Provenance Management for Evolving RDF Datasets

Argyro Avgoustaki[1,2], Giorgos Flouris[2], Irini Fundulaki[2], and Dimitris Plexousakis[1,2]

[1] Department of Computer Science, University of Crete, Greece
[2] Institute of Computer Science, FORTH, Greece
{argiro,fgeo,fundul,dp}@ics.forth.gr

Abstract. Tracking the provenance of information published on the Web is of crucial importance for effectively supporting trustworthiness, accountability and repeatability in the Web of Data. Although extensive work has been done on computing the provenance for SPARQL queries, little research has been conducted for the case of SPARQL updates. This paper proposes a new provenance model that borrows properties from both *how* and *where* provenance models, and is suitable for capturing the triple and attribute level provenance of data introduced via SPARQL INSERT updates. To the best of our knowledge, this is the first model that deals with the provenance of SPARQL updates using algebraic expressions, in the spirit of the well-established model of *provenance semirings*. We present an algorithm that records the provenance of SPARQL update results, and a reconstruction algorithm that uses this provenance to identify a SPARQL update that is *compatible* to the original one, given only the recorded provenance. Our approach is implemented and evaluated on top of Virtuoso Database Engine.

1 Introduction

During the last few years, we have witnessed an explosion in the volume of semantic data available on the Web. These data are usually published using the RDF data model[1], where information is represented using *triples*, organized in *named graphs* [6], thereby forming *quadruples*. Querying and updating RDF data is performed using the W3C standards SPARQL[2] and SPARQL Update[3] respectively.

Nowadays semantic data is the most prominent example of large scale data where one could create new *datasets* (sets of quadruples) by integrating existing ones. In this setting, recording the *provenance* of such data, i.e., their *origin*, which describes from *where* [5] and *how* [12] the data was obtained, is of crucial importance for supporting trustworthiness, accountability and repeatability. This is necessary due to the open and unconstrained nature of the Web of Data and the growing tendency to populate scientific data warehouses through SPARQL updates offered by SPARQL endpoints.

In this work we deal with the problem of *capturing and managing the provenance of quadruples constructed through SPARQL updates*. More specifically, we focus on SPARQL INSERT operations (we refer to them as INSERT updates) used to add newly created triples in a target named graph (i.e., quadruples). The purpose of computing

[1] http://www.w3.org/TR/rdf11-primer/

[2] http://www.w3.org/TR/sparql11-overview/

[3] http://www.w3.org/TR/sparql11-update/

the provenance for such operations is to record from *where* and *how* each quadruple was constructed, thereby allowing us to determine the quadruples and the SPARQL operators that were used to produce it.

The problem of managing provenance information has received considerable attention [5, 9–13, 15, 16, 20, 21], but most works deal with query provenance. W3C published a recommendation [18] concerning the interchange of provenance information, which, however, focuses on providing a syntactic means to represent provenance rather than providing a method for identifying or computing it. Algebraic expressions have been used to capture (query) provenance in varying levels of detail [11, 12, 15]. In the RDF context, provenance is often represented using named graphs [6, 7, 10, 15].

However, the unique requirements associated with SPARQL update provenance do not allow a direct reuse of such approaches. One problem is that the named graph component of a quadruple is defined by the user in the INSERT update, so triples with different origin may be added to the same named graph. Thus, the standard approach of capturing provenance through the named graph of a quadruple is not sufficient, and provenance should be defined for quadruples, rather than triples (as in most works).

In addition, quadruples created via INSERT updates could be the result of combining values found in different quadruples through different SPARQL operators. This creates a unique challenge, because each attribute of a quadruple may have a different provenance. Thus, fine-grained, *attribute level* provenance models are called for, and more expressive models that go beyond the named graphs approach are needed.

Another challenge stems from the persistence of a SPARQL update result, which implies that when a quadruple is accessed, the SPARQL update that generated the quadruple is no longer available. This makes standard *how* provenance models unsuitable for recording provenance at a fine-grained level in this setting. As an example, standard how-provenance approaches will record that a join was used to generate a quadruple, but will not record the components of the quadruples that were joined to produce the result; even though this information is easily available during queries (via the SPARQL query), this is not the case for SPARQL updates (where the SPARQL update is not available). Recording the INSERT update is not an efficient remedy for the situation, because (a) the syntactic form of the actual INSERT update is irrelevant and (b) the INSERT update is no longer relevant, as the dataset has evolved.

Therefore, more fine-grained forms of how-provenance are called for. We define this more demanding form of how-provenance in an indirect manner, by introducing the notion of *reconstructability*, which refers to the ability of using the provenance information for *reconstructing* an INSERT update that is *compatible* (see Definition 4) with the INSERT update that generated this quadruple.

We show that, to satisfy the requirement of reconstructability, the provenance of a quadruple should be expressive enough to identify: (a) the quadruples that contributed to its creation (*where provenance* [8]), and (b) how these quadruples were used (via joins and unions) to generate the new one (*how provenance* [12]), under the more demanding form of how-provenance explained above.

The main contributions of this paper[4] are:

[4] Detailed presentation of our approach including the source code of our implementation can be found in `http://www.ics.forth.gr/isl/provenance`

- The introduction of a *fine-grained and expressive provenance model* that borrows from both *where* and *how* provenance models, is suitable for encoding both *triple* and *attribute* level provenance of quadruples obtained via INSERT updates, and allows the *reconstructability* of such updates from their provenance.
- The provision of algorithmic support for our model via the *provenance construction* and *update reconstruction* algorithms. The former is used for computing and recording the provenance of the result of an INSERT update based on the proposed model, whereas the latter exploits the expressiveness of our model to report on the generation process of a quadruple.
- The implementation, theoretical analysis, and experimental evaluation of these algorithms on top of Virtuoso Database Engine.

2 Preliminaries

We consider provenance in the context of an *RDF dataset* (denoted by D); for simplicity, we assume that an RDF dataset is composed of a set of *quadruples* of the form *(s,p,o,n)*, where *(s,p,o)* is a triple belonging in a *named graph n*.

SPARQL 1.1 is the official W3C recommendation for querying and updating RDF datasets, and is based on the concept of matching patterns against such graphs. Patterns are defined via *quad patterns* which are like quadruples but allow *variables* (prefixed with ?) in the subject, property or object position. Quad patterns can be combined using *SPARQL operators* to form *graph patterns*. In this work, we focus on *union* (UNION) and *join* (".") operators only, ignoring *optional* and *filters* (we plan to deal with these operators in future work). Thus, the considered INSERT updates are of the form: $U :=$ INSERT $\{qp_{ins}\}$ WHERE $\{gp\}$, where qp_{ins} is a quad pattern and *gp* is a graph pattern formed as a union of individual graph patterns, gp^1 UNION ... UNION gp^k. Each gp^i is of the form qp_1^i . qp_2^i . ... . qp_m^i. Note that all INSERT updates containing only union and join operators can be equivalently written in the above form [19]. Note also that INSERT DATA operations can be defined in terms of INSERT [2].

In addition, we require that for each qp_j^i there is a sequence $\langle qp_{j_1}^i, \dots \rangle$ of quad patterns from gp^i, such that each element in the sequence has a common variable with the previous element in the sequence, whereas the first element has a common variable with qp_{ins}. This restriction is necessary to "strip" the graph pattern in the WHERE clause from quad patterns that play no essential role in its evaluation [19].

SPARQL Update specifications do not fully clarify the principles governing transactions with multiple updates [13]; here, we focus on transactions consisting of single atomic updates. Further details on SPARQL are omitted (see [1, 2, 19]).

3 Motivating Example

We provide an example from the medical domain to motivate our approach. Note that this example is used for illustration purposes only and any consequences pertaining to data privacy are out of the scope of our paper. Table 1 shows a dataset D_1 containing four quadruples (with identifiers $c_1, \dots, c_4$), each with a certain provenance ($p_1, \dots, p_4$). These quadruples describe treatments for hypertension that have been provided by different doctors.

	S	P	O	N	PROV
c_1	<hypertension>	<treatedWith>	<diuretics>	<Diabetologist>	p_1
c_2	<hypertension>	<treatedWith>	<diuretics>	<Pathologist1>	p_2
c_3	<hypertension>	<treatedWith>	<diuretics>	<Pathologist2>	p_3
c_4	<hypertension>	<treatedWith>	<b_blockers>	<Pathologist2>	p_4

Table 1: Dataset D_1

Now suppose that a patient visits the hospital and a young doctor diagnoses hypertension. To decide on the proper treatment, he checks the system for previous treatments of hypertension, paying special attention to those proposed by the diabetologist, because the patient's history includes diabetes and some medications may raise the blood sugar levels, a dangerous condition for a diabetic. The result of his query needs also to be recorded in the database, as it will be his suggested treatment, so he executes U:

$$\text{INSERT } \{qp_{ins}\} \text{ WHERE } \{qp_1^1 \text{ UNION } qp_1^2 \,.\, qp_2^2\}$$

where: qp_{ins}: (<hypertension>, <treatedWith>, $?o$, <YoungDoctor>)
qp_1^1: (<hypertension>, <treatedWith>, $?o$, <Diabetologist>)
qp_1^2: (<hypertension>, <treatedWith>, $?o$, <Pathologist1>)
qp_2^2: (<hypertension>, <treatedWith>, $?o$, <Pathologist2>)

The application of U upon D_1 leads to the insertion of c_5, forming dataset D_2, shown in Table 2. The expression p_5 below is used to describe the provenance of c_5:
p_5: $\{\ (\perp, \perp,\ {}_{qp_1^1.o}(c_1)) \quad \oplus \quad (\perp,\ \perp,\ {}_{qp_1^2.o}(c_2\ {}_{\{qp_1^2.o\}}\odot_{\{qp_2^2.o\}}\ c_3))\ \}$
Some explanations on p_5 are in order. First, each operand of $\oplus$ indicates a different way through which c_5 occurred (due to the existence of UNION). The first operand $(\perp, \perp,\ {}_{qp_1^1.o}(c_1))$ resulted from the prescription of the diabetologist; in particular, its subject and property values were dictated by the corresponding constants in qp_{ins} (indicated by the value $\perp$), whereas its object resulted by "copying" the object value (o) of c_1 due to the quad pattern qp_1^1 (denoted by ${}_{qp_1^1.o}(c_1)$).

Similarly, the second operand's subject and property were dictated by the constants of qp_{ins}, whereas the object resulted from the agreeing prescriptions of the two pathologists. In particular, the object value was the result of a join (indicated by $\odot$) between two quadruples, namely c_2, c_3; the join happened between the object position of qp_1^2 (i.e., $qp_1^2.o$) and the object position of qp_2^2 (i.e., $qp_2^2.o$), hence the left and right subscript of $\odot$. Finally, the result of this join was projected over the object position as indicated by the outer subscript $qp_1^2.o$.

	S	P	O	N	PROV
c_1	<hypertension>	<treatedWith>	<diuretics>	<Diabetologist>	p_1
c_2	<hypertension>	<treatedWith>	<diuretics>	<Pathologist1>	p_2
c_3	<hypertension>	<treatedWith>	<diuretics>	<Pathologist2>	p_3
c_4	<hypertension>	<treatedWith>	<b_blockers>	<Pathologist2>	p_4
c_5	<**hypertension**>	<**treatedWith**>	<**diuretics**>	<**YoungDoctor**>	p_5

Table 2: Dataset D_2

The created expression (p_5) is inspired by standard how-provenance expressions [12, 15] used in abstract provenance models, but contains additional information not present in such expressions. In particular, we include, for each attribute of a result quadruple:

- a subscript denoting the quad pattern position in the WHERE clause that the element's value is taken from (arbitrarily we set this to be the first matching position).

- two subscripts in the provenance join operator (${}_{\{\}}\odot_{\{\}}$) to describe the positions of the quad patterns where the joins take place; the first subscript is written to the left of the join operator and refers to the first operand of the join, whereas the second is written to the right and refers to the second operand. This information is important for understanding how c_5 found its way in the dataset (reconstructability).

4 Abstract Provenance Model

Standard *abstract provenance models* are comprised of *abstract identifiers* and *abstract operators* [12, 15]. Abstract identifiers (*quadruple identifiers* in our case, denoted by c_i) are uniquely assigned to RDF quadruples, whereas abstract operators describe the computations performed on quadruples to derive a result quadruple. We additionally introduce the notion of *quad pattern positions*, which are used to describe the position of the occurrence of a constant or a variable in a quad pattern (we provide more details below). Using this infrastructure, RDF quadruples are annotated with complex expressions that involve the identifiers, the operators and the quad pattern positions:

Definition 1. *The provenance p of a quadruple q is defined as $p := \{cpe_1, \ldots, cpe_k\}$. A cpe is a complex provenance expression defined as $cpe := pe^1 \oplus pe^2 \oplus \ldots \oplus pe^m$, where $m \geq 1$, pe^j is a simple provenance expression and $\oplus$ is the provenance operator for union. An expression pe is of the form ($prov_s$, $prov_p$, $prov_o$), where $prov_{pos}$ is the provenance of the attribute pos (described in detail in Definition 2).*

In the above definition, p is the full provenance of the quadruple. Since a quadruple can be the result of more than one INSERT updates applied over the course of time, we use cpe_i to record each such update. As explained in Section 3, each pe^i corresponds to one operand of a UNION operator that leads to the generation of the quadruple, whereas each $prov_{pos}$ describes how the current attribute resulted. Note that $prov_{pos}$ allows the identification of the origin of each element-attribute individually (attribute-level provenance [4]). We are not interested in the provenance of the graph component (the fourth element of a quadruple), as this is explicitly defined by the INSERT update.

Example 1. In our running example (Section 3), $p_5 = \{cpe_1\}$ and $cpe_1 = pe^1 \oplus pe^2$, where $pe^1 = (\bot, \bot, {}_{qp^1_1.o}(c_1))$ and $pe^2 = (\bot, \bot, {}_{qp^2_1.o}(c_2 {}_{\{qp^2_1.o\}} \odot_{\{qp^2_2.o\}} c_3))$; each pe^i results from one operand of the UNION. In pe^1, $prov_s = prov_p = \bot$, whereas $prov_o = {}_{qp^1_1.o}(c_1)$. □

Now let's see how the simple provenance expression pe is constructed. For reasons that will be made apparent later, it is necessary to refer to each individual variable or constant of an update. For this purpose, we arbitrarily number:

a. graph patterns, gp^i ($i \geqslant 1$) indicates the i^{th} graph pattern of the WHERE clause.
b. quad patterns, qp^i_j ($j \geqslant 1$) indicates the j^{th} quad pattern in the graph pattern gp^i.

Moreover, we refer to the quad pattern in the INSERT clause as qp_{ins}.

Using this identification mechanism, each variable or constant in a quad pattern can be uniquely identified by a *quad pattern position*, i.e., $qp^i_j.x$ (or $qp_{ins}.x$), where qp^i_j (qp_{ins}) is the corresponding quad pattern and x is one of $s, p, o,$, to indicate one of the these positions in a quad pattern (e.g., $qp^1_2.o$ denotes the object of the 2^{th} quad pattern of the 1^{st} graph pattern).

Definition 2. *The provenance of attribute pos ($pos \in \{s, p, o\}$), namely $prov_{pos}$, is defined as $prov_{pos}$:= $\perp \mid {}_{varSub}(spe)$, where $\perp$ is a special label, $varSub$ is the var subscript (a quad pattern position) and spe is a standard provenance expression. spe is defined as spe := $(c_{i\ joinSub^1} \odot_{joinSub^2} c_j \cdots {}_{joinSub^{r-1}} \odot_{joinSub^r} c_k)$, where c_x is a quadruple identifier, $joinSub^x$ is a join subscript (quad pattern position IDs) and $\odot$ is the provenance operator of join.*

As proposed in [4, 20], the special label $\perp$ is used to record the case where the INSERT update constructs an element of the new quadruple using a constant, e.g., $prov_s$, $prov_p$ in pe^1 and pe^2 expressions of p_5 in our motivating example. This is the case where the corresponding position in qp_{ins} contains a constant.

If a quad pattern position in qp_{ins} (say $qp_{ins}.pos_1$) contains a variable, then the corresponding value is copied by a quadruple in the dataset, or generated via SPARQL joins. This is recorded using the form ${}_{varSub}(spe)$, where $varSub$ determines the quad pattern position $qp^i_j.pos_2$ that the value should originate from (this position contains the same variable as $qp_{ins}.pos_1$), and spe describes the operation (join or simple "copy") that created it. When there is a copy (in the sense of [4]), spe records the quad pattern position ID from where the value is taken. When there is a join, spe records the joined quadruples, and the positions in said quadruples that were joined (via the left and right subscripts of $\odot$). This is similar to [15], except that [15] does not record the joined positions, which is critical for reconstructability.

Example 2. In our example, $qp_{ins}.s$ and $qp_{ins}.p$ are constants, so the s, p positions of pe^1, pe^2 are set to $\perp$. For the o position, the expression pe^1 contains the var subscript $qp^1_1.o$ because this is the position where the variable *?o* appears in the first operand of the UNION. In this case, the value is taken directly from the corresponding quadruple (c_1), so in pe^1, $prov_o = {}_{qp^1_1.o}(c_1)$. Similarly, for pe^2, the corresponding var subscript is $qp^2_1.o$; note that $qp^2_2.o$ contains the same variable, but we take, by convention, the first valid appearance of said variable. The actual value of the quadruple is generated through a join between the o positions of qp^2_1, qp^2_2, hence the subscripts of the $\odot$ operand; the joined quadruples are c_2 and c_3. Thus, for pe^2, $prov_o = {}_{qp^2_1.o}(c_{2\{qp^2_1.o\}} \odot_{\{qp^2_2.o\}} c_3))$. □

5 Provenance Algorithms

5.1 Provenance Construction Algorithm

The *provenance construction* algorithm (Algorithm 1) is used to record the provenance of quadruples resulting from an INSERT update. This algorithm takes as input an INSERT update U and a dataset D, and returns a provenance expression p_k to associate with each newly created quadruple q_k. Due to space limitations, we will present a simplified version of the algorithm, where the INSERT update generates only one result quadruple; the interested readers can see the full algorithm in [2].

Computing p_k amounts to computing the new cpe_r (resulting from U) to be added; the actual addition happens in line 22 (line 21 determines the corresponding quad q_k). The computation of cpe_r proceeds as follows: the outer FOR (lines 1-20) computes all pe^i (one for each operand of UNION), which are added to cpe_r (line 19), whereas the inner FOR (lines 2-17) computes $prov_s, prov_p, prov_o$ which are composed to form pe^i (line 18). The value of each $prov_{pos}$ is determined by the corresponding $qp_{ins}.pos$: if it is a constant, then $prov_{pos} = \perp$ (line 15); otherwise (if it is a variable), the computation is more complex and is performed in lines 4-13.

Algorithm 1 Provenance Construction Algorithm

```
Input: An INSERT update $U$, a dataset $D$
Output: The provenance $p_k$ of a result quadruple $q_k$, $P$
 1: for all ($gp^i \in$ WHERE clause) do
 2:     for all $qp_{ins}.pos$ do
 3:         if $qp_{ins}.pos \in \mathbb{V}$ then
 4:             Create the set MatchingPatterns $\{mp_1, \ldots mp_z\}$
 5:             $spe$ = FINDIDS($mp_1$)                ▷ "Copy" case
 6:             j = 2
 7:             while $mp_j \neq null$ do                ▷ Join case
 8:                 Create $joinSub^x$ and $joinSub^{x+1}$
 9:                 $spe = spe\ _{joinSub^1} \odot\ _{joinSub^2}$ FINDIDS($mp_j$)
10:                 j++
11:             end while
12:             Create the $varSub$
13:             $prov_{pos} = {}_{varSub}\ (spe)$
14:         else
15:             $prov_{pos} = \perp$
16:         end if
17:     end for
18:     $pe^i = (prov_s, prov_p, prov_o)$
19:     $cpe_r = cpe_r \oplus pe^i$
20: end for
21: $q_k$ = GETQUAD($cpe_r$, $qp_{ins}$)
22: $p_k = p_k \cup cpe_r$
23: return ( $q_k$,$p_k$)
```

In the latter case, we first compute the ordered set *MatchingPatterns* (line 4), which contains all quad pattern identifiers that belong in gp^i and are *related* to the evaluation of the variable in $qp_{ins}.pos$. A quad pattern is related if it contains the specific variable, or if it joins (possibly via another variable) with another related quad pattern.

Then, spe is initially set to be equal to the quad pattern identifier that matches the first item in MatchingPatterns (line 5). If MatchingPatterns has a single item, then we have no joins, i.e., we have a "copy"; lines 7-11 will be skipped, $varSub$ will be computed in line 12, and $prov_{pos}$ in line 13.

If, however, MatchingPatterns has more than one items, then there is one or more joins, which have to be taken into account in the computation of spe. Each join is identified in line 8 (by iterating over the quad patterns and recording the positions where the joins take place by looking at their common variables), line 9 enhances spe with the new join (and the respective quadruple identifier) and the process (lines 7-11) continues until no more MatchingPatterns exist.

It should be noted that there may be more than one quadruple identifiers matching a given quad pattern. In this case, all the different valid combinations are considered by FINDIDS, and each combination results to a different spe and $prov_{pos}$.

Example 3. In our example, $qp_{ins}.s$, $qp_{ins}.p$ are constants, $prov_s = prov_p = \perp$; on the other hand, $qp_{ins}.o = ?o$. For the graph pattern gp^1, we have *MatchingPatterns* $= \{qp_1^1\}$ (which contains the variable *?o*); line 5 will set $spe = c_1$ and line 12 will set $varSub = qp_1^1.o$; the final result (line 18) will be: $pe^1 = (\perp, \perp, {}_{qp_1^1.o}(c_1))$.
For gp^2, the set *MatchingPatterns* $= \{qp_1^2, qp_2^2\}$, indicating that there was a join between qp_1^2, qp_2^2 that created this quadruple. Line 5 sets $spe = c_2$, line 8 identifies the common variable(s) between these two quad patterns ($qp_1^2.o, qp_2^2.o$), and line 9

computes the final $spe = c_{2\{qp_1^2.o\}} \odot_{\{qp_2^2.o\}} c_3$. Note that the evaluation of qp_2^2 also matches c_4, but we ignore it since it does not join with c_2. As before, line 12 will set $varSub = qp_1^2.o$, and the result will be: $pe^2 = (\perp, \perp, {}_{qp_1^2.o}(c_2\ {}_{\{qp_1^2.o\}}\odot_{\{qp_2^2.o\}}\ c_3)$. □

5.2 Update Reconstruction Algorithm

Algorithm 2 exploits the rich semantics of the provenance expression of a quadruple in order to determine how the quadruple found its way in the dataset. It takes as input a complex provenance expression *cpe* that is part of the provenance of the input quadruple q and a dataset D, and returns another INSERT update U'; as we will show below, U' is *compatible* with the original INSERT update that led to the creation of q, i.e., the same in most relevant aspects. The reason why Algorithm 2 takes as input *cpe*, rather than the full provenance, is that each cpe is the result of one INSERT update operation. Before presenting the algorithm, we provide some formal definitions:

Definition 3. *Let gp and gp′ be graph patterns. We say that gp′ is filter-compatible to gp (denoted gp ∼ gp′) iff gp′ differs from gp only in the filters that it may employ.*

Note that Definition 3 refers also to implicit filters created by a constant value in the WHERE clause, e.g., <hypertension> in qp_1^1, qp_1^2, qp_2^2 of our motivating example.

Definition 4. *Let U and U' be* INSERT *updates. We say that U' is compatible to U (denoted $U \rightsquigarrow U'$) if there is a renaming of variables in U', such as $qp_{ins} = qp'_{ins}$ and for each gp' in U' there is a filter-compatible gp in U.*

Intuitively, Definition 4 says that U' is compatible to U iff U contains a subset of the graph patterns in U', modulo filters and variable renaming. As a consequence of Definition 4, the following theorem can be deduced:

Theorem 1. *Let U and U' be UNION-free* INSERT *updates. If U' is compatible to U ($U \rightsquigarrow U'$), then U is also compatible to U' ($U' \rightsquigarrow U$).*[5]

The Algorithm 2 can be split in three parts, each of which computes a different component of the output $U' =$ INSERT $\{qp'_{ins}\}$ WHERE $\{gp'\}$. In particular, lines 1-8 compute qp'_{ins}; lines 9-34 compute gp' and line 35 combines the above to form U'.

For the first part, the graph position (n) of qp'_{ins} is determined by the graph attribute of the input q (line 1). For the s, p, o positions, we exploit the fact that, if $prov_{pos}$ of pe^1 is equal to $\perp$, then the corresponding quadruple attribute was created by a constant, so we set $qp'_{ins}.pos = q.pos$ (note also that in this case, the $prov_{pos}$ of all pe^i will be equal to $\perp$); otherwise, $qp'_{ins}.pos$ is associated with a new variable.

The main part of the algorithm (lines 9-34) contains one FOR loop which computes the graph patterns (gp'^i), each corresponding to one pe^i in cpe; each loop computes all the quad patterns qp'^i_j of gp^i, composes them using join in line 32 (to form gp'^i), and uses the result to progressively built the final graph pattern gp' (line 33).

[5] Proofs for all theorems can be found in [2].

Algorithm 2 Update Reconstruction Algorithm

```
Input: A $cpe$ expression of the form $pe^1 \oplus \ldots \oplus pe^k$, a
    quadruple $q$ $(s, p, o, n)$, a dataset $D$
Output: An INSERT update $U'$, such that $U' \rightsquigarrow U$
 1: $qp'_{ins} = (qp'_{ins}.s, qp'_{ins}.p, qp'_{ins}.o, n)$
 2: for all $pos \in \{s, p, o\}$ do
 3:     if $prov_{pos}$ of $pe^1$ is equal to $\bot$ then
 4:         $qp'_{ins}.pos$ = $q.pos$
 5:     else
 6:         $qp'_{ins}.pos = $ NEWVAR( )
 7:     end if
 8: end for
 9: for all $pe^i$ in $cpe$ do
10:     $l =$ COMPUTEGPSIZE($pe^i$)
11:     ASSIGNGRAPHS($pe^i$)
12:     for all $prov_{pos}$ in $pe^i$, such that $prov_{pos} \neq \bot$ do
13:         $(j, pos')$ = GETPOSFROMVARSUB($prov_{pos}$)
14:         $qp^i_j.pos' = qp'_{ins}.pos$
15:     end for
16:     for all $prov_{pos}$ in $pe^i$, such that $prov_{pos} \neq \bot$ do
17:         Set $AllJoinSubs$ = $\langle joinSub^1$, ...,
    $joinSub^x\rangle$, where $joinSub^r = \langle jp^r_1, \ldots, jp^r_k\rangle$
18:         $r = 1$
19:         while $joinSub^r \neq \emptyset$ do
20:             for all $jp^r_k \in joinSub^r$ do
21:                 if $jp^r_k =$ null then
22:                     $jp^r_k =$ NEWVAR( )
23:                 end if
24:                 $jp^{r+1}_k = jp^r_k$
25:             end for
26:             $r = r + 2$
27:         end while
28:     end for
29:     for all unbound $qp'^i_j.pos$ do
30:         $qp'^i_j.pos =$ NEWVAR( )
31:     end for
32:     $gp'^i$ = $qp'^i_1$ . $qp'^i_2$ . . . . . $qp'^i_l$
33:     $gp'$ = $gp'$ UNION $gp'^i$
34: end for
35: return $U' =$ INSERT $\{qp'_{ins}\}$ WHERE $\{gp'\}$
```

To construct gp'^i, we progressively fill the positions of each quad pattern in gp'^i with variables, taking special care to use the same variables in positions that are joined, and also to reuse the variables already in qp'_{ins} when appropriate.

Initially, we compute the size of gp^i, i.e., the number of quad patterns in gp^i (line 10), by scanning all quad pattern identifiers found in the var or join subscripts of pe^i.

Line 11 deals with the fourth attribute of quad patterns, which does not accept variables, so its value is taken directly by the fourth attribute of the corresponding quadruple. Finding the corresponding quadruple is easy: for a quad pattern appearing in a var subscript, its corresponding quadruple is the first that appears in the respective *spe*, whereas for join subscripts we take the quadruple in the respective “side” of the join.

The most important task is done in lines 12-31, where the s, p, o positions of quad patterns are filled. Lines 12-15 are the starting point: we “read” the $varSub$ of each $prov_{pos} \neq \bot$, in order to identify where each position in qp'_{ins} took its value from. Line 13 finds j (i.e., the proper quad pattern qp'^i_j in gp^i) and the position in said quad pattern (pos'), whereas line 14 fills this position with the variable found in $qp'_{ins}.pos$.

Lines 16-28 essentially “follow the chain of joins” that is recorded in the join subscripts, so as to assign common variable names where appropriate, reusing the variables in qp'_{ins}, or introducing new ones. Recall that each join contains two join subscripts; the number of quad pattern positions in each subscript of the pair depends on the number of positions in which the join is applied. In our algorithm, $AllJoinSubs$ is an ordered list of all such join subscripts (easily found by scanning $prov_{pos}$); by construction,

$joinSub^1$ and $joinSub^2$ appear in the same join (same for $joinSub^3$ and $joinSub^4$ and so on). Each $joinSub^r$ is a sequence of quad patterns $\langle jp_1^r, \ldots, jp_k^r \rangle$.

If $AllJoinSubs$ is empty, then we have a "copy"; lines 19-27 will be skipped, and the only variable assignment necessary is the one already performed in line 14. In the more complex case where $AllJoinSubs$ contains some elements, these are processed in pairs, as indicated by the WHILE in line 19 and the increment in line 26. The idea is to put the same variable in positions that are joined, i.e., the same variable in jp_k^r, jp_k^{r+1} for all pairs $r, r+1$ (for r an odd number). If jp_k^r has already an assigned variable, this was created either by line 14, or by a previous execution of line 24, so this value is copied in jp_k^{r+1}; if not, a "fresh" variable is assigned to jp_k^r (line 22) and the process continues normally. The assumption that the quad pattern position appearing in the var subscript is the first one that matches is critical for this process, because it guarantees that we will not assign a fresh variable when the variables should be taken from qp'_{ins}.

Any *unbound* quad pattern positions (i.e., positions with no assigned variables) remaining after the execution of lines 16-28, are filled with "fresh" variables (line 30).

Example 4. Now, we will explain how Algorithm 2 works for our motivating example. We first determine the graph attribute of qp'_{ins} (<YoungDoctor>), taken from c_5 (line 1). Then (lines 2-8), we note that the s, p values of c_5 resulted from a constant (see pe^1, pe^2), whereas the o value resulted from a "copy" or join; thus, qp'_{ins} = (<hypertension>, <treatedWith>, *?v0*, <YoungDoctor>).
Subsequently, the FOR loop in line 9 is called for each pe^i. For pe^1, only qp_1^1 appears, whose named graph is the one of c_1, i.e., <Diabetologist>. The o position of qp'^1_1 is taken from $qp'_{ins}.o$, as indicated by the var subscript (so $qp'^1_1.o$ = *?v0*. There are no joins, so the block in lines 16-28 has no effect, and fresh variables are assigned in the other positions in line 30. Thus, $qp'^1_1 =$ (*?v1*, *?v2*, *?v0*, <Diabetologist>).
Similarly, for pe^2, we have two quad pattern identifiers, qp_1^2, qp_2^2, whose named graph attributes are taken from c_2, c_3 respectively (lines 10-11). The value of $qp'^1_1.o$ is set to *?v0* (equal to $qp'_{ins}.o$, as indicated by the var subscript of pe^2). In this case, we have a join, so in line 24 we will copy the value of $qp'^1_1.o$ (i.e., *?v0*) to $qp'^1_2.o$; this is due to the form of the join $({}_{\{qp_1^1.o\}} \odot_{\{qp_2^1.o\}}$, which will set $AllJoinSubs = \langle\langle qp_1^1.o\rangle, \langle qp_2^1.o\rangle\rangle)$.
There are no further joins to process, so we put fresh variables in the unbound positions of qp'^1_2, qp'^2_2, resulting into: $qp'^1_2 =$ (*?v3*, *?v4*, *?v0*, <Pathologist1>), $qp'^1_2 =$ (*?v5*, *?v6*, *?v0*, <Pathologist2>).
After the composition of the above quad patterns in lines 32, 33, 35 we get U':

INSERT $\{qp'_{ins}\}$ WHERE $\{qp'^1_1$ UNION $qp'^2_1.\ qp'^2_2\}$

where: qp'_{ins}: (<hypertension>, <treatedWith>, *?v0*, <YoungDoctor>)
qp'^1_1: (*?v1*, *?v2*, *?v0*, <Diabetologist>)
qp'^2_1: (*?v3*, *?v4*, *?v0*, <Pathologist1>)
qp'^2_2: (*?v5*, *?v6*, *?v0*, <Pathologist2>)

Note that U' differs from U only in the (implicit) filters that U employs (<hypertension>, <treatedWith>) in its quad patterns, as well as in the variable names. □

The following theorem proves the correctness of our algorithms:

Theorem 2. *Let U be an* INSERT *update evaluated on a dataset D, q a result quadruple and cpe a complex provenance expression in the provenance of q as computed by*

Algorithm 1. Assume that we run Algorithm 2 with input (cpe, q, D) and we get as output the INSERT *update U'. Then, U' returns q among other quadruples and $U \rightsquigarrow U'$.*

Theorem 2 proves that the output of Algorithm 2 is compatible with the original INSERT update that created the input quadruple; thus, the intended semantics of a provenance expression, as given in Section 4, are correctly recorded by Algorithm 1, and interpreted by Algorithm 2 to reconstruct the original INSERT update.

5.3 Complexities

The time complexity of Algorithm 1 is *linear* with respect to the update size (number of quad patterns in the WHERE clause). To see this, note that lines 2-17 will be executed three times, each run costing $O(m_i)$, where m_i is the number of joined quad patterns in gp^i; thus, the total cost is $O(3 \cdot \sum_i m_i) = O(m)$, where m is the update size. Algorithm 1 is also of *logarithmic* complexity with respect to the dataset size (number of quadruples), say R. Specifically, the dataset is accessed in two occasions: to find the quadruple identifiers (lines 5, 9), and to get the attribute values of q_k (line 21). Each access costs $O(\log R)$ time (assuming appropriate indexes), and happens a constant number of times (assuming a constant update size), so the total cost is $O(\log R)$. The above complexities are related to the cost of annotating the result of the INSERT update with its provenance, and do not include the cost of computing the result itself; an obvious conclusion is that the overhead imposed by the provenance algorithm is negligible.

The time complexity of Algorithm 2 is *linear* with respect to the *size of the cpe*. Lines 9-34 run once for each pe^i, each run costing $O(m_i)$ time (where m_i is the number of quad patterns in pe^i), because each part of provenance is accessed a constant number of times. Hence, the complexity is $O(\sum_i m_i) = O(m)$, where m is the total number of quad patterns in the WHERE clause (i.e., update size). As with Algorithm 1, Algorithm 2 only accesses the dataset in specific points (lines 4, 11), each being run a constant number of times (for a fixed update size) and costing $O(\log R)$ (assuming adequate indexes) over a dataset of size R. Thus, the complexity of Algorithm 2 is *logarithmic* with respect to the number of quadruples in D.

Regarding space complexity, we note that the size of provenance is analogous to the size of the input U (and vice-versa), and that all temporarily stored information is no larger in size than the size of the update/provenance (respectively) in either algorithm. Thus, the space complexity of both algorithms is *linear* with respect to U/cpe.

6 Implementation and Evaluation

6.1 Implementation and Storage (Relational Schema)

Existing SPARQL engines do not support the kind of complex provenance information proposed by our model. Thus, we used Virtuoso Database Engine as our triple store, where quadruples and provenance expressions are stored in relational tables. On top of Virtuoso we built a main memory Java implementation of our algorithms. The quadruples and the related provenance expressions are stored in a relational schema, which uses two tables: *Quads(qid,s,p,o,n)* and *Prov(qid, cpeNo, peNo, prov$_s$, prov$_p$, prov$_o$)*. Table *Quads* stores the quadruple's (ID, subject, property, object, named graph). Table

Prov stores the provenance information of a quadruple: *qid* is the quadruple ID, *cpeNo* and *peNo* are the IDs of *cpe* and *pe* expressions, while $prov_s$, $prov_p$ and $prov_o$ contain the provenance of the corresponding attribute related to the specific *cpe* and *pe*.

6.2 Experiments

In our experiments we used real data that were taken from the Billion Triple Challenge (BTC) dataset (small crawl)[6]. The BTC dataset contains 10 million quadruples, but we used smaller excerpts containing 100, 250 and 500 thousand unique quadruples. Due to the absence of a standard benchmark for provenance, we used our own custom synthetic set of INSERT updates[7]. All experiments were conducted on a Dell OptiPlex 755 desktop with CPU Intel® Core™ 2 Duo CPU Q6600 at 2.40GHz, 6 GB of memory, running Windows 7 Professional x86_64.

We conducted three experiments. EXPERIMENT 1 measures the time required to compute the results of an INSERT update along with their provenance information, whereas EXPERIMENT 2 considers the time required to compute only the result quadruples. The difference in time of EXPERIMENT 1 and EXPERIMENT 2 indicates the overhead for computing the provenance. EXPERIMENT 3 computes the time needed for reconstructing a compatible INSERT update based on a quadruple's provenance.

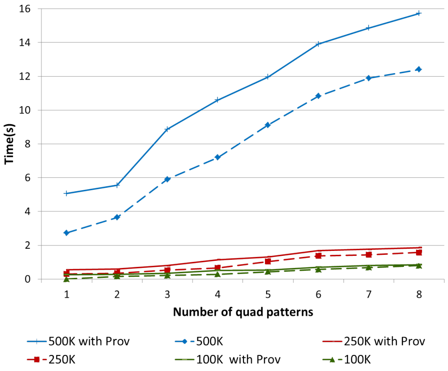

Fig. 1: EXPERIMENT 1, 2

Our experiments confirm the theoretical complexity results above. In particular, the evaluation time depends on the number of quad patterns, and is also affected by the number of quadruples in the dataset, the number of quad patterns in the WHERE clause (i.e., update size), and, of course, the applied SPARQL operators (join, union).

Figure 1 shows the computation time for executing the INSERT update with and without the provenance computation. The graph shows that the provenance computation time increases linearly with the number of quad patterns, and that it is, in all cases, only a fraction of the time required for evaluating the INSERT update. Moreover, note that the dataset's size has a great impact on the evaluation time of both experiments.

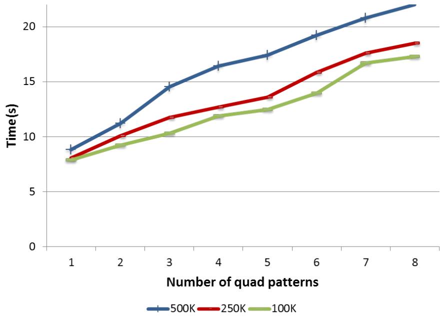

Fig. 2: EXPERIMENT 3

Figure 2 shows how the performance of Algorithm 2 scales as the complexity

[6] https://km.aifb.kit.edu/projects/btc-2009/

[7] http://www.ics.forth.gr/isl/provenance/updates.pdf

of the INSERT update increases, for the considered datasets. We note that the evaluation time increases linearly with respect to the number of quad patterns in the WHERE clause, and that performance is not seriously affected by the dataset size.

7 Related Work

Data provenance has been widely studied in several different contexts such as databases, distributed systems, Semantic Web etc. In [16], Moreau explores the different aspects of provenance in the Web. Likewise, Cheney et al. [8] provide an extended survey that considers the provenance of query results in relational databases regarding the most popular provenance models.

Research on data provenance can be categorized depending on whether it deals with *updates* [3, 4, 10, 13, 20] or *queries* [4, 8–12, 15, 20, 21]; compared to querying, the problem of provenance management for updates is less well-understood.

Another important classification is based on the underlying data model, SQL [5, 12, 20] or RDF [9–11, 13, 15, 21], which determines whether the model deals with the relational or SPARQL algebra operators respectively. Despite its importance, only a few works deal with the problem of update provenance, and even fewer consider the problem in the context of SPARQL updates [13].

A third categorization stems from the expressive power of the employed provenance model, e.g., *how*, *where* and *why* among others. Since our proposed model is based on how and where provenance models, we discuss them thoroughly here. *Where provenance* is a popular provenance model [3, 5, 10, 20] that describes where a piece of data is copied from, i.e., which quadruples contributed to produce a result quadruple in our context. *How provenance* describes not only the quadruples used for producing an output, but also how these source quadruples were combined (through operators) to derive it. In [12], *provenance semirings* are used to record *how provenance* for the relational setting through polynomials; whereas [11, 15] showed how to apply provenance semirings for the RDF/SPARQL setting.

An important work on update provenance for the relational setting is [4], which focuses on the *copy* and *modify* operations. The proposed formalization is based on "tagging" tuples using "colors" propagated along with their data item during the computation of the output. The provenance of the output is the provenance propagated from the input item(s). Our model follows this approach to capture the provenance of a quadruple attribute, but uses identifiers instead of colors, as well as a more expressive provenance model.

In the context of SPARQL update provenance, there are no works that consider abstract provenance models. Instead, RDF named graphs are used to represent both past versions and changes to a graph [13]. This is achieved by modelling the provenance of an RDF graph as a set of history records, including a special provenance graph and additional auxiliary versioning named graphs.

Moreover, our work builds on [15]. This work presents how popular relational data provenance models such as (*how*, *why*) can be adapted to capture the provenance of the results of positive SPARQL queries (i.e., without SPARQL OPTIONAL clauses). More specifically, the authors investigate how provenance models for the positive fragment of the relational algebra (like [12]) can be adapted for unions of conjunctive SPARQL queries. The present paper extends this model in order to address the extra challenges associated with provenance management of SPARQL updates (as opposed to queries).

Another major line of work deals with the different ways in which provenance can be serialized and modelled in an ontology in the form of Linked Data [14, 17, 18]. In [14], Hartig proposes a provenance model that captures information about Web-based data access as well as information about the creation of data. Moreau et al. created the Open Provenance Model [17] that supports the digital representation of provenance for any "thing", no matter how it was produced. In this context, PROV was released as a W3C reccomendation [18]. The goal of PROV is to enable the wide publication and interchange of provenance on the Web and other information systems. PROV can exhibit provenance information using widely available formats such as RDF and XML.

8 Conclusions

As the volume of data made available in the Web is continuously increasing, the need for capturing and managing the provenance of such data becomes all the more important. Our work addresses this problem for RDF data, by proposing a novel, fine-grained and expressive provenance model to record the triple and attribute-level provenance of RDF quadruples generated through SPARQL INSERT updates.

Our work follows the approach of [10, 15], where the use of abstract identifiers and operators is proposed; we build upon the novel notion of *quad pattern positions* in order to provide a richer set of operators, that allow the identification of the attributes of quad patterns that were involved in a join or "copy" operation. Our model is richer than standard query provenance models since it captures fine-grained provenance both at triple and attribute level.

Our model supports the feature of update *reconstructability*. Reconstructability prescribes that the information stored in the provenance of a quadruple allows the identification of an INSERT update that is almost identical (in the sense of *compatibility*) to the original one that was used to create said quadruple. This is a stronger form of *how provenance*. On the algorithmic side, we introduce two algorithms that allow recording the provenance information, as well as interpreting it to identify how the quadruple found its way in the dataset, through the identification of a compatible INSERT update as described above. The overhead imposed by these algorithms in the execution of an INSERT update is negligible. We implemented the *provenance construction* and the *update reconstruction* algorithms on top of Virtuoso Database Engine and conducted a preliminary set of experiments that verified the complexity of the proposed algorithms.

In the future, we plan to consider FILTER and non-monotonic SPARQL operators (OPTIONAL) as well as SPARQL functions. In addition, we will study the SPARQL DELETE, CREATE and DROP operations since all SPARQL update operations can be written as a combination of INSERT, DELETE, CREATE and DROP statements. Furthermore, we plan to take under consideration benchmarks supporting update operations and will try to extend them in order to compute the provenance information using our model.

We also intend to explore the use of PROV approach for representing our model in the form of Linked Data. As a long term plan we aim at working towards a provenance aware triple store in the spirit of TripleProv [21].

References

1. M. Arenas, C. Gutierrez, and J. Perez. On the Semantics of SPARQL. In *Semantic Web Information Management*, pages 281–307. 2009.
2. A. Avgoustaki. Provenance management for SPARQL updates. Master's thesis, University of Crete, 2014. Available at `http://www.ics.forth.gr/isl/provenance/provenance.pdf`.
3. P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *ACM SIGMOD International Conference on Management of Data*, 2006.
4. P. Buneman, J. Cheney, and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. In *International Conference on Database Theory*, 2007.
5. P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *International Conference on Database Theory*, 2001.
6. J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named Graphs. *Journal of Web Semantics*, 3(4):247–267, 2005.
7. J. J. Carroll, C. Bizer, P. J. Hayes, and P. Stickler. Named graphs, provenance and trust. In *International Conference on World Wide Web*, 2005.
8. J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
9. C. V. Damasio, A. Analyti, and G. Antoniou. Provenance for SPARQL Queries. In *International Semantic Web Conference*, 2012.
10. G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides. Coloring RDF triples to capture provenance. In *International Semantic Web Conference*, 2009.
11. F. Geerts, G. Karvounarakis, V. Christophides, and I. Fundulaki. Algebraic Structures for Capturing the Provenance of SPARQL Queries. In *International Conference on Database Theory*, pages 153–164, 2013.
12. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Principles Of Database Systems*, pages 31–40, 2007.
13. H. Halpin and J. Cheney. Dynamic provenance for SPARQL updates. In *International Semantic Web Conference*, 2014.
14. O. Hartig. Provenance information in the Web of Data. In *Proceedings of the 2nd Linked Data on the Web Workshop at the World Wide Web Conference*, 2009.
15. G. Karvounarakis, I. Fundulaki, and V. Christophides. Provenance for Linked Data. In *Search of Elegance in the Theory and Practice of Computation*. 2013.
16. L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2-3):99–241, 2010.
17. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, and J. V. den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computing Systems*, 27(6):743–756, 2011.
18. L. Moreau and P. Missier. PROV-DM: the PROV data model. W3C Recommendation, 2013.
19. J. Perez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *ISWC*, pages 30–43, 2006.
20. S. Vansummeren and J. Cheney. Recording Provenance for SQL Queries and Updates. *IEEE Data Engineering Bulletin*, 30(4):29–37, 2007.
21. M. Wylot, P. Cudré-Mauroux, and P. T. Groth. Tripleprov: efficient processing of lineage queries in a native RDF store. In *International Conference on World Wide Web*, pages 455–466, 2014.