, without considering the underlying database) with respect to the pattern. The output of this step is an "instrumented" program, with relations and rules that "guide" provenance tracking so that only relevant provenance, according to the pattern, is tracked. The second step is then evaluation of the instrumented program with respect to the underlying Database, along side with computing top-k qualifying derivation trees for each fact. The algorithm, detailed in the following sections, 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), 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.

[2]Ties are decided arbitrarily.

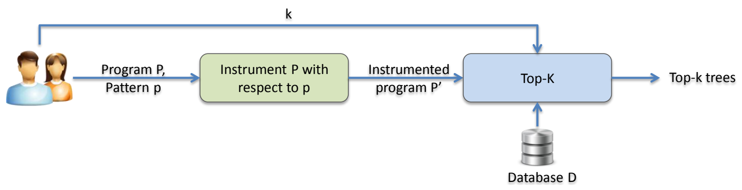


Figure 3: High-level Framework

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.

```
input  : Weighted Program $P$ and a pattern $p$
output: "Instrumented" Program $P_p$
1  foreach pattern node $v \in p$ do
2      Let $v_0, \dots, v_n$ 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 $[R^v(y_0, ..., y_m) : -A(\beta)]$ to $P_p$;
8              end
9              else
10                 foreach $\beta' \in ex(A(\beta), \{v_0, ..., v_n\})$ do
11                     Add $[R^v(y_0, ..., y_m) : -\beta']$ to $P_p$;
12                 end
13             end
14         end
15         if $v$ is a transitive child then
16             foreach $\beta' \in tr - ex(\beta, v)$ do
17                 Add $[R^{v^t}(x_0, ..., x_m) : -\beta']$ to $P_p$;
18             end
19         end
20     end
21     foreach rule $[R^v(y_0, ....y_m) : -\beta]$ added to $P_p$ do
22         Add $[R^{v^t}(y_0, ....y_m) : -\beta]$ to $P_p$;
23     end
24     HandleEDB ();
25 end
26 Clean failed rules in $P_p$ ;
27 return the union of rules in $P$ and $P_p$;
```

Algorithm 1: Program instrumentation according to pattern

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 sub-pattern rooted by v; derivations for R^{v^t} 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 $v_0, ..., v_n$ 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), and in this case we augment A so that a variable x mapped to wildcard, is in fact 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 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 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 the rule from which they originated. 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 9-12), 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" $ex(atoms, \{v_0, ..., v_n\})$ as follows. Consider all one-to-one (but not necessarily onto) functions f that map the set $\{v_0, ..., v_n\}$ to the set $atoms = \{a_0, ..., a_k\}$. Each such function defines a new set of atoms obtained from *atoms* by replacing atom $a_i = R(x_0, ...x_m)$ by $R^{v_j}(x_0, ..., x_m)$ if $f(v_j) = a_i$ and v_j is not a transitive child, or by $R^{v_j^t}(x_0, ..., x_m)$ if v_j is a transitive child (atoms to which no node is mapped remain intact). We then define $ex(atoms, \{v_0, ..., v_n\})$ as the set of all atoms sets obtained for some choice of function f. In Line 11 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 subtree rooted at v, by designing rules that propagate the satisfaction 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 15-18), 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 $R^{v^t}(...)$, Lines 21-22), 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 $R(x_0, ...x_m)$ in *atoms* by $R^{v^t}(x_0, ...x_m)$ (and keeping the other atoms intact), and add the corresponding rules (Line 17).

```
     dealsWith(a, b):- dealsWith(b, a)
     dealsWith(a, b):- imports(a, c), exports(b, c)
     dealsWith(a, b):- dealsWith(a, f), dealsWith(f, b)
[r1'] 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)
[r2'] exports^{v1^t}(Cuba, tobacco):- exports(Cuba, tobacco)
```

Figure 4: The instrumented program

The function HandleEDB adds rules for nodes that locally match edb facts. If v locally matches an edb fact $T(\ldots)$ then we add rules that copy the relevant tuples from the database into the new relations $T^v(\ldots)$ and $T^{v^t}(\ldots)$ (these rules have weight 0).

The final step of the algorithm is "cleanup" (Line 26), removing a subset of the newly added rules that are useless either because some idb relation in their body has no rule that may derive it, or because the rule is not reachable from the rules added for the root node of the pattern.

EXAMPLE 4.1. *Consider the program P given in Examples 2.2, and the tree pattern shown in Figure 2b, where v_0 is the root node in p_2 and v_1 is the leaf. The output program is shown in Figure 4. We just illustrate the generation process of some of these rules. 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 11 the algorithm adds the rule r_1'. Intuitively derivations for facts in `dealsWith`$^{v_0}(...)$ must match the sub-pattern rooted by v_0. Then derivations for facts in `dealsWith`$^{v_1^t}(...)$ must include a sub-tree that matches the sub-pattern rooted by v_1, and generated rules for `dealsWith`$^{v_1^t}(...)$ 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. Rule r_2' is added by* HandleEDB *since $exports(a, b)$ locally matches v_1.*

The instrumented program satisfies the following fundamental property. Given an atom $R(...)$, $R^v(...)$ or $R^{v^t}(...)$ we define its *origin* to be $R(...)$, 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 4.2. *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:*

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

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

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

PROOF.

1. Since $P \subseteq P_p$, every $\tau \in \mathit{trees}(P, D)$ is also in $\mathit{trees}(P_p, D)$ and it holds that $\mathit{origin}(\tau) = \tau$, thus $\mathit{trees}(P, D) \subseteq \bigcup_{t \in \mathit{trees}(P_p,D)} \mathit{origin}(t)$.

   In addition, recall that every node in a derivation tree $\tau \in \mathit{trees}(P_p, D)$ 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 rules in P (possibly with repetitions), and the rules added by `HandleEDB`. $\mathit{origin}(\tau)$ is obtain by removing the annotation from τ and pruning branches added due to the function `HandleEDB`, thus every node in $\mathit{origin}(\tau)$ correspond to a derivation rule in P and therefore $\mathit{trees}(P, D) \supseteq \bigcup_{\tau \in \mathit{trees}(P_p,D)} \mathit{origin}(\tau)$, namely

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

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

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

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

   - v locally matches an edb fact $T(...)$. 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 $T^v(...)$ (and $T^{v^t}(...)$). The derivation tree τ of $T^v(...)$ (and $T^{v^t}(...)$, in the case where v is a transitive node) consist of two nods, a root, $T^v(...)$ (or $T^{v^t}(...)$), and a leaf, $T(...)$, and $\mathit{origin}(\tau)$ is simply $T(...)$ in this case.
   - v locally matches an idb atom $R(...)$ through partial assignment A. In this case, a in line 7 the algorithm adds new rule for each rule in P that its head locally matches v (i.e. $t = R(...) \in P(D) \Leftrightarrow R^v \in P_p(D)$). 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 $\mathit{trees}(P, D)$. Derivation trees τ of $R^v(...)$ consists of one of the rules added in of the form line 7 and the derivation trees of each fact in the rules body, Therefore $\mathit{origin}(\tau)$in the tree obtain by removing the annotation v and it is a derivation tree in $\mathit{trees}(P, D)$.

     For the case where v is transitive, the algorithm adds two types of derivation rule for R^{v^t}, (i) the rules added in line 17 and (ii) in line 22. 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 $\tau \in p|_p(P, D)$) or (2) there exists a node in τ that locally matches v. The rules added in line 22 captures case (1) and this case is similar to the case where v is not transitive.

     The rules added in line 17 captures case (2). Note that the body of such rules contains exactly one annotated relation name $S^{v^t}(...)$ while the rest are facts in $P(D)$ and thus their derivation tree are in $\mathit{trees}(P, D)$. We can thus show by induction that for the proposition holds for $S^{v^t}(...)$. A derivation τ tree that contains type (i) rules must contains a derivation of type (ii) (since initially there are no annotated facts in the database). If $S^{v^t}(...)$ is derived using type (ii) rule, then clearly, by removing the annotations we obtain a derivation tree in $\mathit{trees}(P, D)$. Other the proposition holds by the induction.

   Suppose that the proposition holds for all v s.t. the $p|_v$ is from depth $< k$. Let v a pattern node where $p|_v$ is from depth 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 $\forall p_j\ \exists u$ s.t. u is a child of the root in τ and for the subtree rooted by it τ_j it holds that $\tau_j \in p|_{v_j}(P, D)$. The last derivation step in any derivation tree $\tau \in p|_v(P, D)$ must be done by a derivation rule r the algorithm adds in line 11. The body of the rule β can consists of both annotated and not annotated atoms. Derivation trees of atoms that are not annotated are tree in $\mathit{trees}(P, D)$ for annotated relation it holds that $\tau_j \in p|_{v_j}(P, D)$ and since $p|_{v_j}$are from depth $k - 1$ by the induction hypothesis it holds that

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

     Therefor, the derivation tree obtain by replacing $R^v(...)$ with $R(...)$ and for each subtree τ' rooted by the children of the root in τ with $\mathit{origin}(\tau)$ is $\mathit{origin}(\tau)$ and it holds that $\mathit{origin}(\tau) \in p|_v(P, D)$
   - The case where v is transitive is similar to the case that v is a transitive leaf.

3. The weight of the new rules added by the algorithm are assigned the weight the rule 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 $\mathit{origin}(\tau)$ (due to the construction of the rules added by `HandleEDB`). Therefore we have:

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

□

We refer to v and v^t in $R^v(...)$ and $R^{v^t}(...)$ 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 property will be utilized in the next step, where we evaluate the instrumented program while retrieving only relevant provenance.

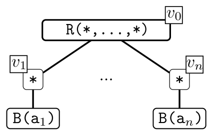
Complexity and output size. Given a datalog program P of size $|P|$ and a pattern p, the algorithm traverses the pattern, and for each node $v \in p$ iterates over the program rules. Let $w(p)$ be the width of p, i.e. the maximal number of children of a node in p. The maximal number of new rules the 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 4.3 (LOWER BOUND). *There is a class of patterns $\{p_1, ...\}$ and a class of programs $\{P_1, ...\}$, 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 4.2 with respect to P_n, p_n.*

PROOF. (sketch) Consider the following datalog program P_n ($a_1, ...a_n$ are constants):

```
R(x_1, x_2, ..., x_n):- R_1(x_1), ... R_n(x_n)
R_1(x):-B(x)
...
R_n(x):-B(x)
```

and the pattern p_n:

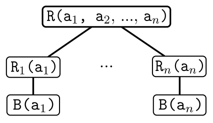


Both the pattern width and program size are polynomial in n (the pattern width $w(p_n)$ is n and the program P_n consists of $n+1$ rules). We claim that every instrumented program P^i satisfying the conditions of Proposition 4.2 must include at least $n!$ rules. To observe that this is the case, first 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":

r_1: $R^{v_0}(x_1, x_2, ..., x_n)$:- $R_1^{v_1}(x_1)$, ... $R_n^{v_n}(x_n)$
...
$R^{v_0}(x_1, x_2, ..., x_n)$:- $R_1^{v_n}(x_1)$, ... $R_n^{v_1}(x_n)$

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 $B(a_i)$ 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).



τ

Alternatively, if P^i "groups" two relation names (w.l.o.g. say R_1^{v1} and R_1^{v2}) together (say using relation name R_1^{v12}), and then e.g. generates the two rules $R_1^{v12}(a_1) : -B(a_1)$ and $R_1^{v12}(a_2) : -B(a_2)$ (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 $t = R^{v_0}(...)$, 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.

□

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.

Negation. The algorithm for intersecting a negation of a pattern is similar to Algorithm 1 with a slight modification, 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 sub-pattern rooted by v and derivations for $R^{\neg v^t}$ should not include a descendant that matches the sub-pattern rooted by v.

We define $neg - ex(atoms, \{v_0, ..., v_n\})$ where $atoms$ is a set of atoms and the v_i's are pattern nodes as follows: if $|atoms| \geq n$, then $neg - ex(...)$ is the set of all atoms sets obtained from $atoms$ by replacing, for each v_i, every atom $R(x_0, ..., x_m)$ in $atoms$ by $R^{\neg v_i^t}(x_0, ..., x_m)$, if v_i is a transitive node and $R^{\neg v_i}(x_0, ..., x_m)$ otherwise. If $|atoms| < n$ then $neg - ex(...)$ is the set that contains only the set $atoms$.

The modifications to the algorithm are then as follows. In the case where $R(...)$ locally-matches v (line 4), if v is a leaf, we do not add a derivation rule for R^v since we want ignore derivation that match v. Otherwise (in line 11), we add for each $\beta' \in neg - ex(A(\beta), \{v_0, ..., v_n\})$ the rule $R^{\neg v}(y_n, ..., y_m) : -\beta'$. Intuitively, a derivation that does not match sub-pattern rooted by v, in the case where the root $r(...)$ locally-matches v, either has less than n children in the derivation, where n is number of the children of v, or the derivation rooted by at least one of $R(...)$'s children does not match one of the children of v, which is captured by the $neg - ex(...)$ set.

In addition, for any rule $R(...) : -\beta$ in P, when $R(...)$ does not locally-match v, we add the rule $R^{\neg v}(...) : -\beta$.

For the case where v is a transitive child (line 15) we modify the new rules body to be $neg - ex(A(\beta), \{v\})$. Finally, instead of adding rules for edb atoms that locally-match v, the function HandleEDB adds the rules $T^{\neg v}(x_0, ..., x_m) : -T(x_0, ..., x_m)$ and $T^{\neg v^t}(x_0, ..., x_m) : -T(x_0, ..., x_m)$ for each edb atom $T(x_0, ..., x_m)$ that does not locally-matches v.

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. A generalization of Proposition 4.2 can be shown for Boolean combinations of patterns.

Complexity. The time complexity remains 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).

5. FINDING TOP-K DERIVATION TREES

The second step of the algorithm is the enumeration of 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, which is (an adaptation [3] of) the generalized dijkstra algorithm of Knuth [38]. The generation of the top-1 tree (out of qualifying trees)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`). 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 p^0 of the pattern.

```
  input  : Weighted Datalog Program $P$, Database $D$
  output: top-1 derivation tree for facts of the form
          $R^{p^0}(...)$
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 [$\arg\max_{t \in Cand}$ Top($t, DTable, P$)] to
      $DTable$
5 end
6 return the entries of all $e \in DTable$ s.t. the fact $t$
  of $e$ is of the form $R^{p^0}(...)$;
```
Algorithm 2: Top-1

[3]The algorithm of [38] works for weighted context free grammars rather than Datalog, see discussion of related work.

EXAMPLE 5.1. *Consider the output program P_p of Algorithm 1 given in Example 4.1, and the Database D shown in Table 2. 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 $t_3 = exports^{v_1^t}(Cuba, tobacco)$ can be derived with weight 1 using the edb fact $t_1 = exports(Cuba, tobacco)$ 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 $(t_3, 1, \{*t_1\})$ 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 $t_4 = dealsWith^{v_1^t}(France, Cuba)$ 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 (In this example 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 the algorithm adds $(t_5, 0.4, \{*t_4\})$ to $DTable$.*

Using $DTable$, the top-1 derivation tree of every fact t can be constructed using the entry of t, by attaching t to the top-1 derivation trees of each one of the facts $t_1, \ldots, t_k$ in the pointers list of t (which are recursively constructed).

5.2 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 2, 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 enumeration of top-k trees for each fact may then simply be done by pointer chasing.

Overall 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 $|P|^{|p|}$. 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 a most $|D|^{|P_p|} = |D|^{(|P|^{|p|})}$, where $|D|$ is the extensional Database size, thus on the i'th step, the size of $DTable$ is bounded by $i \cdot |D|^{(|P|^{|p|})}$. Therefore the time complexity of the i'th step is $O(i^2 \cdot |D|^{O(|P|^{|p|})})$. The complexity of computing the top-k derivation trees is therefore

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

Finally, generating the top-k trees from $DTable$ is linear in the output size, and thus the overall complexity of TOP-K is $O(k^3 \cdot |D|^{O(|P|^{|p|})} + |out|)$, 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 $u_0, \dots, u_m$ 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 v 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 fact outputted by Algorithm 2, 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 the $DTable$. 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 this undesirable 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 sub-tree.

EXAMPLE 5.2. *Reconsider the output program of the algorithm in Example 4.1. The top-2 derivation trees for the fact $dealsWith(Cuba, France)$ are shown in Figure 1, 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 Figure 5a. The nodes u and u_0 correspond to the derivation rule*

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

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`$^{v_1^t}$`(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 Figure 1 (as τ_2 and τ_3 respectively).*

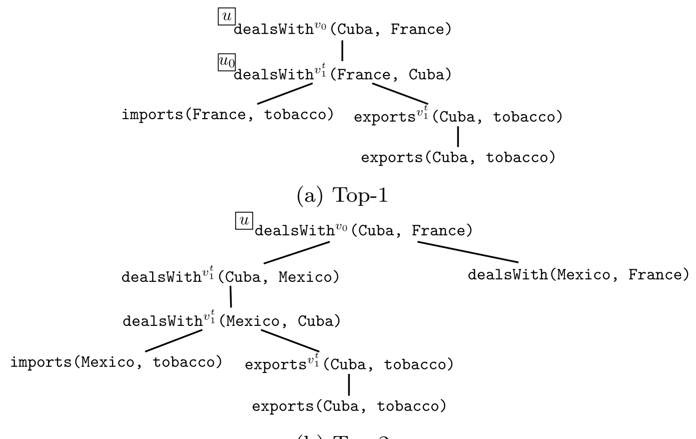

u
dealsWith^v0(Cuba, France)
u0
dealsWith^v1t(France, Cuba)
imports(France, tobacco)
exports^v1t(Cuba, tobacco)
exports(Cuba, tobacco)
(a) Top-1
u
dealsWith^v0(Cuba, France)
dealsWith^v1t(Cuba, Mexico)
dealsWith(Mexico, France)
dealsWith^v1t(Mexico, Cuba)
imports(Mexico, tobacco)
exports^v1t(Cuba, tobacco)
exports(Cuba, tobacco)
(b) Top-2

Figure 5: Top-2 Derivation Trees (with annotations)

Diversification. Our paradigm may be adapted to support *diversification*, by intersecting the program with a negated pattern between computing the top-i and the top-$(i+1)$ results. For instance we may intersect the program with a negated pattern consisting of a new root labeled by wildcard, connected by a transitive edge to a copy of the i'th result; this will make sure that the i'th tree will not appear as a sub-tree in the $i+1$ result. Other notions of diversification are conceivable and some may also be encoded by negation; developing dedicated optimizations for them is left for future work.

6. IMPLEMENTATION AND EXPERIMENTS

We have implemented our algorithms in a system prototype called `selP` (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 `selPQL` 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 `selPQL` queries (different number of requested results and different patterns, varying in size and structure), and for increasingly large output databases [4]. 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 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 the following recursive datalog program consisting of rules mined by AMIE [23], automatically translated into datalog syntax, with weights assigned by AMIE and reflecting rule confidence:
```
hasChild(a, b) :- isMarriedTo(e, a), hasChild(e, b)
hasChild(a, b) :- isMarriedTo(a, f), hasChild(f, b)
isMarriedTo(a, b) :- isMarriedTo(b, a)
dealsWith(a, b) :- dealsWith(a, f), dealsWith(f, b)
isMarriedTo(a, b) :- hasChild(a, c), hasChild(b, c)
dealsWith(a, b) :- dealsWith(b, a)
produced(a, b) :- directed(a, b)
dealsWith(a, b) :- imports(a, c), exports(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)
```
   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 the following 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):
```
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),
Quantity  * TotalSubcost = Cost
```
   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.

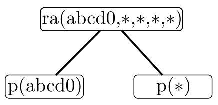
6.2 Experimental Results

Figure 9 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.

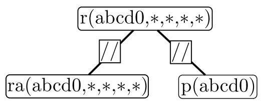
In Figure 9a, the results for the IRIS dataset are presented for 4 different patterns: (p_1) binary tree pattern with three nodes without transitive edges and (p_2) with two transitive edges, (p_3) three nodes chain pattern with two transitive edges, and (p_4) six node pattern with three levels and four transitive edges. The pattern width and structure unsurprisingly has a significant effect on the execution time, but the overhead with respect to 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 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

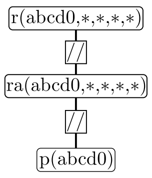
[4]The precise datalog programs and patterns that have been used may be found in the full version [22].



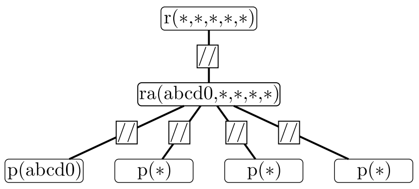
(a) p_1



(b) p_2

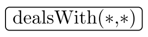


(c) p_3

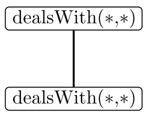


(d) p_4

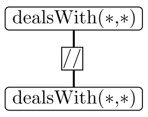
Figure 6: Example Patterns for IRIS



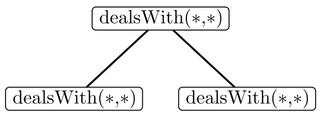
(a) p_5



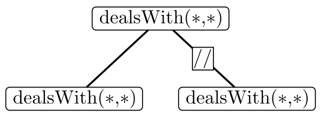
(b) p_6



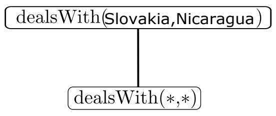
(c) p_7



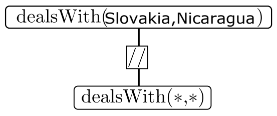
(d) p_8



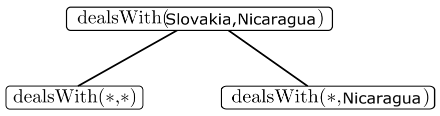
(e) p_9



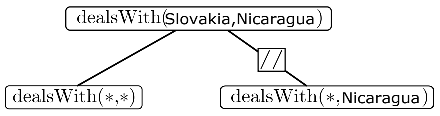
(f) p_6^*



(g) p_7^*

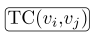


(h) p_8^*

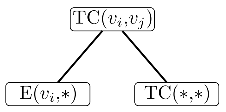


(i) p_9^*

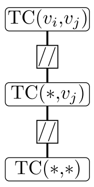
Figure 7: Example Patterns for AMIE



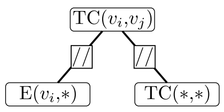
(a) p_{10}



(b) p_{11}

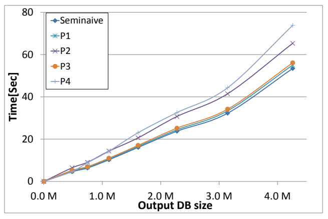


(c) p_{12}

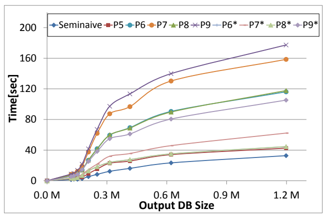


(d) p_{13}

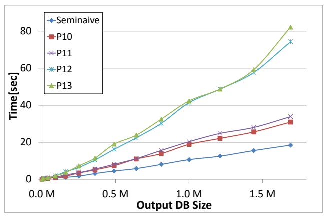
Figure 8: Example Patterns for TC



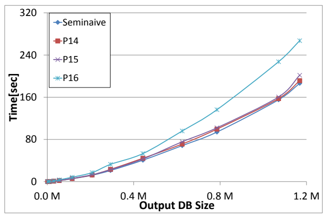
(a) IRIS



(b) AMIE



(c) Transitive Closure



(d) Explain

Figure 9: Time of computation as a function of DB size

(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 9b. 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 seminaive) for p_6^*. For p_7^* and the same number of output tuples it took 62 seconds (less than 2 times slower than seminaive), 44.6 seconds (1.3 times slower than seminaive) for p_8^* and 105 seconds (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 minutes (3.5 times slower than seminaive) for p_6, 2.6 minutes (4.8 times slower) for p_7, and less than 2 and 2.9 minutes (3.6 and 5.4 times slower) for p_8 and p_9 respectively.

In Figure 9c 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 seminaive in average), 33 seconds for p_{11} (1.8 times slower than seminaive in average), 74 seconds for p_{12} (4 times slower) and 82 seconds 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 9d 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 seminaive) for p_{14}, 3.3 minutes (10% overhead w.r.t seminaive) for p_{15} and 4.4 minutes (85% overhead w.r.t seminaive) 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 10 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).

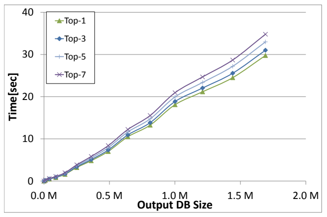


Figure 10: Varying K (Transitive Closure Dataset)

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 for the performance of the top-k step is the size and structure of the obtained instrumented program, which in turn is highly dependent on the size and structure 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 Prop. 4.2, 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). We note that our optimizations aimed at reducing the number of rules, as outlined in section 4, have indeed improved the algorithm's performance by as much as 50%.

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 relational algebra to Nested Relational Calculus, with different provenance models (see e.g. [6, 29, 28, 24, 35, 10, 55, 7, 19]) and applications [54, 42, 50, 41, 26], and with different means for efficient storage (e.g. [4, 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 semirings, 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 understand (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 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, 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 recursive 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 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 "modules", and then to set provenance tracking "on" or "off" for each module. For the chosen modules, the system generates a record of *all instantiations* of rules that has occurred during the program execution. This is a counterpart of our notion of full provenance, since all derivation trees may be obtained from this structure. Once full provenance is tracked, one may analyze it (e.g. using further CORAL queries), or browse through it through a dedicated Graphical User Interface. In contrast to our work, this line of work focuses on analyzing the *the full provenance*, and cannot be used to specify in advance which parts of the provenance to track (the in-advance specification is limited to the very 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 enough to express properties of interest, while allowing for feasibility (and low complexity) of instrumentation (which was not addressed in [3]).* Indeed, `selPQL` allows the specification 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 the other works in this context, such as the debugging system for the LDL deductive database presented in [51]. We note that a feature that is present in [51] and absent from ours is the ability to query *missing* facts, i.e. explore why a fact was not generated. Incorporating such ability in our system (e.g. to find *ranked* explanations for absence of facts) is an intriguing direction for future work.

Program slicing. In [11, 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 [11, 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 [5], 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 derivation in a weighted CFG; other works have studied the problem of finding top-k parses of a given string (thus the derivation size is bounded) in a probabilistic context free grammar. In [12] the author 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 connections 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. Only a very basic idea of the algorithm for top-1 is inspired by that of [38] for finding the (single) maximal *weight* of a tree in a weighted CFG. Novel challenges include having the top-1 algorithm (1) work for datalog, and specifically side-by-side with datalog evaluation and (2) generate a compact representation of the tree itself rather than just its weight, and (3) the entire algorithm for top-k, where a major challenge is in avoiding generation of trees that (when removing the "instrumentation annotations") are duplicates.

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 probability (counterpart of weight 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 effects the further development for top-k computation (see again the discussion above for CFGs). An additional difference is due to our use of a general weight function rather than probabilities. We further note that beyond the difference in model, the *problem* typically considered for probabilistic XML is different than ours. The problem typically studied in these works is 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 restrictions which are not imposed in our case (such as bounding the trees depth) are required 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.

9. REFERENCES

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

[2] A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM*, 1998.

[3] Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. Explaining program execution in deductive systems. In *DOOD*, 1993.

[4] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Sadiq. Efficient provenance storage for relational queries. CIKM, 2012.

[5] Zhuowei Bao, Susan B. Davidson, and Tova Milo. Labeling recursive workflow executions on-the-fly. In *SIGMOD*, 2011.

[6] O. Benjelloun, A.D. Sarma, A.Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17, 2008.

[7] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4), 2008.

[8] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. Query ranking in probabilistic XML data. In *EDBT*, 2009.

[9] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD*, SIGMOD '08, 2008.

[10] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[11] James Cheney, Amal Ahmed, and Umut A. Acar. Database queries that explain their work. *CoRR*, abs/1408.1675, 2014.

[12] Sara Cohen and Benny Kimelfeld. Querying parse trees of stochastic context-free grammars. In *ICDT*, 2010.

[13] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3), 2009.

[14] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

[15] D. Deutch, A. Gilad, and Y. Moskovitch. selp: Selective tracking and presentation of data provenance (demo). In *ICDE*, 2015. to appear.

[16] Daniel Deutch, Christoph Koch, and Tova Milo. On probabilistic fixpoint and markov chain query languages. In *PODS*, 2010.

[17] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. Circuits for datalog provenance. In *ICDT*, 2014.

[18] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2), 1998.

[19] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5), 2012.

[20] I. Foster, J. Vockler, M. Wilde, and A. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *SSDBM*, 2002.

[21] Norbert Fuhr. Probabilistic datalog:a logic for powerful retrieval methods. In *SIGIR*, 1995.

[22] `http://www.cs.tau.ac.il/~danielde/selpFull.pdf`.

[23] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[24] Floris Geerts and Antonella Poggi. On database query languages for k-relations. *J. Applied Logic*, 8(2):173–185, 2010.

[25] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[26] Boris Glavic, Gustavo Alonso, Renée J. Miller, and Laura M. Haas. TRAMP: understanding the behavior of schema mappings through provenance. *PVLDB*, 3(1):1314–1325, 2010.

[27] Boris Glavic, Renée J Miller, and Gustavo Alonso. Using sql for efficient generation and querying of provenance information. In *In Search of Elegance in the Theory and Practice of Computation*. Springer, 2013.

[28] Boris Glavic, Javed Siddique, Periklis Andritsos, and Renée J. Miller. Provenance for data mining. In *Tapp*, 2013.

[29] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[30] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res.*, 34, 2006.

[31] `http://www.iris-reasoner.org`.

[32] Zachary G. Ives, Andreas Haeberlen, Tao Feng, and Wolfgang Gatterbauer. Querying provenance for ranking

and recommending. In *TaPP*, 2012.

[33] Abhay Kumar Jha and Dan Suciu. Probabilistic databases with markoviews. *PVLDB*, 5(11), 2012.

[34] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *SIGMOD*, 2010.

[35] Batya Kenig, Avigdor Gal, and Ofer Strichman. A new class of lineage expressions over probabilistic databases computable in p-time. In *SUM*, pages 219–232, 2013.

[36] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query evaluation over probabilistic XML. *VLDB J.*, 18(5), 2009.

[37] Benny Kimelfeld and Yehoshua Sagiv. Matching twigs in probabilistic XML. In *VLDB*, 2007.

[38] Donald E. Knuth. A generalization of dijkstra's algorithm. *Inf. Process. Lett.*, 6(1), 1977.

[39] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. Top-k keyword search over probabilistic XML data. In *ICDE*, 2011.

[40] B. T. Loo et al. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[41] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12), 2011.

[42] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, 2012.

[43] P. Missier, N. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*, 2010.

[44] Bo Ning, Chengfei Liu, and Jeffrey Xu Yu. Efficient processing of top-k twig queries over probabilistic XML data. *World Wide Web*, 16(3), 2013.

[45] Feng Niu, Ce Zhang, Christopher Re, and Jude W. Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. In *VLDS*, pages 25–28, 2012.

[46] Dan Olteanu and Jakub Zavodny. Factorised representations of query results: size bounds and readability. In *ICDT*, 2012.

[47] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *SIGPLAN*, 2012.

[48] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2), 2006.

[49] Royi Ronen and Oded Shmueli. Automated interaction in social networks with datalog. In *CIKM*, 2010.

[50] Sudeepa Roy and Dan Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, 2014.

[51] Oded Shmueli and Shalom Tsur. Logical diagnosis of LDL programs. *New Generation Comput.*, 9(3/4), 1991.

[52] Y. L. Simhan, B. Plale, and D. Gammon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, 5(2), 2008.

[53] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[54] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[55] Prov-overview, w3c working group note. `http://www.w3.org/TR/prov-overview/`, 2013.