Provenance Management for SPARQL Updates

Argyro Avgoustaki

Thesis submitted in partial fulfilment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes Campus, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

University of Crete
Computer Science Department

Provenance Management for SPARQL Updates

Thesis submitted by
Argyro Avgoustaki
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Argyro Avgoustaki

Committee approvals: ______________________
Dimitris Plexousakis
Professor, Thesis Supervisor

____
Yannis Tzitzikas
Assistant Professor, Committee Member

____
Irini Fundulaki
Principal Researcher, Committee Member

Departmental approval: ______________________
Antonis A. Argyros
Professor, Director of Graduate Studies

Heraklion, December 2014

Abstract

During the last few years we have witnessed an explosion in the publication of data in the Web, mainly in the form of Linked Data. Scientific, corporate or even governmental data are made available for open access and used by applications, individual users and communities. Given the increasing amount and the heterogeneity of this data, it is of crucial importance to be able to track its provenance. Recording the provenance can help us to effectively support trustworthiness, accountability and repeatability in the Web of Data.

A number of models have already been proposed to capture the provenance information of query results; most of them considering RDF or relational data. On the contrary, despite its importance, little research has been conducted in the case of updates and especially of SPARQL updates.

In this thesis, we propose a new provenance model that borrows from both *how* and *where* data provenance models, and is suitable for capturing the triple and attribute level provenance of SPARQL update results. To the best of our knowledge, this is the first model that deals with the provenance of SPARQL updates using *algebraic provenance expressions*, in the spirit of the well-established model of provenance semirings.

On the algorithmic side, we introduce an algorithm that records the provenance of SPARQL update results in terms of the proposed model and a reconstruction algorithm that uses the provenance of a quadruple to identify a SPARQL update that is provably *compatible* to the original one. A SPARQL update is *compatible* to another if they differ only in the variables names that they employ and the first update contains a genuine subset of the unions that appear in the second one. The latter algorithm is a necessary complement in order to fully describe the provenance management, as it shows the determinant role of provenance information in the persistence of SPARQL update results.

Περίληψη

Τα τελευταία χρόνια παρατηρείται μια έκρηξη στη δημοσίευση δεδομένων στον Παγκόσμιο Ιστό, κυρίως με τη μορφή *Συνδεδεμένων Δεδομένων (Linked Data)*. Δεδομένα από διάφορες θεματικές περιοχές, π.χ. επιστημονικά, εταιρικά, κυβερνητικά κτλ., διατίθενται για ανοιχτή πρόσβαση και χρήση από εφαρμογές, μεμονωμένους χρήστες ή ακόμα και κοινότητες χρηστών. Δεδομένου του αυξανόμενου όγκου και της ετερογένειας των δεδομένων αυτών κρίνεται επιτακτική η ανάγκη για καταγραφή της *πληροφορίας προέλευσης (provenance)*. Η γνώση της προέλευσης μάς δίνει τη δυνατότητα να υποστηρίξουμε αποτελεσματικά εφαρμογές που σχετίζονται με την αξιοπιστία, την φερεγγυότητα και την επαναληπτικότητα των δεδομένων.

Ένα πλήθος από μοντέλα έχει ήδη προταθεί για την καταγραφή της πληροφορίας προέλευσης των αποτελεσμάτων μιας επερώτησης (query)· τα περισσότερα από τα οποία αφορούν *RDF* ή *σχεσιακά (relational)* δεδομένα. Αντίθετα, και παρά τη σπουδαιότητα του προβλήματος, η έρευνα για την περίπτωση των *ενημερώσεων (updates)*, και ειδικότερα των *SPARQL ενημερώσεων*, βρίσκεται ακόμα σε πρώιμο στάδιο.

Στην εργασία αυτή, προτείνουμε ένα νέο μοντέλο για την καταγραφή και διαχείριση της πληροφορίας προέλευσης, σε επίπεδο *τριπλέτας (triple)* και *γνωρίσματος (attribute)*, των αποτελεσμάτων των SPARQL updates. Το μοντέλο αυτό, το οποίο δανείζεται χαρακτηριστικά και ιδιότητες από τα ήδη υπάρχοντα μοντέλα του *where* και *how* είναι το πρώτο που υποστηρίζει τη χρήση *αλγεβρικών εκφράσεων* σε ενημερώσεις, ακολουθώντας την προσέγγιση του μοντέλου των *provenance semirings*.

Από αλγοριθμικής σκοπιάς, παρουσιάζουμε έναν αλγόριθμο, ο οποίος υπολογίζει την πληροφορία προέλευσης για τα αποτελέσματα των SPARQL updates με βάση το προτεινόμενο μοντέλο, καθώς και έναν αλγόριθμο ανακατασκευής (reconstruction), ο οποίος χρησιμοποιεί την πληροφορία προέλευσης μιας τετραπλέτας (quadruple) για να δημιουργήσει ένα SPARQL update, αποδεδειγμένα, *συμβατό (compatible)* με το αρχικό. Ένα SPARQL update είναι συμβατό με ένα άλλο, αν διαφέρουν μόνο στα ονόματα των μεταβλητών που χρησιμοποιούν, και το πρώτο update περιέχει ένα γνήσιο υποσύνολο των ενώσεων (unions) που εμφανίζονται στο δεύτερο. Η παροχή ενός αλγορίθμου ανακατασκευής κρίνεται απαραίτητη ώστε να μπορέσουμε να περιγράψουμε πλήρως τη διαχείριση της πληροφορίας προέλευσης, καθώς φανερώνει τον καθοριστικό ρόλο της πληροφορίας αυτής στη διατήρηση της *συνεκτικότητας (persistence)* των αποτελεσμάτων των SPARQL updates.

Στους γονείς μου,
Δημήτρη και Ελένη

Ευχαριστίες

Υπάρχουν τόσα πολλά άτομα που θα ήθελα να ευχαριστήσω, καθέναν για έναν ξεχωριστό λόγο. Αρχικά, θα ήθελα να ευχαριστήσω θερμά τον επόπτη μου, Καθηγητή κ. Δημήτρη Πλεξουσάκη, για την εμπιστοσύνη που μου έδειξε καθώς και για τη στήριξη του καθ' όλη τη διάρκεια των μεταπτυχιακών μου σπουδών.

Επίσης, θα ήθελα να ευχαριστήσω εκ βαθέων τους συνεπιβλέποντες της εργασίας μου, Γιώργο Φλουρή και Ειρήνη Φουντουλάκη, για την καθοδήγηση, τον ενθουσιασμό, τις πολύτιμες συμβουλές καθώς και την υπομονή τους. Οι γνώσεις, η εμπειρία και οι ιδέες τους συνέβαλαν καθοριστικά στην ολοκλήρωση της εργασίας αυτής. Η συνεργασία μας με βοήθησε να εξελιχθώ τόσο σε επαγγελματικό αλλά και προσωπικό επίπεδο, δίνοντας μου ταυτόχρονα τα απαραίτητα εφόδια για τη συνέχεια των σπουδών μου.

Στο σημείο αυτό, θα ήθελα να ευχαριστήσω όλα τα μέλη του εργαστηρίου Πληροφοριακών Συστημάτων για την ευχάριστη συνεργασία. Ιδιαίτερα, ωστόσο, ευχαριστώ τους Γιάννη Ρ., Παναγιώτη και Χριστίνα γιατί εκτός από καλοί συνεργάτες υπήρξαν και καλοί φίλοι. Τα "coffee breaks" μας θα μείνουν στην ιστορία...

Ακόμα, ευχαριστώ τους καλούς μου φίλους Βαλεντίνα, Βιόλα, Κάλλια, Ηρακλή, Νίνα, Γιώργο και Νίκο. Είτε κοντά, είτε μακριά, άλλοι πιο πολύ, άλλοι πιο λίγο έκαναν όλα αυτά τα χρόνια να αξίζουν και μου χάρισαν υπέροχες αναμνήσεις. Κυρίως, όμως, μου πρόσφεραν τη χαρά να έχω δίπλα μου ξεχωριστούς ανθρώπους.

Θα ήθελα να αναφερθώ ιδιαιτέρως στην πολύ καλή μου φίλη Δήμητρα και να την ευχαριστήσω, εκτός των άλλων, για τις εποικοδομητικές συζητήσεις μας αλλά και τις γεμάτες αγάπη και ειλικρίνεια συμβουλές της. Η ωριμότητα της με βοήθησε πολλές φορές να δω από άλλη οπτική γωνία τα γεγονότα.

Επιπλέον, θα ήθελα να ευχαριστήσω από καρδιάς τον αδερφικό μου φίλο Μάνο, για τη συνεχή και ανιδιοτελή αγάπη, υποστήριξη και συμπαράσταση που μου παρέχει από την πρώτη μέρα γνωριμίας μας. Η σχέση μας με έκανε να πιστέψω αυτό που λένε "οι φίλοι είναι η οικογένεια που επιλέγουμε" κι εσύ είσαι ο αδερφός που δεν είχα Στα εύκολα και στα δύσκολα πάντα μαζί...

Το μεγαλύτερο όμως ευχαριστώ ανήκει στην οικογένεια μου και ιδιαίτερα στους γονείς μου, Δημήτρη και Ελένη, που με υπέρμετρη αγάπη, κατανόηση κι υπομονή στηρίζουν πάντα κάθε μου προσπάθεια. Οι αρχές που με δίδαξαν και η διαπαιδαγώγηση που έλαβα με βοήθησαν να χαράξω τη δική μου πορεία στη ζωή. Είμαι τυχερή που σας έχω...

Σας ευχαριστώ πολύ όλους!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last few years, we have witnessed an explosion in the volume of data published in the Web, mainly in the form of Linked Data [1]. The main value of such data stems from the unmoderated nature of data publication, interlinking and reuse. This increases the added-value of interlinked data by identifying unknown correlations and relationships, and by allowing the re-use of concepts and properties.

Data on the web are usually published using the RDF [2] data model. The popularity of the RDF data model is due to the flexible and extensible representation of information under the form of *triples*, organized in *named graphs* [3], thereby forming *quadruples*. An RDF triple (*subject*, *predicate*, *object*) asserts the fact that *subject* is associated with *object* through *predicate*. Querying and updating RDF data is performed using the SPARQL language [4, 5].

The open and unconstrained nature of data published in the Web, makes it imperative to effectively support, e.g., *trustworthiness*, *accountability* and *repeatability*. This is achieved by recording the *provenance* of published data, i.e., their *origin* or *source*, that describes from *where* and *how* the data was obtained [6].

In this work we deal with the problem of *capturing and managing the provenance of quadruples constructed through* SPARQL *updates* [5]. 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. forming quadruples). The purpose of 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.

Even though the problem of provenance has been extensively studied in the literature [6, 7, 8, 9, 10, 11] most of the related works deal with SPARQL query provenance. An approach for recording provenance is via algebraic expressions that describe the origin of data in varying levels of detail [7, 12, 13, 14]; in the RDF context, provenance is recorded via named graphs [3, 9, 14, 15]. Unfortunately, the unique requirements associated with the provenance of SPARQL updates results do not allow a direct reuse of such approaches.

A first problem stems from the fact that the named graph component of a quadruple is defined by the user in the INSERT update. This implies that provenance should be defined for quadruples, rather than triples (as is the case in most works). Furthermore, the same fact implies that 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 in our setting.

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 named graphs approach are needed.

Another challenge stems from the persistence of a SPARQL update result. This implies that when a quadruple is accessed, the SPARQL update that generated may be no longer available. This requirement leads to the notion of *reconstructability*, which refers to the ability of using the provenance expression for *reconstructing* an INSERT update that is *compatible* (Definition 15) with the original INSERT update that generated the quadruple.

Therefore, the provenance of a quadruple should be expressive enough to identify the quadruples that contributed to its creation (*where provenance* [16]), as well as how these quadruples were used to generate the new one (*how provenance* [7]). However, *how provenance* in this setting takes a much more demanding form than in the case of query provenance. As an example, knowing that a join was used to generate a quadruple during a query is enough to understand how it was generated; on the other hand, in the case of INSERT updates, we need to know more fine-grained information, and more specifically which components of a quad pattern were joined to generate the result.

To support the above requirements we introduce a novel *triple* and *attribute level, fine-grained provenance model* that borrows from both *where* and *how* data provenance models [7, 17], as well as algorithms for managing (*recording* and *interpreting*) provenance information. More specifically, the main contributions of this thesis are:

- The introduction of an *expressive provenance model* suitable for encoding triple and attribute level provenance of quadruples obtained via SPARQL INSERT updates, and allowing the *reconstructability* of such updates from their provenance.
- The provision of algorithmic support for our model via the *Provenance Construction* and the *Update Reconstruction* algorithms. The former is used for computing and recording the provenance of the result of a SPARQL INSERT update based on the proposed model. The latter exploits the expressiveness of our model to report on the generation process of a quadruple (using its provenance), in the sense of reconstructing a SPARQL INSERT update that is *compatible* with the original one that created said quadruple.

Structure. In Chapter 2, we briefly discuss basic concepts and definitions of RDF (Section 2.1) and SPARQL (Section 2.2), as well as the most prevalent positive provenance models (Section 2.3). A motivating example that will be used throughout this thesis is provided in the Chapter 3. Chapter 4 describes the semantics of SPARQL Update language. We define our provenance model in Chapter 5. Chapter 6 presents the related algorithms (Sections 6.1, 6.2), their correctness results (Section 6.3), as well as their complexity analysis (Section 6.4). Finally, in Chapter 7 we describe the related work and we conclude in Chapter 8.

Chapter 2

Preliminaries

In this chapter we discuss the Resource Description Framework (RDF) [2], a *data model* used for describing and modelling information that is implemented in Web resources. Additionally, we present SPARQL [4, 5], the official W3C recommendation language for querying and updating data in RDF format. At the end of this chapter we refer to some of the most prevalent positive provenance models that our work builds on.

2.1 RDF

The Resource Description Framework (RDF) [2], a W3C recommendation, is a model for representing information about resources in the World Wide Web (Web resources). RDF enables the encoding, exchange and reuse of structured data, providing therefore the means for publishing both human-readable and machine-processable vocabularies. Nowadays, it is used in a variety of application areas, such as the Linked Data initiative [1], which aims at connecting data sources on the Web, and is employed as a standard for representing information on the Web of Data.

RDF is based on a simple data model that facilitates Web data processing and manipulation. The fundamental idea of RDF model is that everything we wish to describe is a *resource*. A resource may be a title, an author, the modification date of a Web document or even a relation between them, and is identified by using Web identifiers, called Internationalized Resource Identifiers or IRIs (denoted by

$< >$). The building block of the RDF data model is a *triple*.

Assume two pairwise disjoint and infinite sets $\mathbb{I}$ and $\mathbb{L}$, denoting *IRIs* and *literals*, respectively.

Definition 1. *An RDF triple t is a tuple of the form (subject, predicate, object). The set $\mathcal{T} = \mathbb{I} \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{L})$ is the set of all RDF triples.*

An RDF triple asserts the fact that *subject* is associated with *object* through *predicate*. It should be stressed that in this work, we are interested only in ground triples and thus we do not consider blank nodes.

Example 1. For example (<hypertension>, <medication>, <diuretics>) is an RDF triple, with <hypertension> being its subject, <medication> being its predicate and <diuretics> being its object. □

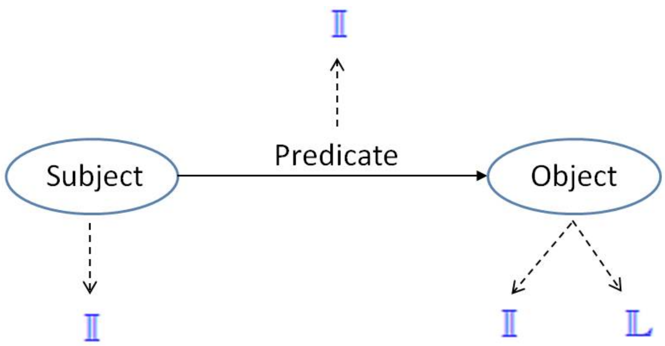

Figure 2.1: Graphical representation of an RDF triple

Definition 2. *An RDF graph $\mathcal{G}$ is a set of RDF triples, $\mathcal{G} \subseteq \mathcal{T}$. An RDF named graph $\mathcal{NG}$ is an RDF graph that is uniquely identified by an IRI from the set $\mathbb{I}$. More specifically, $\mathcal{NG} = (n, \mathcal{G})$ where $n \in \mathbb{I}$ and $\mathcal{G}$ is an RDF graph.*

From this point on, and without loss of generality, we refer to a named graph by using only its name n.

Definition 3. *An RDF quadruple q (subject, predicate, object, named graph) consists of an RDF triple and the IRI of a named graph that triple belongs to. Then, set $\mathcal{Q} = \mathbb{I} \times \mathbb{I} \times (\mathbb{I} \times \mathbb{L}) \times \mathbb{I}$ is the set of all RDF quadruples.*

Subject (S)	**Predicate (P)**	**Object (O)**
<hypertension>	<medication>	<diuretics>
<hypertension>	<medication>	<beta_blockers>
<diuretics>	<slightly_increase>	"glucose"

Table 2.1: Tabular representation of an RDF graph

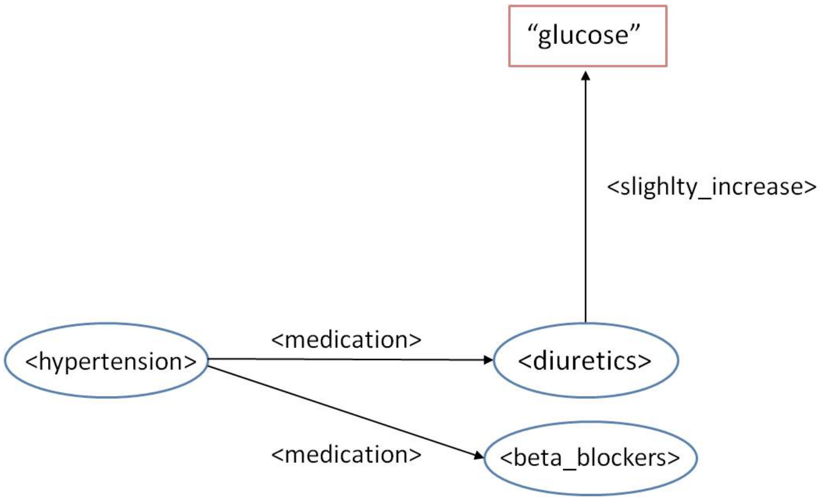

Figure 2.2: Graphical representation of the RDF graph shown in Table 2.1

Example 2. For example, consider (<hypertension>, <medication>, <diuretics>, <Pathologist>) that is an RDF quadruple, with <hypertension> being its subject, <medication> being its predicate, <diuretics> being its object and <Pathologist> being the IRI of a named graph that the aforementioned triple belongs to. □

Subject (S)	**Predicate (P)**	**Object (O)**	**Named Graph (NG)**
<hypertension>	<medication>	<diuretics>	<Pathologist>
<hypertension>	<medication>	<beta_blockers>	<Pathologist>
<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
<hypertension>	<medication>	<diuretics>	<Diabetologist>
<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>

Table 2.2: A set of RDF quadruples

2.2 SPARQL

SPARQL 1.1 [4, 5] is the official W3C recommendation for querying and updating RDF graphs, and is based on the concept of matching patterns against such graphs. Thus, a SPARQL query or a SPARQL update determines the pattern to seek for, and the answer is the part of the RDF graph that matches this pattern.

The building block of a SPARQL statement is a *triple pattern tp* that resembles an RDF triple, but may have a *variable* (prefixed with character *?*) in any of its *subject*, *predicate*, or *object* positions. Intuitively, triple patterns return the triples in an RDF graph that have the form specified by those triple patterns.

In addition to the sets $\mathbb{I}$ and $\mathbb{L}$ we assume the existence of an infinite set $\mathbb{V}$ of variables disjoint from the above sets.

Definition 4. *A triple pattern tp is an element of the set* $\mathcal{TP} = (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{L} \cup \mathbb{V})$.

Intuitively a triple pattern denotes the triples in an RDF graph that are of a specific form.

Example 3. Consider the triple pattern (<hypertension>, *?p*, *?o*) that contains the variables *?p* and *?o*, which can be substituted by any IRI; as such, the previous triple pattern can be used to denote all triples with subject <hypertension>. □

To take into account context information expressed in the form of named graphs, SPARQL 1.1 defines *quad patterns (tp,n)* [4], that are essentially triple patterns with an additional column that denotes the named graph in which said triple pattern must be evaluated against. In this work, we allow only values from the set of IRIs for the named graph column; i.e., variables are not allowed in the graph position.

Definition 5. *A quad pattern qp is an element of the set* $\mathcal{QP} = (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{L} \cup \mathbb{V}) \times \mathbb{I}$.

Note that, as a consequence of Definition 5, a quadruple q can be also considered as a quad pattern.

Example 4. The quad pattern (<hypertension>, *?p*, *?o*, <Diabetologist>) matches all triples with subject <hypertension> in the named graph <Diabetologist>.
In a similar manner, the quad pattern (*?s*, *?p*, *?o*, <Pathologist>) matches all triples in the named graph <Pathologist>. □ .

SPARQL queries and updates use graph patterns. Graph patterns, as triple patterns and quad patterns, are matched against RDF graphs by substituting the variables with matching IRIs or literals.

Definition 6. *A SPARQL* graph pattern *gp is defined recursively as follows:*

- *A triple pattern tp is a graph pattern.*
- *A quad pattern qp is a graph pattern.*
- *If gp and gp′ are graph patterns then (gp . gp′), (gp* UNION *gp′), and (gp* OPTIONAL *gp′) are graph patterns.*
- *If C is a built-in condition, then (gp* FILTER *C) is a graph pattern.*

A SPARQL *built-in* condition is constructed using elements of the set $\mathbb{I} \cup \mathbb{L} \cup \mathbb{V}$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like *bound*, *isBlank*, and *isIRI*, plus other features (see [4] for a complete list).

Example 5. For example the following statements are all graph patterns:

- (<hypertension>, *?p*, *?o*, <Diabetologist>), (*?s*, *?p*, *?o*, <Pathologist>), (<bronchitis>, <treat_with>, "aspirin", <Pneumonologist>)
  These graph patterns are quad patterns as well.
- (*?s*, *?p*, *?o*, <Pathologist>) . (*?o*, <slightly_increase>, "glucose", <Side_Effects>)
  This graph pattern contains a join (on the variable *?o*) between two other graph patterns, (*?s*, *?p*, *?o*, <Pathologist>) and (*?o*, <slightly_increase>, "glucose", <Side_Effects>).
- (*?s*, *?p*, *?o*, <Pathologist>) . (*?o*, <slightly_increase>, "glucose", <Side_Effects>) UNION (<hypertension>, *?p*, *?o*, <Diabetologist>)
  This graph pattern contains a union between two other graph patterns, (*?s*, *?p*, *?o*, <Pathologist>) . (*?o*, <slightly_increase>, "glucose", <Side_Effects>) and (<hypertension>, *?p*, *?o*, <Diabetologist>). □

In our study, we focus on SPARQL INSERT updates containing graph patterns that consider only the union (UNION) and join (".") operators. In particular, we restrict ourselves to INSERT updates of the following form:

Definition 7. *A SPARQL* INSERT *update U is a statement of the form*

$$U := \text{INSERT } \{qp_{ins}\} \text{ WHERE } \{gp\}$$

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^i_1 . ... . qp^i_m. We require that for each qp^i_j there is a sequence $\langle qp^i_{j_1}, \ldots \rangle$ of quad patterns from gp^i, such that $qp^i_j = qp^i_{j_1}$ and 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 essentially corresponds to the class of SPARQL statements containing only union and join operators, as all statements of this class can be equivalently written in the above form [18]. The restriction on the existence of common variables is necessary to "strip" the graph pattern in the WHERE clause from quad patterns that play no essential role in its evaluation [18]. Furthermore, note that the SPARQL statement INSERT DATA is a special case of the previous INSERT update where gp is the empty graph pattern.

The INSERT clause of an update specifies what variables should be returned as results to form the new quadruples. The WHERE clause includes all the quad patterns that must be matched from the results. The full semantics of SPARQL Update are formally described in Section 4.

Example 6. Consider the INSERT update U: INSERT $\{qp_{ins}\}$ WHERE { qp^1_1 . qp^1_2 . qp^1_3 }, where:

qp_{ins}: (*?s*, *?p*, *?o*, <MyGraph>)
qp^1_1: (*?s1*, *?p1*, *?o1*, <n1>)
qp^1_2: (*?s*, *?p*, *?o2*, <n2>)
qp^1_3: (*?s3*, *?p3*, *?o*, <n3>)

We observe that the first quad pattern of the graph pattern in the WHERE clause, (*?s1*, *?p1*, *?o1*, <n1>), belongs to the sequence $\langle\ qp^1_1\ \rangle$, which does not contain an element with a common variable with qp_{ins}. In contrast, the second quad pattern, (?s, ?p, ?o2, <n2>), is related to the sequence $\langle\ qp_2^1\ \rangle$ that has an element with two common variables with qp_{ins}, ?s and ?p. For the third quad pattern, (?s3, ?p3, ?o, <n3>), there is a sequence $\langle\ qp_3^1\ \rangle$ that its first and only element shares a variable (?o) with qp_{ins}. As a result, the first quad pattern is omitted and U can be reworded as INSERT $\{qp_{ins}\}$ WHERE $\{qp_1^1\ .\ qp_2^1\}$, where:

qp_{ins}: (?s, ?p, ?o, <MyGraph>)
qp_1^1: (?s, ?p, ?o2, <n2>)
qp_2^1: (?s3, ?p3, ?o, <n3>)

□

Example 7. Consider the INSERT update U: INSERT $\{qp_{ins}\}$ WHERE $\{\ qp_1^1$ UNION $qp_1^2\ .\ qp_2^2\ \}$, where:

qp_{ins}: (<Alice>, ?b, ?c, <MyGraph>)
qp_1^1: (?a, ?b, ?c, <n1>)
qp_1^2: (?d, ?b, ?c, <n2>)
qp_2^2: (?d, <likes>, ?e, <n3>)

The update U consists of two graph patterns, gp^1 and gp^2, that are the operands of the UNION operation. Then, for the quad pattern qp_1^1 of gp^1 there is a sequence $\langle\ qp_1^1\ \rangle$ that contains only one element, which shares two common variables with qp_{ins}, ?b and ?c. In graph pattern gp^2, the quad pattern (?d, <likes>, ?e, <n3>) joins the quad pattern (?d, ?b, ?c, <n2>) on the variable ?d, and therefore both of them are elements of the sequence $\langle\ qp_1^2,\ qp_2^2\ \rangle$. Furthermore, the first element of this sequence has two common variables (?b, ?c) with qp_{ins}. As a result, we can not omit any quad pattern from the INSERT update U. □

According to SPARQL 1.1 Update [5], a SPARQL update is evaluated on a *Graph Store* that is a mutable container of RDF graphs. For simplicity however, in this thesis we define a Graph Store as:

Definition 8. *A* Graph Store $\mathcal{GS}$ *is a pair* $(\mathcal{Q}_{\mathcal{GS}}, \mathcal{N}_{\mathcal{GS}})$ *where* $\mathcal{Q}_{\mathcal{GS}}$ *is a set of quadruples* $(\mathcal{Q}_{\mathcal{GS}} \subseteq \mathcal{Q})$ *and* $\mathcal{N}_{\mathcal{GS}}$ *is a set of named graphs* $(\mathcal{N}_{\mathcal{GS}} \subseteq \mathbb{I})$*.*

$\mathcal{Q}_{\mathcal{GS}}$

S	P	O	NG
<hypertension>	<medication>	<diuretics>	<Pathologist>
<hypertension>	<medication>	<beta_blockers>	<Pathologist>
<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
<hypertension>	<medication>	<diuretics>	<Diabetologist>
<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>

$\mathcal{N}_{\mathcal{GS}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>

Table 2.3: Tabular representation of a Graph Store $\mathcal{GS}$

For the evaluation of SPARQL graph patterns, we follow the semantics discussed in [18, 19]. More specifically, a *solution mapping*, or simply a *mapping*, μ from $\mathbb{V}$ to $\mathbb{I} \cup \mathbb{L}$ is a partial function $\mu : \mathbb{V} \rightarrow \mathbb{I} \cup \mathbb{L}$. The domain of μ, $dom(\mu)$, is the subset of $\mathbb{V}$ where μ is defined. In case that $dom(\mu) = \emptyset$ then $\mu_\emptyset = \emptyset$; this is the *empty mapping*. Abusing notation, for an arbitrary quad pattern qp we denote by $var(qp)$ the set of variables occurring in qp and by $\mu(qp)$ the result obtained by replacing the variables in qp with their assigned values according to μ. Note that only the triple pattern part (tp) of a quad pattern is permitted to contain variables since n is always an IRI. Then, the evaluation of a quad pattern $qp = (tp, n)$ with respect to a Graph Store $\mathcal{GS}$ returns a *sets of mappings*, denoted as $\Omega = [[tp]]_n^{\mathcal{GS}}$, where.

$$[[tp]]_n^{\mathcal{GS}} = \{\mu \mid dom(\mu) = var(qp) \text{ and } \mu(qp) \subseteq \mathcal{T}_n\} \tag{2.1}$$

with $\mathcal{T}_n$ being the set of triples that are related to the named graph n.

Before discussing the evaluation of a graph pattern we shall refer to some additional notions related to mappings. Two mappings μ_1 and μ_2 are *compatible* if for every $?x \in dom(\mu_1) \cap dom(\mu_2)$ it is the case that $\mu_1(?x) = \mu_2(?x)$, i.e., $\mu_1 \cup \mu_2$ is also a mapping [18, 19]. Note that two mappings with disjoint domains are always compatible, and that the empty mapping $\mu_\emptyset$ is compatible with any other mapping. In addition, the join and the union of two sets of mappings Ω_1 and Ω_2 are defined as:

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$
- $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$.

Then, the evaluation of a SPARQL graph pattern gp with respect to a given Graph Store $\mathcal{GS}$, is defined recursively as:

- $[[tp]]_n^{\mathcal{GS}} \bowtie [[tp']]_{n'}^{\mathcal{GS}}$, if gp is of the form qp . qp'
- $[[tp]]_n^{\mathcal{GS}} \cup [[tp']]_{n'}^{\mathcal{GS}}$, if gp is of the form qp UNION qp'

where $qp = (tp, n)$ and $qp' = (tp', n')$.

Example 8. Consider the Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$), shown in Table 2.2, and the INSERT update U: INSERT $\{qp_{ins}\}$ WHERE $\{qp_1^1\}$, where:

qp_{ins}: ($?s$, $?p$, $?o$, <NewDoctor>)
qp_1^1: ($?s$, $?p$, $?o$, <Pathologist>)

Table 2.4 shows the evaluation of qp_1^1, denoted as Ω_1, where each column corresponds to a variable in the evaluated quad pattern and each row of the table corresponds to a mapping.

	$?s$	$?p$	$?o$
μ_1:	<hypertension>	<medication>	<diuretics>
μ_2:	<hypertension>	<medication>	<beta_blockers>

Table 2.4: Evaluation of quad pattern ($?s$, $?p$, $?o$, <Pathologist>)

According to the INSERT clause of U the result quadruples are formed using values from the evaluation of variable $?s$ for the subject position, $?p$ for the predicate position, $?o$ for the object position and the named graph <NewDoctor>. Hence, the INSERT update U generates the result quadruples (<hypertension>, <medication>, <diuretics>, <NewDoctor>) and (<hypertension>, <medication>, <beta_blockers>, <NewDoctor>).

Note that if U: INSERT $\{qp_{ins}\}$ WHERE $\{qp_1^1\}$, where:

qp_{ins}: (<hypertension>, $?p$, $?o$, <NewDoctor>)
qp_1^1: ($?s$, $?p$, $?o$, <Pathologist>)

Then, the evaluation of quad pattern qp_1^1 remains the same as well as the result quadruples. However, it is worth pointing out that the value of subject position in the result quadruples does not come from the evaluation of the variable $?s$ but from the constant value <hypertension> as defined by the INSERT clause. □

Example 9. Similarly to the previous example, consider the INSERT update U: INSERT $\{qp_{ins}\}$ WHERE { qp_1^1 . qp_2^1 }, where:

qp_{ins}: ($?s$, $?p$, $?o$, <NewDoctor>)
qp_1^1: ($?s$, $?p$, $?o$, <Pathologist>)
qp_2^1: ($?o$, $?x$, $?y$, <Side_Effects>)

Tables 2.5- 2.6 show the evaluation of qp_2^1 (Ω_2) and qp_1^1 . qp_2^1 ($\Omega_1 \bowtie \Omega_2$) respectively; the evaluation of quad pattern qp_1^1 was shown in Table 2.4.

	$?o$	$?x$	$?y$
μ_3:	<diuretics>	<slightly_increase>	"glucose"

Table 2.5: Evaluation of quad pattern ($?o$, $?x$, $?y$, <Side_Effects>)

	$?s$	$?p$	$?o$	$?x$	$?y$
μ_4:	<hypertension>	<medication>	<diuretics>	<slightly_increase>	"glucose"

Table 2.6: Evaluation of graph pattern ($?s$, $?p$, $?o$, <Pathologist>) . ($?o$, $?x$,$?y$, <Side_Effects>)

According to the INSERT clause of U the result quadruples are formed using the values from the evaluation of variable $?s$ for the subject position, $?p$ for the predicate position, $?o$ for the object position and the named graph <NewDoctor>. Hence, the INSERT update U generates only one quadruple (<hypertension>, <medication>, <diuretics>, <NewDoctor>) (based on the evaluation results of the graph pattern in the WHERE clause– see Table 2.6). □

Example 10. Consider the INSERT update U: INSERT $\{qp_{ins}\}$ WHERE { qp_1^1 UNION qp_1^2 }, where:

qp_{ins}: (*?s*, *?p*, *?o*, <NewDoctor>)
qp_1^1: (*?s*, *?p*, *?o*, <Pathologist>)
qp_1^2: (*?s*, *?p*, *?o*, <Diabetologist>)

The evaluation of qp_1^1 (Ω_1) was already shown in Table 2.4. Tables 2.7-2.8 show the evaluation of qp_1^2 (Ω_3) and qp_1^1 UNION qp_1^2 ($\Omega_1 \cup \Omega_3$), respectively.

	?s	*?p*	*?o*
μ_5:	<hypertension>	<medication>	<diuretics>

Table 2.7: Evaluation of quad pattern (*?s*, *?p*, *?o*, <Diabetologist>)

	?s	*?p*	*?o*
μ_6:	<hypertension>	<medication>	<diuretics>
μ_7:	<hypertension>	<medication>	<beta_blockers>
μ_8:	<hypertension>	<medication>	<diuretics>

Table 2.8: Evaluation of graph pattern (*?s*, *?p*, *?o*, <Pathologist>) UNION (*?s*, *?p*, *?o*, <Diabetologist>)

The result quadruples are formed using values from the evaluation of variable *?s* for the subject position, *?p* for the predicate position, *?o* for the object position and the named graph <NewDoctor>. Hence, the INSERT update U generates the quadruples (<hypertension>, <medication>, <diuretics>, <NewDoctor>) (this quadruple is generated with two different ways) and (<hypertension>, <medication>, <beta_blockers>, <NewDoctor>) (based on the evaluation results of the graph pattern in the WHERE clause– see Table 2.8). □

For a thorough presentation of the semantics of the SPARQL language, we urge the interested reader to read the SPARQL specification [4].

2.3 Provenance Models for Queries with Positive Algebra

A great number of provenance models have been proposed so far. Most of them, no matter which data model they support (RDF or relational), deal with the problem of provenance management for the positive fragment of a language (SPARQL or SQL). In particular, the positive fragment of SPARQL consists of statements, queries or updates, that use only the SPARQL operators SELECT, AND, FILTER and UNION [10], whereas the positive fragment of SQL is comprised of the operators σ (filtering), π (projection), $\cup$ (union) and $\bowtie$ (natural join) [7].

In this thesis, we propose a novel provenance model that is suitable to record the provenance of SPARQL update results. As already described in Section 2.2, we restrict our attention to unions of conjunctive INSERT updates and therefore our model deals with the positive fragment of SPARQL language. In this Section we will discuss the positive provenance models that our work builds on.

The most popular model among those to be discussed is the *provenance semirings*; the notion of *how provenance*, i.e., how an output tuple is derived according to a given query, was articulated for first time in this work. Green et al. [7] propose an algebraic approach that consider various forms of annotated (tagged) relational data and their transformations in the context of positive relational queries. A transformation refers to the operations that can be applied to the source tuples. Thus, source tuples can be either joined via a join operation (defined by the operator "$\cdot$"), or merged as an effect of a union or a projection operation (defined by the operator "+"). Then, abstract tags and operators are combined to create algebraic expressions that describe how source tuples generate a result tuple. These expressions are in fact polynomials in a commutative semiring (K, $+$, $\cdot$, 0, 1). Furthermore, the authors propose polynomials with integer coefficients —the universal provenance semiring- and show that positive algebra semantics for any commutative semiring factors through the provenance semantics.

In [10], authors extend the previous model and show that semirings approach is sufficient for positive SPARQL queries on annotated RDF data as well. More specifically, Karvounarakis et al. investigate how popular relational provenance models, such as *how* and *why*, can be leveraged to capture the data provenance of unions of conjunctive queries over Linked Data, despite their subtle differences. In addition, they identify the limitations of these models (mainly because of the SPARQL operator OPTIONAL) and advocate the need for new provenance models for SPARQL queries. We urge the interested reader to read [12, 13] for a full representation of SPARQL algebra using abstract relational provenance models.

The model of *where provenance* was introduced by Buneman et al. [16], and it was firstly defined for a deterministic semi-structured data model and an associated query language. In contrast to how (and why) provenance that describe the relationship between the source and the result tuples of a query, where provenance indicates the origin of an attribute of a result tuple, i.e., from which *location(s)* this attribute was copied. A location refers to an attribute of a tuple with respect to a relation [6]. In [20], Buneman et al. extended the aforementioned work for a relational model with SPJRU queries (in terms of selection (S), projection (P), join (J), renaming (R) and union (U) operators) and defined the semantics of where provenance through a set of *annotation propagation rules*. These rules determine how annotations related to the source locations propagate to result locations in order to form the where provenance of an attribute in a result tuple.

The Figure 2.3 shows a comparison of the main characteristics between the previous models and the proposed one.

	Green et al.	**Karvounarakis et al.**	**Buneman et al.**	**Proposed Model**
Operation	queries	queries	queries updates	updates
Data Model	relational	RDF	relational	RDF
Supported Operators	projection natural join union	projection filter join union	projection rename join union	projection (``copy'') join union
Provenance Model	how	how why	where	how where
Reconstruction of Operation	no	no	no	yes

Figure 2.3: Comparison between Green et al., Karvounarakis et al., Buneman et al. and proposed model

Chapter 3

Motivating Example

In the last years there is an increasing interest for the use of RDF technologies in the field of e-health and more specifically in medical applications [21, 22]. Scientists are especially enthusiastic about using RDF, since it gives users the ability to create descriptions in a very flexible and powerful way. Therefore, it is essential for scientists to be able to have access to this huge and heterogeneous amount of information, and at the same time track its *provenance*.

We will use, for illustration purposes, a simple example taken from the medical domain[1]. Table 3.1 illustrates the Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$) (presented in Section 2.2) that we will be considering, where each row of $\mathcal{Q}_{\mathcal{GS}}$ corresponds to an RDF quadruple, and columns **S**, **P**, **O**, **NG** stand for the *subject*, *predicate*, *object* and *named graph* of the RDF quadruple. Additionally, we have included column **PROV** that is used to store the *provenance* of a quadruple and the unique identifiers c_i for referring to a quadruple q_i. Furthermore, each row of $\mathcal{N}_{\mathcal{GS}}$ corresponds to a named graph.

Suppose now that a patient visits the hospital because of an urgent health issue. The doctor diagnosed hypertension and decided to prescribe diuretic medication. However, the patient's history includes diabetes; diuretics may increase the blood glucose [23], which is a dangerous condition for diabetics. For this reason, doctor prefers to prescribe a medication based on other doctors' opinion, stored in the on-line medical system; the final medication is inserted in the on-line system as

[1] <http://www.nhlbi.nih.gov/>

well. To support this request, he executes the SPARQL INSERT update U:

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

where:

qp_{ins}: (<hypertension>, $?p$, $?o$, <NewDoctor>)

qp_1^1: ($?s$, $?p$, $?o$, <Pathologist>)

qp_2^1: ($?o$, <slightly_increase>, "glucose", <Side_Effects>)

qp_1^2: (<hypertension>, $?p$, $?o$, <Diabetologist>)

$\mathcal{Q}_{\mathcal{GS}}$

	S	P	O	NG	PROV
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>	p_1
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>	p_2
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>	p_3
c_4	<hypertension>	<medication>	<diuretics>	<Diabetologist>	p_4
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>	p_5
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>	p_6

$\mathcal{N}_{\mathcal{GS}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>

Table 3.1: Tabular representation of Graph Store $\mathcal{GS}$ with additional information for provenance and quadruple identifiers

Intuitively, the INSERT update U will insert in the Graph Store information about a medicine that is a cure for hypertension and cause a slightly increase in the blood glucose levels (by <Pathologist>' point of view), or just a medicine that is a cure for hypertension (by <Diabetologist>' point of view; we consider that a Diabetologist would never suggest a medicine that would be harmful for a diabetic).

The INSERT clause determines the form of the result quadruples while the WHERE clause determines the values (through the evaluation process) for these quadruples. In our example, the WHERE clause contains a JOIN expression between the quad patterns qp_1^1 and qp_2^1 on the variable *?o*, and a UNION expression between graph patterns qp_1^1 . qp_2^1 (forms the graph pattern gp^1) and qp_1^2 (forms the graph pattern gp^2). Furthermore, it computes the values for the variables *?s*, *?p* and *?o*.

Tables 3.2 - 3.4 show the evaluation of qp_1^1 (Ω_1), qp_2^1 (Ω_2) and qp_1^2 (Ω_3), where each column corresponds to a variable in the evaluated quad pattern and each row of the table corresponds to a mapping. Similarly, Table 3.5 shows the evaluation of the join between qp_1^1 and qp_2^1 ($\Omega_1 \bowtie \Omega_2$), or, more precisely, the join of the corresponding mappings: μ_1 joins μ_3 over variable *?o*, resulting to the mapping μ_5. The evaluation of the union between qp_1^1 . qp_2^1 and qp_1^2 $((\Omega_1 \bowtie \Omega_2) \cup \Omega_3)$, shown in Table 3.6, is much simpler as it is the union of the corresponding mappings μ_5 and μ_4 (coming from the evaluation of the individual graph patterns gp^1 and gp^2).

	?s	***?p***	***?o***
μ_1:	<hypertension>	<medication>	<diuretics>
μ_2:	<hypertension>	<medication>	<beta_blockers>

Table 3.2: Evaluation of quad pattern (*?s*, *?p*, *?o*, <Pathologist>)

	?o
μ_3:	<diuretics>

Table 3.3: Evaluation of quad pattern (*?o*, <slightly_increase>, "glucose", <Side_Effects>)

	?p	***?o***
μ_4:	<medication>	<diuretics>

Table 3.4: Evaluation of quad pattern (<hypertension>, *?p*, *?o*, <Diabetologist>)

	?s	***?p***	***?o***
μ_5:	<hypertension>	<medication>	<diuretics>

Table 3.5: Evaluation of graph pattern (*?s*, *?p*, *?o*, <Pathologist>) . (*?o*, <slightly_increase>, "glucose", <Side_Effects>)

	?s	*?p*	*?o*
μ_4:		<medication>	<diuretics>
μ_5:	<hypertension>	<medication>	<diuretics>

Table 3.6: Evaluation of graph pattern (*?s*, *?p*, *?o*, <Pathologist>) . (*?o*, <slightly_increase>, "glucose", <Side_Effects>) UNION (<hypertension>, *?p*, *?o*, <Diabetologist>)

For the evaluation of the INSERT clause we are interested only in variables found in qp_{ins} (*?p*, *?o*); each mapping of Table 3.6 is used to extract the values for these variables. These values correspond, therefore, the predicate and object of the result quadruple, respectively. Note that the subject of the result quadruple, (<hypertension>), was introduced as a constant value by the update itself, whereas the graph attribute is user-defined.

The result quadruple (<hypertension>, <medication>, <diuretics>, <NewDoctor>) (c_7) and the named graph <NewDoctor> are inserted in $\mathcal{Q}_{\mathcal{GS}}$ and $\mathcal{N}_{\mathcal{GS}}$ of $\mathcal{GS}$, respectively, forming thereby the new Graph Store $\mathcal{GS}_2$ ($\mathcal{Q}_{\mathcal{GS}_2}$, $\mathcal{N}_{\mathcal{GS}_2}$), shown in Table 3.7.

$\mathcal{Q}_{\mathcal{GS}_2}$

	S	**P**	**O**	**NG**	**PROV**
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>	p_1
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>	p_2
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>	p_3
c_4	<hypertension>	<medication>	<diuretics>	<Diabetologist>	p_4
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>	p_5
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>	p_6
$\mathbf{c_7}$	**<hypertension>**	**<medication>**	**<diuretics>**	**<NewDoctor>**	$\mathbf{p_7}$

$\mathcal{N}_{\mathcal{GS}_2}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>

Table 3.7: Tabular representation of Graph Store $\mathcal{GS}_2$ with additional information for provenance and quadruple identifiers

The expression p_7 below is used to describe the provenance of quadruple c_7:

$$p_7\text{: } \{\ (\perp,\ {}_{qp_1^1.p}(c_1\ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}}\ c_3),\ {}_{qp_1^1.o}(c_1\ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}}\ c_3))$$
$$\oplus$$
$$(\perp,\ {}_{qp_1^2.p}(c_4),\ {}_{qp_1^2.o}(c_4))\ \}$$

Note that p_7 records the fact that c_7 originates with two different ways (illustrated by the provenance UNION operator $\oplus$), either via join (e.g., first operand of UNION), or via "copy" values (e.g., second operand of UNION). In the first case, we record the fact that the derivation involves a join over the object-subject positions (**O**-**S**) of qp_1^1, qp_2^1, whose evaluation results to quadruples c_1, c_3 (cf. $c_1\ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}}\ c_3$). Further, it states that the subject (**S**) of the new quadruple c_7 is a constant value ($\perp$), the predicate (**P**) originates from the predicate (**P**) of quadruple c_1 (cf. ${}_{qp_1^1.p}(\ldots)$), whereas its object (**O**) originates from the object (**O**) of quadruple c_1 (cf. ${}_{qp_1^1.o}(\ldots)$). In the second case, we record the fact that some attributes of the new quadruple derived from the quadruple c_4 and, additionally, that the subject (**S**) of the new quadruple c_7 is a constant value ($\perp$), its predicate (**P**) originates from the predicate (**P**) of quadruple c_4 (cf. ${}_{qp_1^2.p}(\ldots)$) and its object (**O**) originates from the object (**O**) of quadruple c_4 (cf. ${}_{qp_1^2.o}(\ldots)$).

The created expression (p_7) is inspired by standard provenance expressions [7, 14] used in abstract provenance models, but contains additional information not present in standard how provenance expressions. In particular, we include, for each attribute of the new quadruple:

- a subscript denoting the information for the position of the quad pattern in the WHERE clause that this element's value is taken from (arbitrarily we define 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. This information is important for understanding how c_7 found its way in the Graph Store; as it turns out, this information is also enough for *reconstructing* a *compatible* SPARQL INSERT update.

Chapter 4

SPARQL Update Language Semantics

In the following sections, we discuss the formal semantics for the different operations of SPARQL Update according to our approach. SPARQL 1.1 Update [5] supports two categories of update operations on a Graph Store, the *Graph Update* (Section 4.1) and the *Graph Management* (Section 4.2) operations.

A SPARQL update can read from and write to several named graphs at the same time. For simplicity, we restrict our attention to updates that affect only a single RDF named graph each time, i.e., it is permitted to read from only one graph and write to as well one graph (we refer to this graph as *target graph*) at the same time (see Section 2). Let n_u be the IRI of the target named graph and $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$) be a Graph Store. The result of the execution of a SPARQL update operation on $\mathcal{GS}$ is a newly constructed Graph Store $\mathcal{GS}'$ ($\mathcal{Q}'_{\mathcal{GS}}$, $\mathcal{N}'_{\mathcal{GS}}$).

Note that in case that a graph is not related to any quadruple after an operation, then it is not removed from the set of graphs $\mathcal{N}_{\mathcal{GS}}$ in the Graph Store. According to SPARQL 1.1 Update semantics it is up to the implementation to decide whether an empty graph will be removed or not. Also, if the inserted data are related to a graph that does not exist in the Graph Store then the graph is created and added to the set of graphs $\mathcal{N}_{\mathcal{GS}}$ in the Graph Store.

For ease of readability we define the auxiliary function EVAL (qp, Ω) that will be used to determine the semantics of some update operations:

- eval($quad\ pattern\ qp, set\ of\ mappings\ \Omega$) = $\{\mu_i(qp) \mid \mu_i \in \Omega\}$

  The function returns a set of quadruples obtained by substituting the variables in qp according to each mapping μ_i in the set of mappings Ω and assigning to them as graph attribute the corresponding value of quad pattern qp.

For the rest of this Chapter we will consider the Graph Store $\mathcal{GS}_2$ ($\mathcal{Q}_{\mathcal{GS}_2}$, $\mathcal{N}_{\mathcal{GS}_2}$) of our Motivating Example (Chapter 3) for the in-line examples.

4.1 Graph Update Operations

This category concerns the addition and removal of quadruples within the Graph Store, e.g., INSERT, DELETE, CLEAR, LOAD operations.

1. **INSERT DATA**

   Let q (s,p,o,n_u) be a ground quadruple. Then:

   INSERT DATA $\{q\}$

   INSERT DATA adds the quadruple q to the Graph Store $\mathcal{GS}$ and more specifically to $\mathcal{Q}_{\mathcal{GS}}$. If the quadruple already exists in $\mathcal{Q}_{\mathcal{GS}}$ then no action is performed for it. Note that INSERT DATA is a special case of the INSERT operation, where grounded quadruples are inserted to the Graph Store. In particular, we write:

   INSERT $\{q\}$ **WHERE** $\{ \}$

   We define formally the semantics of the operation as follows:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
insert data(q, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \cup \{ q \}$	$\mathcal{N}_{\mathcal{GS}} \cup \{ n_u \}$

Example 11. The following INSERT DATA operation adds the quadruple (<ace_inhibitors>, <lower>, "blood pressure", <HeartFailure>) into the Graph Store. This quadruple is used to determine a treatment in case of heart failure disease. We write here the update operation following the syntax of SPARQL 1.1. Update:

```
INSERT DATA {
    GRAPH <HeartFailure> { <ace_inhibitors> <lower> "blood pressure" }
}
```

We write the same update operation following our abstract syntax:

```
INSERT DATA {
    (<ace_inhibitors>, <lower>, "blood pressure", <HeartFailure>)
}
```

The quadruple c_8 and the named graph <HeartFailure> are inserted in the Graph Store $\mathcal{GS}_2$, forming consequently the new Graph Store $\mathcal{GS}_3$, shown in Table 4.1.

2. **DELETE DATA**

   Let q (s,p,o,n_u) be a ground quadruple. Then:

   DELETE DATA $\{q\}$

   DELETE DATA deletes the quadruple q from the Graph Store $\mathcal{GS}$ and more specifically from $\mathcal{Q}_{\mathcal{GS}}$. If the quadruple does not exist in $\mathcal{Q}_{\mathcal{GS}}$ then no action is performed for it. Note that DELETE DATA is a special case of the DELETE operation, where grounded quadruples are deleted from the Graph Store. In particular, we write:

   DELETE $\{q\}$ **WHERE** $\{\ \}$

$\mathcal{Q}_{\mathcal{GS}_3}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_4	<hypertension>	<medication>	<diuretics>	<Diabetologist>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_7	<hypertension>	<medication>	<diuretics>	<NewDoctor>
c_8	**<ace_inhibitors>**	**<lower>**	**"blood pressure"**	**<HeartFailure>**

$\mathcal{N}_{\mathcal{GS}_3}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>

Table 4.1: Graph Store $\mathcal{GS}_3$ (INSERT DATA operation)

We define formally the semantics of the operation as follows:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
delete data(q, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \setminus q$	$\mathcal{N}_{\mathcal{GS}}$

Example 12. The following DELETE DATA operation removes the quadruple (<hypertension>, <treat1>, <diuretics>, <NewDoctor>) from the Graph Store. Following the syntax of SPARQL 1.1. Update, we write:

```
DELETE DATA {
    GRAPH <NewDoctor> { <hypertension> <treat1> <diuretics> }
}
```

Following our abstract syntax, we write:

```
DELETE DATA {
```

(<hypertension>, <medication>, <diuretics>, <NewDoctor>)

}

The quadruple c_7 is deleted from the Graph Store $\mathcal{GS}_3$, forming consequently the new Graph Store $\mathcal{GS}_4$, shown in Table 4.2.

$\mathcal{Q}_{\mathcal{GS}_4}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_4	<hypertension>	<medication>	<diuretics>	<Diabetologist>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
~~c_7~~	~~**<hypertension>**~~	~~**<medication>**~~	~~**<diuretics>**~~	~~**<NewDoctor>**~~
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>

$\mathcal{N}_{\mathcal{GS}_4}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>

Table 4.2: Graph Store $\mathcal{GS}_4$ (DELETE DATA operation)

Note that the named graph <NewDoctor> is not removed from the Graph Store $\mathcal{GS}_3$ ($\mathcal{N}_{\mathcal{GS}_3}$), despite the fact that it is associated with no quadruple any more.

3. **INSERT**

Let $qp_{ins} = (tp_{ins}, n_u)$ be a quad pattern, gp be 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 and Ω is the evaluation result of gp (see Section 2.2 for details). Then:

INSERT $\{qp_{ins}\}$ **WHERE** $\{gp\}$

INSERT adds quadruples to the Graph Store based on the evaluation results of qp_{ins} on the set of mappings obtained from the evaluation of graph pattern gp specified in the WHERE clause (see Section 2.2).

Formally, we define:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
insert(qp_{ins}, gp, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \cup \text{eval}(qp_{ins}, \Omega)$	$\mathcal{N}_{\mathcal{GS}} \cup \{n_u\}$

Example 13. The following INSERT update modifies the predicate value of the quadruples associated with the graph <Diabetologist> and adds them as newly constructed quadruples into the Graph Store. Using the SPARQL 1.1. Update syntax, we write:

INSERT { GRAPH <Diabetologist> { *?disease* <treatment> *?medicine* } }
WHERE { GRAPH <Diabetologist> { *?disease* *?property* *?medicine* } }

We write the same operation using our abstract syntax:

INSERT {(*?s*, <treatment>, *?o*, <Diabetologist>) }
WHERE { (*?s*, *?p*, *?o*, <Diabetologist>).
}

The quadruple c_9 is inserted into the Graph Store $\mathcal{GS}_4$, forming consequently the new Graph Store $\mathcal{GS}_5$, shown in Table 4.3.

4. **DELETE**

Let $qp_{del} = (tp_{del}, n_u)$ be a quad pattern, gp be a graph pattern formed as a union of individual graph patterns, gp^1 UNION ... UNION gp^k. Each gp^i is of the form $qp^i_1 \;.\; qp^i_2 \;.\; \ldots \;.\; qp^i_m$ and Ω is the evaluation result of gp (see Section 2.2). Then:

$\mathcal{Q}_{\mathcal{GS}_5}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_4	<hypertension>	<medication>	<diuretics>	<Diabetologist>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_9	**<hypertension>**	**<treatment>**	**<diuretics>**	**<Diabetologist>**

$\mathcal{N}_{\mathcal{GS}_5}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>

Table 4.3: Graph Store $\mathcal{GS}_5$ (INSERT operation)

DELETE $\{qp_{del}\}$ **WHERE** $\{gp\}$

DELETE removes quadruples from the Graph Store based on the evaluation results of qp_{del} on the set of mappings obtained from the evaluation of graph pattern gp specified in the WHERE clause.

We define formally the semantics of the operation as follows:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
$\text{delete}(qp_{del},\, gp,\, \mathcal{GS})$	$\mathcal{Q}_{\mathcal{GS}} \setminus \text{eval}(qp_{del}, \Omega)$	$\mathcal{N}_{\mathcal{GS}}$

Example 14. The following DELETE update removes from the Graph Store the quadruples that are related to the graph <Diabetologist> and have com-

mon subject and predicate values in graphs <Diabetologist> and <Pathologist>. Using the SPARQL 1.1. Update syntax, we write:

DELETE { GRAPH <Diabetologist> { *?s ?p ?o* } }
WHERE { GRAPH <Diabetologist> {*?s ?p ?o* } .
GRAPH <Pathologist> {*?s ?p ?o1*} }

The same operation is written using our abstract syntax as:

DELETE {(*?s*, *?p*, *?o*, <Diabetologist>) }
WHERE { (*?s*, *?p*, *?o*, <Diabetologist>).
(*?s*, *?p*, *?o1*, <Pathologist>) }

The quadruple c_4 is removed from the Graph Store $\mathcal{GS}_5$, forming consequently the new Graph Store $\mathcal{GS}_6$, shown in Table 4.4.

$\mathcal{Q}_{\mathcal{GS}_6}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
~~c_4~~	~~<hypertension>~~	~~<medication>~~	~~<diuretics>~~	~~<Diabetologist>~~
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_9	<hypertension>	<treatment>	<diuretics>	<Diabetologist>

$\mathcal{N}_{\mathcal{GS}_6}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>

Table 4.4: Graph Store $\mathcal{GS}_6$ (DELETE operation)

5. **DELETE/INSERT**

Let $qp_{del} = (tp_{del}, n_u)$, $qp_{ins} = (tp_{ins}, n_u)$ be quad patterns, gp be 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^i_1 . qp^i_2 . .... qp^i_m and Ω is the evaluation result of gp (see Section 2.2). Then:

DELETE $\{qp_{del}\}$ **INSERT** $\{qp_{ins}\}$ **WHERE** $\{gp\}$

DELETE/INSERT is a shortcut for removing and adding quadruples from/to the Graph Store based on the evaluation results of qp_{del} and qp_{ins} on the set of mappings obtained from the evaluation of graph pattern gp specified in the WHERE clause.

In the same manner as in INSERT and DELETE operations, we define formally:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
delete/insert(qp_{del}, qp_{ins}, gp, $\mathcal{GS}$)	$(\mathcal{Q}_{\mathcal{GS}} \setminus eval(qp_{del}, \Omega))$ $\cup$ eval(qp_{ins}, Ω)	$\mathcal{N}_{\mathcal{GS}} \cup \{n_u\}$

Example 15. The following DELETE/INSERT removes from the Graph Store the quadruples that are related to the graph <Diabetologist>. Additionally, it inserts new quadruples with respect to the treatment of hypertension. Using the SPARQL 1.1. Update syntax, we write:

DELETE { GRAPH <Diabetologist> { *?s ?p ?o* } }
INSERT { GRAPH <Pathologist> { *?s* <treat3> *?o1* } }
WHERE { GRAPH <Diabetologist> {*?s ?p ?o* } UNION
{ GRAPH <Pathologist> {*?s ?p ?o* .
GRAPH <HeartFailure> {*?o1 ?p1 ?s1*} } }

The same operation is written using our abstract syntax as:

DELETE {(*?s*, *?p*, *?o*, <Diabetologist>) }
INSERT {(*?s*, <treat3>, *?o1*, <Pathologist>) }
WHERE { (*?s*, *?p*, *?o*, <Diabetologist>) UNION

(*?s*, *?p*, *?o1*, <Pathologist>) .
(*?o1*, *?p1*, *?s1*, <HeartFailure>) }

The quadruple c_9 is removed from the Graph Store $\mathcal{GS}_6$, whereas the quadruple c_{10} is inserted to it, forming thereby the new Graph Store $\mathcal{GS}_7$, shown in Table 4.5.

$\mathcal{Q}_{\mathcal{GS}_7}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
~~c_9~~	~~**<hypertension>**~~	~~**<treatment>**~~	~~**<diuretics>**~~	~~**<Diabetologist>**~~
c_{10}	**<hypertension>**	**<treat3>**	**<ace_inhibitors>**	**<Pathologist>**

$\mathcal{N}_{\mathcal{GS}_7}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>

Table 4.5: Graph Store $\mathcal{GS}_7$ (DELETE/INSERT shortcut)

6. **LOAD**

Let n_{from} be the IRI of the named graph, whose data we want to load. Then:

LOAD n_{from} INTO n_u

LOAD reads the RDF named graph n_{from} and inserts its triples into the

Graph Store, after appending to them as graph attribute the value n_u (forming thereby quadruples). Note that graph n_{from} does not necessarily belong to the Graph Store.

We define formally the semantics of the operation:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
load($n_{from}, n_u, \mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \cup \{ (s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_{from}} \}$	$\mathcal{N}_{\mathcal{GS}} \cup \{ n_u \}$

with $\mathcal{T}_{n_{from}}$ being the set of triples that are related to the named graph n_{from}.

Example 16. The following LOAD operation inserts the quadruples formed by the triples in graph <HypertensionDrugs> and the graph <Drugs>. We write the operation following the SPARQL 1.1. Update syntax:
LOAD <HypertensionDrugs> INTO GRAPH <Drugs>

We write the same operation using our abstract syntax:
LOAD <HypertensionDrugs> INTO <Drugs>

S	P	O
<lasix>	<class>	<diuretics>
<diuril>	<class>	<diuretics>
<lopressor>	<class>	<beta_blockers>
<accupril>	<class>	<ace_inhibitors>
<monopril>	<class>	<ace_inhibitors>

Table 4.6: Tabular representation of named graph <HypertensionDrugs>[1]

This operation adds the quadruples c_{11}, c_{12}, c_{13}, c_{14}, c_{15} and the named graph <Drugs> to the Graph Store $\mathcal{GS}_7$, forming thereby the new Graph Store $\mathcal{GS}_8$, shown in Table 4.7.

7. **CLEAR**

This operation can be defined as:

[1] goo.gl/NACUXq

$\mathcal{Q}_{\mathcal{GS}_8}$

	S	P	O	NG
c_1	<hypertension>	<medication>	<diuretics>	<Pathologist>
c_2	<hypertension>	<medication>	<beta_blockers>	<Pathologist>
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_{10}	<hypertension>	<treat3>	<ace_inhibitors>	<Pathologist>
c_{11}	**<lasix>**	**<class>**	**<diuretics>**	**<Drugs>**
c_{12}	**<diuril>**	**<class>**	**<diuretics>**	**<Drugs>**
c_{13}	**<lopressor>**	**<class>**	**<beta_blockers>**	**<Drugs>**
c_{14}	**<accupril>**	**<class>**	**<ace_inhibitors>**	**<Drugs>**
c_{15}	**<monopril>**	**<class>**	**<ace_inhibitors>**	**<Drugs>**

$\mathcal{N}_{\mathcal{GS}_8}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>
<Drugs>

Table 4.7: Graph Store $\mathcal{GS}_8$ (LOAD operation)

CLEAR n_u

The CLEAR operation removes the quadruples that are associated with the specified graph n_u from the Graph Store.

Formally, we define the semantics for this operation:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
clear(n_u, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \setminus \{(s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_u}\}$	$\mathcal{N}_{\mathcal{GS}}$

where $\mathcal{T}_{n_u}$ is the set of triples that are related to the named graph n_u.

Example 17. The following CLEAR operation removes from the Graph Store $\mathcal{GS}_8$ all quadruples that are related to the graph <Pathologist>. Following the syntax of SPARQL 1.1. Update we write:

CLEAR GRAPH <Pathologist>

The same operation can be written using our abstract syntax as:

CLEAR <Pathologist>

This operation removes the quadruples c_1, c_2 and c_{10} from the Graph Store $\mathcal{GS}_8$, forming thereby the new Graph Store $\mathcal{GS}_9$, shown in Table 4.8.

4.2 Graph Management Operations

This category concerns the creation and deletion of graphs within the Graph Store, as well as convenient shortcuts for Graph Update operations often used during graph management (to add, move, and copy all quadruples that are related to a graph), e.g., CREATE, DROP, COPY, MOVE, ADD.

1. **CREATE**

   We define this operation as:

   CREATE n_u

   CREATE operation creates an empty named graph n_u and inserts it into the Graph Store $\mathcal{GS}$ and more specifically in $\mathcal{N}_{\mathcal{GS}}$. If the specified named graph already exists in the Graph Store then no action is performed.

$\mathcal{Q}_{\mathcal{GS}_9}$

	S	P	O	NG
~~c_1~~	~~**<hypertension>**~~	~~**<medication>**~~	~~**<diuretics>**~~	~~**<Pathologist>**~~
~~c_2~~	~~**<hypertension>**~~	~~**<medication>**~~	~~**<beta_blockers>**~~	~~**<Pathologist>**~~
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
~~c_{10}~~	~~**<hypertension>**~~	~~**<treat3>**~~	~~**<ace_inhibitors>**~~	~~**<Pathologist>**~~
c_{11}	<lasix>	<class>	<diuretics>	<Drugs>
c_{12}	<diuril>	<class>	<diuretics>	<Drugs>
c_{13}	<lopressor>	<class>	<beta_blockers>	<Drugs>
c_{14}	<accupril>	<class>	<ace_inhibitors>	<Drugs>
c_{15}	<monopril>	<class>	<ace_inhibitors>	<Drugs>

$\mathcal{N}_{\mathcal{GS}_9}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>
<Drugs>

Table 4.8: Graph Store $\mathcal{GS}_9$ (CLEAR operation)

Formally, the semantics of this operation can be defined as:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
create(n_u, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}}$	$\mathcal{N}_{\mathcal{GS}} \cup \{ n_u \}$

Example 18. The following CREATE update operation inserts into the Graph Store $\mathcal{GS}_9$ the graph <Hypertension>, forming thereby the newly constructed Graph Store $\mathcal{GS}_{10}$, shown in Table 4.9. Following the syntax of SPARQL 1.1. Update we write:

CREATE GRAPH <Hypertension>

The same operation can be written using our abstract syntax as:

CREATE <Hypertension>

$\mathcal{Q}_{\mathcal{GS}_{10}}$

	S	P	O	NG
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_5	<bronchitis>	<treat_with>	<antibiotics>	<Pneumonologist>
c_6	<bronchitis>	<treat_with>	"aspirin"	<Pneumonologist>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_{11}	<lasix>	<class>	<diuretics>	<Drugs>
c_{12}	<diuril>	<class>	<diuretics>	<Drugs>
c_{13}	<lopressor>	<class>	<beta_blockers>	<Drugs>
c_{14}	<accupril>	<class>	<ace_inhibitors>	<Drugs>
c_{15}	<monopril>	<class>	<ace_inhibitors>	<Drugs>

$\mathcal{N}_{\mathcal{GS}_{10}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<Pneumonologist>
<NewDoctor>
<HeartFailure>
<Drugs>
<Hypertension>

Table 4.9: Graph Store $\mathcal{GS}_{10}$ (CREATE operation)

2. **DROP**

We define the operation as:

DROP n_u

The DROP operation removes the named graph n_u and the corresponding quadruples from the Graph Store. If the graph does not exist in the Graph

Store, then no action is performed.

The semantics of the operation are defined as:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
drop(n_u, $\mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \setminus \{(s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_u}\}$	$\mathcal{N}_{\mathcal{GS}} \setminus \{ n_u \}$

with $\mathcal{T}_{n_u}$ being the set of triples that are related to the named graph n_u.

Example 19. The following DROP update operation removes from the Graph Store $\mathcal{GS}_{10}$ the graph <Pneumonologist> and its corresponding quadruples c_5 and c_6. The newly constructed Graph Store $\mathcal{GS}_{11}$ is shown in Table 4.10. We write the previous operation following the syntax of SPARQL 1.1. Update:
DROP GRAPH <Pneumonologist>

Using our abstract syntax the same operation can be written as:
DROP <Pneumonologist>

3. **COPY**

Let n_{from} be the IRI of the named graph whose data we want to copy. Then:

COPY n_{from} **TO** n_u

COPY operation inserts the triples that are related to the graph n_{from} into the Graph Store, as newly constructed quadruples with graph value n_u. Data related to the input graph n_{from} is not affected, but data related to the target graph n_u, if any, is removed before insertion.

We define formally the semantics:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
copy(n_{from}, n_u, $\mathcal{GS}$)	$(\mathcal{Q}_{\mathcal{GS}} \setminus \{ (s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_u} \})$ $\cup \{ (s',p',o',n_u) \mid (s',p',o') \in \mathcal{T}_{n_{from}} \}$	$\mathcal{N}_{\mathcal{GS}} \cup$ $\{ n_u \}$

$\mathcal{Q}_{\mathcal{GS}_{11}}$

	S	P	O	NG
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
~~c_5~~	~~<bronchitis>~~	~~<treat_with>~~	~~<antibiotics>~~	~~<Pneumonologist>~~
~~c_6~~	~~<bronchitis>~~	~~<treat_with>~~	~~"aspirin"~~	~~<Pneumonologist>~~
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_{11}	<lasix>	<class>	<diuretics>	<Drugs>
c_{12}	<diuril>	<class>	<diuretics>	<Drugs>
c_{13}	<lopressor>	<class>	<beta_blockers>	<Drugs>
c_{14}	<accupril>	<class>	<ace_inhibitors>	<Drugs>
c_{15}	<monopril>	<class>	<ace_inhibitors>	<Drugs>

$\mathcal{N}_{\mathcal{GS}_{11}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
~~<Pneumonologist>~~
<NewDoctor>
<HeartFailure>
<Drugs>
<Hypertension>

Table 4.10: Graph Store $\mathcal{GS}_{11}$ (DROP operation)

where $\mathcal{T}_{n_u}$, $\mathcal{T}_{n_{from}}$ are the sets of triples that are related to the named graphs n_u and n_{from} respectively.

Example 20. The following COPY operation inserts the quadruples that formed by the triples related to the graph <HeartFailure> and the graph value <Hypertension>, i.e., c_{16}, into the Graph Store $\mathcal{GS}_{11}$. The newly constructed Graph Store $\mathcal{GS}_{12}$ is shown in Table 4.11. We write here the update operation following the syntax of SPARQL 1.1 Update:
COPY GRAPH <HeartFailure> TO GRAPH <Hypertension>

Using our abstract syntax the same operation can be written as:
COPY <HeartFailure> TO <Hypertension>

$\mathcal{Q}_{\mathcal{GS}_{12}}$

	S	P	O	NG
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_{11}	<lasix>	<class>	<diuretics>	<Drugs>
c_{12}	<diuril>	<class>	<diuretics>	<Drugs>
c_{13}	<lopressor>	<class>	<beta_blockers>	<Drugs>
c_{14}	<accupril>	<class>	<ace_inhibitors>	<Drugs>
c_{15}	<monopril>	<class>	<ace_inhibitors>	<Drugs>
c_{16}	**<ace_inhibitors>**	**<lower>**	**"blood pressure"**	**<Hypertension>**

$\mathcal{N}_{\mathcal{GS}_{12}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<NewDoctor>
<HeartFailure>
<Drugs>
<Hypertension>

Table 4.11: Graph Store $\mathcal{GS}_{12}$ (COPY operation)

4. **MOVE**

Let n_{from} be the IRI of a named graph from which we want to move all data. Then, we define:

MOVE n_{from} **TO** n_u

MOVE operation inserts the triples related to the named graph n_{from} into the Graph Store, as newly constructed quadruples with graph value n_u. The input graph n_{from} is removed after insertion and data related to the target graph n_u, if any, is removed before insertion.

Formally, the semantics of MOVE operation can be defined as:

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
move($n_{from}, n_u, \mathcal{GS}$)	$((\mathcal{Q}_{\mathcal{GS}} \setminus \{ (s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_u} \})$ $\cup \{ (s',p',o',n_u) \mid (s',p',o') \in \mathcal{T}_{n_{from}} \})$ $\setminus \{ (s',p',o',n_{from}) \mid (s',p',o') \in \mathcal{T}_{n_{from}} \}$	$\mathcal{N}_{\mathcal{GS}}$ $\cup \{ n_u \}$ $\setminus \{ n_{from} \}$

where $\mathcal{T}_{n_u}$, $\mathcal{T}_{n_{from}}$ are the sets of triples that are related to the named graphs n_u and n_{from} respectively.

Example 21. This MOVE operation inserts the quadruples that consist of the triples in graph <Drugs> and the graph <Hypertension>, i.e., c_{17}, c_{18}, c_{19}, c_{20}, c_{21}, into the Graph Store $\mathcal{GS}_{12}$; before the insertion the quadruple c_{16} is deleted. In addition, the graph <Drugs> and its corresponding quadruples are removed from the Graph Store $\mathcal{GS}_{12}$. The newly constructed Graph Store $\mathcal{GS}_{13}$ is shown in Table 4.12. Following the syntax of SPARQL 1.1 Update we write:
MOVE GRAPH <Drugs> TO GRAPH <Hypertension>

Using our abstract syntax this operation can be written as:
MOVE <Drugs> TO <Hypertension>

5. **ADD**

Let n_{from} be the IRI of the named graph whose data we want to add in another named graph. Then:

ADD n_{from} **TO** n_u

ADD inserts all triples related to the graph n_{from} into the Graph Store, as newly constructed quadruples with graph value n_u. Data related to the input graph n_{from} is not affected, and initial data related to the target graph n_u, if any, is kept intact.

The semantics of this operation can be defined as follows:

$\mathcal{Q}_{\mathcal{GS}_{13}}$

	S	**P**	**O**	**NG**
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
~~c_{11}~~	~~<lasix>~~	~~<class>~~	~~<diuretics>~~	~~<Drugs>~~
~~c_{12}~~	~~<diuril>~~	~~<class>~~	~~<diuretics>~~	~~<Drugs>~~
~~c_{13}~~	~~<lopressor>~~	~~<class>~~	~~<beta_blockers>~~	~~<Drugs>~~
~~c_{14}~~	~~<accupril>~~	~~<class>~~	~~<ace_inhibitors>~~	~~<Drugs>~~
~~c_{15}~~	~~<monopril>~~	~~<class>~~	~~<ace_inhibitors>~~	~~<Drugs>~~
~~c_{16}~~	~~<ace_inhibitors>~~	~~<lower>~~	~~"blood pressure"~~	~~<Hypertension>~~
c_{17}	<lasix>	<class>	<diuretics>	<Hypertension>
c_{18}	<diuril>	<class>	<diuretics>	<Hypertension>
c_{19}	<lopressor>	<class>	<beta_blockers>	<Hypertension>
c_{20}	<accupril>	<class>	<ace_inhibitors>	<Hypertension>
c_{21}	<monopril>	<class>	<ace_inhibitors>	<Hypertension>

$\mathcal{N}_{\mathcal{GS}_{13}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<NewDoctor>
<HeartFailure>
~~<Drugs>~~
<Hypertension>

Table 4.12: Graph Store $\mathcal{GS}_{13}$ (MOVE operation)

	$\mathcal{Q}'_{\mathcal{GS}}$	$\mathcal{N}'_{\mathcal{GS}}$
add($n_{from}, n_u, \mathcal{GS}$)	$\mathcal{Q}_{\mathcal{GS}} \cup \{ (s,p,o,n_u) \mid (s,p,o) \in \mathcal{T}_{n_{from}} \}$	$\mathcal{N}_{\mathcal{GS}} \cup \{ n_u \}$

with $\mathcal{T}_{n_u}$ being the set of triples that are related to the named graph n_{from}.

Example 22. This ADD operation inserts the quadruples formed by the triples of graph <Side_Effects> and the graph <Impacts> (c_{22}) into the Graph Store $\mathcal{GS}_{13}$. The newly constructed Graph Store $\mathcal{GS}_{14}$ is shown in

Table 4.13. Following the syntax of SPARQL 1.1 Update we write:
ADD GRAPH <Side_Effects> TO GRAPH <Impacts>

Using our abstract syntax this operation can be written as:
ADD <Side_Effects> TO <Impacts>

$\mathcal{Q}_{\mathcal{GS}_{14}}$

	S	P	O	NG
c_3	<diuretics>	<slightly_increase>	"glucose"	<Side_Effects>
c_8	<ace_inhibitors>	<lower>	"blood pressure"	<HeartFailure>
c_{17}	<lasix>	<class>	<diuretics>	<Hypertension>
c_{18}	<diuril>	<class>	<diuretics>	<Hypertension>
c_{19}	<lopressor>	<class>	<beta_blockers>	<Hypertension>
c_{20}	<accupril>	<class>	<ace_inhibitors>	<Hypertension>
c_{21}	<monopril>	<class>	<ace_inhibitors>	<Hypertension>
c_{22}	**<diuretics>**	**<slightly_increase>**	**"glucose"**	**<Impacts>**

$\mathcal{N}_{\mathcal{GS}_{14}}$

NG
<Pathologist>
<Side_Effects>
<Diabetologist>
<NewDoctor>
<HeartFailure>
<Hypertension>
<Impacts>

Table 4.13: Graph Store $\mathcal{GS}_{14}$ (ADD operation)

Chapter 5

Abstract Provenance Model

An *abstract provenance model* is comprised of *abstract identifiers* and *abstract operators* [7, 10, 14]. Abstract identifiers (we refer to them as *quadruple identifiers* and we denote them by c_i) are uniquely assigned to RDF quadruples, whereas abstract operators describe the computations performed between source quadruples to derive a result quadruple.

Unlike previous abstract provenance models, we introduce the notion of *quad pattern positions*. Quad pattern positions are used to describe the occurrence of a constant or a variable in a quad pattern. We will refer to this notion in detail below.

Using this infrastructure, RDF quadruples are then annotated with complex algebraic provenance expressions that involve the identifiers, the operators and the quad pattern positions of the abstract model. Formally:

Definition 9. *The provenance p of a quadruple q is defined as $p := \{cpe_1, \ldots, cpe_k\}$, where cpe_i is a complex provenance expression.*

Definition 10. *A complex provenance expression cpe is 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 commutative binary operator of union.*

Definition 11. *A simple provenance expression pe is of the form $(prov_s, prov_p, prov_o)$, where $prov_{pos}$ being the provenance of the attribute pos.*

Example 23. Consider the provenance p_7 of quadruple c_7 (see Chapter 3). The provenance p_7 contains the complex provenance expression cpe_1 that consists of the simple provenance expressions, pe^1 and pe^2, combined using the operator $\oplus$. The simple provenance expression pe^1 consists of $prov_s$ $(\bot)$ that is the provenance of subject attribute, $prov_p$ $({}_{qp^1_1.p}(c_{1\{qp^1_1.o\}} \odot {}_{\{qp^1_2.s\}}c_3))$ that is the provenance of predicate attribute and $prov_o$ $({}_{qp^1_1.o}(c_{1\{qp^1_1.o\}} \odot {}_{\{qp^1_2.s\}}c_3))$ that is the provenance of object attribute. The simple provenance expression pe^2 consists of $prov_s$ $(\bot)$, $prov_p$ $({}_{qp^2_1.p}(c_4))$ and $prov_o$ $({}_{qp^2_1.o}(c_4))$. □

A quadruple can be resulted more than once from either a single or different INSERT updates applied over the course of time. To capture this feature, a complex provenance expression *cpe* (Definition 10) records each way of generating the new quadruple, whereas provenance *p* (Definition 9) encodes all the different ways, structured in a set.

Example 24. Consider the update U_1: INSERT $\{qp_{ins}\}$ WHERE $\{qp^1_1\}$, where:

qp_{ins}: (*?s*, *?p*, <steroids>, <NewDoctor>)
qp^1_1: (*?s*, *?p*, *?o*, <Pneumonologist>)

Intuitively, the INSERT update U_1 will insert in the Graph Store information which determines that <NewDoctor> suggests as a treatment for pulmonary ailments the <steroids>. The update U_1 is evaluated on the Graph Store $\mathcal{GS}_2$ (see Chapter 3). The result quadruple c_8 : (<bronchitis>, <treat_with>, <steroids>, <NewDoctor>) is inserted in the newly constructed Graph Store $\mathcal{GS}_3$; the named graph <NewDoctor> already exists in the Graph Store $\mathcal{GS}_2$. There are two ways to obtain c_8, either through copying the subject and predicate value from quadruple c_5 or through copying these values from quadruple c_6; object value is a constant value in both cases.

The provenance of the result quadruple c_8 is:

$$p_8 = \{({}_{qp^1_1.s}(c_5),\ {}_{qp^1_1.p}(c_5), \bot),\ ({}_{qp^1_1.s}(c_6),\ {}_{qp^1_1.p}(c_6), \bot)\}$$

Note that, in this case, $cpe_1 = ({}_{qp^1_1.s}(c_5),\ {}_{qp^1_1.p}(c_5),\ \bot)$ and $cpe_2 = ({}_{qp^1_1.s}(c_6),\ {}_{qp^1_1.p}(c_6),\ \bot)$, which represent the first and the second way, respectively, to obtain c_8. The complex provenance expression cpe_1 consists of a simple provenance expression pe^1, where $prov_s$ is equal to ${}_{qp^1_1.s}(c_5)$, $prov_p$ is equal to ${}_{qp^1_1.p}(c_5)$ and $prov_o$ is $\bot$. In a similar manner, we find the individual provenance expressions for cpe_2. □

As already stated, INSERT updates may use the UNION operator. In such updates, a result quadruple is generated from *one* or *more operands* of a UNION expression. In the first case (when the quadruple is generated from only one operand), the provenance management is identical to the provenance management of UNION-free updates, then $cpe = pe^1$. In the second case (when the quadruple is generated from more than one operands), each operand of the operator $\oplus$ represents the provenance of an operand of the UNION expression.

Example 25. Consider the update U and its result quadruple c_7 (see Chapter 3). The quadruple c_7 is obtained from both operands (qp_1^1 . qp_2^1, qp_1^2) of the UNION expression. As a result, its provenance p_7 contains two simple provenance expressions:
$pe^1 = (\bot, {}_{qp_1^1.p}(c_1 \ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}} \ c_3), {}_{qp_1^1.o}(c_1 \ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}} \ c_3))$
$pe^2 = (\bot, {}_{qp_1^2.p}(c_4), {}_{qp_1^2.o}(c_4))$

Each one of the simple provenance expressions pe^1 and pe^2 is standing for the provenance of c_7 derived from the operand (graph pattern) qp_1^1 . qp_2^1 and qp_1^2, respectively. □

Now let's see how the simple provenance expression pe (Definition 11) is constructed. For reasons that will be made apparent later in Chapter 6, it is necessary to refer to each individual variable or constant of an update. For this purpose, we arbitrarily number:

- graph patterns, based on the order that they appear in the WHERE clause. Then, the graph pattern gp^i, $i \geqslant 1$, indicates the i^{th} graph pattern of the WHERE clause.

- quad patterns, based on the order that they appear in a graph pattern gp^i. Then, the quad pattern qp_j^i, $j \geqslant 1$, indicates the j^{th} quad pattern in the graph pattern gp^i. A qp_j^i is called a *quad pattern identifier*.

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

A quad pattern $qp = (tp, n)$ has three *positions* (*pos*) for the *subject* s, *predicate* p and *object* o of its corresponding triple pattern tp (same as quadruples). Thus, each constant or variable of an INSERT update can be uniquely identified through the quad pattern identifier and its position *pos*, where *pos* can be one of s, p, o. For instance, $qp^1_2.s$ denotes the subject of the second quad pattern of the first graph pattern in the WHERE clause (i.e., *?o* in our Motivating Example), whereas $qp_{ins}.p$ denotes the predicate of the quad pattern in the INSERT clause (i.e., *?p* in our Motivating Example).

As shown in Definition 11, a simple provenance expression *pe* is broken down in $prov_s$, $prov_p$, $prov_o$, which records the provenance of the subject, predicate and object of the quadruple respectively. This allows the identification of the origin of each element-attribute individually (attribute-level provenance [17]). 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. Formally, we define:

Definition 12. *The provenance of attribute pos, namely* $prov_{pos}$*, is an expression of the form* $prov_{pos} := \perp \mid {}_{varSub}(spe)$*, where* $\perp$ *is a special label, varSub is the var subscript and spe is a standard provenance expression.*

Definition 13. *A standard provenance expression spe can be defined as* $spe := (c_i \; {}_{\{joinSub^1\}} \odot {}_{\{joinSub^2\}} \; c_j) \ldots {}_{\{joinSub^{r-1}\}} \odot {}_{\{joinSub^r\}} \; c_k$*, where* c_x *is a quadruple identifier,* $joinSub^z$ *is a join subscript and* $\odot$ *is the binary operator of join.*

As proposed in [8, 17], the special label $\perp$ is used in Definition 12 to record the case where the INSERT update constructs an element of the new quadruple using a constant, e.g., $prov_s$ in pe^1, pe^2 of provenance p_7 in our Motivating Example.

Instead of using a constant, we can alternatively construct an element of the new quadruple by copying a value from an existing quadruple. This quadruple may be in the Graph Store itself, or generated via SPARQL joins. This alternative is recorded using the form ${}_{varSub}(spe)$ of $prov_{pos}$.

This form is composed of the *varSub* subscript, namely *var subscript*, and a standard provenance expression *spe*. The var subscript represents a quad pattern position $qp^i_j.pos$, which denotes that the attribute *pos* of the new quadruple, originates from the variable in $qp^i_j.pos$, after applying the operation described in *spe*. Recall, though, that the attribute *pos* is generated from the evaluation of the variable in $qp_{ins}.pos$ (cf. Chapter 3), i.e., $qp^i_j.pos$ shares the same variable with

$qp_{ins}.pos$. As there could be multiple quad pattern positions in a gp^i (e.g., joins) that use the same variable with $qp_{ins}.pos$, the recorded quad pattern position in the var subscript is by convention the first one that matches.

Example 26. In our Motivating Example, the expression pe^1 contains the var subscripts $qp^1_1.p$ and $qp^1_1.o$ that appear in the provenance of predicate ($prov_p$) and object ($prov_o$) attributes, respectively. The quad pattern position $qp^1_1.p$ shares the variable *?p* with $qp_{ins}.p$ that generates the predicate attribute <medication> of the result quadruple c_7. Similarly, $qp^1_1.o$ has the same variable (*?o*) with $qp_{ins}.o$ that generates the object attribute <diuretics> of c_7. Note that *?o* appears in the quad pattern position $qp^1_2.s$ as well, because of an existing join on this variable. However, we record $qp^1_1.o$ as var subscript as it is the first quad pattern position of the current gp^i that shares the same variable with $qp_{ins}.pos$.

Similarly, we compute that expression pe^2 is associated with the var subscripts $qp^2_1.p$ and $qp^2_1.o$ for the predicate and object positions, respectively. □

The standard provenance expression *spe* is closely related to the evaluation process as it is composed of quadruple identifiers and potentially of quad pattern positions too. Quadruple identifiers represent the quadruples that resulted from the evaluation of the corresponding quad patterns, whereas quad pattern positions describe the existing joins. Hence, if *spe* is a quadruple identifier, then we have a "copy" in the sense of [17], e.g., $prov_p$, $prov_o$ in pe^2 of provenance p_7.

On the contrary, if *spe* is a more complex expression, then it describes a join operation e.g., $prov_p$, $prov_o$ in pe^1 of provenance p_7. The latter case is indicated by the existence of the *binary operator of join* $\odot$ (initially defined in [14]), where each operand of the operator $\odot$ is a subscript, namely a *join subscript*.

We use join subscripts to record the quad pattern positions that were joined (i.e. a join subscript is a set of quad pattern positions). Then, each operand of the operator $\odot$ represents the quad pattern positions of the corresponding operand of the SPARQL JOIN expression that participates in a join. We can easily figure out which quad pattern positions share the same variable since the i^{th} quad pattern position of the first join subscript of $\odot$ operator (e.g. $joinSub^1$, $joinSub^3$, ...) joins the i^{th} quad pattern position of the second join subscript ($joinSub^2$, $joinSub^4$, ...). This allows determining the actual quad pattern positions that joins performed on, an information critical for reconstructability as we will see below.

Example 27. Consider the INSERT update U of our Motivating Example. In the WHERE clause we meet the JOIN expression qp_1^1 . qp_2^1, where qp_1^1 joins qp_2^1 on the variable *?o*. We create, therefore, the $joinSub^1 = \{qp_1^1.o\}$ and $joinSub^2 = \{qp_2^1.s\}$ that represent the quad pattern positions of qp_1^1 and qp_2^1, respectively, that participate in the join. Moreover, from the evaluation of the JOIN expression (see Table 3.5) it arises that we the result quadruple takes its values from the quadruple c_1 (evaluation result of qp_1^1) and c_3 (evaluation result of qp_2^1). Thus, the resulting *spe* expression is $spe = c_1 \ {}_{\{qp_1^1.o\}} \odot {}_{\{qp_2^1.s\}} \ c_3$. □

Chapter 6

Provenance Algorithms

In this chapter we introduce the *Provenance Construction* (Section 6.1) and the *Update Reconstruction* (Section 6.2) algorithms, as well as their correctness results (Section 6.3) and their complexity analysis (Section 6.4). The first algorithm (Algorithm 1 in Section 6.1) is used to record the provenance of quadruples resulting from a SPARQL INSERT update. This algorithm takes as input an INSERT update U and a Graph Store $\mathcal{GS}$, and returns a provenance expression p_i to associate with each newly created quadruple q_i. Each provenance expression p_i is expressed under the semantics of the proposed model (Chapter 5).

The second algorithm (Algorithm 3 in Section 6.2), provides the means to exploit the rich semantics of the provenance expression of a quadruple in order to determine how the quadruple found its way in the Graph Store. In particular, this algorithm takes as input a complex provenance expression *cpe* that is part of the provenance of the input quadruple q and returns a compatible INSERT update U'. It is worth noting the fact that the algorithm requires only a complex provenance expression, instead of the full provenance, since a *cpe* is the minimum computed provenance result of an INSERT update and therefore it is quite enough to be used for the reconstruction of another INSERT update.

In Section 6.3, we present the correctness theorems of the above algorithms. More specifically, Theorem 1 is used to prove the reciprocal relationship between two compatible UNION-free INSERT updates. Furthermore, in Theorem 2 we prove that the output U' of Algorithm 3 is *compatible* (see Definition 15) with the INSERT update U that was used to create q in the first place. This theorem is also a correctness theorem, as it shows that the intended semantics of the provenance model are correctly implemented by Algorithm 1 and utilized by Algorithm 3.

Finally, in the last section of this chapter (Section 6.4), we discuss the complexity of *provenance construction* and *update reconstruction* algorithms.

6.1 Provenance Construction Algorithm

As shown in Algorithm 1, to compute the provenance p_k (Definition 9) of a newly created quadruple q_k, we have to compute the corresponding complex provenance expressions cpe generated via the update U. Recall that the provenance p of a single quadruple is of the form $p = \{cpe_1, \ldots, cpe_j\}$, where $cpe = pe^1 \oplus \ldots \oplus pe^m$. Hence, for each graph pattern gp^i of the WHERE clause we call the algorithm PE_COMPUTATION, which computes the individual simple provenance expressions pe^i. The pe^i expressions are then used to form an expression cpe that is appended to the provenance p of a quadruple q. For readability purposes, we define:

- $PE^i = \{(q_1, pe^i_{1\_1}), (q_1, pe^i_{1\_2}) \ldots (q_j, pe^i_{j\_l-1}), (q_j, pe^i_{j\_l})\}$,
  where $pe^i_{k\_m}$ is the m^{th} simple provenance expression that created using the graph pattern gp^i for the quadruple q_k. Note that there may be created more than one pe^i_k expressions for a quadruple q_k forming its corresponding cpe_k expression.
- $CPE = \{(q_1, cpe_{1\_1}), (q_1, cpe_{1\_2}) \ldots (q_j, cpe_{j\_l-1}), (q_j, cpe_{j\_l})\}$,
  where cpe_{k_r} is the r^{th} complex provenance expression created for the quadruple q_k. Note that there may be created more than one cpe_k expressions for a quadruple q_k forming its provenance p_k.
- $P = \{(q_1, p_1), \ldots (q_j, p_j)\}$,
  where p_k is the provenance of quadruple q_k

Moreover, we define the following operations between them:

- $CPE \oplus PE^i$
  This operation appends each simple provenance expression $pe^i_{k\_m}$ of PE^i to the corresponding cpe_{k_r} expression, e.g., $\{(q_1, cpe_{1\_1})\} \oplus \{(q_1, pe^i_{1\_1})\} = \{(q_1, cpe_{1\_1} \oplus pe^1_{1\_1})\}$.
- $P \cup CPE$
  This operation appends each complex provenance expression cpe_{k_r} to the

corresponding provenance p_k, e.g., $\{(q_1, p_1)\} \cup \{(q_1, cpe_{1\_1})\} = \{(q_1, p_1 \cup cpe_{1\_1})\}$.

Algorithm 1 Provenance Construction Algorithm

Input: An INSERT update U, a Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$)
Output: The provenance p_k of each result quadruple q_k, P

for all ($gp^i \in$ WHERE clause) **do**
  PE^i = PE_COMPUTATION(gp^i, qp_{ins}, $\mathcal{GS}$)
  $CPE = CPE \oplus PE^i$
return $P \cup CPE$

The algorithm PE_COMPUTATION (see Algorithm 2), which is the main algorithm of the provenance construction, is used to compute the provenance of the subject, predicate and object attributes for each result quadruple of the update U.

We will explain how this is done for an arbitrary attribute (specified by *pos*) but, as shown in Algorithm 2 (line 1), we follow the same process for the provenance computation of subject ($pos = s$), predicate ($pos = p$) and object attribute ($pos = o$). For the rest of this Section we will consider for our examples the update U and the Graph Store $\mathcal{GS}_2$, presented in our Motivating Example (Chapter 3).

To compute the provenance of the attribute *pos* we examine the value of $qp_{ins}.pos$. Recall that the attribute *pos* of a result quadruple is generated from the evaluation of the corresponding position in the INSERT clause ($qp_{ins}.pos$). The value of $qp_{ins}.pos$ can be either a constant or a variable. In the first case (line 15), the provenance computation of attribute *pos* ($prov_{pos}$) is quite simple, since we only assign to it the special label $\perp$ (line 16) and we proceed to the provenance computation of the next attribute (if any).

Example 28. The quad pattern position $qp_{ins}.s$ of U (Chapter 3) contains the constant value <hypertension>. Then, the provenance of attribute s is $prov_s = \perp$ both in case of gp^1 or gp^2 input. □

In the second case (line 2), the computation of provenance is more complicated, as we have to evaluate the *gp* parameter and identify the joins (if any) that were involved in the construction of a quadruple (lines 2-14).

As a first step in the latter case, we determine the *MatchingPatterns* set (line 3). This set contains the quad pattern identifiers that appear in the input graph pattern gp (mp_j denotes the j^{th} quad pattern identifier in the set) and are related directly or indirectly to the evaluation of the variable in $qp_{ins}.pos$. A quad pattern is *directly* related to the evaluation of a variable, if any of its positions contains this specific variable, or *indirectly*, if any of its positions joins (implicitly, via another variable, or explicitly) a position in a quad pattern that contains the evaluated variable.

Example 29. Consider the graph pattern gp^1: qp_1^1 . qp_2^1 of the INSERT update U (Chapter 3). The created $MatchingPatterns$ set is $\{mp_1, mp_2\}$, where mp_1, mp_2 denote the quad patterns qp_1^1 and qp_2^1, respectively. Note that the $MatchingPatterns$ set is the same both in case of the variable *?p* ($qp_{ins}.p$) and *?o* ($qp_{ins}.o$). In the first case the variable *?p* is contained in qp_1^1 and qp_2^1 is related indirectly to it, since it joins implicitly the variable *?o*. In the second case the variable *?o* is contained in qp_1^1 and qp_2^1 is related directly to it, since qp_2^1 contains also this variable.

In the same manner, we compute that $MatchingPatterns$ set is $\{mp_1\}$, where mp_1 denotes the quad pattern qp_1^2, both for variables *?p* and *?o*, if gp^2 is given as input. □

In the simple case that *MatchingPatterns* set has only one element, then we have no joins, i.e. we have a "copy" operation. Then, it is sufficient to compute the quadruple identifiers (using the $findIDs$ function) that result from the evaluation of the variable in $qp_{ins}.pos$ (line 4) and the *var subscript* (line 13). Each quadruple identifier forms a new *spe* expression that entails the creation of different $prov_{pos}$ expressions, e.g., in Example 24 we create a different *spe* expression for each of c_5 and c_6. The *var subscript* value is computed as defined in Chapter 5.

Eventually, the provenance of the attribute *pos* (line 14) for a "copy" operation is of the form:

$$prov_{pos} =_{mp_1} (c_a)$$

where $varSub = mp_1$ and $spe = c_a$, with c_a belonging to the quadruple identifiers result of $findIDs$ function (line 5).

Example 30. Consider the $MatchingPatterns$ set of gp^2, created in the previous example, which contains only one element ($\{mp_1\}$). We apply the $findIDs$ function to mp_1 and we get from the evaluation of qp_1^2 the quadruple identifier c_4; this is the evaluation result both in case of $qp_{ins}.p$ or $qp_{ins}.o$.

The var subscripts are $qp_1^2.p$ and $qp_1^2.o$ respectively for $prov_p$ and $prov_o$. As a consequence, we create the expression $pe^2_{1\_1} = (\bot, {}_{qp_1^2.p}(c_4), {}_{qp_1^2.o}(c_4))$. Note that pe^2 and $pe^2_{1\_1}$ refer actually to the same expression. Then, we use the $getQuad$ function to get the quadruple q_1 (<hypertension>, <medication>, <diuretics>, <NewDoctor>). Eventually, the output of PE_COMPUTATION regarding gp^2 is $\{(q_1, pe^2_{1\_1})\}$. □

In the more complex case, where *MatchingPatterns* has more than one elements, we have to identify the corresponding JOIN expressions and record the related joins, by iterating over them and recording the involved quadruple identifiers and the quad pattern positions (in the form of join subscripts– see Chapter 5) where the joins take place (lines 7-12). A JOIN expression is of the form $joinOp_1$. $joinOp_2$, where $joinOp_1$ and $joinOp_2$ are graph patterns denoting the first and second operand of the join operation. By convention, we identify the JOIN expressions sequentially based on their occurrence order in the WHERE clause (lines 8, 10, 11).

As already mentioned, for each JOIN expression we have to compute the corresponding join subscripts (line 9) and quadruple identifiers. We can easily compute join subscripts just by looking at the common variables of $joinOp_1$, $joinOp_2$ (see Chapter 5 for details); quadruple identifiers are computed using the $findIDs$ function (line 10). The computed *spe* is used to form the final provenance result of the algorithm for the specific position. Note that we create a different *spe* expression for each quadruple identifiers combination. For instance, consider the combination $[c_1]\ {}_{joinSub^1} \odot {}_{joinSub^2}\ [c_2, c_3]$, then we create two *spe* expressions for this position, $c_1\ {}_{joinSub^1} \odot {}_{joinSub^2}\ c_2$ and $c_1\ {}_{joinSub^1} \odot {}_{joinSub^2}\ c_2$.

Eventually, the provenance of attribute *pos* (line 14) for a join operation is of the form:

$$prov_{pos} =_{mp_k} ((c_a\ {}_{\{joinSub^1\}} \odot {}_{\{joinSub^2\}}\ c_b) \cdots {}_{\{joinSub^{r-1}\}} \odot {}_{\{joinSub^r\}}\ c_d)$$

where $spe = (c_a\ {}_{\{joinSub^1\}} \odot {}_{\{joinSub^2\}}\ c_b) \cdots {}_{\{joinSub^{r-1}\}} \odot {}_{\{joinSub^r\}}\ c_d$ (line 10) and $varSub = mp_k$ (line 13). Note that we create a $prov_{pos}$ for each different

Algorithm 2 PE_COMPUTATION

Input: A graph pattern gp, the Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$), the quad pattern qp_{ins} of U

Output: The pe_{k_m} expressions for each q_k quadruple, $\{(q_1, pe_{1\_1}), (q_1, pe_{1\_2}) \dots (q_j, pe_{j\_l})\}$

for all $qp_{ins}.pos$ **do**
  if valueOf($qp_{ins}.pos$) $\in \mathbb{V}$ **then**
    Create the set $MatchingPatterns$ $\{mp_1, mp_2 \dots mp_x\}$
    spe = FINDIDS(mp_1)
    Let $joinOp_1$, $joinOp_2$ be the two operands of a JOIN expression; $joinOp_1 = mp_1$, $joinOp_2 = null$
    j = 1
    while $mp_{j+1} \neq null$ **do**
      $joinOp_2 = mp_{j+1}$
      Create the $joinSub^1$ and $joinSub^2$
      $spe = spe\ {}_{joinSub^1} \odot_{joinSub^2}$ FINDIDS(mp_{j+1})
      $joinOp_1 = joinOp_1\ .\ joinOp_2$
      j++
    Create the $varSub$
    $prov_{pos} = {}_{varSub}\ (spe)$
  else
    $prov_{pos} = \bot$
$pe = (prov_s, prov_p, prov_o)$

for all created pe_k **do**
  q_k = GETQUAD(pe_k, qp_{ins})
return $\{(q_1, pe_{1\_1}), (q_1, pe_{1\_2}) \dots (q_j, pe_{j\_l})\}$

spe.

Finally, we combine the computed provenance for subject, predicate and object attributes to create a pe expression. Each different combination of $prov_s$, $prov_p$, $prov_o$ requires the creation of a new pe expression.

Example 31. Consider the $MatchingPatterns$ for gp^1, created in the Example 29, which contains the elements mp_1 and mp_2. Using the function $findIDs$, we get that the quadruple identifiers resulted from the evaluation of mp_1 (qp_1^1) are c_1 and c_2. Afterwards, we identify the only existing JOIN expression for $qp_{ins}.p$, where $joinOp_1 = mp_1$ (qp_1^1) and $joinOp_2 = mp_2$ (qp_2^1); the JOIN expression is the same in case of $qp_{ins}.o$ as well. Following the semantics of our model, we compute the join subscripts, $joinSub^1 = \{qp_1^1.o\}$ and $joinSub^2 = \{qp_2^1.s\}$ and we apply once again the $findIDs$ function to compute the quadruple identifiers for mp_2 (qp_2^1), c_3. As presented in Table 3.5, only c_1 and c_3 meet the evaluation requirements of the join between $joinOp_1$. $joinOp_2$. Therefore, the created *spe* expression for both $qp_{ins}.p$ and $qp_{ins}.o$ is $c_1 \;_{\{qp_1^1.o\}} \odot \;_{\{qp_2^1.s\}}\; c_3$.

The computed var subscripts, $qp_1^1.p$ and $qp_1^1.o$, are, then, used to form the corresponding *pe* expression, $pe_{1\_1}^1 = (\bot,\; _{qp_1^1.p}\; (c_1 \;_{\{qp_1^1.o\}} \odot \;_{\{qp_2^1.s\}}\; c_3)\;,\; _{qp_1^1.o}\; (c_1 \;_{\{qp_1^1.o\}} \odot \;_{\{qp_2^1.s\}}\; c_3))$. Note that $pe_{1\_1}^1$ and pe^1 represents the same expression. Then, we use $getQuad$ to get the quadruple q_1 (<hypertension>, <medication>, <diuretics>, <NewDoctor>). Eventually, the output of $PE\_COMPUTATION$ regarding gp^1 is $\{(q_1,\, pe_{1\_1})\}$.

Going back to Algorithm 1, we get that $PE^1 = \{(q_1,\, pe_{1\_1}^1)\}$ (based on the output of Algorithm 2 for gp^1– see this example) and $PE^2 = \{(q_1,\, pe_{1\_1}^2)\}$ (based on the output of Algorithm 2 for gp^2– see Example 30). Then, PE^1 and PE^2 are combined through the union operator $\oplus$ setting thereby $CPE = \{(q_1,\, cpe_{1\_1}\}$, where $cpe_{1\_1} = pe_{1\_1}^1 \oplus pe_{1\_1}^2$. Finally, the output of *provenance construction* algorithm is $P = \{(q_1,\, cpe_{1\_1})\}$. □

6.2 Update Reconstruction Algorithm

As already mentioned, the purpose of the reconstruction algorithm is to output a SPARQL update U', which is *compatible* with the original update that created the input quadruple. Theorem 2 (see Section 6.3), which is a correctness theorem, is used to prove this claim. Before proceeding to the presentation of algorithm, we formally define the *filter-compatible graph patterns* and the *compatible* INSERT *updates*:

Definition 14. *Let gp and gp′ be graph patterns. We say that gp′ is filter-compatible to gp (denoted* $gp \sim gp'$*) iff gp′ differs from gp only in the filters that it may employ.*

Note that Definition 14 refers as well to implicit filters created by a constant value in the WHERE clause, e.g., “glucose” in qp_2^1 of our Motivating Example.

Definition 15. *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'^i in U' there is a filter-compatible gp^i in U.*

Reconstructing an INSERT update requires both the quad pattern qp_{ins} of the INSERT clause and the graph pattern gp of the WHERE clause. For the former, we consider the global quad pattern qp'_{ins}, which represents the quad pattern in the INSERT clause of the compatible update U'; qp'_{ins} gets its values during the execution of Algorithms 3, 4. For the latter, we use the Algorithm UPD_RECONSTRUCTION that utilizes the pe^i expressions of cpe to reconstruct the individual graph patterns of gp'. Towards a better understanding of context we will provide in line examples considering the provenance p_7 of quadruple c_7 and the Graph Store $\mathcal{GS}_2$ ($\mathcal{Q}_{\mathcal{GS}_2}$, $\mathcal{N}_{\mathcal{GS}_2}$), presented in our Motivating Example (Chapter 3). Recall that c_7: (<hypertension>, <medication>, <diuretics>, <NewDoctor>) and $p_7 = \{cpe_1\}$, where $cpe_1 = pe^1 \oplus pe^2$, $pe^1 = (\bot, {}_{qp^1_1.p}(c_1 \; {}_{\{qp^1_1.o\}} \odot {}_{\{qp^1_2.s\}} \; c_5), {}_{qp^1_1.o}(c_1 \; {}_{\{qp^1_1.o\}} \odot {}_{\{qp^1_2.s\}} \; c_5))$ and $pe^2 = (\bot, {}_{qp^2_1.p}(c_6), {}_{qp^2_1.o}(c_6))$.

Algorithm 3 Update Reconstruction Algorithm

Input: A complex provenance expression cpe of the form $pe^1 \oplus \ldots \oplus pe^k$, a quadruple q (s, p, o, n), a Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$)
Output: An INSERT update U'

1: Let $qp'_{ins} = (tp'_{ins}, n)$
2: **for all** pos **do**
3: $\quad qp'_{ins}.pos =$ NEWVAR( )
4: **for all** $pe^i \in cpe$ **do**
5: $\quad gp^i =$ UPD_RECONSTRUCTION(pe^i, q, $\mathcal{GS}$, qp'_{ins})
6: $\quad gp' = gp'$ UNION gp^i
7: $U' =$ INSERT $\{qp'_{ins}\}$ WHERE $\{gp'\}$

As shown in Algorithm 3, we can determine the graph attribute (n) of qp'_{ins} using the fourth attribute of the input quadruple q (line 1). For example, we can determine the graph <NewDoctor> from c_7. Then, we spawn a new variable for each position of qp'_{ins} (lines 2,3), e.g., qp'_{ins} = (*?v1*, *?v2*, *?v3*, <NewDoctor>).

The UPD_RECONSTRUCTION (Algorithm 4) is called for each pe^i expression to reconstruct the corresponding graph pattern gp^i (lines 4-6). The individual graph patterns gp^i, then form the graph pattern gp' in the WHERE clause of U'.

As a first step of Algorithm 4, we compute the var subscript that exists in each $prov_{pos}$ and assign to it the value of $qp'_{ins}.pos$. Note that if $prov_{pos} = \perp$, then there is no var subscript to be determined because this attribute has been created through the assignment of a constant value.

Example 32. In our Motivating Example, the computed var subscripts for $prov_p$, $prov_o$ of pe^1 are $qp_1^1.p$ and $qp_1^1.o$, respectively. Then, we set $qp_1^1.p = qp'_{ins}.p = ?v2$ and $qp_1^1.o = qp'_{ins}.o = ?v3$. Similarly, we compute the var subscripts $qp_1^2.p$, $qp_1^2.o$ for $prov_p$ and $prov_o$, respectively in pe^2 expression. As a result, $qp_1^2.p = qp'_{ins}.p = ?v2$ and $qp_1^2.o = qp'_{ins}.o = ?v3$. Note that the attribute provenance $prov_s$ is not associated to any var subscript. □

Subsequently, we create the *SubsPatterns* set (line 4). This set contains the different quad pattern identifiers (sp_m denotes the m^{th} quad pattern identifier in the set) that appear in the subscripts of all $prov_{pos}$ in the input pe^i. As defined earlier, though, $prov_{pos}$ is either of the form $\perp$ or ${}_{varSub}(spe)$ (Definition 12).

If $prov_{pos}$ is of the first form, then there is no quad pattern to be identified. Otherwise, we determine the quad pattern identifiers by checking the subscripts of spe (join subscripts) and afterwards the $varSub$ (var subscript). Note, however, that we ignore multiple instances of the same quad pattern identifier, i.e. each quad pattern identifier exists only once in *SubsPatterns*, and that we take into account the occurrence order of the quad patterns, i.e. *SubsPatterns* is an ordered set. Moreover, note that each element of *SubsPatterns* indicates a quad pattern in the output gp^i.

Example 33. Considering our Motivating Example, if pe^1 is the given input, then $SubsPatterns$ set is $\{sp_1, sp_2\}$, where sp_1, sp_2 identify qp_1^1 and qp_2^1, respectively. On the contrary, if pe^2 is the given input, then $SubsPatterns = \{sp_1\}$, with qp_1^2 being identified by sp_1. □

In addition, we create the ordered set *PeGraphs* (line 5) that contains the graphs implied by the quadruple identifiers of pe^i expression. In more detail, for each quadruple identifier existing in pe^i we identify and record its corresponding graph. As with *SubsPatterns* set, we take into account only the first occurrence of a graph.

Example 34. Back to our Mmotivating Example, the pe^1 expression contains the quadruple identifiers c_1, c_3, and therefore $PeGraphs$ = {<Pathologist>, <Side_Effects>}. In the same manner, we compute that $PeGraphs$ is equal to {<Diabetologist>} for pe^2 expression, because of the existence of c_4. □

Algorithm 4 UPD_RECONSTRUCTION

Input: A simple provenance expression pe^i ($prov_s$, $prov_p$, $prov_o$), a quadruple q (s, p, o, n), a Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$)
Output: A graph pattern gp^i

for all $prov_{pos}$ **do**
  $varSub$ = GETVARSUBSCRIPT($prov_{pos}$)
  valueOf($varSub$) = valueOf($qp'_{ins}.pos$)
Create the set $SubsPatterns$ $\{sp_1, sp_2, \ldots, sp_l\}$
Create the set $PeGraphs$ $\{n_a, n_b, \ldots, n_d\}$
ASSIGNGRAPHS($SubsPatterns$, $PeGraphs$)
for all $prov_{pos} \in pe^i$ **do**
  if $prov_{pos} \neq \perp$ **then**
    Create the set $JoinSubs$ $\{joinSub^1, joinSub^2, \ldots, joinSub^{x-1}, joinSub^x\}$
    Let $joinSub^r$ be the r^{th} element in $JoinSubs$, and jp^r_k be the k^{th} element of $joinSub^r$
    $r = 1$ $k = 1$
    while $joinSub^r \neq$ null **do**
      while $jp^r_k \neq$ null **do**
        if valueOf(jp^r_k) = null **then**
          valueOf(jp^r_k) = NEWVAR( )
        valueOf($jp^{(r+1)}_k$) = valueOf(jp^r_k)
        k++
      r = r+2
  else
    valueOf($qp'_{ins}.pos$) = valueOf($q.pos$)
for all $sp_m \in SubsPatterns$ **do**
  $UnboundPos$ = GETUNBOUNDPOS(sp_m)
  for all $qp^i_j.pos \in UnboundPos$ **do**
    $qp^i_j.pos$ = NEWVAR( )
$gp^i = qp^i_1 \,.\, qp^i_2 \,.\, \ldots \,.\, qp^i_l$
return gp^i

So far, we know the quad patterns ($SubsPatterns$) that constitute the output graph pattern gp^i and the graphs ($PeGraphs$) appearing in them. Thus, since the two sets are ordered, we can properly relate a quad pattern with the correct graph by applying the following simple rule: the k^{th} graph of $PeGraphs$ is assigned to the graph attribute of the k^{th} quad pattern of the $SubsPatterns$ set; this is done using the $assignGraph$ function (line 6).

Example 35. Applying the $assignGraph$ function for pe^1 and pe^2 of our Motivating Example, results $qp_1^1 = (tp_1^1,$ <Pathologist>), $qp_2^1 = (tp_2^1,$ <Side_Effects>) and $qp_1^2 = (tp_1^2,$ <Diabetologist>), respectively. □

At this point, we have to compute the values that appear in the s, p, o positions of each created quad pattern. Hence, we exploit the information provided by the provenance of each attribute ($prov_s$, $prov_p$, $prov_o$). We will explain how this is done for an arbitrary attribute (specified by pos) but, as shown in line 4, the process is identical for the subject ($pos = s$), predicate ($pos = p$) and object ($pos = o$) attribute.

If $prov_{pos} = \perp$ (line 19), then the attribute pos of quadruple q was created via a constant value. As a consequence, we override the value of $qp'_{ins}.pos$ and set it to be the same as the value of this attribute in the input quadruple q (line 20). For example, consider $prov_s$ both in pe^1 and pe^2. In that instance, we set the value of $qp'_{ins}.s$ to be equal to <hypertension>.

On the contrary, if $prov_{pos} = {}_{varSub}(spe)$ (line 8), then the attribute pos of quadruple q was created via a construction. Hence, we have to determine if the construction was the result of a "copy" or a join operation (see Chapter 5 for details). To figure out the kind of operation we use the $JoinSubs$ set (line 9). As it is implied by its name, this set contains the join subscripts (denoted as $joinSub^1$, ...) that appear in the current $prov_{pos}$. In the simple case that $JoinSubs$ has no elements, we have a "copy" operation and the block in lines 10-18 will be skipped. Hence, the var subscript value is sufficient to indicate the variable that appear in this position.

Example 36. The attribute provenances $prov_p$ and $prov_o$ of pe^2 expression in our Motivating Example witness that the predicate and object attributes of c_7 have been constructed via a "copy" operation. Then, the corresponding quad pattern positions $qp_1^2.p$ (*?v2*) and $qp_1^2.o$ (*?v3*) have already assigned to a variable via the var subscripts computation. □

In the more complex case, where $JoinSubs$ contains some elements, we process them in order to appropriately set the variables of the quad patterns so that those that are involved in a join to have common variable names (line 10). Recall that a join subscript is a set of quad pattern positions that participate in a join, and that each JOIN expression requires two join subscripts to be represented.

Assume that jp_k^r denotes the k^{th} element of $joinSub^r$, then the element jp_k^r joins the element jp_k^{r+1}; $joinSub^r$ and $joinSub^{r+1}$ have always the same number of elements. If jp_k^r has already an assigned variable name, it is implied that jp_k^r participates as well in the provenance of other attributes that have been already processed or it determines a var subscript. Otherwise, we use the function $NewVar$ to spawn a new variable name and assign it to jp_k^r (lines 14-16).

Example 37. Unlike pe^2 (see previous example), $prov_p$ and $prov_o$ of pe^1 expression indicate that the predicate and object attributes of c_7 have been constructed via join operations. Then, we create the $JoinSubs$ set that is both for $prov_p$ and $prov_o$ equal to $\{joinSub^1, joinSub^2\}$, where $joinSub^1 = \{qp_1^1.o\}$ and $joinSub^2 = \{qp_2^1.s\}$. This implies that $qp_1^1.o$ joins $qp_2^1.s$. Since, $qp_1^1.o$ has an assigned variable already (*?v3*), we set $qp_2^1.s = qp_1^1.o =$ *?v3*. □

Until now, we have assigned variable names to any quad pattern position that is related somehow to a $prov_{pos}$. However, *unbound* quad pattern positions may exist. A quad pattern position is called unbound, if it has not been assigned any variable name. To find the unbound quad pattern positions, we search the created quad patterns using the *getUnboundPos* function (line 22). The output of this function is the *UnboundPos* set. In our example, *UnboundPos* $= \{qp_1^1.s,\ qp_2^1.p,\ qp_2^1.o,\ qp_1^2.s\}$. Then, each element of this set is being assigned a "fresh", random variable (lines 24).

Finally, we combine the created quad patterns into a big join that forms the returned graph pattern gp^i (line 25). In our example, the reconstructed compatible update is U':

$$\textsf{INSERT}\ \{qp'_{ins}\}\ \textsf{WHERE}\ \{qp_1^1\ .\ qp_2^1\ \textsf{UNION}\ qp_1^2\}$$

where:

qp'_{ins}: (<hypertension>, *?v2*, *?v3*, <NewDoctor>)
qp^1_1: (*?v4*, *?v2*, *?v3*, <Pathologist>)
qp^1_2: (*?v3*, *?v5*, *?v6*, <Side_Effects>)
qp^2_1: (*?v7*, *?v2*, *?v3*, <Diabetologist>)

Note that U' differs from the INSERT update U of our Motivating Example only in the filters that U employs ("glucose" in qp^1_2 and <hypertension> in qp^2_1) as well as in their syntactic form (i.e. the variable names).

6.3 Correctness Results

As a consequence of the definition of compatible INSERT updates (Definition 15), 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' ($U' \rightsquigarrow U$).*

Proof. Assume that U is of the form U: INSERT $\{qp_{ins}\}$ WHERE $\{gp^1\}$ and U' is of the form U': INSERT $\{qp'_{ins}\}$ WHERE $\{gp'^1\}$. If U' is compatible to U, then it is implied that there is a renaming such as $qp_{ins} = qp'_{ins}$ and $gp^1 \sim gp'^1$ (definition of *compatible* INSERT updates). However, the definition of *filter-compatible graph patterns* (Definition 14) implies that $gp'^1 \sim gp^1$ as well. Then, $qp'_{ins} = qp_{ins}$ and $gp'^1 \sim gp^1$, and therefore U is a compatible INSERT update to U' ($U' \rightsquigarrow U$). □

Lemma 1. *Let U be an* INSERT *update and U' be a compatible* INSERT *update of it. U' was created via the Update Reconstruction algorithm with given input (cpe, q, $\mathcal{GS}$), where q (s, p, o, n) is a result quadruple of U, cpe is a complex provenance expression that belongs to the provenance of q (as computed by the Provenance Construction algorithm) and $\mathcal{GS}$ is the Graph Store where U was evaluated against. Then, U' differs from U in its syntactic form (variables' names) and in the filter conditions that U may employ.*

Intuitively, we want to prove that U' contains a consistent renaming of the variables that appear in the quad pattern positions of U. For example, assume that

$valueOf(qp_{ins}.p) = valueOf(qp_2^1.s) = ?x$ in U, then we will prove that $valueOf$ $(qp_{ins}.p) = valueOf(qp_2^1.s) = ?y$ in U'. Note that variables names are insignificant since they play no role in the evaluation process.

Proof. Following the semantics of our proposed model (see Section 4), we consider the following forms for U, U', cpe and pe:

- U: INSERT $\{qp_{ins}\}$ WHERE $\{gp\}$
- U': INSERT $\{qp'_{ins}\}$ WHERE $\{gp'\}$
- $cpe := pe^1 \oplus pe^2 \ldots \oplus pe^m$
- $pe := (prov_s, prov_p, prov_o)$, where $prov_{pos}$ is the provenance of attribute pos

We distinguish different cases based on the cpe format to prove the correctness of Lemma 1.

1. $cpe := pe^1$ or simply $cpe := pe$

   This is the case of UNION-free INSERT updates. In this case, we have to examine the provenance of each constituent of pe ($prov_{pos}$) to determine potential differences between U and U'. The attribute provenance $prov_{pos}$ may have one of the following forms:

   a. $prov_{pos} := \bot$

      This case implies that the attribute pos has been created through the assignment of a constant value. However, the value of attribute pos in a result quadruple q is determined through the evaluation of $qp_{ins}.pos$ and therefore $valueOf(q.pos) = valueOf(qp_{ins}.pos)$ (line 20 in Algorithm 2). Additionally, every result quadruple q' of U' will have the same value in pos attribute as the quadruple q since $valueOf(qp'_{ins}.pos) = valueOf(q.pos)$ (line 20 of Algorithm 4). Then, $qp'_{ins}.pos$ and $qp_{ins}.pos$ will have the same value in the specific position of the INSERT clause. As a result, U and U' will always return exactly the same value for the attribute pos no matter what variables exist in the WHERE clause.

   b. $prov_{pos} := {}_{varSub}(spe)$

      This case implies that the attribute pos has been constructed through a "copy" or a join operation. By definition the var subscript ($varSub$) represents the first quad pattern position, $qp_j^i.pos_2$, in the WHERE clause that shares the same variable with $qp_{ins}.pos_1$, i.e., $valueOf(qp^i_j.pos_2) = valueOf(qp_{ins}.pos_1)$ (see Section 4 for details). Line 13 of Algorithm 2 guarantees that. In addition, line 3 of Algorithm 4 assures that the quad pattern position $qp'^k_l.pos_4$, denoted by the *varSub*, will have the same value as $qp'_{ins}.pos_3$, i.e., $valueOf\ (qp'^k_l.pos_4) = valueOf(qp'_{ins}.pos_3)$. Moreover, lines 2 (Algorithm 4), 14 (Algorithm 2) imply that $qp^i_j.pos_2 = qp'^k_l.pos_4$, i.e., $i = k$, $j = l$ and $pos_2 = pos_4$, and $qp_{ins}.pos_1 = qp'_{ins}.pos_3$, i.e., $pos_1 = pos_3$. Therefore, $qp'_{ins}.pos_3$, $qp_{ins}.pos_1$ and $qp'^k_l.pos_4$, $qp^i_j.pos_2$ refer to the same quad pattern positions and differ only in the variables' names that they employ. As a consequence, we have to examine the different forms of *spe*:

i. $spe := c_i$

This is the case of "copy" operation. In this case, there is only one quad pattern position in the WHERE clause that contains the same variable with $qp_{ins}.pos_1$ and it is mapped to a constituent of c_i through the evaluation process (lines 4, 20 of Algorithm 2). Since this quad pattern position is unique it will coincide with the *varSub* $qp^i_j.pos_2$, which has already been proved that refers to the same quad pattern position as $qp'^k_l.pos_4$.

ii. $spe := (c_a\ {}_{joinSub^1} \odot {}_{joinSub^2}\ c_b) \ldots {}_{joinSub^{x-1}} \odot {}_{joinSub^x}\ c_d$

This is the case of a join operation. A $joinSub^r$ is a set of quad pattern positions that participate in a join. Then, two *join subscripts* (e.g. $joinSub^{r-1}$, $joinSub^r$) are used to describe the existing joins between two operands of a JOIN expression; the values of the corresponding quad pattern positions in the two sets have to be equal (see Section 4 for details). In Algorithm 4, lines 9-18 claim the previous statement, whereas Algorithm 2 ensures it in lines 5-12. Moreover, line 9 in Algorithm 2 and lines 9-10 in Algorithm 4 assert that the join subscripts of U and U' will refer exactly to the same quad pattern positions.

Until now, we have proved that each quad pattern position of INSERT and WHERE clause of U that is associated somehow with an attribute provenance $prov_{pos}$ of pe, will also appear in the INSERT or WHERE clause of U'. Nevertheless, the same quad pattern positions may have different variables' names in U and U'. The rest of quad pattern positions of U may contain a constant value or a variable. These positions are being characterized as *unbound quad pattern positions* in U'. Then, we distinguish the following cases:

A. An unbound position of U' contains a constant value in U
This is a filter condition. According to Algorithm 4 every unbound quad pattern position is being assigned a new random variable (line 24). Then, U' will return for this quad pattern position the maximum number of results that match this variable including the constant value too.

B. An unbound position of U' contains a variable in U
Following the previous consideration we have that an unbound position of U' is being assigned a new random variable (line 24 of Algorithm 4). Then, U' will return for this quad pattern position the same evaluation results as U.

2. $cpe := pe^1 \oplus pe^2 \ldots \oplus pe^m$
A *cpe* expression of this form consists of individual simple provenance expressions (pe^x) that are constructed through Algorithm 4 and combined using the operator $\oplus$ (lines 2,3 of Algorithm 1). Then, the proof for this form is traced back to the previous case.

Eventually, we conclude that U' is a filter-free version of U with respect to *cpe* that may differ from it in the variables' names that they employ. □

Corollary 1. *Let U be an* INSERT *update and U' be a compatible* INSERT *update of it, created via the* Update Reconstruction *algorithm with given input (cpe, q, $\mathcal{GS}$); q (s, p, o, n) is a result quadruple of U, cpe is a complex provenance expression that belongs to the provenance of q (as computed by the* Provenance Construction *algorithm) and $\mathcal{GS}$ is the Graph Store where U was evaluated against. Let also Q_U and $Q_{U'}$ be the result sets of U and U' respectively. Then $q \in Q'_U$.*

Proof. As a consequence of Lemma 1, U' returns a set of quadruples ($Q_{U'}$) that contains all quadruples of the result set of U (Q_U) that are related to at least one simple provenance expression pe^i of cpe; q is related to every pe^i as implied by the hypothesis of this corollary. As a result, $q \in Q_{U'}$. □

The following theorem (Theorem 1) proves that the output of Algorithm 3 in the previous Section is compatible with the original INSERT update that created the input quadruple. Thus, the intended semantics of a provenance expression, as given in Section 5, are correctly recorded by Algorithm 1 (Section 6.1), and interpreted by Algorithm 3 (Section 6.2).

Theorem 2. *Let U be an* INSERT *update evaluated on the Graph Store $\mathcal{GS}$ ($\mathcal{Q}_{\mathcal{GS}}$, $\mathcal{N}_{\mathcal{GS}}$), q a result quadruple and cpe a complex provenance expression that belongs to the provenance of q as computed by the Provenance Construction Algorithm. Assume that we run the Update Reconstruction Algorithm with input (cpe, q, $\mathcal{GS}$) and we get as output the* INSERT *update U'. Then, U' returns q among other quadruples and $U \rightsquigarrow U'$.*

Proof. In Corollary 1 we have proved that q belongs to the result set of U and U' as well. Then, it is sufficient to prove that U' is a compatible INSERT update to U. By definition, an INSERT update U' is compatible to an INSERT update U if there is a renaming of variables in U', such as $qp'_{ins} = qp_{ins}$ and for each gp'^i in U' there is a filter-compatible gp^i in U (Definition 5). In Lemma 1 we proved that U' is a filter-free version of U with respect to cpe and these two updates may differ only in their variables names. Consequently, we prove that $U \rightsquigarrow U'$. □

6.4 Complexity Analysis

The complexity of *Provenance Construction* algorithm (Algorithm 1) is considered with respect to *a) the update size* and *b) the size of the input Graph Store.* The update size refers to the number of quad patterns in the WHERE clause. The complexity regarding this parameter is linear, namely $O(m)$ where m is the number of quad patterns. To see this, note that we have to execute lines 2-17 of Algorithm 2 three times, where each execution running for one evaluated position of qp_{ins} (s,p,o). Each of these runs costs $O(m_i)$, where m_i is the number of quad patterns in the input gp^i that participate in a join. The algorithm runs for all qp^i of the WHERE clause, so, in the worst-case, where all quad patterns are involved in joins, we have that the total computational cost is $O(3 \cdot \sum_i m_i) = O(m)$.

The size of the Graph Store refers to the number of quadruples that exist in the Graph Store, more specifically in $\mathcal{Q}_{\mathcal{GS}}$, where the input INSERT update will be evaluated. In this case, the complexity is $O(logR)$, where R is the number of quadruples that exist in the Graph Store. More specifically, we need $O(logR)$ time to compute the corresponding quadruple identifiers resulting from the evaluation of a quad pattern, assuming that quadruples have been sorted based on their identifier (binary search). Additionally, we need three accesses in the Graph Store to compute the s, p, o attributes of each quadruple; each access in the Graph Store costs $O(logR)$ time (totally $3 * O(logR)$). Therefore, the total time complexity is $O(logR) + 3 * O(logR) = 4 * O(logR) = O(logR)$.

The complexity of *Update Reconstruction* algorithm (Algorithm 3) is considered regarding the *size of the input cpe expression*. In particular, we are interested in the number of unions (as determined by the appearance of $\oplus$) that exist in *cpe*. Recall that *cpe* is of the form $cpe := pe^1 \oplus \ldots \oplus pe^m$. Then, each operand pe^i of a union operator requires time O(x_i), where x_i is the number of quad patterns that exist in pe^i. Hence, the complexity is $O(\sum_i x_i) = O(m)$, where m is the total number of quad patterns in the WHERE clause.

Chapter 7

Related Work

Data provenance has been widely studied in several different contexts such as databases, distributed systems, Semantic Web etc. In [11], Moreau explores the different aspects of provenance in the Web. Likewise, Cheney et al. [6] 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* [8, 9, 17, 24, 25] or *queries* [7, 8, 9, 12, 13, 14, 17, 26]. 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 [7, 8, 17] or RDF [9, 12, 13, 14, 25, 26], 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 [25].

A third categorization stems from the expressive power of the employed provenance model, e.g., *how*, *where*, *why*, *lineage* etc. Since our proposed model is based on how and where provenance models, we discuss them thoroughly here. *Where provenance* is a popular data provenance model [8, 9, 14, 17, 24, 16] 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 [7], *provenance semirings* are used to record *how provenance* for the relational setting through polynomials; whereas [12, 13, 14] showed how to apply provenance semirings for the RDF/SPARQL setting. Our provenance model is inspired by these models (see 2.3 for details).

Another relevant dimension of provenance is granularity. In standard relational settings, three granularity levels are admitted (*attribute*, *tuple* and *table*), but most works deal only with *tuple-level provenance* (an exception is [17], which deals with all levels of provenance). Our approach deals both with *triple* (aka tuple) and *attribute level* provenance.

An important work on update provenance for the relational setting is [17], 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 [25]. 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 [14]. This work presents how abstract relational data provenance models can be adapted to capture the provenance of the results of positive SPARQL queries, i.e., without SPARQL OPTIONAL clauses (see Section 2.3 for details). The present work 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 ([27, 28, 29]). In [28], 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 [29] 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 [27]. The goal of PROV is to enable the wide publication and interchange of provenance on the Web and other information systems. PROV enables one to represent and interchange provenance information using widely available formats such as RDF and XML.

Chapter 8

Conclusions and Future Work

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 [9, 14], where the use of abstract identifiers and operators is proposed. Abstract identifiers are uniquely assigned to RDF quadruples, whereas abstract operators describe how a result quadruple was derived. In addition, we introduce the notion of quad pattern positions, which allows the identification of the attributes of quad patterns that were involved in a join or a "copy" operation. Hence, identifiers, operators and quad pattern positions are combined to create abstract algebraic expressions to annotate RDF quadruples. Our model is richer than standard query provenance models since it captures fine-grained provenance both at triple and attribute level.

Our main contribution is the exploitation of the expressive power of the proposed provenance model to introduce the feature of *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 the implied quadruple. This can be viewed as 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 Graph Store, through the identification of a compatible INSERT update as described above.

We are currently working on a first implementation of our ideas on top of the Virtuoso database engine that aims to test the correctness of the proposed algorithms. In the future, we plan to experimentally evaluate the performance of our model with more complex data and real world applications, e.g., health care, as well as its performance and its scalability for large INSERT updates and/or updates with a large output. We also plan to consider FILTER and non-monotonic SPARQL operators. This would lead to a stronger version of reconstructability, i.e., being able to reconstruct an INSERT update that is equivalent (modulo variable naming) to the original one. In addition, we will study the SPARQL DELETE, CREATE and DROP operations since all SPARQL operations can be written as a combination of INSERT, DELETE, CREATE and DROP statements. Finally, we intend to explore the use of PROV and CIDOC CRM [30] approaches for representing our model in the form of Linked Data.

Bibliography

[1] "W3C Linking Open Data," World Wide Web Consortium, Tech. Rep. [Online]. Available: http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

[2] F. Manola and E. Miller, Eds., *RDF Primer*. W3C, 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[3] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named Graphs," *Journal of Web Semantics*, vol. 3, no. 4, pp. 247–267, 2005.

[4] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," World Wide Web Consortium, Tech. Rep., 2013. [Online]. Available: http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[5] P. Gearon, A. Passant, and A. Polleres, "SPARQL 1.1 Update," World Wide Web Consortium, Tech. Rep., 2013. [Online]. Available: http://www.w3.org/TR/2013/REC-sparql11-update-20130321/

[6] J. Cheney, L. Chiticariu, and W.-C. Tan, *Foundations and Trends in Databases*, vol. 1, no. 4, pp. 379–474, 2009. [Online]. Available: http://dx.doi.org/10.1561/1900000006

[7] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Principles Of Database Systems*. ACM, 2007, pp. 31–40.

[8] S. Vansummeren and J. Cheney, "Recording Provenance for SQL Queries and Updates." *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 29–37, 2007.

[9] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides, "Coloring RDF Triples to Capture Provenance," in *International Semantic Web Conference*, A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, Eds., vol. 5823. Springer, 2009, pp. 196–212.

[10] G. Karvounarakis, I. Fundulaki, and V. Christophides, "Provenance for linked data," in *In Search of Elegance in the Theory and Practice of Computation*, ser. Lecture Notes in Computer Science, V. Tannen,

L. Wong, L. Libkin, W. Fan, W.-C. Tan, and M. Fourman, Eds. Springer Berlin Heidelberg, 2013, vol. 8000, pp. 366–381. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41660-6_19

[11] L. Moreau, "The foundations for provenance on the web," *Foundations and Trends in Web Science*, vol. 2, no. 2-3, pp. 99–241, 2010. [Online]. Available: http://dx.doi.org/10.1561/1800000010

[12] F. Geerts, G. Karvounarakis, V. Christophides, and I. Fundulaki, "Algebraic Structures for Capturing the Provenance of SPARQL Queries," in *International Conference on Database Theory*. ACM, 2013, pp. 153–164.

[13] C. V. Damasio, A. Analyti, and G. Antoniou, "Provenance for SPARQL Queries," in *International Semantic Web Conference*, P. Cudrré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, Eds., vol. 7649. Springer, 2012, pp. 625–640.

[14] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides, "On Provenance of Queries on Semantic Web Data," *IEEE Internet Computing*, vol. 15, no. 1, pp. 31–39, 2011.

[15] J. J. Carroll, C. Bizer, P. J. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, 2005, pp. 613–622.

[16] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *Proceedings of the 8th International Conference on Database Theory*, ser. ICDT '01. Springer-Verlag, 2001, pp. 316–330. [Online]. Available: http://dl.acm.org/citation.cfm?id=645504.656274

[17] P. Buneman, J. Cheney, and S. Vansummeren, "On the Expressiveness of Implicit Provenance in Query and Update Languages." in *International Conference on Database Theory*, T. Schwentick and D. Suciu, Eds., vol. 4353. Springer, 2007, pp. 209–223.

[18] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," in *International Semantic Web Conference*, I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, Eds., vol. 4273. Springer, 2006, pp. 30–43.

[19] M. Arenas, C. Gutierrez, and J. Perez, "On the Semantics of SPARQL," in *Semantic Web Information Management*, R. D. Virgilio, F. Giunchiglia, and L. Tanca, Eds. Springer, 2009, pp. 281–307.

[20] P. Buneman, S. Khanna, and W.-C. Tan, "On propagation of deletions and annotations through views," in *Proceedings of the ACM Symposium on*

Principles of Database Systems, ser. PODS '02. ACM, 2002, pp. 150–158. [Online]. Available: http://doi.acm.org/10.1145/543613.543633

[21] R. Krummenacher, E. P. B. Simperl, D. Cerizza, E. D. Valle, L. J. B. Nixon, and D. Foxvog, "Enabling the european patient summary through triplespaces," *Computer Methods and Programs in Biomedicine*, vol. 95, no. 2-S1, pp. 33–43, 2009.

[22] D. Schmidt, G. Lindemann, and T. Schrader, "First steps towards an intelligent catalogue within the open european nephrology science center?open.sc," *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, vol. 2, pp. 39–44, 2007.

[23] E. Grossman, P. Verdecchia, A. Shamiss, F. Angeli, and G. Reboldi, "Diuretic treatment of hypertension," *Diabetes Care*, vol. 34, no. Supplement 2, pp. S313–S319, 2011.

[24] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 539–550. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142534

[25] H. Halpin and J. Cheney, "Dynamic provenance for SPARQL updates using named graphs," in *Theory and Practice of Provenance*, 2011.

[26] M. Wylot, P. Cudré-Mauroux, and P. T. Groth, "Tripleprov: efficient processing of lineage queries in a native RDF store," in *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, 2014, pp. 455–466. [Online]. Available: http://doi.acm.org/10.1145/2566486.2568014

[27] "An Overview of the PROV Family of Documentsl," World Wide Web Consortium, Tech. Rep., 2013. [Online]. Available: http://www.w3.org/TR/2013/NOTE-prov-overview-20130430

[28] O. Hartig, "Provenance Information in the Web of Data," in *Proceedings of the WWW2009 Workshop on Linked Data on the Web, LDOW 2009, Madrid, Spain, April 20, 2009.*, 2009. [Online]. Available: http://ceur-ws.org/Vol-538/ldow2009_paper18.pdf

[29] 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 Comp. Syst.*, vol. 27, no. 6, pp. 743–756, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.future.2010.07.005

[30] M. Theodoridou, Y. Tzitzikas, M. Doerr, Y. Marketakis, and V. Melessanakis, "Modeling and querying provenance by extending cidoc crm," *Distrib. Parallel Databases*, vol. 27, no. 2, pp. 169–210, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1007/s10619-009-7059-2