PlanetData

Network of Excellence

FP7 – 257641

D3.2 Provenance Management and Propagation Through SPARQL Query and Update Languages

Coordinator: Giorgos Flouris

With contributions from:

Irini Fundulaki, Vassilis Papakonstantinou, Argiro Avgustaki, Anastasia Analyti, Grigoris Antoniou, Carlos Viegas Damasio, Vassilis Christophides, Grigoris Karvounarakis

1st Quality Reviewer: Oscar Corcho

2nd Quality Reviewer: Luciano Serafini

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M36
Actual delivery date:	M36
Version:	1.0
Total number of pages:	38
Keywords:	provenance, SPARQL, linked (open) data, LOD, provenance management

Abstract

Recording the origin (provenance) of data is important to determine reliability, accountability, attribution or reputation, and to support various applications, such as trust, access control, privacy policies etc. The unmoderated nature of the Linked Open Data paradigm emphasizes this need. This deliverable studies the problem of provenance management under different settings, and theoretically evaluates the use of abstract models for this purpose, with emphasis in the case of dynamic datasets which support inference (as is the case with RDFS datasets). Additionally, abstract models are applied to study the problem of representing the provenance that is needed for annotating the results of SPARQL queries, as well as for determining the provenance of triples that were added to the dataset during SPARQL Update operations.

Executive Summary

Storing the provenance of information, i.e., from where, and how, each piece of data was generated, is highly important for many applications, because it allows evaluating the trustworthiness, reliability and quality of the data, it provides accountability in case of errors, and allows the reproduction of data which is necessary for most scientific applications. Thus, provenance is used in various domains to support trust, access control, privacy, digital rights management, quality management and assessment, or other applications. These arguments are even more emphasized in the context of the Linked Open Data (LOD) initiative, where one can expect to find various (possibly conflicting) datasets, coming from different sources of varying quality and reliability.

To represent provenance in RDF, the de facto standard language for data representation in LOD, various proposals advocate the use of tags associating each RDF triple with its provenance. These models are often referred to as *concrete*, because they associate each RDF triple with a concrete token representing its provenance (or trust level, accessibility information etc, depending on the application of provenance considered).

The concrete representation is simple, but falls short in cases where the annotated dataset supports inference (as is the case, e.g., for linked datasets that use RDFS constructs). In the presence of inference, the value of the annotations of inferred triples requires a certain amount of computation. However, concrete models do not store *how* the annotation tag was computed, only the final result. Thus, there is no way to know which tags are affected by any given change, so all changes on the dataset or on the semantics of the computation, would cause a complete recomputation of all annotation tags, even those that are not relevant with the change.

An additional shortcoming of the concrete representation is the fact that several applications require full knowledge of the computation process that led to the final result. This is necessary to support applications of provenance that were not envisioned at design time, or to support scenarios where the same dataset is being used by different users or applications which need to impose different semantics on the provenance information.

In this deliverable we will study the use of abstract models as a means around these problems. Abstract models store *how* the provenance is computed, rather than the result of the computation itself, thus circumventing the above problems. Abstract models are defined by means of abstract tokens and abstract operators. Abstract tokens are used to represent provenance of triples, whereas abstract operators are used to describe how tokens are combined to express the provenance of triples which originate from triples coming from different sources (e.g., via RDFS inference).

The use of abstract models (rather than concrete ones) has several advantages in the dynamic setting, such as the ability to adopt different concrete values for the abstract tokens by the same (or different) application(s) in order to implement different provenance semantics, as well as the possibility to perform efficient recomputation of provenance information in the case of changes.

An additional advantage of abstract models is that they allow to efficiently annotate SPARQL query results. In this respect, the challenge is to identify the provenance that should be associated with a triple returned by a SPARQL query; such provenance information can be viewed as a kind of "justification" of why said triple appears in the result.

This problem can be solved by adapting well-known works from the classical relational literature (where they study the similar problem of annotating SQL query results). However, this approach would work only for the monotonic fragment of SPARQL; as a matter of fact, the incorporation of non-monotonic keywords in SPARQL (such as OPTIONAL) creates an important challenge, which we address in this deliverable.

Finally, this deliverable studies the similar problem of computing the provenance values of triples inserted during SPARQL Updates. In this respect, the problem amounts to identifying the provenance value of triples included in a dataset via complex SPARQL Update statements. This is similar to the problem of annotating the result of SPARQL queries, but raises some additional challenges which are explored in this deliverable.

Document Information

IST Project Number	FP7 – 257641	Acronym	PlanetData
Full Title	PlanetData		
Project URL	http://www.planet-data.eu/		
Document URL	`http://planet-data-wiki.sti2.at/web/D3.2`		
EU Project Officer	Leonhard Maqua		

Deliverable	Number	D3.2	Title	Provenance Management and Propagation Through SPARQL Query and Update Languages
Work Package	Number	WP3	Title	Provenance and Access Policies

Date of Delivery	Contractual	M36	Actual	M36
Status	version 1.0		final ☒	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Giorgos Flouris (FORTH)			
Responsible Author	Name	Giorgos Flouris	E-mail	fgeo@ics.forth.gr
	Partner	FORTH	Phone	+302810391674

Abstract (for dissemination)	Recording the origin (provenance) of data is important to determine reliability, accountability, attribution or reputation, and to support various applications, such as trust, access control, privacy policies etc. The unmoderated nature of the Linked Open Data paradigm emphasizes this need. This deliverable studies the problem of provenance management under different settings, and theoretically evaluates the use of abstract models for this purpose, with emphasis in the case of dynamic datasets which support inference (as is the case with RDFS datasets). Additionally, abstract models are applied to study the problem of representing the provenance that is needed for annotating the results of SPARQL queries, as well as for determining the provenance of triples that were added to the dataset during SPARQL Update operations.
Keywords	provenance, SPARQL, linked (open) data, LOD, provenance management

Version Log			
Issue Date	Rev. No.	Author	Change
13/08/2013	0.5	Giorgos Flouris	First full draft
21/08/2013	0.7	Giorgos Flouris	Consolidated draft
05/09/2013	0.9	Giorgos Flouris	Ready for the internal review process
30/09/2013	1.0	Giorgos Flouris	Final version, to submit to EU

Table of Contents

List of Tables

1 Introduction

Provenance refers to the origin of information and is used to describe *where* and *how* the data was obtained. Provenance is versatile and could include various types of information, such as the source of the data, information on the processes that led to a certain result, date of creation or last modification, authorship, and others.

Recording and managing the provenance of data is of paramount importance, as it allows supporting trust mechanisms, access control and privacy policies, digital rights management, quality management and assessment, in addition to reputability, reliability and accountability of sources and datasets. In the context of scientific communities, provenance information is used in the proof of the correctness of results and in general determines the quality of scientific work. In this respect, provenance is not an end in itself, but a means towards answering a number of questions concerning data and processes.

The absence, or non-consideration, of provenance can cause several problems; interesting examples of such *provenance failures* can be found in [19]. One such case was the publication, in 2008, of an undated document regarding the near bankruptcy of a well-known airline company; even though the document was 6 years old, and thus irrelevant at the time, the absence of a date in the document caused panic, and the company's share price fell by 75% [19].

Provenance becomes even more important in the context of the Linked Open Data (LOD)[1] initiative, which promotes the free publication and interlinking of large datasets in the Semantic Web [8, 5]. The LOD cloud is experiencing rapid growth since its conception in 2007; hundreds of interlinked datasets compose a knowledge space which currently consists of more than 31 billion RDF triples. Such datasets include ontologies created from Wikipedia[2] or other sources[3], data from e-science, most notably in the area of life sciences[4], Web 2.0 information mashups [48], and others.

The unconstrained publication, use and interlinking of datasets that is encouraged by the LOD initiative is both a blessing and a curse. On the one hand, it increases the added-value of interlinked datasets by allowing the re-use of concepts and properties. On the other hand, the unmoderated data publication raises various additional challenges including making the need for clear and efficient recording of provenance even more imperative for resolving problems related to data quality, data trustworthiness, privacy, digital rights etc [29, 41].

In this deliverable, we consider the problem of representing and storing provenance in an RDF setting, which is the de facto standard language for data representation in LOD. Note that most LOD datasets also employ RDFS constructs, thus introducing semantics and inference in the picture. The main challenge in this respect is not the storage of provenance per se, but the handling of provenance for inferred information (i.e., in the presence of RDFS or custom inference [59] rules), as well as the efficient management of provenance information during queries and updates.

Regarding the former problem (handling the provenance of inferred triples) the main problems occur in the presence of changes, either in the data or in the semantics of the application (e.g., trust, access control) that uses the corresponding provenance information. In such a setting, standard annotation models (also called *concrete models*) are inefficient, because a complete recomputation of all the implicit annotation tokens is required after each change.

To address this shortcoming, we discuss the use of *abstract provenance representation models* [37] which can be used to store *how* the provenance of a triple was computed; thanks to this feature, the recomputation of implicit tags after a change becomes very efficient, because knowing how the provenance of a triple is computed allows knowing which triples are affected by each change, and how.

An additional advantage of abstract models is that they allow for uses and applications of the provenance information that were not necessarily envisioned at design time. Knowing how the provenance of a triple is computed can be used to support any additional application and semantics of provenance over the system. Along similar lines, abstract models are necessary in scenarios where the same dataset is being used by different users or applications, which need to impose different semantics on the provenance information. For example, two

[1] `http://linkeddata.org/`

[2] `http://www.dbpedia.org`

[3] `http://www.informatik.uni-trier.de/~ley/db`

[4] `http://www.geneontology.org/`

different users may have different trust models, so the computed trust level of the same triple (with the same abstract annotation tag) may be different. In such cases, storing the abstract provenance tag is more efficient than storing a multitude of different concrete tags.

Note that models which focus on what type of metadata information is important for provenance (e.g., when the data was created, how, by whom etc), as well as on how to model such information in an ontology, are orthogonal to the discussion on concrete or abstract provenance models; examples of such approaches are the PROV model[5], the Open Provenance Model (OPM) [51], the approach of [39] for SPARQL updating, and the vocabulary proposed in [41], which is based on the Vocabulary of Interlinked Datasets (voiD) [2].

The first part of this deliverable focuses on elaborating on abstract models, namely defining them, explaining their underlying intuitions, and outlining their usage, applications, merits and drawbacks for provenance management with respect to the aforementioned problems. More specifically, we will describe how abstract models can be used to represent provenance information, as well as how they can be applied to efficiently compute the effects of a change on the annotation labels of triples.

Building on this analysis, we will then explain how abstract models can be used to annotate the result of SPARQL queries. In that setting, the user may wish to know the exact provenance value of the returned results, or in other words a kind of "justification" of why the results were there. This information can be used for different applications, such as trustworthiness evaluation, access control, privacy etc; thus, the system cannot predict the use of such information by the user, so it should not perform any kind of ad-hoc aggregation of information (and return a single provenance value) as performed by standard annotation models. Thus, the expressive power of abstract models is necessary to address this case also.

The approach of using abstract models to annotate query results has been extensively used by the database community to annotate the results of SQL queries [12, 32, 7, 37, 33, 20, 35]. Adapting that work for the RDF/SPARQL setting has attracted a lot of attention recently [60, 45], and the main related challenge was identified to be the incorporation of the non-monotonic features of SPARQL inside the models [60]. In this deliverable, we will describe two different ways of dealing with the non-monotonic features of SPARQL, which are based on the algebraic structures of m-semirings and spm-semirings [21, 34].

Another similar problem that can be resolved using abstract provenance models is the problem of determining the provenance of triples generated via SPARQL update operations. In this respect, the problem amounts to identifying the provenance value of triples included in a dataset via complex SPARQL Update statements. This problem has received much less attention; most related works apply for the relational setting, and essentially adapt works that annotate the results of SQL queries [63, 39]. This is reasonable, because the two problems are very similar; however, in this deliverable we show that there are several additional challenges involved in the case of updating, we analyze those challenges and study some possible solutions.

This deliverable is structured along the above problems. In particular, Chapter 2 provides some basic facts on RDF, RDFS, SPARQL and SPARQL Update, to be used as reference points for the rest of the deliverable. Chapter 3 describes concrete and abstract models at an intuitive level, through an example, and explains the problems associated with concrete models, as well as how abstract models can be used to overcome these problems. In Chapter 4, we focus on explaining how abstract models can be used for provenance management of dynamic datasets in the presence of RDFS inference; in the same chapter we also provide a more elaborate description of abstract models, including various alternative ways to define abstract models (Section 4.1), formal definitions for one useful, and quite general, alternative (Section 4.2), as well as algorithms for provenance management using abstract models (Section 4.3). Further, Chapter 5 discusses the problem of annotating and managing provenance information for SPARQL query results, with emphasis on the non-monotonic part of SPARQL (which is more challenging); in this respect, we present two alternative approaches for addressing the problem in Sections 5.4, 5.5. Chapter 6 discusses the problems associated with managing the provenance of triples generated via SPARQL Update operations. Finally, we conclude in Chapter 7.

[5] http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/

2 Preliminaries

The W3C Linked Data Initiative has boosted the publication and interlinkage of massive amounts of scientific, corporate, governmental and crowd-sourced data sets on the emerging Data Web. This data is commonly published in the form of RDF data, and often uses RDFS semantics; it is queried using the SPARQL query language and updated using the SPARQL Update language. In this section we briefly repeat the basic facts regarding RDF, RDFS, SPARQL and SPARQL Update, for reference in the rest of this deliverable.

2.1 RDF and RDFS

The Resource Description Framework (RDF) [49], a W3C recommendation, is used for representing information about Web resources. It enables the encoding, exchange, and reuse of structured data, while it provides the means for publishing both human-readable and machine-processable vocabularies. It is used in a variety of application areas, and has become the de facto standard for representing information on the Web of Data.

RDF [49] is based on a simple data model that makes it easy for applications to process Web data. In RDF everything we wish to describe is a *resource*. A resource may be a person, an institution, a thing, a concept, or a relation between other resources. A resource is uniquely identified by its Universal Resource Identifier (URI). Literals are also used for numbers, dates, strings etc.

The building block of the RDF data model is a *triple*, which is of the form $(subject, predicate, object)$. The subject and predicate of a triple are URIs, whereas the object can be either a URI or a literal.

It has been argued [17, 16] that it is useful to extend the syntax of RDF to include RDF graphs nameable by URIs, which are called *named graphs*. Formally, a named graph is just a set of triples identified by a URI. Named graphs can find various applications, as they can describe various metadata on the triples; for example, they can be used to describe the provenance of an RDF triple (all triples belonging to a given named graph have a certain provenance).

A *dataset* consists of a set of triples. A *graph store* consists of a set of named graphs.

$$R1: \frac{(p, subPropertyOf, q), (q, subPropertyOf, r)}{(p, subPropertyOf, r)} \qquad R2: \frac{(p, subPropertyOf, q), (x, p, y)}{(x, q, y)}$$

$$R3: \frac{(x, subClassOf, y), (z, type, x)}{(z, type, y)} \qquad R4: \frac{(x, subClassOf, y), (y, subClassOf, z)}{(x, subClassOf, z)}$$

Table 2.1: RDFS Inference Rules

The RDF Schema (RDFS) language [10] is used to introduce useful semantics to RDF triples. RDFS discriminates resources in various *types*, such as *classes* (sets of resources), *properties* (relations between resources) and *instances* (individual resources). RDFS also provides a built-in vocabulary for asserting user-defined schemas within the RDF data model. This vocabulary can be used to specify specific URIs as being of a specific type (classes, properties, instances), or to denote special relationships between URIs, including subsumption, instantiation, and domain and range of properties.

For example, the URIs $rdfs{:}Class$ $(Class)$ and $rdf{:}Property$ $(Property)$[1], are used to specify classes and properties respectively (e.g., $(A, type, Class)$ denotes that A is a class), whereas subsumption between classes/properties is denoted using the $rdfs{:}subClassOf$ $(subClassOf)$ and $rdfs{:}subPropertyOf$ $(subPropertyOf)$ predicates respectively. For more details on RDF and RDFS, the reader is referred to [49, 10].

In addition to the vocabulary elements, RDFS also specifies the semantics of these elements (e.g., subsumption is defined to be transitive). Part of this semantics is a set of *inference rules* [10] which are used to infer new triples from existing ones. We will refer to such triples as *implicit triples* (or *inferred triples*), whereas the

[1] In parenthesis are the terms we use to refer to the RDFS built-in classes and properties.

triples in the original dataset are called *explicit*. The set consisting of all the implicit and explicit triples in a dataset D is called the *closure* of the dataset, denoted by $Cn(D)$.

Table 2.1 depicts the inference rules we consider in this deliverable; for the full set, the reader is referred to [10]. Rules $R1$ and $R4$ discuss the transitivity of $subPropertyOf$ and $subClassOf$ properties resp., whereas $R2$ and $R3$ discuss the transitivity for the $subClassOf$ and $subPropertyOf$ relations and *class* and *property* instances. Additional inference rules can be neatly included in our model but are omitted for simplicity (see also Subsection 4.2.1). For technical purposes that will be made apparent later, we assume that the class and property hierarchies of the dataset are acyclic.

2.2 SPARQL

SPARQL [56] is the official W3C recommendation for querying RDF datasets, and is based on the concept of matching patterns against a dataset. Thus, a SPARQL query determines the pattern to seek for, and the answer is the part(s) of the dataset that match this pattern.

More specifically, SPARQL uses *triple patterns* which resemble an RDF triple, but may have a *variable* (prefixed with character `?`) in any of the subject, predicate, or object positions in the RDF triple. Intuitively, triple patterns denote the triples in a dataset that have the form specified by the triple patterns. For example, the triple pattern $tp_1 = (?x, subClassOf, ?y)$ contains two variables $(?x, ?y)$, which can be substituted by any URI or literal; as such, tp_1 is used to denote all triples in the dataset whose predicate is $subClassOf$.

SPARQL *graph patterns* are produced by combining triple patterns through the *join*, *optional* and *union* SPARQL operators. Graph patterns may contain filters, using the `FILTER` expression to specify conditions on the triple patterns. More formally, a SPARQL graph pattern is defined recursively as follows: a triple pattern p is the simplest graph pattern; if p_1 and p_2 are graph patterns, then the expressions p_1 . p_2, p_1 `OPTIONAL` p_2, and p_1 `UNION` p_2 are graph patterns; finally, if p is a graph pattern and C is a SPARQL built-in condition, then the expression p `FILTER` C is a graph pattern. As with triple patterns, graph patterns are matched against a dataset by substituting the variables with matching URIs, literals or blank nodes.

The SPARQL syntax follows the SQL select-from-where paradigm. The select clause determines the form of the result. In this deliverable, we will mostly employ the `CONSTRUCT` clause, which is used for queries whose returned results are triples (i.e., the query result is a set of triples). The `CONSTRUCT` clause specifies the names of the variables which should be returned; these variables (along, possibly, with other variables) are subsequently used in the graph pattern in the `WHERE` clause of the query, which determines the triples that should be returned by the query. To determine the returned triples, we identify the triples that match the graph pattern in the `WHERE` clause, and replace the variables in the `CONSTRUCT` clause with the corresponding URIs, forming triples.

The interested reader can find a more detailed description of the semantics of the SPARQL language in [55, 54].

2.3 SPARQL Update

SPARQL Update version 1.1 [31] was specified with the objective of being a standard language for specifying updates to named graphs in a graph store. It is a W3C Recommendation since March 2013. It supports various types of updates, including inserting and deleting triples, operations on named graphs (create, drop etc), as well as operations on the content of named graphs (copy, move etc). In this section, we give an outline of the semantics of the operations supported by SPARQL Update 1.1; more details can be found in [31].

The syntax of SPARQL Update is very similar to the syntax used in the SPARQL query language. A SPARQL Update statement determines the operation to perform (insert a triple, drop a named graph, copy the contents of a named graph to another etc), as well as the parameters of the operation (e.g., in the case of inserting a triple, the parameters determine which triples should be added, and the named graphs where those triples should be added, whereas in the case of copying the contents of a named graph to another, the parameters determine those named graphs). These parameters are URIs denoting the subject, property and object of the triple(s) involved, the affected named graphs etc, or they can be variables (in the subject, property, object or

graph position) which are mapped to URIs using triple and graph patterns as in the SPARQL query language. In the following paragraphs, we briefly describe the semantics and parameters of each of the operations supported by SPARQL Update 1.1. For more details, and the exact syntax of each operator, the reader is referred to [31].

INSERT DATA: INSERT DATA is an operation used to add ground triples to a named graph. The allowed parameters are the ground triples and the named graph; note that this operation does not allow variables, so the inserted triples must be explicitly specified.

DELETE DATA: This operation is used to delete ground triples from a named graph. As with INSERT DATA, the allowed parameters is the ground triples and the named graph, and no variables or blank nodes are allowed.

INSERT: INSERT is similar to INSERT DATA, except that it allows the inserted triples to be specified using variables. The INSERT operation accepts as parameters the triples (or triple patterns) to insert, and the named graph(s) where they should be inserted; in the case where a triple pattern (rather than a ground triple) is inserted, the graph pattern that determines the bindings of the variables, as well as the named graph(s) that it is evaluated against, must be provided also.

DELETE: DELETE is used to delete triples from one or more named graphs, where those triples can be specified using variables. The parameters are the same as those of INSERT.

DELETE/INSERT: This is a composite operation that both deletes and inserts triples from/to one or more named graphs. It is essentially a shortcut for a DELETE operation, followed by an INSERT operation. The parameters are the same as those of INSERT and DELETE. The specified graph pattern applies for both the inserted and the deleted triples.

LOAD: LOAD is used to copy data (triples) from one named graph to another. The parameters are the two named graphs (source and destination), and no variables are allowed. Note that the source graph may not be part of the graph store (e.g., it could be an RDF document); this differentiates this operation from other similar operations (COPY, MOVE, ADD) described below.

CLEAR: CLEAR is an operator that deletes the contents of one or more named graphs from a graph store. Note that this operator only deletes the contents of named graphs (triples), not the named graphs themselves. The only parameter of the operation is the set of named graphs to clear.

CREATE: The operation of CREATE is used to create a new empty graph. The only accepted parameter is the named graph to create.

DROP: DROP removes one or more named graphs from a graph store. Note that (unlike CLEAR) this operator does not only delete the contents of a named graph (triples), but also the named graphs themselves. The only accepted parameter is the set of named graphs to drop.

COPY: The COPY operation is used to copy all triples from a source graph to a destination graph. Data from the source graph is not affected, but data from the destination graph, if any, is removed before insertion. The accepted parameters are the two named graphs. Both named graphs must be part of the graph store.

MOVE: MOVE is similar to COPY (copies all triples from a source graph to a destination graph), with the only difference being that the source graph is dropped after the operation is complete. The parameters of MOVE are the same as in COPY. As with COPY, both named graphs must be part of the graph store.

ADD: ADD is also similar to COPY (copies all triples from a source graph to a destination graph), except that the original contents of the destination graph are not affected during copying. Again, the only parameters are the two named graphs (source and destination). This operation is essentially identical to LOAD, the only difference being that, for ADD, both the source and the destination named graphs must be part of the graph store.

3 Representation Models for Provenance

Provenance of data is of course data itself that needs to be adequately represented and stored. In this section, we will describe two representation models for provenance information, namely *concrete* and *abstract* ones, and informally explain the merits and drawbacks of each. We will argue that abstract models are more suitable in the presence of updates (i.e., for dynamic datasets) and explain how they can be used to support the provenance management for dynamic RDFS datasets (where the combination of inference and updates makes concrete models inadequate). In the following chapters, we will elaborate on the use of abstract models for representing provenance and also apply them for provenance management of SPARQL queries and updates.

To explain the two approaches, we will use an example in which provenance is being used in the context of a trust assessment application, where various linked sources are composed to form a linked dataset, and each source is associated with a trust value. Note that, in principle, it could be the case that a given source is associated with several trust values; for example, even though a sports web site could publish information on sports, but also a weather forecast, the former should be more trusted than the latter. This more complicated case can be similarly supported by assuming a richer (i.e., more fine-grained) set of sources, i.e., treating the different information as coming from different sources. In the following, for reasons of simplicity and without loss of generality, we assume the simple case where each source is associated with a single trust value. In Section 4.2 we drop this assumption. Moreover, it should be noted that, even though our running example is about trust assessment, most of our observations in this deliverable hold for any application of provenance, unless mentioned otherwise.

3.1 Concrete Representation Models

3.1.1 Description of Concrete Models

Concrete representation models are actually straightforward and consist in associating each RDF triple with an *annotation tag* that describes its provenance. This essentially transforms a triple into a *quadruple*, the fourth field being the triple's provenance (which associates the triple with its source in an RDF/S setting). The most common method for the representation of quadruples is through named graphs [17]. Table 3.1 shows a simple example dataset, inspired by the FOAF ontology[1], that we will use as a running example. Each triple in the dataset is associated with its source, forming a quadruple. Note that namespaces are omitted from URIs for readability purposes.

subject	*property*	*object*	*source*
$Student$	$subClassOf$	$Person$	s_1
$Person$	$subClassOf$	$Agent$	s_2
$\&a$	$type$	$Student$	s_3
$\&a$	$firstName$	`Alice`	s_4
$\&a$	$lastName$	`Smith`	s_4
$Person$	$subClassOf$	$Agent$	s_1

Table 3.1: Concrete Provenance Representation of Explicit Triples

Implicit information raises some complications, because it may be inferrable from triples coming from different sources [27]. For example, the triple $(\&a, type, Person)$ is implied from $(\&a, type, Student)$ and $(Student, subClassOf, Person)$, whose provenance is s_3 and s_1 respectively. Thus, the provenance of $(\&a, type, Person)$ is the *combination* of s_3 and s_1. This is different from the case where a triple originates from different sources, in which case it has two *distinct* provenance tags [27], as, e.g., is the case with $(Person, subClassOf, Agent)$, which originates from s_2 and s_1 (cf. Table 3.1). To address this problem, concrete policies associate specific

[1] `http://www.foaf-project.org/`

subject	*property*	*object*	*trust level*
Student	*subClassOf*	*Person*	$0, 9$
Person	*subClassOf*	*Agent*	$0, 9$
$\&a$	*type*	*Student*	$0, 3$
$\&a$	*firstName*	`Alice`	$0, 1$
$\&a$	*lastName*	`Smith`	$0, 1$
Student	*subClassOf*	*Agent*	$0, 9$
$\&a$	*type*	*Person*	$0, 3$
$\&a$	*type*	*Agent*	$0, 3$

Table 3.2: Trustworthiness Assessment (Concrete Representation)

semantics to the two types of provenance combination (combining provenance under inference, or combining provenance in multiply-tagged triples).

We will explain how the interrelationship between inference and provenance works in an example, considering a trust assessment application. This application associates each of the four sources $(s_1, \ldots, s_4)$ in our example, with a trust level annotation value in the range $0 \ldots 1$ (where 0 stands for "fully untrustworthy" and 1 stands for "fully trustworthy"). Let's say that s_1, s_2, s_3, s_4 are associated with the values $0, 9$, $0, 6$, $0, 3$ and $0, 1$ respectively. Let us also assume that the trustworthiness of an implied triple is equal to the minimum trustworthiness of its implying ones. When a triple originates from several different sources, its trustworthiness is equal to the maximum trustworthiness of all such sources.

This semantics would lead to the dataset shown in Table 3.2, where the last column represents the trustworthiness level of each quadruple. The first five quadruples are explicit ones, and result directly from the quadruples in Table 3.1, whereas the rest are implicit. For example, $(Person, subClassOf, Agent)$ originates from both s_1 and s_2 (see Table 3.1) so its trustworthiness equals the maximum of $0, 6$ and $0, 9$. Similarly, the implicit triple $(Student, subClassOf, Agent)$ is implied by $(Student, subClassOf, Person)$ (whose provenance is s_1) and $(Person, subClassOf, Agent)$ (that originates from two sources and hence has provenance s_1 and s_2). The pair (s_1, s_1) results to a trustworthiness of $0, 9$, whereas the pair (s_1, s_2) would give trustworthiness $0, 6$; thus, the final trustworthiness level of $(Student, subClassOf, Agent)$ is $0, 9$ (the maximum of the two, per our semantics). Using similar arguments, one can compute the trustworthiness levels of the other triples, as shown in Table 3.2.

In summary, annotating triples under a concrete model amounts to computing the trust level of each triple, according to its provenance and the considered semantics, and associating each with its (explicit or computed) trust value. The annotation semantics differ depending on what this annotation is representing (e.g., trust, fuzzy truth value etc [37]) and the application context. Concrete provenance models have been used for applications such as access control [1, 22, 43, 46, 52, 57, 64] and trust [17, 16, 40, 61] among others.

3.1.2 Discussion on Concrete Models

Despite its simplicity and efficiency, concrete models have certain drawbacks, which are related to the fact that the information on how the provenance of the implicit triple was constructed is lost. This information may be relevant in several cases, e.g., when dynamic information is involved, or when we don't know a priori all the uses of the provenance information, or when different applications/users want to impose different semantics (e.g., combination semantics) on the same data.

Linked data are in general dynamic and volatile[2]. The recent SPARQL Update recommendation [31] is specifically the answer to the need of having a standardized way to perform such changes. However, SPARQL Update, as well as all the related technologies and systems dealing with changes, focus on the effects of changes

[2] `http://www.w3.org/wiki/DatasetDynamics`, `http://blog.semantic-web.at/2010/04/26/a-dynamic-web-of-data`

on the data itself, rather than the provenance of the data [65]. The latter is an important problem when provenance is stored using concrete models. In particular, the loss of the information on how the implicit provenance is constructed causes a lot of unnecessary computations during changes.

For example, assume that the linked dataset corresponding to source s_1 is updated and no longer includes triple $(Person, subClassOf, Agent)$. Formally, this amounts to the deletion of the quadruple $(Person, subClassOf, Agent,$ This deletion affects the trustworthiness of $(Student, subClassOf, Agent)$ in Table 3.2, but since concrete models do not record the computation steps that led to each annotation value, there is no way to know which annotations are affected, or how they are affected; thus, we have to recompute all annotations to ensure correctness. Similar problems appear in the case of additions, as well as when the application semantics change, e.g., if we revise the trustworthiness associated to some source, or the function that computes the overall trust level of an implicit or multiply-tagged triple.

Note that the problem with dynamic information becomes a real issue only in the presence of inference (as in the case of RDFS information). If inferred knowledge was not considered, then concrete models would cause no unnecessary overhead during changes; all we would have to do without inference would be to update (add/delete) the triples affected by the change, along with their provenance information.

But even when dealing with static datasets, it is sometimes the case that we need to know how the provenance was constructed, because, e.g., we want to be prepared for unexpected uses of the provenance information, or because different users/applications impose different semantics on the same provenance data. In that case, we want to store the provenance per se, rather than, e.g., how this provenance affects the trust or the accessibility level of a triple. All such information is lost when concrete models are employed.

Summarizing, even though concrete models are simple to understand and implement, they cannot be used to store how the provenance of a triple was constructed, and are thus inadequate when dynamic information is involved or when full access to the construction method is necessary (e.g., to account for unexpected or multiple simultaneous uses of the provenance information). In the next section, we present abstract models which can be used to store exactly how the provenance of a triple is constructed, thus overcoming the above limitations.

3.2 Abstract Representation Models

3.2.1 Description of Abstract Models

As mentioned above, the underlying intuition behind *abstract models* [37, 44] is to record *how* the trust level (or provenance) associated with each triple should be computed (rather than the trust level/provenance itself). To do so, each explicit triple is annotated with a unique *annotation tag* (a_i). A *label* is an expression that is used to annotate a triple. Labels are shown in the column "label" of Table 3.3. In its simplest form, a label can be a simple annotation tag (e.g., q_1, whose label is a_1), but it can also involve operators applied over annotation tags (e.g., q_7 whose label is $a_1 \odot a_2$).

Note that an annotation tag identifies both the triple and its source, so the same triple coming from different sources would have a different tag and appear more than once in the dataset (cf. a_2, a_6 in Table 3.3). The annotation tag should not be confused with the quadruple ID (q_i) that is not an integral part of the model and is used just for reference. Even though this tag uniquely associates each explicit quadruple with its source, Table 3.3 includes the source as well for a better understanding of the approach.

The annotation of an inferred triple in abstract models is the composition of the labels of the triples used to infer it. This composition is encoded with the operator $\odot$. For example, the triple $(\&a, type, Person)$ results from triples $(\&a, type, Student)$ (with tag a_3) and $(Student, subClassOf, Person)$ (with tag a_1); this results to the annotation $a_3 \odot a_1$, as shown in quadruple q_9 in Table 3.3. When a triple can be inferred by two or more combinations of quadruples, each such combination results to a different quadruple (cf. q_7, q_8 in Table 3.3).

The concrete trust level is not stored, but can be computed on-the-fly through the annotations using the trust levels and semantics associated with each tag and operator. For example, if we assume the semantics described for the concrete model of Section 3.1, the annotation of $(Student, subClassOf, Person)$ is a_1, which corresponds to s_1 that has a trustworthiness level of $0,9$, per the association of sources to trustworthiness described in the previous subsection. Similarly, the trustworthiness level of $(Person, subClassOf, Agent)$ is,

	subject	*property*	*object*	*label*	*source*
q_1:	*Student*	*subClassOf*	*Person*	a_1	s_1
q_2:	*Person*	*subClassOf*	*Agent*	a_2	s_2
q_3:	$\&a$	*type*	*Student*	a_3	s_3
q_4:	$\&a$	*firstName*	`Alice`	a_4	s_4
q_5:	$\&a$	*lastName*	`Smith`	a_5	s_4
q_6:	*Person*	*subClassOf*	*Agent*	a_6	s_1
q_7:	*Student*	*subClassOf*	*Agent*	$a_1 \odot a_2$	–
q_8:	*Student*	*subClassOf*	*Agent*	$a_1 \odot a_6$	–
q_9:	$\&a$	*type*	*Person*	$a_3 \odot a_1$	–
q_{10}:	$\&a$	*type*	*Agent*	$a_3 \odot a_1 \odot a_2$	–
q_{11}:	$\&a$	*type*	*Agent*	$a_3 \odot a_1 \odot a_6$	–

Table 3.3: Trustworthiness Assessment (Abstract Representation)

per our semantics, the maximum trust level of the sources of a_1, a_6, i.e., $0,9$. Finally, the trustworthiness of $(\&a, type, Agent)$ is the maximum of the trustworthiness associated with q_{10}, q_{11}; per the annotation of q_{10} and our semantics, the trustworthiness of q_{10} is found by looking at the quadruples whose label is a_3, a_1, a_2 (whose trustworthiness is $0,3$, $0,9$, $0,6$ respectively) and taking the minimum, i.e., $0,3$. Similarly the trustworthiness of q_{11} is $0,3$, so the final trustworthiness of $(\&a, type, Agent)$ is $0,3$ (the maximum of the two).

This computation is essentially identical to the computation taking place in concrete models; the only difference is that concrete models execute this computation at initialization/loading time, whereas abstract models perform this computation *on demand*, i.e., during queries, using stored information that specifies *how* to perform the computation. Abstract models have been considered for provenance (e.g., [37]), as well as for access control (e.g., [53]).

3.2.2 Discussion on Abstract Models

Abstract models can be used to overcome the problems related to concrete models that were described in Subsection 3.1.2. In fact, they have been used in different ways to form various algebraic structures that allow the representation of provenance for inferred triples [62, 15, 66]. Note that abstract models are reminiscent of Truth Maintenance Systems (TMS) that have been used in AI in the past (e.g., [24, 11, 47]).

Regarding the problem of dynamic information, it turns out that most changes in the dataset or in the annotation semantics can be handled easily using abstract models. For example, if we delete quadruple q_6 (i.e., if we realize that the origin of $(Person, subClassOf, Agent)$ is not s_1), we can easily identify the quadruples to delete (namely, q_8, q_{11}). Even better, if one decides that source s_1 has, e.g., trustworthiness $0,8$ after all, then no recomputation is necessary, because any subsequent computations will automatically consider the new value; the same is true if the semantics for computing the trustworthiness of an implicit triple change.

As far as the unexpected uses of provenance information is concerned, again, abstract models fare better, as, by definition, they store the full provenance information (i.e., how the provenance label was computed) and can thus be used in various different ways, even in ways that were not envisioned at design time.

The same argument applies when the same data is used by different users or applications with different semantics for provenance: given that the full provenance information is stored, each user/application can use its own semantics. For example, a trust application and an access control application could work on the same data, implementing different semantics for the $\odot$ operator.

However, abstract models come with a price. In particular, abstract models cause some overhead in storage space, because storing the full provenance information is more complex than storing a value, and annotations can get quite complex. Moreover, computing said annotations at initialization time may generate an overhead as opposed to standard models. Furthermore, since the annotations of relevant triples have to be computed at

query time to determine the actual value of the annotation (e.g., the corresponding trust value, the accessibility information etc, depending on the application), abstract models cause also an overhead at query time.

Thus, the tradeoffs involved in the use of abstract models (as opposed to concrete ones) are clear: they provide significant gains in update efficiency, at the expense of a worse performance during initialization (for creating the complex annotations), query answering and storage space. The evaluation of these tradeoffs will appear in the upcoming deliverable D3.3 of the PlanetData project, which is due on M42.

3.3 Representation Models and SPARQL

Challenges related to the representation of provenance information appear also in the context of SPARQL queries [56, 55] and SPARQL Updates [31]. In that case, the provenance of the constructed or inserted triples (or tuples, in general) is the combination of the provenance of the triples that were used to obtain the result, so challenges similar to the case of representing the provenance of inferred information appear. However, this is a much more complicated case, because construction in SPARQL queries and updates uses (via the graph patterns) various different operators (union, join, projection, selection etc); this is in contrast to inference, where inference rules is the only way to combine triples and their provenance.

As with inference, concrete models can (in principle) be used to obtain the provenance of a query result, or an inserted triple, based on the defined semantics. The only difference is that now more complex computation semantics need to be defined for capturing each of the SPARQL operators (or their combination) that can be involved in the computation of the query result.

However, the problems with concrete models that were described in Subsection 3.1.2 also surface in this case. The problem with dynamic information appears when the result of a SPARQL query is pre-computed, e.g., in the context of materialized SPARQL views. In such a scenario, the provenance of triples in the result of the SPARQL query needs to be revisited (and recomputed) when changes in the underlying dataset happen. Note that this is part of the more general and well-known problem of managing materialized views when the underlying data changes [38].

More importantly, the problem with the unexpected uses of provenance information is emphasized in the SPARQL case. In many cases, users may not be interested only in the result of the query, but also in the provenance of the information found in the result. This provenance information can be used in various different and unexpected ways by the users, so it should not be artificially aggregated by the system under any reasonable semantics, but instead presented in the form of an abstract expression, which, in essence, "justifies" (or "explains") the presence of each triple in the result. Same arguments apply for the case where triples are inserted as part of SPARQL Update operations. This fact is the main reason why works related to computing and representing the provenance during SPARQL queries and updates typically use abstract models.

For the case of querying, examples of such works are [37, 63, 9, 32, 7, 35, 33], which are mostly focusing on queries in the relational setting (i.e., when the underlying dataset is a relational database, and the queries are expressed using SQL). Attempts for adapting this line of work for the RDF/SPARQL case have identified several problems, mainly related to the non-monotonic fragment of SPARQL [34, 21, 60]. An indirect way of overcoming the problem appears in [66], where an extension of SPARQL is proposed, in which the user can express queries that explicitly manipulate both data and annotations (provenance). Chapter 5 contains more details on the problem of annotating the result of a SPARQL query.

Most works related to the updating problem try to adapt the solutions proposed for the case of (SQL or SPARQL) querying [63, 13, 14]. This is reasonable, because the underlying problem is that of identifying the annotation of a newly constructed triple, regardless of whether this is constructed in order to be returned as the result of a query or if it is constructed in order to be inserted in a dataset/database. Having said that, it is also true that there are several differences that require a different approach to address this problem. Chapter 6 contains more details on this issue.

4 Abstract Models for Provenance Management of Dynamic RDFS Datasets

In this chapter we elaborate on abstract models, giving various alternative definitions (Section 4.1). In addition, we formally define abstract models for one popular, and quite powerful, alternative (Section 4.2) and provide algorithms for provenance management (Section 4.3).

This chapter is focusing on the simple version of the problem, i.e., when only inference, rather than the more complex SPARQL operators, is considered. However, we also briefly describe the extensions that are necessary to handle the more complex case; more details on SPARQL provenance management can be found in Chapter 5.

4.1 Alternatives for the Definition of Abstract Models

Abstract models can be implemented in different ways; in this section, we briefly consider various alternatives which differ along four orthogonal axes, namely: (a) whether multiple annotations of a given triple should be combined or not; (b) the type of identifiers used in complex annotations (provenance IDs, triple IDs or annotation tags); (c) whether one type of composition (inference) or several types of composition (SPARQL construction of tuples) is considered; (d) the considered provenance model (how-provenance, why-provenance or lineage [18]).

The first dimension is related to the choice of having several, or just one, annotation(s) for a given triple. Our chosen representation in this deliverable (described in the motivating example, and also in Section 4.2) is to use several annotations for the same triple, i.e., if a triple comes from two different sources, or it can be inferred in two different ways, then it will appear twice in the table (in different quadruples). Alternatively, one could require that each triple appears only once in the table, so different annotation tags of the same triple would have to be combined using a second operator, say $\oplus$. For example, q_2, q_6 in Table 3.3 would be combined in a single quadruple (say $q_2{}'$) having as annotation the expression $a_2 \oplus a_1$. The advantage of the alternative representation is that the number of quadruples is smaller, but the annotations are usually more complex.

The second dimension is related to the type of identifiers used for labels. In Table 3.3, we used annotation tags, which is the most fine-grained option, as it explicitly describes the dependencies between quadruples. Alternatively, one could use the provenance or triple ID as the main identifier used in the labels. The use of provenance IDs facilitates the computation of concrete tags (because we don't need to go through the annotation tags to identify the trust level of a complex annotation), but is ambiguous as far as the dependencies between quadruples is concerned. The use of triple IDs gives a more compact representation; for example, only one of q_7, q_8 would have to be recorded under this representation (because both are inferred by the same set of triples). On the other hand, it forces an extra join to identify the annotations associated with a given triple during trust level assessment.

The third dimension is whether we consider inference as the only way of composing triples, or also alternative ways. The latter is necessary when SPARQL operators are involved, but also when provenance information can be composed or propagated between triples in various different ways (e.g., via the propagation operator in the access control context, as proposed in [53], or via custom inference rules [59]). Note that for each different type of composition/propagation, a new abstract operator (apart from $\odot$) should be defined.

The fourth dimension is related to the provenance model used. There are three main provenance models that have been discussed in the literature [18]:

- *Lineage* is the most coarse-grained one and records the input data (labels in our context) that led to the result (complex label), but not the process itself. For example, the lineage label of $(\&a, type, Agent)$ would be $\{a_3, a_1, a_2, a_6\}$, because these are the tags involved in computing its label.

- *Why-provenance* is more fine-grained; it records the labels that led to the result, but different computations (called *witnesses*) are stored in different sets. In the same example, the why-provenance label of $(\&a, type, Agent)$ would be $\{\{a_3, a_1, a_2\}, \{a_3, a_1, a_6\}\}$; each set in this representation corresponds to one different way to infer $(\&a, type, Agent)$. This is reminiscent to the graphsets approach for storing provenance [27].

- Finally, *how-provenance* (the approach we employ here) is the most fine-grained type; it is similar to why-provenance, except that it also specifies the operations that led to the generation of the label. Thus, how-provenance requires the use of operators and would produce expressions like the ones appearing in Table 3.3.

In the simple case of our motivating example, where only inference is considered, why-provenance and how-provenance could be equivalently used. This is not true in general, even in the simple case where there is only one way of composing triples (inference), because why-provenance cannot capture the order in which the triples/quadruples were used to infer the new triple (which may be relevant for some applications); moreover, why-provenance cannot record the case where a quadruple is used twice in an inference (again, this cannot happen in our setting, but it is possible in general). More importantly, the equivalence would break if we considered more operators than just $\odot$, e.g., when we need to represent the how-provenance of triples produced via SPARQL queries.

4.2 Formalities

4.2.1 Basic Concepts

Now let's see how the above ideas regarding abstract provenance models can be formalized. As explained in Section 3.2, we will assume that each triple gets a universally unique, source-specific *abstract annotation tag* (taken from a set $\mathcal{A} = \{a_1, a_2, \ldots\}$). This tag identifies it across all sources, so if the same triple appears in more than one source, then each appearance will be assigned a different tag. Note that this implies that the source of a triple is assumed to be known; if we drop this assumption, we would have to use some special "source" representing the fact that the source is unknown, essentially incorporating neatly this case in our model. This has already been used in some contexts, where certain sources have special semantics, such as "unknown" or "default", depending on the setting [27, 53]; for simplicity, we don't consider this case here.

We define a set of *abstract operators*, which are used to "compose" annotation tags into algebraic expressions. Abstract operators encode the computation semantics of annotations during composition (e.g., for inferred triples). For the simple setting, we only need one operator, the *inference accumulator operator*, denoted by $\odot$, which encodes composition during inference; more operators may be necessary depending on the needs of the application and our assumptions. For example, if we required to have only one annotation per triple, then the *annotation composition operator* $\oplus$, (see Section 4.1) would be necessary as well; the same would be true if we considered the more general problem of composing triples using SPARQL queries [34, 21, 60], or propagation semantics such as those proposed in [53, 30, 26], or if we had custom inference rules [59] in which we would like to assign different composition semantics than the composition semantics of standard inference rules. In general, for each different type of "composition semantics", a different operator should be used.

Depending on the application, abstract operators may be constrained to satisfy properties such as commutativity, associativity or idempotence [37, 53]. These properties provide the benefit that they can be used for efficiently implementing an abstract model. Here, we require that the inference accumulator operator satisfies commutativity ($a_1 \odot a_2 = a_2 \odot a_1$) and associativity ($(a_1 \odot a_2) \odot a_3 = a_1 \odot (a_2 \odot a_3)$). These properties are reasonable for the inference accumulator operator, because the provenance of an implicit triple should be uniquely determined by the provenance of the triples that imply it, and not by the order of application of the inference rules. Idempotence is not considered for the inference accumulator operator, because there may be applications where this is not desirable; for example, in a trust application, the trust of an inferred triple may be less than those that are used to infer it, whereas in an access control application we may want to allow access to the inferred triple (with label, e.g., $a_1 \odot a_1$) but not the inferring ones (whose label is, e.g., a_1).

The proposed version of the inference accumulator operator supports only binary inference rules, i.e., rules that consider exactly two triples in their body. To support other types, this operator should be defined as a unary operator over $2^{\mathcal{A}}$ (sets of annotation tags). Since we only consider binary inference rules here (see Table 2.1), we will use the simple version.

A *label* l is an algebraic expression consisting of abstract tags and operators. In our case, a label is of the form $a_1 \odot \ldots \odot a_n$, for $n \geq 1$, $a_i \in \mathcal{A}$.

$f_{\mathcal{A}}(a_1)$	$=$	$0,9$	$f_{\mathcal{A}}(a_4)$	$=$	$0,1$
$f_{\mathcal{A}}(a_2)$	$=$	$0,6$	$f_{\mathcal{A}}(a_5)$	$=$	$0,1$
$f_{\mathcal{A}}(a_3)$	$=$	$0,3$	$f_{\mathcal{A}}(a_6)$	$=$	$0,9$

Table 4.1: Defining $f_{\mathcal{A}}$ for our Running Example

A *quadruple* is a triple annotated with a label. A quadruple is called *explicit* iff its label is a single annotation tag of the form a_i; by definition, such quadruples come directly from a certain source (the one associated with a_i). A quadruple is called *implicit* iff its label is a complex one, containing annotation tags and operators (i.e., of the form $a_1 \odot \ldots \odot a_n$, for $n > 1$ in our case); implicit quadruples have been produced by inference, thus the complex label.

Given a label $l = a_1 \odot \ldots \odot a_n$ and an annotation tag a, we say that a *is contained in* l (denoted $a \sqsubseteq l$) iff $a \in \{a_1, \ldots, a_n\}$. Intuitively, this means that the quadruple labeled using a was used to produce (via inference) the quadruple with label l.

We overload the notion of a *dataset* $\mathcal{D}$ to refer to a set of quadruples. Similarly, we overload the notion of closure: the *closure* of a dataset $\mathcal{D}$ (denoted by $Cn(D)$) is the set of all the implicit and explicit quadruples in said dataset (given a set of inference rules); analogously, a dataset $\mathcal{D}$ is called *closed* with respect to said inference rules, iff the application of these inference rules does not generate any new quadruples (i.e., if $D = Cn(D)$).

As already mentioned (Section 2.1), we require that the class and property hierarchies in a dataset are acyclic. This is necessary to avoid repeated (infinite) generation of new quadruples (that correspond to the same triples) due to the repeated applications of rules $R1, R4$, which could occur in the event of a cycle in the hierarchy.

4.2.2 Defining Abstract Provenance Models

An *abstract provenance model* is comprised of an *abstract annotation algebra* and *concrete semantics* for this algebra. An abstract annotation algebra consists of a set of abstract annotation tags and a set of abstract operators, as defined above. The concrete semantics ($\mathcal{S}$) is used to assign "meaning" to these tags and operators, in order to associate labels with concrete values depending on the application at hand (e.g., trustworthiness evaluation). The concrete semantics is composed of various subcomponents, as described below.

First, concrete semantics specifies a *set of concrete annotation values*, denoted by $\mathcal{A}_{\mathcal{S}}$, which contains the concrete values that a triple can be annotated with. Depending on the needs of the application, this set can be simple (e.g., $\{trusted, untrusted\}$) or more complex (e.g., the $0 \ldots 1$ continuum).

Secondly, concrete semantics associate abstract tokens with concrete values, through a mapping $f_{\mathcal{A}} : \mathcal{A} \mapsto \mathcal{A}_{\mathcal{S}}$. Recall that the abstract annotation tag uniquely identifies the source of the corresponding triple. Thus, given two tags, a_1, a_2 which correspond to the same source (i.e., the triples that have these tags come from the same source), one would expect that $f_{\mathcal{A}}(a_1) = f_{\mathcal{A}}(a_2)$. This is a desirable property, but we don't impose it here, in order to support cases where the same source has more than one associated concrete values (recall the example of Chapter 3). In our running example, the mapping $f_{\mathcal{A}}$ was defined as shown in Table 4.1 (cf. Subsection 3.1.1).

To compute the annotation value of implicit labels (which include operators like $\odot$), we need a concrete definition for abstract operators, i.e., the association of each abstract operator with a concrete one. Moreover, we need a function that will combine different concrete values into a single one to cover the case where a triple is associated to more than one labels. In our running example, $\odot$ was defined to be equal to the min function, whereas the combination of different tags was done using the max function (cf. Subsection 3.1.1). Note that the operators' concrete definitions should satisfy the properties set by the algebra (e.g., $\odot$ should be commutative and associative).

The concrete semantics allows us to associate each triple with a concrete value (see Subsection 3.2.1 for a description of the process), in the same way that concrete models compute concrete labels. For example, the triple $(Student, subClassOf, Agent)$ is associated with $a_1 \odot a_2$ and $a_1 \odot a_6$ (cf. quadruples q_7, q_8 in Table 3.3). The corresponding labels of a_1, a_2 are $0,9$, $0,6$ respectively per $f_{\mathcal{A}}$ in Table 4.1; since $\odot$ is mapped to the min function, the final annotation tag of q_7 is $min\{0,9,\ 0,6\} = 0,6$. Using similar arguments, the

concrete annotation tag of q_8 is $0, 9$. To find the final trustworthiness level of $(Student, subClassOf, Agent)$, we need to apply the max operator over $0, 6$, $0, 9$, getting $0, 9$, which is the trustworthiness level of said triple $(Student, subClassOf, Agent)$.

As explained before, trustworthiness computation in abstract models takes place *on demand* (e.g., during queries), whereas concrete models perform it at storage time. This causes an overhead at query time for abstract models, but allows them to respond efficiently to changes in the semantics or the data itself. In addition, abstract models allow one to associate different semantics to the data (e.g., per user role), or periodically changing semantics (e.g., over time, or in response to certain events), as well as to experiment/test with different semantics; all this can be supported without costly recomputations that would be necessary for concrete models.

4.3 Algorithms

In this section we will present algorithms for: (a) annotating triples; (b) evaluating triples; (c) updating a dataset. Our presentation considers only the (binary) inference rules R1-R4 that appear in Table 2.1 of Chapter 2 (see also [10]). It is simple to extend the corresponding algorithms for the full list of inference rules in [10] (but in that case the inference accumulator operator should be defined differently – see Subsection 4.2.1).

4.3.1 Annotation

Algorithm 1: Annotation

Data: A dataset $\mathcal{D}$
Result: The closure of $\mathcal{D}$
1 $rdfs1 = \emptyset$; $rdfs2 = \emptyset$; $rdfs3 = \emptyset$; $rdfs4 = \emptyset$;
2 $rdfs1 = Apply\_R1(\mathcal{D})$;
3 $rdfs2 = Apply\_R2(\mathcal{D} \cup rdfs1)$;
4 $rdfs4 = Apply\_R4(\mathcal{D})$;
5 $rdfs3 = Apply\_R3(\mathcal{D} \cup rdfs4)$;
6 **return** $\mathcal{D} \cup rdfs1 \cup rdfs2 \cup rdfs3 \cup rdfs4$;

The objective of annotation is to compute all implicit quadruples (i.e., to compute a closed dataset) given a set of explicit ones (i.e., a dataset). The process is shown in Algorithm 1 which calls subroutines executing each of the considered RDFS inference rules sequentially to compute all implicit quadruples. The order of execution that was selected in Algorithm 1 is important, because some rules could cause the firing of others (e.g., an implicit class subsumption relationship could cause the inference of additional instantiation relationships, so R3 must be applied after R4).

The subroutines called by Algorithm 1 return all the quadruples that can be inferred (using the corresponding inference rule) from their input dataset. Algorithm 2 shows how such a procedure should be implemented for rule R4 (class subsumption transitivity). To identify all transitively induced subsumption relationships, Algorithm 2 needs to identify all subsumption "chains", regardless of length. In the first step (lines 2-4) the algorithm will identify all chains of length 2, by joining the provided explicit quadruples. Longer chains are identified by joining the explicit quadruples with the newly produced ones (lines 9-11). More specifically, in the first execution of the while loop (lines 5-11), explicit quadruples will be joined with newly generated implicit ones (which correspond to chains of length 2) to compute all chains of length 3. These quadruples are put in tmp_now (line 11). In the next execution of the while loop, recently generated implicit quadruples are added to I (the output variable) and joined again with explicit quadruples to generate all chains of length 4. The process continues until we reach a certain chain length in which the while loop does not generate any new quadruples (in which case $tmp\_now = \emptyset$).

The correctness of Algorithm 2 critically depends on the assumption regarding the commutativity and associativity of $\odot$; in fact, the while loop fails to produce some of the quadruples, but these are provably equivalent

Algorithm 2: Class subsumption transitivity rule (R4)

Data: A dataset $\mathcal{D}$
Result: All quadruples implied by $\mathcal{D}$ via R4

```
1  I = ∅; tmp_now = ∅; tmp_prev = ∅;
2  forall the q1 = (u1, subClassOf, u2, l1) ∈ D do
3      forall the q2 = (u2, subClassOf, u3, l2) ∈ D do
4          tmp_now = tmp_now ∪ {(u1, subClassOf, u3, l1 ⊙ l2)}
5  while tmp_now ≠ ∅ do
6      I = I ∪ tmp_now;
7      tmp_prev = tmp_now;
8      tmp_now = ∅;
9      forall the q1 = (u1, subClassOf, u2, l1) ∈ D do
10         forall the q2 = (u2, subClassOf, u3, l2) ∈ tmp_prev do
11             tmp_now = tmp_now ∪ {(u1, subClassOf, u3, l1 ⊙ l2)};
12 return D ∪ I
```

to a produced quadruple, thanks to associativity. This is different from the fact that the same triple could be generated more than once (with different labels) from different sets of quadruples, which is captured by the algorithm by generating multiple quadruples corresponding to the same triple. Algorithm 2 is guaranteed to terminate as long as the class hierarchy is acyclic.

The corresponding algorithm for R1 is similar to Algorithm 2 (just replace $subClassOf$ with $subPropertyOf$). Same comments hold for this algorithm's termination and correctness. The other two considered inference rules (R2, R4) are simpler, as they consist of a simple join of the quadruples of the appropriate form (in the sense of the for loop in lines 2-4 of Algorithm 2). In all cases, identifying the label is straightforward (see lines 4, 11 in Algorithm 2).

4.3.2 Evaluation

The evaluation process is used to compute the concrete values of a given (set of) triple(s) in a closed annotated dataset, according to the association of abstract tags and operators with concrete ones provided by the concrete semantics. The process was explained in Subsection 3.2; the algorithm is straightforward, given adequate subroutines for computing the intermediate results (e.g., $\odot$, $f_{\mathcal{A}}$) and omitted.

4.3.3 Adding Quadruples

The main challenge when adding a quadruple q is the efficient computation of the newly-inferred quadruples. Note that we don't support the addition of stand-alone triples, but assume that the provenance information of the triple (via the corresponding explicit label) is an integral part of the update request (e.g., via named graphs [39, 17, 16]). Note that the computation of the provenance of the inserted triple is part of the problem considered in Chapter 6. Algorithm 3 describes the process of adding a quadruple q, and is based on a partitioning of quadruples into 4 types, depending on the value of the property (p) of q:

Class subsumption quadruples, which are of the form $(s, subClassOf, o, l)$, can fire R4 (transitivity of class subsumption) and R3 (class instantiation) rules only, and are handled in lines 5-15 of Algorithm 3. In the first step (lines 6-7) the new implied quadruple is joined with existing ones that are found "below" the new one in the hierarchy. Then, all produced subsumptions are joined with subsumptions that are found "above" the new one in the hierarchy (lines 8-10). Finally, all new subsumptions are checked to determine whether new instantiations should be produced, according to R3 (lines 11-13).

Algorithm 3: Addition of a Quadruple

Data: A closed dataset $\mathcal{D}$ and an explicit quadruple $q = (s, p, o, a)$
Result: The smallest closed dataset $\mathcal{D}'$ that is a superset of $\mathcal{D} \cup \{q\}$

1 **if** $q \in \mathcal{D}$ **then**
2 **return** $\mathcal{D}$
3 $\mathcal{D}' = \mathcal{D} \cup \{q\}$;
4 $tmp_1 = \emptyset$; $tmp_2 = \emptyset$;
5 **if** $p = subClassOf$ **then**
6 **forall the** $q_1 = (u_1, subClassOf, s, l_1) \in \mathcal{D}$ **do**
7 $tmp_1 = tmp_1 \cup \{(u_1, subClassOf, o, l_1 \odot a)\}$;
8 **forall the** $q_2 = (o, subClassOf, u_2, l_2) \in \mathcal{D}$ **do**
9 $tmp_2 = tmp_2 \cup \{(s, subClassOf, u_2, a \odot l_2)\}$ **forall the** $q_{tmp} = (u_1, subClassOf, o, l_{tmp}) \in tmp_1$ **do**
10 $tmp_2 = tmp_2 \cup \{(u_1, subClassOf, u_2, l_{tmp} \odot l_2)\}$;
11 **forall the** $q_3 = (u_1, subClassOf, u_2, l_3) \in tmp_1 \cup tmp_2 \cup \{q_\}$ **do**
12 **forall the** $q_4 = (u, type, u_1, l_4) \in \mathcal{D}$ **do**
13 $\mathcal{D}' = \mathcal{D}' \cup \{(u, type, u_2, l_3 \odot l_4)\}$;
14 $\mathcal{D}' = \mathcal{D}' \cup tmp_1 \cup tmp_2$;
15 **else**
16 **if** $p = subPropertyOf$ **then**
17 ...;
18 Similar to lines 7-21;
19 ...;
20 **else**
21 **if** $p = type$ **then**
22 **forall the** $q_1 = (o, subClassOf, u_1, a_1) \in \mathcal{D}$ **do**
23 $\mathcal{D}' = \mathcal{D}' \cup \{(s, type, u_1, a \odot a_1)\}$
24 **else**
25 **forall the** $q_1 = (p, subPropertyOf, p_1, a_1) \in \mathcal{D}$ **do**
26 $\mathcal{D}' = \mathcal{D}' \cup \{(s, p_1, o, a \odot a_1)\}$
27 **return** $\mathcal{D}'$

Property subsumption quadruples, which are of the form (s, $subPropertyOf$, o, l), can fire R1 (transitivity of property subsumption) and R2 (property instantiation) rules only. The process for this case is similar to the previous one and omitted for brevity (lines 16-19).
Class instantiation quadruples, which are of the form $(s, type, o, l)$, fire R3 (class instantiation) only. These triples are handled in lines 21-23 of Algorithm 3.
Property instantiation quadruples, which are of the form (s, p, o, l) (for $p \neq subClassOf, p \neq subPropertyOf, p \neq type$), can fire R2 (property instantiation) only, and are handled in lines 24-26 of Algorithm 3.

4.3.4 Deleting Quadruples

Deletion of quadruples is one of the operations where the strength of abstract models becomes apparent. The process consists in identifying (and deleting) all implicit quadruples whose label contains the label of the deleted

one (lines 4-5 of Algorithm 4). The limit case where the deleted quadruple is not in the dataset to begin with is handled in lines 1-2.

Algorithm 4: Deletion of a Quadruple

Data: A closed dataset $\mathcal{D}$ and an explicit quadruple $q = (s, p, o, a)$
Result: The largest closed dataset $\mathcal{D}'$ that is a subset of $\mathcal{D} \setminus \{q\}$
1 **if** $q \notin \mathcal{D}$ **then**
2 **return** $\mathcal{D}$
3 $\mathcal{D}' = \mathcal{D} \setminus \{q\}$;
4 **forall the** $q_1 \in \mathcal{D}$ *such that* $q_1 = (s_1, p_1, o_1, l_1)$ *and* $a \sqsubseteq l_1$ **do**
5 $\mathcal{D}' = \mathcal{D}' \setminus \{q_1\}$;
6 **return** $\mathcal{D}'$

Note that Algorithm 4 supports only the deletion of explicit quadruples; deleting implicit quadruples raises some additional complications which are out of the scope of this work. In particular, to delete an implicit quadruple, one needs to delete at least one explicit as well, otherwise the implicit quadruple will re-emerge due to inference. For example, when deleting q_7 (cf. Table 3.3), one needs to delete either q_1 or q_2. Choosing which quadruple to delete is a non-trivial problem, because the choice should be based on extra-logical considerations and/or user input [28], and is not considered here.

4.3.5 Changing the Semantics

Changing the semantics of the annotation algebra is an important and versatile category of changes, which includes changes like: changing the concrete value associated to an abstract annotation tag; changing the semantics of $\odot$, or the way that different concrete tags associated with the same triple are combined; or changing the concrete values that a triple can take ($\mathcal{A}_\mathcal{S}$).

None of these, seemingly complex, operations requires changes in the dataset. Since we only record how the various quadruples were created, the dataset is not affected by these changes, which are related to how the concrete value associated with a triple is computed. This allows the system administrator to freely experiment with different policies and semantics, without the need for costly recomputations of the values in the dataset.

5 Approaches for SPARQL Provenance Management

5.1 Introduction to the Problem

When queried with SPARQL, annotated data undergoes certain transformations (joins, projections etc), which must be recorded as part of the annotation of the final result. For example, when assessing the trustworthiness of a triple that is in the query result, one must use the trust values of the input triples that were used to derive the output triple [6, 40]. Similar ideas apply for other applications of provenance [50, 42, 53].

As already mentioned, concrete approaches have several drawbacks, such as the inability to deal with multiple users/applications, inefficient updating of data etc. The same problems appear in the case of annotating the results of SPARQL queries, as analyzed in Section 3.3. For this reason, most approaches employ abstract provenance models to tackle these problems. However, in the case of SPARQL query annotation the simple model presented in Section 4.2 is not enough; as explained in Section 4.1, the model should be extended by using one abstract operator for every type of transformation that the query could apply on the input triples (i.e., a single inference accumulator operator is not enough). The various models of provenance differ in the exact operators used and their properties.

5.2 The Relational/SQL Setting

As usual, inspiration to address this problem comes from the relational setting, where the problem has been considered for SQL queries in various works. In that setting, the various operators of relational algebra (σ, π, $\bowtie$ etc) have been reused to denote provenance and annotate the corresponding data in a way that shows which operators were used (and how) to produce it [18, 37]. As an example, when a tuple is the result of a join between two other tuples, then its provenance annotation should be $a_1 \bowtie a_2$, where a_1, a_2 are the provenance annotations of the input tuples and $\bowtie$ is the abstract operator that corresponds to joins. Under this general idea, different abstract models for the relational algebra, with different expressiveness, have been developed.

An important consideration in this respect is the granularity of provenance. Some works apply provenance at the attribute level [12, 32, 63, 14], others consider provenance at the level of tuples [7, 37, 12, 20, 35, 63, 14], whereas others study provenance at the relation level [12, 14].

Another consideration is the fragment of the relational algebra considered. The so-called *positive fragment* is simpler; this fragment allows only *filter-project-join-union* queries, i.e., the monotonic part of SQL. Most works deal with this fragment (e.g., [18, 37, 14, 45]).

The seminal work in this respect is [37], where an abstract provenance model with two abstract operators was defined. The two operators accepted by the model, $+$ and $\cdot$, correspond to the union and natural join operators of relational algebra, and are provably sufficient to express the full positive fragment of SQL. In addition to the operators, two distinguished elements 0 and 1, indicating that a certain tuple is "out of" or "in" the relation (respectively) were defined. Then, the authors argued that any reasonable concretization of the above abstract model should satisfy a number of properties, which essentially stem from the properties of relational algebra; these properties allowed proving that the structure $(K, +, \cdot, 0, 1)$, for any given set K such that $0, 1 \in K$, is a commutative semiring.

When *non-monotonic* operators, i.e., (left-)outer join or relational difference, enter the picture, the problem becomes more difficult. The extra challenge in the more general case is that the existence of a tuple in the query result is sometimes due to the *absence* of a tuple. This is harder to record in the annotation of the result and is certainly not expressible in the semiring framework of [37]. Works addressing this problem (such as [33, 3, 4, 35]) proposed extensions of the semiring model with additional operators that would capture the semantics of non-monotonic operators.

Of particular interest in this respect is the work of Geerts et al. [33], whose extension included a *monus* operator that captures the semantics of relational difference. This approach thus uses three operators, namely $\oplus$ (corresponding to $+$ of [37]), $\otimes$ (corresponding to $\cdot$ of [37]) and the new $\ominus$ (the monus operator, which is used

to capture relational difference). This led to a structure called *monus-semiring* (or *m-semiring* for short) which allowed the annotation of query results for the full SQL.

5.3 The RDF/SPARQL Setting

The obvious solution to solve the problem of annotating the results of queries for the RDF/SPARQL case would be to adapt the corresponding relational/SQL solutions. SPARQL, like SQL, allows monotonic operators (AND, UNION etc) as well as non-monotonic ones (OPTIONAL, DIFFERENCE, MINUS). Extending the relational/SQL solutions to apply in the RDF/SPARQL case was proven easy for the positive fragment [60, 23]; in particular, the classical approach of provenance semirings [37] was shown to be adequate to express provenance for the monotonic SPARQL fragment [60].

However, as with the relational case, the approach of [37] is not adequate for the non-monotonic case. Moreover, for the RDF/SPARQL case an additional difficulty was caused by the fact that the semantics of the DIFFERENCE and OPTIONAL operators of SPARQL [55, 54] is slightly different than the corresponding SQL operators. In particular, when taking the SPARQL difference (say $P_1 \setminus P_2$), SPARQL uses bag semantics for P_1, but set semantics for P_2 (i.e., the multiplicities in P_2 are ignored) [34]. As a result, extensions to the semiring framework that capture the non-monotonic fragment of SQL, such as [33], could not be applied directly for the SPARQL case [34, 60, 45].

In the following sections, we describe two possible solutions for addressing this problem [21, 34]. For reasons of brevity and clarity, we refrained from giving the full technical details of these approaches in this deliverable. For a full technical presentation, the reader is referred to the corresponding papers [21, 34].

5.4 SPARQL Provenance Management Using m-semirings

The first proposed approach towards solving the problem of SPARQL provenance management [21] employs the formalism of m-semirings [33] in order to model the non-monotonic operators of SPARQL using a translation technique (from SPARQL to SQL) to overcome the discrepancy between the semantics of SPARQL and SQL.

More precisely, (m, δ)-semirings are employed, which are m-semirings extended with δ, a generalization of the duplicate elimination operator that represents constant annotations [33]; this is necessary to capture the hybrid semantics of SPARQL DIFFERENCE (which employs both set and bag semantics, as explained above).

The approach is based on a translation of SPARQL queries into equivalent relational ones along the lines of [25]. More precisely, each triple pattern as well as each graph pattern that can appear in a SPARQL query is translated into an appropriate SQL expression; SPARQL DIFFERENCE in particular is encoded through a complex relational expression involving joins, relational set difference and duplicate elimination. Based on this translation, any SPARQL query can be translated into a (possibly complex) SQL query.

Using this translation, the result of a SPARQL query can be obtained via the result of an SQL query. Given that the provenance of the results of an SQL query (for both the monotonic and the non-monotonic fragment) can be obtained using the approach of m-semirings [33], this approach solves the problem of representing the provenance of the result of SPARQL queries.

In summary, to capture the provenance of the result of a SPARQL query, one must first translate the query into the corresponding SQL one, and take the result. The provenance of the SQL result is represented using the abstract model of m-semirings that is described in [33]: the abstract operators of $\oplus, \otimes$ and $\ominus$ are used to describe the provenance, and are concretized according to the application at hand (trustworthiness evaluation, data quality management etc). Note that the concretization should satisfy the properties of $\oplus, \otimes, \ominus$ that are defined in [33].

The only problem with this approach is that the final abstract expression that models the provenance of the query result may be overly complicated, due to the complex translation that is necessary to obtain it. Moreover, the structure of the original SPARQL query is not always preserved during the translation, which could lead sometimes to unintuitive provenance annotations for the resulting triples.

5.5 SPARQL Provenance Management Using spm-semirings

Another effort of solving the problem of SPARQL provenance management for both the monotonic and the non-monotonic operators of SPARQL uses an algebraic structure called *spm-semirings* [34] ("spm" stands for SPARQL minus). The proposed solution is also inspired by the idea of m-semirings [33]: like m-semirings, the approach extends the original semirings framework with a new operator, $\ominus$, whose semantics are adequate for capturing SPARQL non-monotonic operators [34].

$\sigma_{?x=?x}(P_1) \equiv P_1$	$P_1 \bowtie \emptyset \equiv \emptyset$
$P_1 \cup \emptyset \equiv P_1$	$P_1 \cup P_2 \equiv P_2 \cup P_1$
$P_1 \bowtie P_2 \equiv P_2 \bowtie P_1$	$P_1 \cup (P_2 \cup P_3) \equiv (P_1 \cup P_2) \cup P_3$
$P_1 \bowtie (P_2 \bowtie P_3) \equiv (P_1 \bowtie P_2) \bowtie P_3$	$P_1 \bowtie (P_2 \cup P_3) \equiv (P_1 \bowtie P_2) \cup (P_1 \bowtie P_3)$
$P_1 \setminus P_1 \equiv \emptyset$	$P_1 \setminus (P_2 \cup P_3) \equiv (P_1 \setminus P_2) \setminus P_3$
$P_1 \bowtie (P_2 \setminus P_3) \equiv (P_1 \bowtie P_2) \setminus P_3$	$(P_1 \setminus (P_1 \setminus P_2)) \cup (P_1 \setminus P_2) \equiv P_1$

Table 5.1: SPARQL Equivalences

In more detail, spm-semirings accept three operators, $\oplus, \otimes, \ominus$, where the latter is used to capture SPARQL DIFFERENCE (and thus the operator OPTIONAL as well). As already mentioned, the semantics of SPARQL DIFFERENCE are slightly different than the semantics of the corresponding SQL operator, in the sense that SPARQL DIFFERENCE employs both set and bag semantics in a hybrid manner; therefore the semantics of $\ominus$ in [33] is different than the one used in the proposal presented here.

$x \otimes 1 = x$	$x \otimes 0 = 0$
$x \oplus 0 = x$	$x \oplus y = y \oplus x$
$x \otimes y = y \otimes x$	$(x \oplus y) \oplus z = x \oplus (y \oplus z)$
$(x \otimes y) \otimes z = x \otimes (y \otimes z)$	$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
$x \ominus x = 0$	$x \ominus (y \oplus z) = (x \ominus y) \ominus z$
$x \otimes (y \ominus z) = (x \otimes y) \ominus z$	$(x \ominus (x \ominus y)) \oplus (x \ominus y) = x$

Table 5.2: Identities for spm-semirings

The idea is to start by identifying the equivalences that should hold for SPARQL operators; these appear in Table 5.1, and were shown to hold for any SPARQL expressions P_1, P_2, P_3 [58, 34]. Then, spm-semirings are defined to be structures of the form $(K, \oplus, \otimes, \ominus, 0, 1)$ that satisfy the identities in Table 5.2. It can be easily seen that a structure of the form $(K, \oplus, \otimes, \ominus, 0, 1)$ is an spm-semiring if and only if the corresponding SPARQL K-annotated algebra satisfies the equivalences of Table 5.1.

The above formal approach allowed showing that the semantics of $\ominus$ (as expressed in Table 5.2) cannot be captured using $\oplus, \otimes$ alone. This means that the inclusion of the new operator ($\ominus$) in the spm-semiring structure is unavoidable. Further, it was shown that the identities $x \otimes 0 = 0$ and $x \oplus 0 = x$ (from Table 5.2) are not necessary, in the sense that a structure $(K, \oplus, \otimes, \ominus, 0, 1)$ is an spm-semiring iff it satisfies all the remaining identities from Table 5.2. Finally, SPARQL query equivalences, as implied by the SPARQL semantics [55], are respected in the spm-semiring structure (and the corresponding annotations). This means that equivalent queries would lead to equivalent annotations for the (same) output triples.

The above results show that spm-semirings are adequate for our purposes, but do not give any hints as to how to construct an spm-semiring, i.e., the methodology that should be followed in order to define operators that satisfy the identities of Table 5.2. Towards this aim, a structure called *seba structure* (from semiring boolean algebra) was defined, which is a hybrid structure consisting of a commutative semiring and a boolean algebra, as well as mappings between them.

Seba-structures solve the above problem because they provide an easy constructive way to generate an spm-semiring; in particular, the operators $\oplus, \otimes$ stem directly from the corresponding operations of the commutative

semiring in the seba-structure, whereas the $\ominus$ operator is defined constructively using the $\otimes$ operator of the semiring, the complement operator of the boolean algebra and the mappings between the semiring and the boolean algebra of the seba-structure. It can be shown that the above construction gives a direct correspondence between spm-semirings and seba-structures, so spm-semirings can be constructed via the more intuitive seba-structures.

Further, seba-structures can be used to construct the "universal spm-semiring", i.e., the most general spm-semiring. This universal structure provides a concise representation of the provenance of RDF data and SPARQL queries involved.

Summarizing, an abstract model based on the above operators can capture the semantics of SPARQL provenance management, provided that the operators are concretized in a way that they satisfy the identities of Table 5.2. Such concretizations can be used to model various different applications of provenance, including trust, access control, quality management and assessment and others.

6 Provenance Management for SPARQL Updates

6.1 Introduction to the Problem

In previous chapters, we assumed that whenever a new triple is inserted in the dataset, its provenance is given as part of the insert operation. We also assumed that this provenance is a simple label (not a complex one), i.e., the triple comes directly from some source without any intermediate manipulations. In this chapter, we revisit those two assumptions, and address the problem of identifying the provenance tag that is associated with a new triple that was inserted as part of a SPARQL Update operation.

The main problem here, as also in the case of SPARQL queries, is that a new triple may have occurred as the result of a set of transformations (projections, joins etc) over the existing annotated data (as given by the graph pattern associated with the SPARQL Update statement), and these transformations should be recorded as part of the annotation of the inserted triple. Thus, the problem becomes very similar to the problem of annotating the result of SPARQL queries, and in fact many works deal with both problems in the same manner (e.g., [63, 14]). Abstract models are again the underlying model of choice for these works, and same considerations as in the case of querying apply.

Note that the problem of provenance management during updates is interesting for additions (INSERT operations) only. Deletions (DELETE and DELETE DATA) are not interesting from the perspective of provenance, because they simply cause the deletion of a triple (along with its provenance), and no provenance needs to be recorded [63, 14, 13]. Further, INSERT DATA is a special case of INSERT, so it is not considered. DROP and CREATE are about deleting and inserting named graphs rather than triples; in this first version of the work, we are focusing on triples only, so these operations are not considered either. Finally, other operations such as DELETE/INSERT, LOAD, CLEAR, COPY, MOVE, ADD, can be expressed as adequate sequences of the above operations so they are omitted as well.

6.2 Related Work

Compared to querying, the problem of provenance management during updates is less well-understood. To the best of our knowledge, all the related work is actually dealing with the relational setting. Moreover, most existing works deal with provenance at the tuple level (the approach of [14] is an exception in this respect, as it deals with all three levels of provenance granularity, namely at the attribute, tuple and relation level).

In most approaches, the creation (and insertion) of a new tuple that is "constructed" from constant values or from values taken from different tuples is assumed to take a "default" provenance (often called "empty", or "blank" provenance), indicated by a special symbol ($\perp$) [63, 13]. This loss of information on how said triple was created is an inherent problem of dealing with provenance at the tuple level; discriminating between the (different) provenance of each attribute can only be done in models that work at the attribute level. In addition, the provenance model considered is why and where provenance, rather than how provenance; as a result, these works do not consider how the tuple was created (i.e., the transformations that the source data underwent), only the origin of the source data. Given these shortcomings, the only case that these works essentially address is when a tuple is copied from one table (or database) to another; in this case the provenance records the origin of the original tuple [63, 13].

Another related approach is [36], where schema mappings are used to express the provenance of a tuple; this tuple can be inserted from the user of the database or it can be copied from another database. It is worth mentioning that this model is more expressive than why-provenance or lineage, since it relies on the properties of semirings (see [37]).

6.3 Some Thoughts on Provenance Management for SPARQL Updates

The problem of identifying the provenance tag of an inserted triple is quite similar to the problem of computing the provenance of the result of a SPARQL query, but raises some additional complications. Nevertheless, as we

will show below, abstract models are in principle capable of addressing the problem of provenance management during SPARQL Updates, albeit using a more complex representation. In the following, we explain the differences between the two problems and propose some ideas towards overcoming the associated difficulties, based on the general ideas presented in previous chapters.

One important complication arises because of the fact that inserted triples are actually placed in some user-defined named graph. This is in contrast to the case of triples produced via inference (which are placed in custom, system-defined graphs, such as named graphs representing graphsets [27]), and is also different from triples constructed via SPARQL queries (which are not placed in any named graph whatsoever). This fact raises some complications as related to the representation of provenance: named graphs are no longer adequate for this purpose.

To be more specific, it has been argued that named graphs (or their extension via graphsets [27]) can be used to store the provenance of a triple [17, 16]. Despite the inherent difficulties of modeling a complex provenance label (such as those occurring during complex inference paths) via a named graph, it is, in principle, possible to associate each such label with a distinct URI, which plays the role of the corresponding named graph (and represents the triple's provenance); thus, the above argument is valid for the case of inference and querying.

However, this is no longer the case when it comes to modeling the provenance of newly inserted triples. In fact, SPARQL Update statements explicitly specify the named graph where the newly inserted triple should be placed. As a result, two triples with a totally irrelevant provenance label may be forced to co-exist in the same named graph.

To address this issue, we must define the provenance of a triple to be something different than the named graph that it is stored in. To avoid ambiguities, we will use the term *core provenance* (or simply *provenance*) to refer to the actual origin of the triple and the term *named graph* to refer to the named graph that the triple resides in.

An associated problem is what exactly is the core provenance. In this respect, we can reuse works such as those mentioned in Chapter 5; more precisely, the core part should be an abstract expression, that describes the operations (i.e., transformations, such as join, selection etc) that took place in order for said triple to be created, and should also identify the triples that were involved in such operations. Obviously, all the previous work that has been proposed in classical abstract provenance models (e.g., [37, 33]) can be exploited here.

To identify and describe the triples and the corresponding operations that were involved in the provenance of a triple we need identifiers. However, using just the triple ID is not enough, because the same triple could appear twice in different, or even in the same, named graph. Thus, the ID used should uniquely identify the triple itself, its core provenance, and the named graph it resides in. This is in contrast to the annotation tags (introduced in Chapter 3), which uniquely identify the triple and the named graph only. At a physical level we could use some URI as the ID of the triple, and associate this URI/ID with the triple itself, the named graph and the provenance information using, e.g., reification techniques.

ID	$subject$	$property$	$object$	$namedgraph$	$provenance$
ID_1	p_1	$parent$	p_2	n_1	$\perp$
ID_2	p_2	$parent$	p_3	n_2	$\perp$
ID_3	p_3	$parent$	p_4	n_3	$\perp$

Table 6.1: Modeling Provenance Information During Updates (Initial)

For visualization purposes, it is much simpler to use a table with several columns. For example, Table 6.1 shows a dataset with three triples representing parenthood information, which are located in named graphs n_1, n_2, n_3 and have provenance information $\perp$, indicating that these triples have unknown provenance; their IDs are ID_1, ID_2, ID_3 respectively.

Now suppose a SPARQL Update statement that requests a copy of ID_1 into n_2. This should create a new row (with ID ID_4) in the above table, as shown in Table 6.2. Note that we used ID_1 as the provenance of the new triple, to indicate that said triple originates via a copy from the triple with ID ID_1. If the new triple (ID_4) is subsequently copied back in n_1, then a new triple (with ID ID_5 – cf. Table 6.2) would be created. Note that

this triple's provenance is ID_4; to trace back the fact that this triple originally stems from ID_1, one needs to recursively navigate the provenance information of said triples.

ID	*subject*	*property*	*object*	*namedgraph*	*provenance*
ID_1	p_1	*parent*	p_2	n_1	$\perp$
ID_2	p_2	*parent*	p_3	n_2	$\perp$
ID_3	p_3	*parent*	p_4	n_3	$\perp$
ID_4	p_1	*parent*	p_2	n_2	ID_1
ID_5	p_1	*parent*	p_2	n_1	ID_4

Table 6.2: Modeling Provenance Information During Updates (After Copy)

Another important observation here is that it so happens that the triple $(p_1, parent, p_2)$ appears twice in n_1 (ID_1, ID_5). Such duplicates could also appear through a complex transformation that reconstructs a triple (through SPARQL Update statements). These duplicates should be maintained, as they have a different core provenance, and could subsequently be used for the construction of other triples. A possible alternative would be to join the core provenance labels of all duplicates into one large label, but here we opt for the simpler solution of keeping the duplicates – see also Section 4.1 for a related discussion. Our approach of generating IDs that uniquely identify the tuple (triple, named graph, provenance) allows supporting this. In contrast, note that abstract annotation tags (described in Section 4.2 and used, e.g., in Table 3.3) only identify the triple and its named graph (which is also its provenance in the models presented in previous chapters), but this is no longer enough for the problem considered here.

The provenance tag associated to the inserted triples should reflect the transformations that the original triples underwent to construct the new ones. In the above example, the "transformation" is just the copying, but more complicated cases can arise. In such cases, as already mentioned, we could reuse the classical approaches on annotating the results of SPARQL queries (e.g., [37, 33, 21, 34, 60]) in order to determine the correct annotation.

As an example, let's consider again the original dataset (Table 6.1), and suppose that the user executes the following SPARQL Update statement:

```
INSERT {GRAPH <n4> {?s <grandparent> ?o} }
WHERE {
  ?s <parent> ?x .
  ?x <parent> ?o
}
```

This would join the pair (ID_1, ID_2), as well as the pair (ID_2, ID_3), producing the triples ID_6, ID_7 respectively (see Table 6.3); using the general approach of abstract models, the core provenance of the new triples would be $ID_1 \otimes ID_2$, $ID_2 \otimes ID_3$ respectively, under the notation of [34].

ID	*subject*	*property*	*object*	*namedgraph*	*provenance*
ID_1	p_1	*parent*	p_2	n_1	$\perp$
ID_2	p_2	*parent*	p_3	n_2	$\perp$
ID_3	p_3	*parent*	p_4	n_3	$\perp$
ID_6	p_1	*grandparent*	p_3	n_4	$ID_1 \otimes ID_2$
ID_7	p_2	*grandparent*	p_4	n_4	$ID_2 \otimes ID_3$

Table 6.3: Modeling Provenance Information During Updates (After Join)

The above example leads us to the second major difference between modeling the provenance of query results, as opposed to modeling the provenance of newly created triples. This difference is derived from the fact that a triple returned by a query (or even during inference) is not subsequently used anywhere (i.e., it may not

affect the provenance of triples in future queries), whereas an inserted triple is actually stored (like ID_6, ID_7 in Table 6.3); thus, the provenance of triples acquired in future SPARQL queries or SPARQL Update statements may be affected by its existence. In other words, the provenance label acquired by a triple during querying is lost after the query, as the triple is not physically stored anywhere, but in the case of INSERT, the triple is physically stored, so its provenance label could be used later to produce other, more complex provenance labels.

As an example, let's assume that we want to identify (and add to the dataset) all the great-grandparents that are known to the dataset. There are two (equivalent) ways to do that: we could either perform a triple join on the *parent* property, or we could re-use the existing *grandparent* information (see Table 6.3) and perform a double join on a pair of *grandparent-parent* relationships. The two alternative SPARQL Update statements are shown below:

```
INSERT {GRAPH <n5> {?s <greatgrandparent> ?o} }
WHERE {
  ?s <parent> ?x .
  ?x <parent> ?y .
  ?y <parent> ?o
}

INSERT {GRAPH <n5> {?s <greatgrandparent> ?o} }
WHERE {
  ?s <grandparent> ?x .
  ?x <parent> ?o
}
```

The two methods are equivalent, in terms of the triples they will produce; the equivalence stems from the definition of the *grandparent* property. Nevertheless, they are different in terms of the provenance of the produced triples. More specifically, in the first case, the produced triple, namely $(p_1, greatgrandparent, p_4)$, with ID ID_8, would have a core provenance $ID_1 \otimes ID_2 \otimes ID_3$; in the second case, the same triple would be produced (say with ID ID_9), but with provenance $ID_6 \otimes ID_3$.

Our first observation is that the second methodology would produce a "nested" provenance label (see the label of ID_9); recall that this case appears also in the case of triples produced via inference, but is not an issue for provenance labels associated with SPARQL query results. A simpler form of "nesting" appeared also in Table 6.2 (see ID_5 in said table). This constitutes another difference between the provenance models of SPARQL query and update.

The second observation is that the provenance labels produced in the two alternative cases are not equivalent. This means that "unfolding" the provenance of ID_9 into $ID_1 \otimes ID_2 \otimes ID_3$ would not be correct, because the query explicitly joined ID_6 with ID_3. Recall that this "unfolding" is possible in the case of inferred triples, thanks to the associativity and commutativity property of the corresponding operator ($\odot$); in the case of the SPARQL Update provenance, despite the fact that the $\otimes$ operator is defined as associative and commutative [37, 33], this unfolding is not possible.

Even though the problem of identifying and representing the provenance of newly created triples is similar to the problem of representing the provenance of SPARQL query results, this chapter showed that there are important differences too. In summary, the provenance of SPARQL query results can be represented using named graphs, but this is not the case for newly inserted triples, where a more complex structure is necessary. Furthermore, provenance for SPARQL query results does not produce "nested" provenance labels, whereas SPARQL Updates produce unfoldable nested labels. The above results show that further research and understanding of the properties of provenance representation for the case of SPARQL Update is needed.

7 Conclusions

Recording the origin (provenance) of data is important to determine reliability, accountability, attribution or reputation, and to support various applications, such as trust, access control, privacy policies etc. This is even more imperative in the context of LOD, whose unmoderated nature emphasizes the need to know and record the source of the data.

This deliverable studies the problem of provenance management under different settings, and theoretically evaluates the use of abstract models for this purpose. Abstract models can be used to support various applications of provenance, such as trust or access control, and are particularly useful for dynamic datasets (in the presence of inference), as well as in cases where the exact provenance needs to be recorded (rather than its consequence, such as the exact trust value or accessibility token it implies); this can happen, e.g., when unexpected uses of the provenance information could appear or when several users/application access the same data, but employ different semantics on the provenance values.

In this deliverable, we elaborated on abstract models, by describing in detail how abstract models can be used for representing provenance, as well as for the applications of provenance, such as trust. Moreover, we defined abstract models and showed algorithms for provenance management in the presence of inference and updates.

Our analysis showed that abstract models are more suitable for dynamic datasets, as well as in environments where the application supported by provenance (e.g., trust or access control) has dynamic semantics, or needs to support diverse policies/semantics which are fluctuating in a periodical, user-based or event-based fashion. Thus, they are adequate for dynamic, interlinked datasets in which both the dataset itself and/or our assessment regarding their quality or trustworthiness may change.

On the other hand, as explained in Subsection 3.2.2, there is also a tradeoff involved, namely that abstract models provide significant gains in update efficiency, at the expense of a worse performance during initialization, query answering and storage space. Future work (to appear in the upcoming Deliverable D3.3) will experimentally evaluate this tradeoff.

Another case where abstract models are useful is in the context of annotating the results of SPARQL queries. In this respect, our efforts (as well as most of the related literature) concentrated on adapting the corresponding results from the database community (used to annotate the results of SQL queries) for the RDF/SPARQL context. The main challenges in both settings are related to the non-monotonic constructs of SQL/SPARQL, so this deliverable focused on the non-monotonic SPARQL features. In particular, we proposed two different ways to deal with the problem using the algebraic structures of m-semirings and spm-semirings [21, 34].

Finally, this deliverable studied the problem of identifying the provenance value of triples included in a dataset via complex SPARQL Update statements. We showed that, despite the similarities, the problem is more difficult than the corresponding problem of identifying the provenance of SPARQL query results and presented some preliminary ideas towards a solution.

REFERENCES

[1] F. Abel, J. L. De Coi, N. Henze, A. Wolf Koesling, D. Krause, and D. Olmedilla. Enabling advanced and context-dependent access control in RDF stores. In *Proceedings of the 6^{th} International Semantic Web Conference and the 2^{nd} Asian Semantic Web Conference (ISWC/ASWC-07)*, 2007.

[2] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *Proceedings of the Linked Data on the Web Workshop (LDOW-09)*, 2009.

[3] Yael Amsterdamer, Daniel Deutch, and Val Tannen. On the limitations of provenance for queries with difference. In *TaPP*, 2011.

[4] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, 2011.

[5] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.

[6] D. Artz and Y. Gil. A survey of trust in computer science and the semantic web. *Web Semantics*, 5(2), 2007.

[7] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American-American Edition*, 2001.

[9] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *The VLDB Journal*, 14:373–396, 2005.

[10] D. Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF Schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210`, 2004.

[11] J. Broekstra and A. Kampman. Inferencing and truth maintenance in RDF schema. In *Proceedings of the Workshop on Practical and Scalable Semantic Systems (PSSS-03)*, 2003.

[12] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, 2001.

[13] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD-06)*, pages 539–550, 2006.

[14] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions in Database Systems*, 33(4):28:1–28:47, 2008.

[15] Peter Buneman and Egor V. Kostylev. Annotation algebras for RDFS. In *Proceedings of the 3^{rd} International Workshop on the role of the Semantic Web in Provenance Management (SWPM-11)*, 2011.

[16] J.J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *Proceedings of the 3^{rd} International Semantic Web Conference (ISWC-04)*, 2004.

[17] J.J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs. *Journal of Web Semantics*, 3(4), 2005.

[18] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how and where. *Foundations and Trends in Databases*, 1(4):379–474, 2007.

[19] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: A future history. In *Proceedings of the 24^{th} ACM SIGPLAN Conference Companion on Object Oriented Programming Systems, Languages and Applications (OOPSLA-09)*, pages 957–964, 2009.

[20] Yingwei Cui and Jennifer Widom. Lineage Tracing for General Data Warehouse Transformations. In *VLDB*, 2001.

[21] C.V. Damasio, A. Analyti, and G. Antoniou. Provenance for SPARQL queries. In *Proceedings of the* 12^{th} *International Semantic Web Conference (ISWC-13)*, 2013.

[22] S. Dietzold and S. Auer. Access control on RDF triple store from a Semantic Wiki perspective. In *Proceedings of the ESWC Workshop on Scripting for the Semantic Web*, 2006.

[23] R. Dividino, S. Sizov, S. Staab, and B. Schueler. Querying for provenance, trust, uncertainty and other meta knowledge in RDF. *Journal of Web Semantics*, 7(3), 2009.

[24] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):251–272, 1979.

[25] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *Proceedings of the 2009 International Database Engineering & Applications Symposium (IDEAS-09)*, pages 31–42, 2009.

[26] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598, 2004.

[27] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides. Coloring RDF triples to capture provenance. In *Proceedings of the* 8^{th} *International Semantic Web Conference (ISWC-09)*, 2009.

[28] G. Flouris, G. Konstantinidis, G. Antoniou, and V. Christophides. Formal foundations for RDF/S KB evolution. *International Journal on Knowledge and Information Systems (KAIS)*, 2013.

[29] G. Flouris, Y. Roussakis, M. Poveda-Villalon, P. N. Mendes, and I. Fundulaki. Using provenance for quality assessment and repair in Linked Open Data. In *Proceedings of the* 2^{nd} *Joint Workshop on Knowledge Evolution and Ontology Dynamics (EvoDyn-12)*, 2012.

[30] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and provenance. In *Proceedings of the ACM SIGMOD/PODS Conference*, 2008.

[31] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 update. `http://www.w3.org/TR/sparql11-update/`, March 2013.

[32] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. In *ICDE*, 2006.

[33] F. Geerts and A. Poggi. On database query languages for k-relations. *Journal of Applied Logic*, 8(2), 2010.

[34] Floris Geerts, Grigoris Karvounarakis, Vassilis Christophides, and Irini Fundulaki. Algebraic structures for capturing the provenance of SPARQL queries. In *Proceedings of the Joint EDBT/ICDT 2013 Conference*, 2013.

[35] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the* 25^{th} *International Conference on Data Engineering (ICDE-09)*, 2009.

[36] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the* 33^{rd} *International Vonference on Very Large Databases (VLDB-07)*, pages 675–686, 2007.

[37] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the* 26^{th} *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-07)*, pages 31–40, 2007.

[38] A. Gupta and I.S. Mumick, editors. *Materialized views: techniques, implementations, and applications*. MIT Press, Cambridge, MA, USA, 1999.

[39] H. Halpin and J. Cheney. Dynamic provenance for SPARQL updates using named graphs. In *Workshop on the Theory and Practice of Provenance (TaPP-11)*, 2011.

[40] O. Hartig. Querying trust in RDF data with tSPARQL. In *Procedings of the* 6^{th} *European Semantic Web Conference (ESWC-09)*, 2009.

[41] O. Hartig and J. Zhao. Publishing and consuming provenance metadata on the web of linked data. In *Proceedings of the* 3^{rd} *International Provenance and Annotation Workshop (IPAW-10)*, 2010.

[42] H. Huang and C. Liu. Query Evaluation on Probabilistic RDF Databases. In *WISE*, 2009.

[43] A. Jain and C. Farkas. Secure resource description framework. In *Proceedings of the* 11^{th} *ACM Symposium on Access Control Models and Technologies (SACMAT-06)*, 2006.

[44] G. Karvounarakis and T.J. Green. Semiring-annotated data: queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[45] Grigoris Karvounarakis, Irini Fundulaki, and Vassilis Christophides. Provenance for linked data. In Val Tannen, editor, *Festschrift celebrating Peter Buneman*. (to appear).

[46] J. Kim, K. Jung, and S. Park. An introduction to authorization conflict problem in RDF access control. *Innovation in Knowledge-Based and Intelligent Engineering Systems (KES) journal*, 2008.

[47] J. Kotowski and F. Bry. A perfect match for reasoning, explanation, and reason maintenance: OWL 2 RL and semantic wikis. In *Proceedings of* 5^{th} *Semantic Wiki Workshop*, 2011.

[48] D. Le-Phuoc, A. Polleres, M. Hauswirth, G. Tummarello, and C. Morbidoni. Rapid prototyping of semantic mash-ups through semantic web pipes. In *Proceedings of the* 18^{th} *International World Wide Web Conference (WWW-09)*, 2009.

[49] F. Manola, E. Miller, and B. McBride. RDF primer. `www.w3.org/TR/rdf-primer`, 2004.

[50] Mauro Mazzieri and Aldo Franco Dragoni. A Fuzzy Semantics for the Resource Description Framework. In *URSW*, 2008.

[51] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J.V. den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27:743–756, 2011.

[52] H. Muhleisen, M. Kost, and J.-C. Freytag. SWRL-based access policies for linked data. In *Proceedings of the* 2^{nd} *Workshop on Trust and Privacy on the Social and Semantic Web (SPOT-10)*, 2010.

[53] V. Papakonstantinou, M. Michou, I. Fundulaki, G. Flouris, and G. Antoniou. Access control for RDF graphs using abstract models. In *Proceedings of the* 17^{th} *ACM Symposium on Access Control Models and Technologies (SACMAT-12)*, 2012.

[54] J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Retrieved from: `http://users.dcc.uchile.cl/~cgutierr/ftp/sparql_semantics.pdf`.

[55] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3), 2009.

[56] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[57] P. Reddivari, T. Finin, and A. Joshi. Policy-based access control for an RDF store. In *IJCAI Workshop on Semantic Web for Collaborative Knowledge Acquisition*, 2007.

[58] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proceedings of the Joint EDBT/ICDT 2010 Conference*, 2010.

[59] C. Strubulis, Y. Tzitzikas, M. Doerr, and G. Flouris. Evolution of workflow provenance information in the presence of custom inference rules. In *Proceedings of the* 3^{rd} *International Workshop on the Role of Semantic Web in Provenance Management (SWPM-12)*, 2012.

[60] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, 2011.

[61] D. Tomaszuk, K. Pak, and H. Rybinski. Trust in RDF graphs. In *Proceedings of the* 17^{th} *East-European Conference on Advances in Databases and Information Systems (ADBIS-13)*, 2013.

[62] Octavian Udrea, Diego Reforgiato Recupero, and V. S. Subrahmanian. Annotated RDF. *ACM Transactions on Computational Logic*, 11(2), 2010.

[63] S. Vansummeren and J. Cheney. Recording provenance for SQL queries and updates. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 30(4):29–37, 2007.

[64] S. Villata, N. Delaforge, F. Gandon, and A. Gyrard. An access control model for linked data. In *Proceedings of On The Move to Meaningful Internet Systems Workshops 2011 (OTM-11)*, 2011.

[65] F. Zablith, G. Antoniou, M. d'Aquin, G. Flouris, H. Kondylakis, E. Motta, D. Plexousakis, and M. Sabou. Ontology evolution: A process centric survey. *Knowledge Engineering Review (KER)*, (to appear).

[66] Antoine Zimmermann, Nuno Lopes, Axel Polleres, and Umberto Straccia. A general framework for representing, reasoning and querying with annotated Semantic Web data. *Journal of Web Semantics*, 11, 2012.