Controlling Access to RDF Data using Abstract Models

Vassilis Papakonstantinou

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisor: Prof. *Dimitris Plexousakis*

This work has been performed at the **University of Crete, School of Science and Engineering, Computer Science Department** and at the **Institute of Computer Science (ICS) - Foundation for Research and Technology - Hellas (FORTH), Heraklion, Crete, GREECE.**

The work is partially supported by the **PlanetData NoE (FP7:ICT-2009.3.4, #257641)**

University of Crete
Computer Science Department

Controlling Access to RDF Data using Abstract Models

Thesis submitted by
Vassilis Papakonstantinou
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Vassilis Papakonstantinou

Committee approvals: _____
Dimitris Plexousakis
Professor, University of Crete

_____
Irini Fundulaki
Principle Researcher, Foundation for Research and Technology (FORTH)

_____
Maria Papadopouli
Associate Professor, University of Crete

Departmental approval: _____
Angelos Bilas
Professor, Director of Graduate Studies, University of Crete

Heraklion, February 2013

Abstract

The number of applications that publish and exchange possibly sensitive RDF data continuously increases in a large number of domains ranging from bioinformatics to e-government. In light of the sensitive nature of the available information, the issue of securing RDF content and ensuring the selective exposure of information to different classes of users is becoming all the more important.

This thesis studies the problem of providing *secure access to RDF data* taking into account RDFS *inference* and *propagation* of access labels along the RDFS *class* and *property hierarchies*.

The majority of the state of the art approaches for RDF access control use annotation models where each triple is assigned a *concrete value* as *access label* that determines whether the triple is *accessible or not*. In these models the computation of the access label of a triple (via implication or propagation) is done once, in a fixed manner according to predefined semantics. Hence, when the initial assignment of the access labels to triples or the semantics on how the implied labels are computed change, then the labels of all the implied triples in the dataset must be recomputed. This also holds when data, or even the way that labels are assigned to triples change.

To address those shortcomings, we propose the use of abstract access control models, in which the access label of a triple is not a concrete value, but an *algebraic expression* that encodes exactly *how* the access label of an implied or propagated triple was computed, that is which triples were involved in the implication or propagation thereof. This way, we can easily determine the triples that are affected by each change in the dataset or in the authorizations, and act accordingly, by recomputing only the affected labels, rather than the labels of entire dataset.

The flexibility of the proposed model to handle different applications with diversified needs, simplifies the maintenance of an access control-enhanced dataset. The abstract approach generalizes in a straightforward manner the existing RDF access control models that consider RDFS semantics since they can be considered as specific concretizations of the general model. More specifically, the model can be used in situations that consider different and/or dynamic datasets, authorizations, application requirements and access control semantics.

Περίληψη

Ο αριθμός των εφαρμογών που δημοσιεύουν και ανταλλάσσουν ενδεχομένως ευαίσθητα RDF δεδομένα αυξάνει συνεχώς σε ένα μεγάλο αριθμό θεματικών περιοχών (βιοπληροφορική, ηλεκτρονική διακυβέρνηση). Υπό το φως της ευαίσθητης φύσης των διαθέσιμων πληροφοριών, ο ελέγχος πρόσβασης και η διασφάλιση της επιλεκτικής έκθεσης πληροφοριών RDF έχουν αναδειχθεί σε πολύ σημαντικά ερευνητικά ζητήματα.

Η συγκεκριμένη εργασία μελετά το πρόβλημα της ασφαλούς πρόσβασης σε δεδομένα RDF λαμβάνοντας υπόψη (α) τις *συλλογιστικές διαδικασίες* (*inference rules*) όπως ορίζονται από την γλώσσα αναπαράστασης σχημάτων RDF, RDFS και (β) τις *διαδικασίες διάδοσης* (*propagation rules*) πάνω από τις *ιεραρχίες* RDF *κλάσεων* και *ιδιοτήτων*. Η πλειοψηφία των υπαρχόντων προσεγγίσεων για τον έλεγχο πρόσβασης σε δεδομένα RDF, χρησιμοποιεί *μοντέλα σχολιασμού* (*annotation model*) όπου στην κάθε *τριπλέτα* (*triple*) ανατίθεται μια συγκεκριμένη τιμή ως ετικέτα πρόσβασης η οποία προσδιορίζει εάν η τριπλέτα είναι *προσβάσιμη* (*accessible*) ή όχι από ένα χρήστη. Στα συγκεκριμένα μοντέλα, ο υπολογισμός της τιμής της ετικέτας γίνεται μόνο μία φορά με διαδικασίες οι οποίες έχουν καλώς ορισμένη σημασιολογία.

Ως εκ τούτου, όταν η αρχική ανάθεση των ετικετών πρόσβασης σε τριπλέτες RDF αλλάζει, τότε πρέπει να επανυπολογιστούν οι ετικέτες των τριπλετών που προκύπτουν από συλλογιστικές διαδικασίες ή από την διάδοση στις ιεραραχίες κλάσεων και ιδιοτήτων. Το ίδιο συμβαίνει και στην περίπτωση στην οποία αλλάζει ο τρόπος με τον οποίο υπολογίζεται η ετικέτα των τριπλετών που προκύπτουν από συλλογιστικές διαδικασίες.

Για την αντιμετώπιση των παραπάνω θεμάτων, προτείνουμε τη χρήση ενός *αφηρημένου μοντέλου ελέγχου πρόσβασης* (*abstract access control model*) σε δεδομένα RDF. Η ετικέτα πρόσβασης μίας τριπλετας δεν είναι μία συγκεκριμένη τιμή αλλά μία *αλγεβρική έκφραση* (*algebraic expression*) η οποία κωδικοποιεί ακριβώς πως η ετικέτα πρόσβασης μιας, παραγόμενης μέσω συλλογιστικών διαδικασιών ή διαδικασιών διάδοσης, τριπλέτας υπολογίζεται. Πιο συγκεκριμένα, ποιές είναι εκείνες οι τριπλέτες οι οποίες συμμετείχαν στον υπολογισμό της.

Με αυτό τον τρόπο, μπορούμε να προσδιορίσουμε εύκολα τις τριπλέτες που επηρεάζονται από αλλαγές στο σύνολο των δεδομένων. Συνεπώς, δεν χρειάζεται να υπολογιστούν εκ νέου οι ετικέτες όλων των τριπλετών παρά μόνο εκείνων οι οποίες σχετίζονται με αυτές οι οποίες έχουν επηρεαστεί από αλλαγές.

Η ευελιξία του προτεινόμενου μοντέλου για να χειριστεί διαφορετικές εφαρμογές με διαφοροποιημένες ανάγκες, απλοποιεί τον έλεγχο πρόσβασης σε σύνολα δεδομένων τα οποία επιδέχονται αρκετές αλλαγές. Η προσέγγιση που προτείνεται στην συγκεκριμένη εργασία, γενικεύει με ένα απλό τρόπο τα υπάρχοντα μοντέλα ελέγχου πρόσβασης για RDF δεδομένα.

Acknowledgements

There are so many people that I would like to thank, each one helped me with their own special way. First and foremost I offer my sincerest gratitude to my advisor Professor Dimitris Plexousakis for supervising, guiding and having trust in this work.

In addition, I would like to deeply thank my co-supervisors Dr. Irini Fundulaki and Dr. Giorgos Flouris for their valuable guidance and their useful advice. Their ideas were crucial to the completion of this work, but mainly I would like to thank them for their continuous enthusiasm and willingness to help me at every stage of this thesis. During my studies in the Master's degree they introduced me to the notion of research, giving me the bases for the continuation of my studies and for my future career, so I am grateful to them.

Also, I need to express my gratitude to the University of Crete and the Department of Computer Science for providing me with proper education; as well as the Institute of Computer Science of the Foundation for Research and Technology (ICS-FORTH) for supporting me.

Moreover, I would like to give my appreciation to my close friends, Anastasios Papagiannis and Antonis Papaioannou for the encouragement and the support during my studies, but also for the great moments we had together during these last two years. Furthermore, I would like to thank Maria, Georgia and especially Panagiotis for their valuable comments and suggestions during the preparation of my thesis presentation.

Last but not least I am grateful to my father Theofanis, my mother Evaggelia, as well as my sisters Eleni and Maria for the encouragement and the support in every single aspect of my life.

Herakleion–Crete, Vassilis Papakonstantinou
February 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

RDF [1] has established itself as a widely used standard for representing data in the Semantic Web. Several commercial and academic efforts, such as the W3C Linked Open Data initiative [2], target the development of RDF datasets. The popularity of the RDF data model [1] and the RDF Schema language (RDFS) [3] is due to the flexible and extensible representation of information under the form of *triples*. An RDF triple *(subject, property, object)* asserts the fact that *subject* is associated with *object* through *property*. RDFS is used to add semantics to RDF triples, through the *inference rules* specified in the RDF Schema language [4] W3C Standard that entail new *implicit (implied)* triples.

The number of applications that publish and exchange possibly sensitive RDF data continuously increases in a large number of domains ranging from bioinformatics [5] to e-government[1]. In light of the sensitive nature of the available information, the issue of *securing* RDF content and *ensuring the selective exposure of information* to different classes of users is becoming all the more important.

In our work we focused on the problem of providing secure access to RDF data taking into account RDFS *inference* and the *propagation of access labels* along the RDFS *class* and *property* hierarchies. In the case of inference the scenario we consider is that the label of an implied triple depends on the labels of the ones that imply it [6]. Propagated labels are useful when one wants to enforce the inheritance of labels along the RDFS class and property hierarchies: for example, an application may consider that an object should inherit the label of its class. This is a common approach in XML access control where the labels are inherited from parent to its children nodes in an XML tree.

The majority of the state of the art approaches for RDF access control [7, 8, 9, 10, 11] use *annotation models* where each triple is assigned a *concrete value* as access label that determines whether the triple is accessible or not. In these models

[1] `http://data.gov.uk/`, `http://www.data.gov/`

the computation of the access label of a triple (via implication or propagation) is done in a fixed manner according to predefined semantics, and any change in the dataset or the authorizations would force the recomputation of the triple's access permission [12]. The same would be true when one chooses to change the way that the labels relate to each other, or the access control policy in general. For instance, an application may consider a concrete policy where the access label of a triple takes one of the values "public", "confidential", "secret" and "top-secret". The semantics of the application are hard-coded inside the policy specification: the policy considers that a triple gets the "public" access label if all the triples that contributed to its implication, are labeled as "public" (we will use the term "implying triples" to refer to this set). Consequently, when the initial assignment of the access labels to triples or the semantics on how the implied labels are computed change, then the labels of all the implied triples in the dataset must be recomputed.

One possible solution to this problem is to store *how the access label of a triple is computed*. To do so, we advocate the use of an *annotation model* in which we *do not commit* to a specific assignment of values as access labels of triples and predefined semantics for computing the access labels of those triples obtained through *inference* and *propagation*. Instead, we use an *abstract access control model* defined by *abstract tokens and operators*. Tokens are assigned to explicit triples through *authorizations* or *access control rules*. Operators are used to encode the operators employed to compute the *labels of implied triples* and those labels obtained through *propagation*. In this way, the access label of a triple is an *abstract expression*, consisting of tokens and operators, which describes *how the access label of said triple is computed*. This allows the quick identification of the labels affected by any given change, making the recomputation of the entire dataset's accessibility information unnecessary. Our approach is inspired by *how provenance models* [13] that have been defined for relational data provenance.

To determine the actual label of a triple associated with an abstract expression in order to decide whether a triple is accessible or not, one should concretize the respective abstract tokens and operators. This is done using a *concrete policy*, defined by the corresponding application. Using this policy, one can compute the value of the abstract expression, i.e., the concrete label associated with the triple in question. Based on the computed value and the semantics of the policy, one can decide whether the triple is accessible or not.

The main benefits of the proposed model are sketched below:

- When abstract models are considered, the abstract labels for implied triples and the propagated labels are computed *only once*. The application can then adopt a concrete policy to serve its needs. The concrete labels of triples are only computed at query time (on a need-to-have basis), hence applications can easily experiment with different concrete access policies without needing

to recompute the access labels each time since only the value of an abstract expression needs to be computed. Similarly, our approach can support applications that dynamically adapt their policies, e.g., on a per-user basis.

- In standard annotation models, a change in the assigned access label of a triple would require a complete recomputation of the access labels of all triples obtained through propagation and inference. This must be done in order to ensure the correctness of annotations [12]. In the case of abstract models, the abstract expressions make explicit the tokens and operators involved in the computation of the complex label. Consequently, various types of changes (in both the dataset and in the associated authorizations) can be supported more efficiently; this can lead to important gains especially when large datasets are considered.

- The flexibility of the proposed model to handle different applications with diversified needs simplifies the maintenance of an access control-enhanced dataset. The abstract approach generalizes in a straightforward manner the existing RDF access control models that consider RDFS semantics since they can be considered as specific concretizations of the general model. More specifically, the model can be used in situations that consider different and/or dynamic datasets, authorizations, application requirements and access control semantics. This ability essentially simplifies the maintenance of an access control-enhanced dataset.

In summary, the main contributions of the work are:

- The definition and use of an *abstract access control model* to provide secure access to RDF *triples*, that allows us to determine how the access label of a triple was computed. Triples can be accessed by a SPARQL *construct* query that returns a *subset* of the triples in the queried RDF graph. The abstract control model works on *triples* rather than *resources* (i.e., nodes in an RDF graph). This approach is motivated by the fact that the semantics in an RDF graph are not given by the resources themselves, but by relations as expressed in triples.

  An abstract model is defined by a set of *abstract tokens* and a set of *abstract operators* the latter used to determine how an access label was computed. The abstract operators are used to *(i)* compute the access labels of implied RDF triples and *(ii)* determine the propagated labels of triples. Abstract tokens are assigned to triples through *authorizations* or *access control rules* forming *quadruples*. An authorization is specified through a query that determines the triples that should be assigned the respective token.

- The extension of the standard *RDFS inference rules* for quadruples in order to determine the access labels of implied triples, as well as the definition and formalization of *propagation rules*, that determine how access labels are propagated along the RDFS *class* and *property* hierarchies.

- The description of how abstract access control labels can be used to annotate triples with their access control label (*annotation process*), how concrete policies can be used to determine the accessibility of triples annotated with said abstract labels (*evaluation process*), and how different types of changes in the dataset and/or the accessibility information can be handled without the need to recompute the access control labels of all the triples in the dataset.

- An implementation of our ideas on top of the MonetDB open source RDBMs using the state of the art schemas for storing RDF graphs. Also a set of experiments that shows the overhead of our approach regarding the annotation, as well as its gains in the cases of evaluation and changes.

1.1 Motivating Example

We will use, for illustration purposes, a simple example taken from the FOAF ontology[2] that captures information about agents, persons, organizations and their relationships. To present our work we consider the part of the schema that considers *persons* and we have extended it with class *Student*. Table 1.1 shows RDF graph $\mathcal{G}$ (in tabular form) that we will be considering where each row corresponds to an RDF triple, and columns s, p, o stand for the *subject*, *predicate* and *object* of the RDF triple.

Table 1.1 shows (in tabular form) a set of RDF triples from the FOAF ontology that we will use for illustration purposes.

Access control authorizations are used to assign an *abstract access control token* to RDF triples, as specified by means of a *query*. We say that the triples returned from the evaluation of the query are *in the scope of the authorization*. We rely on authorizations that make use of the SPARQL [14] language to determine the RDF triples concerned by it.

Table 1.2 shows a set of access authorizations defined for the set of RDF triples of Table 1.1. In our work, we use *quadruples* to encode the access label of an RDF triple, and write (s, p, o, l) to denote that l is the access label assigned to triple (s, p, o). In general, quadruples are used to represent *context*, *provenance* or other types of information. Named Graphs [15] are one way of implementing RDF quadruples. Pediaditis et. al. [16] showed that named graphs alone do not suffice

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

	s	p	o
t_1 :	*Student*	sc	*Person*
t_2 :	*Person*	sc	*Agent*
t_3 :	&a	type	*Student*
t_4 :	&a	*firstName*	Alice
t_5 :	&a	*lastName*	Smith
t_6 :	*Agent*	type	class

Table 1.1: RDF Triples

in capturing the labels of implied triples but a more expressive structure is needed. Therefore, they cannot be used as the basis for representing abstract labels.

In discretionary access control models *queries* are used to specify the data objects that are concerned by an authorization or access control rule. We follow a similar approach where we rely on the SPARQL W3C Recommendation Language [14] for querying RDF data to specify queries that determine the triples concerned by the authorization. The SPARQL queries we use are of the form

$$\begin{array}{ll} \texttt{CONSTRUCT} & \{tp\} \\ \texttt{WHERE} & \{gp\} \end{array}$$

with tp and gp being a SPARQL *triple* and a *graph* pattern respectively. Such queries return an RDF graph that is, sets of RDF triples and not a simple assignment of variables to values as in the case of SPARQL `SELECT` queries. Each authorization is a tuple of the form $A_i = (q_i, at_i)$ where q_i is SPARQL query of the above form and at_i is an abstract access token. For instance, the following query for authorization $\mathcal{A}_1$ returns the set of triples with predicate *firstName* whose *subject* resource participates in triples with *predicate type* and *object Student*.

$$\begin{array}{ll} \texttt{CONSTRUCT} & \{?x\ firstName\ ?y\} \\ \texttt{WHERE} & \{?x\ type\ Student\} \end{array}$$

Table 1.3 shows the RDF quadruples obtained by evaluating the authorizations of Table 1.2 to the set of triples in Table 1.1. For instance quadruples q_1, q_2 are obtained from the evaluation of authorization $\mathcal{A}_2$ that assigns to triples with predicate sc (triples t_1 and t_2) the abstract token at_2. Quadruple q_3 is obtained from the evaluation of authorization $\mathcal{A}_3$ that assigns to triples with predicate type and object *Student* the token at_3 (triple t_3). Authorization $\mathcal{A}_1$ assigns to triples with predicate *firstName* the access token at_1 (triple t_4) thus obtaining quadruple q_4. Quadruple q_6 is obtained by assigning token at_4 to triples with predicate type

$\mathcal{A}_1$:	(CONSTRUCT	$\{?x$ *firstName* $?y\}$
	WHERE	$\{?x$ type *Student*$\}$, at_1)
$\mathcal{A}_2$:	(CONSTRUCT	$\{?x$ sc $?y\}$, at_2)
$\mathcal{A}_3$:	(CONSTRUCT	$\{?x$ type *Student*$\}$, at_3)
$\mathcal{A}_4$:	(CONSTRUCT	$\{?x$ type class$\}$, at_4)
$\mathcal{A}_5$:	(CONSTRUCT	$\{?x$ $?p$ *Person*$\}$, at_5)

Table 1.2: Access Control Authorizations

and *object* class through authorization $\mathcal{A}_4$ (triple t_6). Finally, authorization $\mathcal{A}_5$ assigns to triples with *object Person* access token at_5 (triple t_1) thereby obtaining quadruple q_7. Triple t_5 is not in the scope of any of the authorizations $\mathcal{A}_1 - \mathcal{A}_5$, so we assign to it the *default token* $\perp$ (quadruple q_5).

	s	p	o	l
q_1 :	*Student*	sc	*Person*	at_2
q_2 :	*Person*	sc	*Agent*	at_2
q_3 :	&a	type	*Student*	at_3
q_4 :	&a	*firstName*	Alice	at_1
q_5 :	&a	*lastName*	Smith	$\perp$
q_6 :	*Agent*	type	class	at_4
q_7 :	*Student*	sc	*Person*	at_5

Table 1.3: RDF Quadruples

Note that in our example each of the authorizations uses a different access token. Nevertheless, our model does not forbid different authorizations from using the same token. In addition, the same triple may obtain labels from different authorizations. This is the case for triple t_1 that is in the scope of authorizations $\mathcal{A}_2$ and $\mathcal{A}_5$, thereby obtaining quadruples q_1 and q_7 respectively.

In the above discussion we have not taken into account the RDFS inference rules [4] or the propagation of access labels along the RDFS class and property hierarchies. RDFS inference rules compute *implied* triples from *explicit* ones. The RDFS inference rules that we consider in our work refer to the transitivity of *subClassOf* (sc), *subPropertyOf* (sp) and *type* (type) hierarchies. For instance, when applying transitivity for the RDFS sc and type hierarchies (i.e., a resource is an instance of all the superclasses of its class) for triples t_1 and t_3 shown in Table 1.1 we obtain triple (&a, type, *Person*).

RDFS inference rules can be naturally extended for quadruples and the natural question that comes to mind in this case is *"what is the access label of the implied quadruple?"*. Consider for instance quadruples q_3 and q_1. The label of the implied quadruple (&a, type, *Person*, l) cannot be one of tokens at_3 or at_2 but a *composite* label that involves both tokens at_2 and at_3 (as in [6]). We model the composite label of such an implied quadruple using the *inference abstract operator* denoted by $\odot$ that operates on the labels of its implying quadruples. Using $\odot$, said quadruple will be (&a, type, *Person*, $at_3 \odot at_2$) (q_{10} in Table 1.4).

By applying the RDFS inference rules on the type and sc hierarchies in all possible ways, we obtain the quadruples shown in Table 1.4. Note that the order in which the explicit triples are combined to produce the implied triples is irrelevant. In this way, the *inference operator* $\odot$ is commutative. Implicit quadruples may also be involved in inferences, resulting in more complex expressions; for example q_{11} is obtained using q_{10} and q_2. Quadruple q_{12} is obtained from quadruples q_3 and q_9.

	s	p	o	l
q_8 :	*Student*	sc	*Agent*	$at_2 \odot at_2$
q_9 :	*Student*	sc	*Agent*	$at_2 \odot at_5$
q_{10} :	&a	type	*Person*	$at_3 \odot at_2$
q_{11} :	&a	type	*Agent*	$(at_3 \odot at_2) \odot at_2$
q_{12} :	&a	type	*Agent*	$at_3 \odot (at_2 \odot at_5)$
q_{13} :	&a	type	*Person*	$at_2 \odot at_5$

Table 1.4: Implied Quadruples

The RDFS semantics associated with the class and property hierarchies cause several authors to consider the *propagation* of labels along such hierarchies [10]. For example, an application may require that a triple that defines an instantiation relation between an instance and a class, inherits the access label of the triple defining such class (an instance inherits the label(s) of its class). To support this feature, we define the abstract *unary propagation operator* denoted by $\otimes$.

Recall that in our framework we assign access labels to *triples* rather than *resources*, so the label of a class C is specified by the label of quadruple (C, type, class, l). Similarly, the label of an instance x of a class C is defined by quadruple (x, type, C, l). Under this understanding, quadruple q_6 should propagate its label (at_4) to all instances of *Agent*, thus obtaining the quadruple q_{14} shown in Table 1.5.

This approach is motivated by the fact that the semantics in an RDF graph are not given by the resources themselves, but by their interconnections as expressed in triples. Thus, triples are the "first-class citizens" in an RDF graph; triples can de-

	s	p	o	l
q_{14} :	$\&a$	type	*Agent*	$\otimes\ at_4$

Table 1.5: Propagated Quadruple

scribe resources (e.g., the fact that a given resource is a class), whereas the opposite is not possible (e.g., resources cannot describe a subsumption relationship).

As explained above, our framework does not bound the abstract tokens (at_i), the default token ($\perp$) and the operators $\odot$ and $\otimes$ to concrete values. Instead, each application, depending on its needs defines a *concrete policy* that *maps* every abstract access token to a concrete value and specifies the concrete operators that implement the abstract ones. The concrete policy also specifies how conflicting labels are resolved. This case arises when multiple quadruples that refer to the same triple exist but with different labels. In our example, the concrete policy might determine that the abstract tokens at_1, at_2 and at_3 are mapped to *true* whereas at_4 and at_5 to *false*. The policy semantics may specify that the label of an implied quadruple is *true* if and only if the labels of its implying quadruples are both *true* (i.e., $\odot$ is mapped to conjunction), and that the labels are propagated as such (i.e., $\otimes$ is mapped to identity). In the case of conflicting labels for a triple the application favors quadruples with the *false* label. In this scenario quadruples q_1, q_2, q_3, q_4 will get a *true* label, whereas q_5, q_6 and q_7 label *false*. Implied quadruples q_8, q_{10} and q_{11} will get a *true* label whereas q_9 and q_{12} the *false* label. Note that quadruples q_8 and q_9 refer to the same triple (*Student*, sc, *Agent*). According to the semantics of the policy as implemented by the enforcement mechanism, the resulting quadruple will obtain label *false*. Last, quadruple q_{13} will get the *false* label since the propagation operator is mapped to identity and the label propagated is the *false* label (at_4).

The same process could be used for any other concretization imposed by an application. Note that, for more complex concretizations, an *access function* is necessary to determine whether the triple is accessible or not; for example, if the final concrete label of a given quadruple was M (indicating a medium confidentiality level), the access function would determine who has access to it (e.g., managers of the company may be able to access such data, whereas secretaries may not). Another important characteristic of our framework is that our abstract model can simultaneously support different application semantics using the same data (triples and their abstract labels). Also, changing the semantics of an application is easy, and requires no recomputation or other changes on the underlying data.

Chapter 2

Preliminaries

2.1 RDF and RDF Schema

The Resource Description Framework (RDF) [1], 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, such as the Linked Data initiative and has become the de fact standard for representing information on the Web of Data.

RDF [1] 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 or an institution, or the relation a person has with that institution. A resource is uniquely identified by its Universal Resource Identifier (URI). The building block of the RDF data model is a *triple*.

Definition 2.1 *An RDF triple (subject, predicate, object) is any element of the set* $\mathcal{T} = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$*, where* $\mathbb{U}$ *and* $\mathbb{L}$ *are disjoint,* $\mathbb{U}$ *is the set of URIs, and* $\mathbb{L}$ *the set of literals. A set of RDF triples is called an RDF* graph.

The RDF Schema (RDFS) language [3] provides a built-in vocabulary for asserting user-defined schemas in the RDF data model. For instance, RDFS names *rdfs*:*Class* (class) and *rdf*:*Property* (prop)[1] can be used to specify *class* and *property* types. Furthermore, one can assert *instanceOf* relationships of resources with the RDF predicate *rdf*:*type* (type), whereas *subsumption* relationships among classes and properties are expressed with the RDFS *rdfs*:*subClassOf* (sc) and *rdfs*:*subPropertyOf* (sp) predicates respectively. In addition, RDFS *rdfs*:domain (domain) and *rdfs*: range (range) predicates allow one to specify the domain and range of the properties in an RDFS vocabulary.

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

$$\mathcal{R}_1 : \frac{(p,\ \mathsf{sp},\ q), (q,\ \mathsf{sp},\ r)}{(p,\ \mathsf{sp},\ r)} \qquad \mathcal{R}_2 : \frac{(p,\ \mathsf{sp},\ q), (x,\ p,\ y)}{(x,\ q,\ y)}$$

$$\mathcal{R}_3 : \frac{(x,\ \mathsf{sc},\ y), (z,\ \mathsf{type},\ x)}{(z,\ \mathsf{type},\ y)} \qquad \mathcal{R}_4 : \frac{(x,\ \mathsf{sc},\ y), (y,\ \mathsf{sc},\ z)}{(x,\ \mathsf{sc},\ z)}$$

Table 2.1: RDFS Inference Rules

An *RDF Graph* $\mathcal{G}$ is defined as a *set of RDF data and schema triples*, i.e., $\mathcal{G} \subseteq \mathcal{T}$. In this work we consider graphs in which the sc and sp relations are *acyclic*. This assumption is introduced in order to avoid the repeated generation of new quadruples with new labels that can occur when cycles exist. Note that acyclicity holds in the large majority of RDF schemas used in real applications [17], and is a common assumption made for efficiency (e.g., query optimization [18]) in many RDF applications. An RDF graph can be represented as a directed node and edge labeled graph where nodes in the graph are URIs and literals and edges represent properties between the subject and object components of an RDF triple.

RDFS defines a set of *inference rules* [4] depicted in Table 2.1, which are used to compute the *closure* of an RDF graph $\mathcal{G}$, denoted by $Cn(\mathcal{G})$. Rules $\mathcal{R}_1$ and $\mathcal{R}_4$ discuss the transitivity of sp and sc properties resp., whereas $\mathcal{R}_2$ and $\mathcal{R}_3$ discuss the transitivity for the sc and sp relations and *class* and *property* instances.

2.2 SPARQL

SPARQL [14] is the official W3C recommendation for querying RDF graphs, and is based on the concept of matching patterns against the RDF graph. Thus, a SPARQL query determines the pattern to seek for, and the answer is the part(s) of the RDF graph that match this pattern.

More specifically, SPARQL defines *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 an RDF graph that have the form specified by the triple patterns. 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 that specify conditions on the triple patterns.

For example, triple pattern $tp_1 = (?x,\ \mathsf{sc},\ ?y)$ 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 RDF graph with predicate sc. In our example of Figure 1.1, the triples that match tp_1 are t_1 and t_2.

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 the RDF graph by substituting the variables with matching URIs/literals. Finally, a `FILTER` expression specifies explicitly a condition on query variables.

The SPARQL syntax follows the SQL select-from-where paradigm. The `SELECT` clause specifies the variables that should appear in the query results. The SPARQL queries we use in our work use the `CONSTRUCT` clause, and return a single RDF graph (i.e., set of triples) specified by a graph pattern in the `WHERE` clause of the query. For the purposes of our work, we use the `CONSTRUCT` clause since we assign access labels to *triples* and not to nodes in the RDF graph. Thus, the SPARQL queries used in this work are of the form:

$$Q = \texttt{CONSTRUCT}\ gp_1\ \mathsf{where}\ gp_2$$

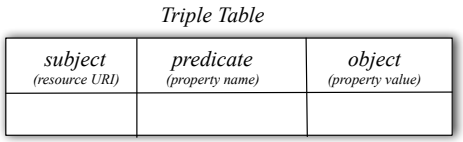
where gp_1 and gp_2 are graph patterns. The result of such a query is an RDF graph formed by performing a set union on the set of triples that match graph pattern gp_1.

The interested reader can find a more detailed description of the semantics of the SPARQL language in [19]. Note that SPARQL does not support functionalities necessary for navigating the RDFS `sc` and `sp` property hierarchies. Consequently, to query the closure of the RDF graph, one must either compute it before hand (by applying the rules in Tables 2.1) or evaluate it on the fly. In this work we focus on the computation of the access labels for the implied RDF triples and not on efficient ways of how to compute the closure of an RDF graph.

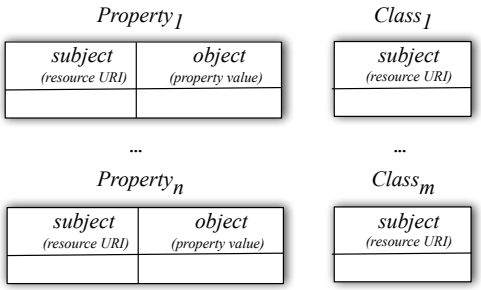
2.3 Storage Schemas for RDF Data

The three widely used as logical storage schemes for shredding RDF/S resource descriptions into relational tables are:

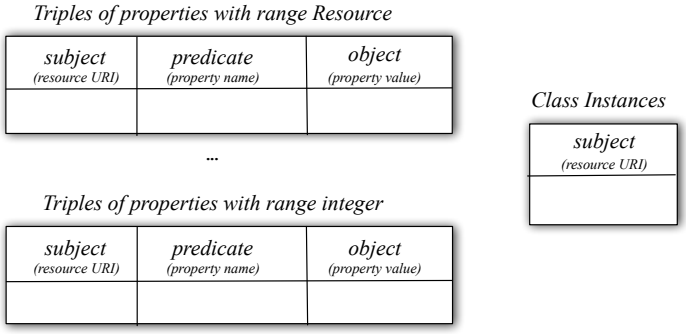
- *schema agnostic* in which RDF triples are stored in a large *triple table*. The triple table's attributes are *subject*, *predicate* and *object* and refer to the three components of an RDF triple. attribute This approach has been followed in the majority of works that deal with the storage and processing RDF data [20, 21, 22, 23, 24, 25, 26, 27] (see Figure 2.1a).
- *schema aware* [28, 29] in which for each property $Property_i$in a set of RDF triples, a binary table is created that stores the *subject* and *object* components of the triple with predicate $property_i$. This approach has been used in [28, 29] and is known as *vertical partitioning* (see Fig. 2.1b).



(a) Schema Agnostic Approach



(b) Schema Aware Approach



(c) Hybrid Approach

Figure 2.1: Logical Storage Schemas for RDF data

- *hybrid* that combines elements of the previous approaches. In the hybrid approach, one relational table per RDF class $Class_i$ is created, that stores the instances of the class. In a similar manner, dependending on the type of the property value (resource, string, integer), a table that stores the instances of the corresponding value types is created (see Figure 2.1c).

In the case of the schema agnostic approach, one triple table is created for any RDFS schema. On the other hand, in the case of schema aware approach, the properties and classes defined at the RDFS schema are considered to define the property and class tables of the logical schema where the data will be stored. Finally, the hybrid approach uses the RDF meta-schema (classes, attributes, value types) to define the logical data storage schema.

The advantages of the schema agnostic approach is the ease of RDF data representation and the decoupling of the logical schema from the existence of an RDFS schema. This decoupling makes schema evolution a trivial process, since the addition/deletion of a class or schema property corresponds to the addition/deletion of a set of triples. In a similar manner, in the case of the hybrid approach, this corresponds to deletion of triples from the property tables and the deletion of tuples from the table that stores the class instances. On the other hand, in the case of the schema aware approach, a change in the schema corresponds to the addition/deletion of a class or property table. The disadvantage of the schema agnostic approach is the loss of information related to the type of property values (i.e., the object of a triple), since all values of the object component of a triple are stored as a string. The hybrid and schema aware approaches record this kind of information which can be used for query processing.

As far as SPARQL query evaluation is concerned, the main disadvantage of the schema agnostic approach is that the algebraic plan obtained for a query involves a large number of *self joins* over a large triple table, whereas in the other two cases, the plan involves joins between the appropriate class and property tables. The schema agnostic approach is advantageous in the cases in which a property is unbound (i.e., use of a variable and not a URI). On the other hand, in the case of schema aware and hybrid approaches, the initial user query should be translated into as many queries as the number of predicates in the dataset [20].

Chapter 3

Access Control

3.1 Abstract Access Control Model

An *abstract access control model* is comprised of *abstract tokens* and *abstract operators*. Abstract tokens are assigned to RDF triples through *authorizations* or *abstract access control rules*, whereas abstract operators describe *(i)* the computation of access labels for implied triples and *(ii)* the propagation of access labels along the RDFS class and property hierarchies. RDF triples are either annotated with abstract tokens or with a complex algebraic expression that involves the abstract tokens and operators of the abstract model.

In our work *annotated RDF triples* are represented as *quadruples*. A quadruple is of the form (s, p, o, l) where s, p, o are the *subject*, *property* and *object* and l is an *abstract access control expression*. We denote by $\mathcal{D}$ the set of *quadruples*.

As discussed in previous sections, for a given RDF graph $\mathcal{G}$ and set of tokens $\mathcal{L}$, access control authorizations are used to assign tokens (from $\mathcal{L}$) to RDF triples (from $\mathcal{G}$). It might happen that a set of authorizations assigns no, or multiple, access label(s) to a given triple. To handle the case of missing labels, a triple is labeled with a special *default* token ($\bot$) that is not in the set of abstract tokens. To address RDFS inference [4] and the propagation of access labels along the RDFS *subClassOf* and *subPropertyOf* hierarchies, we introduce the *abstract inference accumulator* operator ($\odot$) and *abstract propagation* operator ($\otimes$) respectively. An *access label*, denoted by al, is either an *access token* from $\mathcal{L}$, or a *complex expression* formulated using abstract tokens combined with the above abstract operators.

Now we are ready to define the notion of *access control model*.

Definition 3.1 *An abstract access control model $\mathcal{M}$ is a tuple $\mathcal{M} = \langle \mathcal{L}, \bot, \odot, \otimes \rangle$ where:*

- *$\mathcal{L}$ is the set of abstract access tokens*

- $\perp$ *is the default access token that is assigned to triples that are not in the scope of some authorization rule,* $\perp \notin \mathcal{L}$
- $\odot$ *is the binary inference operator*
- $\otimes$ *is the unary propagation operator*

The inference and propagation operators are defined over the tokens in $\mathcal{L} \cup \{\perp\}$. The access label of a quadruple is an expression that is defined over the abstract tokens in $\mathcal{L} \cup \{\perp\}$.

The abstract operatos will be described in detail in the following sections.

3.1.1 Inference Operator

Implied triples are obtained from the application of RDFS inference rules [4] on an RDF graph. The RDFS inference rules shown in Table 2.1 discuss the transitivity along the *subClassOf* and *subPropertyOf* hierarchies (rules $\mathcal{R}_1$ and $\mathcal{R}_4$) and of *type* property ($\mathcal{R}_2$, $\mathcal{R}_3$). An implied triple is annotated by a complex expression that involves the labels of the triples that imply it associated through the binary *inference operator* $\odot$.

The inference rules shown in Table 3.1 extend those specified in Table 2.1 in a straightforward manner for quadruples. An implied triple is annotated by a complex expression that involves the labels of its implying triples associated through the binary *inference operator* $\odot$.

For instance, in our motivating example of Section 1.1 quadruple q_{12} is obtained by applying the transitivity on the sc hierarchies (rule $\mathcal{R}_4$) for quadruples q_7, q_2.

$$\mathcal{QR}_1 \ : \ \frac{(P, \mathsf{sp}, Q, al_1), (Q, \mathsf{sp}, R, al_2)}{(P, \mathsf{sp}, R, (al_1 \odot al_2))}$$

$$\mathcal{QR}_2 \ : \ \frac{(P, \mathsf{sp}, Q, al_1), (x, P, y, al_2)}{(x, Q, y, (al_1 \odot al_2))}$$

$$\mathcal{QR}_3 \ : \ \frac{(x, \mathsf{sc}, y, al_1), (z, \mathsf{type}, x, al_2)}{(z, \mathsf{type}, y, (al_1 \odot al_2))}$$

$$\mathcal{QR}_4 \ : \ \frac{(x, \mathsf{sc}, y, al_1), (y, \mathsf{sc}, z, al_2)}{(x, \mathsf{sc}, z, (al_1 \odot al_2))}$$

Table 3.1: RDFS Inference Rules on quadruples

In order for the inference operator $\odot$ to be compliant with its role of "composing" labels during inference, we require it to be commutative and associative. These properties are necessary, because the access label of an implicit triple should be

uniquely determined by the access labels of the triples that imply it and not by the order of application of the inference rules. Note that we do not require the inference operator to be idempotent, since we might need to take into account multiple appearances of the same label. Formally:

Definition 3.2 *The* abstract inference operator*, denoted by $\odot$, is a binary operator defined over abstract tokens from $\mathcal{L} \cup \{\bot\}$ with the following properties:*

$$\begin{array}{lcll} al_1 \odot al_2 & = & al_2 \odot al_1 & \textit{(Commutativity)} \\ (al_1 \odot al_2) \odot al_3 & = & al_1 \odot (al_2 \odot al_3) & \textit{(Associativity)} \end{array}$$

Example 3.1 *Consider our motivating example and rule $\mathcal{QR}_3$ from Table 3.1. When this rule is applied to quadruples $q_1 = (Student, \mathsf{sc}, Person, at_2)$ and $q_3 = (\&a, \mathsf{type}, Student, at_3)$ in Figure 1.3, we obtain quadruple $q_{10} = (\&a, \mathsf{type}, Person, (at_2 \odot at_3))$ shown in Figure 1.4.*

3.1.2 Propagation Operator

$$\mathcal{QR}_5: \quad \frac{(x, \mathsf{type}, \mathsf{class}, al_1), (y, \mathsf{sc}, x, al_2), (y, \mathsf{type}, \mathsf{class}, al_3)}{(y, \mathsf{type}, \mathsf{class}, \otimes(al_1))}$$

$$\mathcal{QR}_6: \quad \frac{(x, \mathsf{type}, \mathsf{class}, al_1), (y, \mathsf{type}, x, al_2)}{(y, \mathsf{type}, x, \otimes(a_1))}$$

$$\mathcal{QR}_7: \quad \frac{(x, \mathsf{type}, \mathsf{prop}, al_1), (y, \mathsf{sp}, x, al_2), (y, \mathsf{type}, \mathsf{prop}, al_3)}{(y, \mathsf{type}, \mathsf{prop}, \otimes(al_1))}$$

$$\mathcal{QR}_8: \quad \frac{(P, \mathsf{type}, \mathsf{prop}, al_1), (x, P, y, al_2)}{(x, P, y, \otimes(al_1))}$$

Table 3.2: Propagation Rules

The idea of propagation of access labels is found in XML access control models. These models take into account the hierarchical nature of XML data and *propagate* labels to the descendants or ancestors of a node in an XML tree [30]. The propagation rules we advocate in our work *do not generate new triples* since this is the role of the inference rules discussed previously. They simply assign new labels to *existing* triples hence producing *new quadruples*. In this work we focus on *"downward"* propagation rules along the sc, sp and type hierarchies, where the labels are propagated from the upper level of a hierarchy to the lower levels (e.g., from a class to its instances). It is straightforward to model propagation rules for the opposite direction.

We model the propagation of labels with the *abstract propagation operator*, denoted by $\otimes$. Currently, we consider the propagation of a *single label* and therefore $\otimes$ is a unary operator over abstract access expressions. Table 3.2 shows the propagation rules that we consider in our work. Note that this is just one possible set of propagation rules that one can use. Different applications may have different needs, hence employ different propagation rules, or omit them altogether. In our framework, the handling of different sets of propagation rules is similar.

We require $\otimes$ to be idempotent so that multiple applications on the same label would not give new quadruples. Without this property, each of the propagation rules could be applied arbitrarily many times, each time producing a new quadruple: $(t, \otimes al)$, $(t, \otimes \otimes al)$, ..., $(t, \otimes \ldots \otimes al)$, ..., resulting in an infinite number of quadruples for the same triple (see Example 3.2).

Definition 3.3 *The* abstract propagation operator*, denoted by $\otimes$, is a unary operator defined over labels, with the property:*

$$\otimes(\otimes(al)) \quad = \quad \otimes(al) \quad (Idempotence)$$

Example 3.2 *Consider again our motivating example, and rule $\mathcal{QR}_6$ shown in Table 3.2. The rule states that the label of a class x (defined by quadruple (x,* type*,* class*, al_1)) is propagated to its instances (defined by quadruple (y,* type*, x, al_2)), thereby obtaining quadruple (y,* type*, x, $\otimes(al_1)$)). When this rule is applied to the quadruples $q_{11} = (\&a,$* type*, Agent, $(at_3 \odot at_2) \odot at_2$) from Figure 1.4 and $q_6 =$ (Agent,* type*,*class*, at_4) from Figure 1.3 we obtain quadruple $q_{13} = (\&a,$* type*, Agent, $\otimes at_4$). Now note that the same propagation rule can be applied again for quadruples q_{11} and q_{13}. Thanks to the idempotence property of $\otimes$, such an application is unnecessary, as it would result to the same quadruple. Without idempotence, we would get a new quadruple with label $\otimes\otimes(at_4)$, which in turn would fire another application of $\mathcal{R}_6$ and so on, leading to an infinite loop.*

3.1.3 Computing Abstract Expressions

Now consider an RDF graph $\mathcal{G}$ (i.e., set of triples) and a set of *authorization rules* $\mathcal{A}$ that assign abstract access tokens to triples by means of SPARQL queries. In order to obtain the set of explicit quadruples $\mathcal{E}$ we first annotate triples in $\mathcal{G}$ by evaluating the authorization rules on the set of RDF triples; after the evaluation of all the authorization rules, the triples that did not receive any label are annotated with the default access token ($\perp$).

We then have to apply the inference and propagation rules on the set of explicit quadruples $\mathcal{E}$, in order to obtain the *implicit* ones $\mathcal{I}$, along with their labels. These rules are implemented using Algorithms $1 - 6$ and described as follows.

The first two of our rules are $\mathcal{QR}_1$ and $\mathcal{QR}_4$ of Table 3.1, which model the transitive closure of sp and sc relations respectively. As shown in Algorithm 1, these rules are first applied to explicit quadruples (Lines $2-8$) and then recursively to explicit and implicit ones, that were produced via previous application of current rule (lines $10-22$). This is necessary because implicit triples may cause new applications of $\mathcal{QR}_1$ or $\mathcal{QR}_4$, leading to new implicit triples. This recursive application continues until no more implicit quadruples are produced.

Algorithm 1 TCL_SC_SP

Input: $\mathcal{E}$ the set of Explicit Quadruples
pred, one of sc, sp properties
Output: Set $\mathcal{I}'$ of implicit quadruples that are produced through the application of one of rules $\mathcal{QR}_4$ and $\mathcal{QR}_1$ (transitivity of the sc and sphierarchies resp.)

1: let S_pred, $S\_tmp\_pred = \emptyset$
2: **for all** $q_1 = (s_1, pred, o_1, at_1)$ in $\mathcal{E}$ **do**
3: **for all** $q_2 = (s_2, pred, o_2, at_2)$ in $\mathcal{E}$ **do**
4: **if** $o_1 = s_2$ **then**
5: $S\_pred = S\_pred \cup \{ (s_1, pred, o_2, (at_1 \odot at_2)) \}$
6: **end if**
7: **end for**
8: **end for**
9: $\mathcal{I}' = S\_pred$
10: **while** $S\_pred \neq \emptyset$ **do**
11: **for all** $q_1 = (s_1, pred, o_1, at_1)$ in $\mathcal{E}$ **do**
12: **for all** $q_2 = (s_2, pred, o_2, al_2)$ in S_pred **do**
13: **if** $o_1 = s_2$ **then**
14: $S\_tmp\_pred = S\_tmp\_pred \cup \{ (s_1, pred, o_2, (al_1 \odot al_2)) \}$
15: **end if**
16: **end for**
17: **end for**
18: $S\_pred = \emptyset$
19: $S\_pred = S\_tmp\_pred$
20: $S\_tmp\_pred = \emptyset$
21: $\mathcal{I}' = \mathcal{I}' \cup S\_pred$
22: **end while**
23: **return** $\mathcal{I}'$

Algorithms 2 and 3 implement *inference rules* $\mathcal{QR}_2$ and $\mathcal{QR}_3$ that are shown in Table 3.1. $\mathcal{QR}_2$ must be applied to explicit quadruples of the form of (s_1, p_1, o_1, al_1), provided that explicit or implicit (produced through the application of $\mathcal{QR}_1$ quadruples of the form (p_1, sp, o_2, al_2) exist in the dataset. Respectively, $\mathcal{QR}_3$

must be applied to explicit quadruples of the form of (s_1, type, o_1, al_1), provided that explicit or implicit (produced through the application of $\mathcal{QR}_4$ quadruples of the form (o_1, sc, o_2, al_2) exist in the dataset. Note that both these rules require no recursive application, because the triples produced by their application cannot fire the same rules again.

Algorithm 2 TRANS_PROP_INST

Input: $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of implicit quadruples
Output: Set $\mathcal{I}'$ of implicit quadruples that were produced by rule $\mathcal{QR}_2$

```
1: I' = ∅
2: for all q1 = (s1, p1, o1, al1) in E do
3:     for all q2 = (s2, sp, o2, al2) in E ∪ I do
4:         if p1 = s2 then
5:             I' = I' ∪ { (s1, o2, o1, (al1 ⊙ al2)) }
6:         end if
7:     end for
8: end for
9: return I'
```

Note that in the case in which we use vertical partitioning (which will be described in later section) as our storage schema, we have to add appropriate constraints when selecting the quadruples to join in order not to add erroneous triples in the dataset. For instance, in the case of TRANS_PROP_INST, we must select only the quadruples where p_1 is *not* one of sp, sc and type.

Algorithm 3 TRANS_CLASS_INST

Input: $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of implicit quadruples
Output: Set $\mathcal{I}'$ of implicit quadruples that produced through current rule application ($\mathcal{QR}_3$)

```
1: I' = ∅
2: for all q1 = (s1, type, o1, al1) in E do
3:     for all q2 = (s2, sc, o2, al2) in E ∪ I do
4:         if o1 = s2 then
5:             I' = I' ∪ { (s1, type, o2, (al1 ⊙ al2)) }
6:         end if
7:     end for
8: end for
9: return I
```

Continuing to our four *propagation* rules we first have $\mathcal{QR}_5$ and $\mathcal{QR}_7$ rules of Table 3.2 which both implemented through Algorithm 4. As shown in Algorithm 4,

$\mathcal{QR}_5$ is applied to explicit quadruples of the form of (s_1, type, class, al_1) and (s_3, type, class, al_3), provided that explicit or implicit quadruples of the form (s_3, sc, s_1, al_2) exist in the dataset. Respectively, $\mathcal{QR}_7$ rule is applied to explicit quadruples of the form of (s_1, type, prop, al_1) and (s_3, type, prop, al_3), provided that explicit or implicit quadruples of the form (s_3, sp, s_1, al_2) exist in the dataset.

Algorithm 4 PROPAGATION_CLASS_PROP

Input: $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of implicit quadruples
pred, one of sc, sp values
obj, one of class, prop values
Output: Set $\mathcal{I}'$ of implicit quadruples that produced through the application of rule $\mathcal{QR}_5$

```
1: I' = ∅
2: for all q1 = (s1, type, obj, al1) in E do
3:     for all q2 = (s2, pred, o2, al2) in E ∪ I do
4:         for all q3 = (s3, type, obj, al3) in E do
5:             if s1 = o2 ∧ s2 = s3 then
6:                 I' = I' ∪ { (s3, type, obj, ⊗(al1)) }
7:             end if
8:         end for
9:     end for
10: end for
11: return I'
```

The last two of our *propagation* rules are $\mathcal{QR}_6$ and $\mathcal{QR}_8$, which are implemented through Algorithms 5 and 6 respectively. $\mathcal{QR}_6$ rule is applied to explicit and implicit quadruples of the form of (s_1, type, class, al_1), provided that explicit or implicit ones of the form (s_2, type, s_1, al_2) exist in the dataset. On the same manner, $\mathcal{QR}_8$ is applied to explicit and implicit quadruples of the form of (s_1, type, prop, al_1), provided that explicit or implicit ones of the form (s_2, s_1, o_1, al_2) exist in the dataset.

Taking into consideration the algorithms that described above we are able to first apply the *inference* and then the *propagation* rules. The process of applying the various inference and propagation rules in order to produce new quadruples is called *annotation* and uses the previously presented algorithms (each of which handles a single inference or propagation rule). The process is described in Algorithm 7.

Note that Algorithm 7 applies the rules in a specific order (namely: $\mathcal{QR}_1$, $\mathcal{QR}_2$, $\mathcal{QR}_4$, $\mathcal{QR}_3$, $\mathcal{QR}_5$, $\mathcal{QR}_6$, $\mathcal{QR}_7$, $\mathcal{QR}_8$), which is necessary because the quadruples produced by certain rules may cause the firing of other rules. In particular: $\mathcal{QR}_1$ will fire inference rules $\mathcal{QR}_2$ and $\mathcal{QR}_4$ and propagation rule $\mathcal{QR}_5$; $\mathcal{QR}_2$ would fire rule inference rule $\mathcal{QR}_4$ and propagation rule $\mathcal{QR}_7$.

Algorithm 5 PROPAGATION_CLASS_INST

Input: $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of implicit quadruples
Output: Set $\mathcal{I}'$ of implicit quadruples that are produced through the application of $\mathcal{QR}_6$

1: $\mathcal{I}' = \emptyset$
2: **for all** $q_1 = (s_1, \mathsf{type}, \mathsf{class}, al_1)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
3: **for all** $q_2 = (s_2, \mathsf{type}, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
4: **if** $s_1 = o_2$ **then**
5: $\mathcal{I}' = \mathcal{I}' \cup \{ (s_2, \mathsf{type}, o_2, \otimes(al_1)) \}$
6: **end if**
7: **end for**
8: **end for**
9: **return** $\mathcal{I}'$

Algorithm 6 PROPAGATION_PROP_INST

Input: $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of implicit quadruples
Output: Set $\mathcal{I}'$ of implicit quadruples that were produced by rule $\mathcal{QR}_6$

1: $\mathcal{I}' = \emptyset$
2: **for all** $q_1 = (s_1, \mathsf{type}, \mathsf{prop}, al_1)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
3: **for all** $q_2 = (s_2, p_2, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
4: **if** $s_1 = p_2$ **then**
5: $\mathcal{I}' = \mathcal{I}' \cup \{ (s_2, p_2, o_2, \otimes(al_1)) \}$
6: **end if**
7: **end for**
8: **end for**
9: **return** $\mathcal{I}'$

In more details, the first rule that applied is $\mathcal{QR}_1$ (Line 2). The reason why we fist apply $\mathcal{QR}_1$ is that its output have to be used as input in application of $\mathcal{QR}_2$ rule. So, the next one is $\mathcal{QR}_2$ (Line 3). Similarly, we next apply $\mathcal{QR}_4$ rule (Line 4) and then, by using its output, we apply $\mathcal{QR}_3$ (Line 5). Continuing to the application of *propagation* rules we first apply $\mathcal{QR}_5$ rule by taking as input the explicit quadruples and the implicit ones that produced through $\mathcal{QR}_4$ (Line 6). The next one is $\mathcal{QR}_6$ (Line 7) which takes as its input the explicit quadruples and the implicit ones that produced through $\mathcal{QR}_3$ and $\mathcal{QR}_5$ rules. Similarly, the last two propagation rules that applied are $\mathcal{QR}_7$ and $\mathcal{QR}_8$. $\mathcal{QR}_7$ is applied to explicit and implicit quadruples that produced through $\mathcal{QR}_1$ rule (Line **??**), and $\mathcal{QR}_8$ to explicit quadruples and the implicit ones that produced through $\mathcal{QR}_2$ and $\mathcal{QR}_7$ (Line 9).

We have to note here that the assignment of access tokens to the explicit triples is not part of the *annotation* process that is presented in Algorithm 7. The assignment is done by evaluating the SPARQL queries of the authorizations on the explicit triples of the input dataset.

Algorithm 7 Graph Annotation

Input: $\mathcal{E}$ the set of Explicit Quadruples

Output: RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the produced set of Implicit Quadruples, $\mathcal{I} \neq \emptyset$

1: S_rdfs5, S_rdfs7, S_rdfs11, S_rdfs9, S_prop1, S_prop2, S_prop3, $S\_prop4 = \emptyset$;
2: $S\_rdfs5 = \text{TCL\_SC\_SP}(\mathcal{E}, \mathsf{sp})$;
3: $S\_rdfs7 = \text{TRANS\_PROP\_INST}(\mathcal{E}, S\_rdfs5)$;
4: $S\_rdfs11 = \text{TCL\_SC\_SP}(\mathcal{E}, \mathsf{sc})$;
5: $S\_rdfs9 = \text{TRANS\_CLASS\_INST}(\mathcal{E}, S\_rdfs11)$;
6: $S\_prop1 = \text{PROPAGATION\_CLASS\_PROP}(\mathcal{E}, S\_rdfs11, \mathsf{sc}, \mathsf{class})$
7: $S\_prop2 = \text{PROPAGATION\_CLASS\_INST}(\mathcal{E}, S\_rdfs9 \cup S\_prop1)$
8: $S\_prop3 = \text{PROPAGATION\_CLASS\_PROP}(\mathcal{E}, S\_rdfs5, \mathsf{sp}, \mathsf{prop})$
9: $S\_prop4 = \text{PROPAGATION\_PROP\_INST}(\mathcal{E}, S\_rdfs7 \cup S\_prop3)$
10: $\mathcal{I} = S\_rdfs11 \cup S\_rdfs5 \cup S\_rdfs9 \cup S\_rdfs7 \cup S\_prop1 \cup S\_prop2 \cup S\_prop3 \cup S\_prop4$
11: **return** $\mathcal{G}' = (\mathcal{E}, \mathcal{I})$

At the end of the above process (Algorithm 7), the access label of any quadruples of the form:

$$\begin{aligned} proplabel &:= \otimes aclabel \mid at_i \\ aclabel &:= at_i \mid aclabel \odot aclabel \end{aligned}$$

where at_i is an abstract token from $\mathcal{L}$ or $\perp$.

Note that we follow this order in computing the (propagated) access labels of implicit triples since it is the only order that ensures the correct computation of labels of triples. The idea is that we first need to compute *all* the *implied* triples.

For instance, consider the set of quadruples shown in Table 3.3. Applying first the propagation rules on quadruples q_a to q_e we obtain quadruples q_f, q_g, but we are missing quadruple q_e since we have not computed quadruple (B,sc,C, $\otimes(at_4)$.

	s	p	o	l
q_a :	A	type	B	at_1
q_b :	B	type	class	at_2
q_c :	B	sc	C	at_3
q_d :	C	sc	D	at_3
q_e :	D	type	class	at_4
q_f :	A	type	class	$\otimes(at_2)$
q_g :	C	type	class	$\otimes(at_4)$
q_e :	A	type	class	$\otimes(at_4)$

Table 3.3: Example Quadruples

3.2 Concrete Policies

As discussed in Section 1.1, in order for an application to determine the triples in an RDF graph that are accessible, it must assign specific values to the abstract tokens and operators. This process is called *evaluation* and is informally described in Section 1.1 through our motivating example. The *evaluation* process uses a *concrete policy* $\mathcal{P}$ that specifies the

- *concrete tokens*
- *mapping* between the concrete and abstract tokens
- *concrete operators*
- *conflict resolution operator*
- and the *access function.*

More specifically, a concrete policy uses a set of *concrete tokens* $\mathcal{L}_\mathcal{P}$; Note that $\perp$ is a special token, it does not belong in $\mathcal{L}$, and is mapped to a concrete value that does not belong in $\mathcal{L}_\mathcal{P}$.

The mapping is used to associate abstract tokens from the set of abstract tokens $\mathcal{L}$ to concrete ones in $\mathcal{L}_\mathcal{P}$. To simplify the presentation and without loss of generality we use $\perp$ to refer to both the abstract default token and the concrete token that is mapped to.

The *concrete operators* implement the abstract inference ($\odot$) and propagation ($\otimes$) operators in the sense that they provide a specific definition of these operators over the set of concrete tokens and $\perp$(i.e., over the set $\mathcal{L}_\mathcal{P} \cup \{\perp\}$). Note that these implementations must respect the properties defined for their abstract counterparts. The concrete operators are defined over the set $\mathcal{L}_\mathcal{P} \cup \{\perp\}$, i.e., they should specify how to handle $\perp$ as well.

In our framework, it is possible that a triple is assigned several different concrete tokens through different quadruples, and/or the special value to which the default access token ($\perp$) is mapped. To handle such cases, we use the *conflict resolution operator*, denoted by $\oplus$, to select the final concrete token of said triple. To respect the intended semantics of $\perp$ as the "default" token, we require that it is ignored by $\oplus$ in the case in which the triple is assigned other concrete tokens from $\mathcal{L}_\mathcal{P}$. Formally,

Definition 3.4 *$\oplus$ is a selection function over subsets of $\mathcal{L}_\mathcal{P} \cup \{\perp\}$ (i.e., $2^{\mathcal{L}_\mathcal{P} \cup \{\perp\}}$)) that has the following property:*

$$\oplus\mathcal{X} = \begin{cases} x \in \mathcal{X} & \text{if and only if } \mathcal{X} \neq \emptyset \\ \perp & \text{if and only if } \mathcal{X} = \emptyset \text{ or } \mathcal{X} = \{\perp\} \end{cases}$$

The *access function*, denoted by *access*() is the final component of a concrete policy and determines whether a triple associated with a certain (concrete) token, as computed during the evaluation process, is accessible or not. Note that, if a triple is associated with several different concrete tokens, then the conflict resolution operator should be applied first, in order to select one of the tokens for consideration by *access*(). Formally, *access*() is a function mapping each concrete token to *allow* or *deny*, indicating that the corresponding triple is accessible or not accessible respectively.

To visualize our approach we will use two reasonable, and rather simple, concrete policies, **C1** and **C2**. The former is based on boolean tokens, whereas the latter is based on numerical access levels. More specifically, in **C1**, the abstract tokens are mapped to boolean values: $\mathcal{L}_\mathcal{P} = \{true, false\}$. A mapping determines how the values in $\mathcal{L}$ are mapped to $\mathcal{L}_\mathcal{P}$. The access function assigns *"allow"* to *true* value (i.e., a quadruple with a *true* label is accessible), and *"deny"* to *false* value. Triples with no label (i.e., triples assigned the special token $\perp$) are assigned the

value *"deny"*. The abstract operator $\odot$ is mapped to the following operator:

$$al_1 \odot al_2 = \begin{cases} al_1 \wedge al_2 & \text{if } al_1 \text{ and } al_2 \text{ are different from } \bot \\ al_i & \text{if } al_i \neq \bot,\ al_j = \bot,\ i \neq j \\ \bot & \text{if } al_1 \text{ and } al_2 \text{ are equal to } \bot \end{cases}$$

where $\wedge$ is the standard boolean conjunction. The propagation operator is simply defined as $\otimes al = al$. According to **C1**, an implied is accessible if and only if its implying triples are accessible. If one of its implying triples has as label the default token, then the triple gets the label of the other implying triple. The propagation operator is the identity function, that is the label is inherited with no change. In the case of conflicting labels, *false* overrides *true*, that is the policy is very conservative.

$$\oplus\mathcal{X} = \begin{cases} \mathit{false} & \text{if } \mathit{false} \in \mathcal{X} \\ \mathit{true} & \text{if } \mathit{false} \notin \mathcal{X},\ \mathit{true} \in \mathcal{X} \\ \bot & \text{if } \mathit{false},\ \mathit{true} \notin \mathcal{X} \end{cases}$$

The concrete policy **C2** uses positive integer values corresponding to increasing levels of confidentiality (i.e., *1* means minimum confidentiality). Thus, $\mathcal{L}_{\mathcal{P}} = \mathcal{N}$. The operator $\odot$ is mapped to addition, ignoring $\bot$, i.e., $n \odot m = n + m$, $n \odot \bot = n$. The propagation operator is the identity: $\otimes al = al$. The conflict resolution operator selects the most restrictive (maximum) token, ignoring $\bot$. More specifically,

$$\oplus\mathcal{X} = \begin{cases} max(\mathcal{X} \setminus \{\bot\}) & \text{if } \mathcal{X} \setminus \{\bot\} \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

For this policy, the implied triples have higher confidentiality level than their implying ones. As before, the propagation operator is the identity function. In the case of conflicting labels, the conflict is resolved in favor of the triple with the highest value.

The access function specifies that all quadruples whose confidentiality token is higher than a certain number (say *2*) are accessible (i.e., $access(n) = allow$ iff $n \geq 2$), and inaccessible otherwise (thus, e.g., $access(\bot) = deny$).

To handle cases where multiple users with different roles access the same dataset, one could define different concrete policies, or concrete policies with different access functions. For example consider a health case scenario in which we have different users that access the same data but have different privileges. For instance, a doctor can see the medical record of a patient, but a nurse, or a lab technician can see only a subset of this information. To achieve this, in our framework, we can define a different concrete policy per user (or per role), or specialize different components of a concrete policy (i.e., access function, conflict resolution operator etc.).

Chapter 4

Updates

The main advantage of our framework compared to standard (concrete) models is that it can efficiently handle various types of changes in the dataset, authorizations and concrete policies. This is possible since we store for each quadruple the quadruples that contributed to the computation thereof. In the following subsections, we discuss the different type of changes that are supported by our framework, and describe the algorithms that can handle them.

4.1 Changes in the Dataset

The dataset can be changed by either adding or deleting triples from it. Recall that a triple is associated with more than one quadruples since the triple might be in the scope of multiple authorizations. Hence, an addition or deletion of a triple generally results in the addition or deletion (respectively), of several quadruples.

4.1.1 Adding a new triple

As shown in Algorithm 8, assuming the set of access authorizations $\mathcal{A}$, when a triple t is added, then one should determine the authorizations that have the newly added triple in their scope (Lines $2-6$) and add the corresponding quadruple(s) to the set of explicit ones, $\mathcal{E}$ (Line 7). Then, we should consider the inference and propagation rules to determine their new applications in the presence of the new triple (and quadruples), and add the corresponding implicit quadruples as necessary (Line 8).

The addition of implicit quadruples, is implemented through AddImplicitQuads procedure and described in Algorithm 9. As shown in this algorithm, our analysis differs depending on the type of triple that is going to be added. In particular, we consider six types of triples, as shown in Table 4.1, and each of them is related to specific inference/propagation rules only. For example, the addition of a triple of type 1 cannot trigger new inferred quadruples through rule $\mathcal{QR}_1$, even though it

Algorithm 8 AddTriple

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
$\mathcal{A}$ the set of Authorizations
Output: Updated Graph $\mathcal{G}' = (\mathcal{E}', \mathcal{I}')$

```
1: S = ∅
2: for all A_i = (q_i, at_i) in A do
3:     for t in the scope of q_i do
4:         S = S ∪ {(s, p, o, at_i)}
5:     end for
6: end for
7: E' = E ∪ S
8: I' = AddImplicitQuads(t, G, S)        // Algorithm 9
9: return G' = (E', I')
```

can trigger rule $\mathcal{QR}_4$. Thus, this discrimination in types allows us to skip certain inference/propagation rules depending on the case and leads to improved efficiency (cf. Algorithm 9). It is also important because certain processes are implemented differently depending on the type of triple concerned (cf. Algorithms 11 − 15).

1. $?x$ sc $?y$
2. $?x$ sp $?y$
3. $?x$ type class
4. $?x$ type prop
5. $?x$ type $?y$ (where $?y \neq$ class $\wedge$ $?y \neq$ prop)
6. $?x$ P $?y$

Table 4.1: Type of triples

In more details, as shown in Algorithm 9, for each triple type of Table 4.1 the rules that applied, in the order they mentioned, are the following:

- Type 1: $\mathcal{QR}_4$, $\mathcal{QR}_3$, $\mathcal{QR}_5$ and $\mathcal{QR}_6$ (Lines 3 − 6)
- Type 2: $\mathcal{QR}_1$, $\mathcal{QR}_2$, $\mathcal{QR}_7$ and $\mathcal{QR}_8$ (Lines 9 − 12)
- Type 3: $\mathcal{QR}_5$ and $\mathcal{QR}_6$ (Lines 16 − 17)
- Type 4: $\mathcal{QR}_7$ and $\mathcal{QR}_8$ (Lines 20 − 21)

Algorithm 9 AddImplicitQuads

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
S the set of explicit quadruples that are going to be added
Output: Set $\mathcal{I}'$ of implicit quadruples

1: let S_cTrans, S_rdfs5, S_rdfs9, S_rdfs7, S_prop1, S_prop2, S_prop3, $S\_prop4 = \emptyset$
2: **if** $p =$ sc **then** // Type 1
3: $S\_rdfs11 =$ UPDT_TCL_SC_SP(t, $\mathcal{G}$, S)
4: $S\_rdfs9 =$ UPDT_TRANS_CLASS_INST(t, $\mathcal{E}$, S, S_rdfs11)
5: $S\_prop1 =$ UPDT_PROPAGATION_CLASS_PROP(t,$\mathcal{E}$,S,S_rdfs11,sc,class)
6: $S\_prop2 =$ UPDT_PROPAGATION_CLASS_INST(t,$\mathcal{E}$,S_rdfs11,S_rdfs9)
7: $\mathcal{I}' = S\_rdfs11 \cup S\_rdfs9 \cup S\_prop1 \cup S\_prop2$
8: **else if** $p =$ sp **then** // Type 2
9: $S\_rdfs5 =$ UPDT_TCL_SC_SP(t, $\mathcal{G}$, S)
10: $S\_rdfs7 =$ UPDT_TRANS_PROP_INST(t, $\mathcal{E}$, S, S_rdfs5)
11: $S\_prop3 =$ UPDT_PROPAGATION_CLASS_PROP(t,$\mathcal{E}$,S,S_rdfs5,sp,prop)
12: $S\_prop4 =$ UPDT_PROPAGATION_PROP_INST(t, $\mathcal{E}$, S_rdfs5, S_rdfs7)
13: $\mathcal{I}' = S\_rdfs5 \cup S\_rdfs7 \cup S\_prop3 \cup S\_prop4$
14: **else if** $p =$ type **then**
15: **if** $o =$ class **then** // Type 3
16: $S\_prop1 =$ UPDT_PROPAGATION_CLASS_PROP(t, $\mathcal{G}$, S, sc, class)
17: $S\_prop2 =$ UPDT_PROPAGATION_CLASS_INST(t, $\mathcal{G}$, S_prop1)
18: $\mathcal{I}' = S\_prop1 \cup S\_prop2$
19: **else if** $o =$ prop **then** // Type 4
20: $S\_prop3 =$ UPDT_PROPAGATION_CLASS_PROP(t, $\mathcal{G}$, S, sp, prop)
21: $S\_prop4 =$ UPDT_PROPAGATION_PROP_INST(t, $\mathcal{G}$, S_prop3)
22: $\mathcal{I}' = S\_prop3 \cup S\_prop4$
23: **else** // Type 5
24: $S\_rdfs9 =$ UPDT_TRANS_CLASS_INST(t, $\mathcal{G}$, S)
25: $S\_prop2 =$ UPDT_PROPAGATION_CLASS_INST(t, $\mathcal{G}$, S_rdfs9)
26: $\mathcal{I}' = S\_rdfs9 \cup S\_prop2$
27: **end if**
28: **else if** $p =$ P **then** // Type 6
29: $S\_rdfs7 =$ UPDT_TRANS_PROP_INST(t, $\mathcal{G}$, S)
30: $S\_prop4 =$ UPDT_PROPAGATION_PROP_INST(t, $\mathcal{G}$, S_rdfs7)
31: $\mathcal{I}' = S\_rdfs7 \cup S\_prop4$
32: **end if**
33: **return** $G' = (E', I')$

- Type 5: $\mathcal{QR}_3$ and $\mathcal{QR}_6$ (Lines 24 − 25)
- Type 6: $\mathcal{QR}_2$ and $\mathcal{QR}_8$ (Lines 29 − 30)

The reason for following a specific order for the application of rules application is the same as in the *annotation* process (described in Section 3.1.3), i.e., the output of one rule may be used as input in a later one.

To understand the above we will describe the case of the addition of a triple (x sc y) (**Type 1**). In this case, rules $\mathcal{QR}_4$, $\mathcal{QR}_3$, $\mathcal{QR}_5$ and $\mathcal{QR}_6$ must be considered. Rule $\mathcal{QR}_4$ models the transitivity of the sc relation and is implemented through UPDT_TCL_SC_SP procedure (Algorithm 10). For a given triple $t = (s, p, o)$ to be added, we first compute the set S of quadruples that correspond to triple t. Then, the new transitive closure is obtained by adding incrementally to the old transitive closure the following [31]:

1. all quadruples obtained by joining a quadruple $q \in S$, with already existing ones that have as object s (i.e., $q_2 = (s_2, p, s, al_2)$) (Line 4).
2. all quadruples obtained by joining a quadruple $q \in S$, with already existing ones that have as subject o (i.e., $q_3 = (o, p, o_3, al_3)$) (Line 7).
3. all quadruples obtained by joining a quadruple $q \in S$, with already existing ones that have as as object s (i.e., $q_2 = (s_2, p, s, al_2)$) and subject o (i.e., $q_3 = (o, p, o_3, al_3)$) (Line 6).

Algorithm 10 UPDT_TCL_SC_SP

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
S the set of explicit quadruples that are going to be added
Output: Set $\mathcal{I}$ of implicit quadruples

```
1:  S' = ∅
2:  for all q = (s, p, o, al_1) in S do
3:      for all q_2 = (s_2, p, s, al_2) in E ∪ I do
4:          S' = S' ∪ { (s_2, p, o, (al_1 ⊙ al_2)) }
5:          for all q_3 = (o, p, o_3, al_3) in E ∪ I do
6:              S' = S' ∪ { (s_2, p, o_3, (al_1 ⊙ (al_2 ⊙ al_3))) }
7:              S' = S' ∪ { (s, p, o_3, (al_1 ⊙ al_3)) }
8:          end for
9:      end for
10: end for
11: return I ∪ S'
```

Implicit quadruples that were produced through $\mathcal{QR}_4$ may fire rules $\mathcal{QR}_3$ and $\mathcal{QR}_5$ of Tables 3.1 and 3.2 respectively. The input to $\mathcal{QR}_3$, which is implemented through UPDT_TRANS_CLASS_INST (Algorithm 11) is the set of explicit quadruples S that are going to be added and the set of implicit quadruples S_rdfs11 that were produced through $\mathcal{QR}_4$ rule application. The rule is applied to explicit quadruples of the form of $s_1, \mathsf{type}, o_1, al_1$, provided that ones of the form of $o, \mathsf{sc}, o_2, al_2$ exist in S or S_rdfs11 (Lines $3-9$ of Algorithm 11)

Algorithm 11 UPDT_TRANS_CLASS_INST

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $G = (E, I)$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
S, the set of explicit quadruples that are going to be added
S_rdfs11, a subset of implicit quadruples that produced through previous $\mathcal{QR}_4$ rule application
Output: Set $\mathcal{I}$ of implicit quadruples

1: let $S' = \emptyset$
2: **if** $p = \mathsf{sc}$ **then**
3: **for all** $q_1 = (s_1, \mathsf{type}, o_1, al_1)$ in $\mathcal{E}$ **do**
4: **for all** $q_2 = (s_2, \mathsf{sc}, o_2, al_2)$ in $S \cup S\_rdfs11$ **do**
5: **if** $o = s_2$ **then**
6: $S' = S' \cup \{ (s, type, o_2, (al_1 \odot al_2)) \}$
7: **end if**
8: **end for**
9: **end for**
10: **else** // $p = type \wedge o \neq Prop \wedge o \neq Class$
11: **for all** $q = (s, p, o, al_1)$ in S **do**
12: **for all** $q_2 = (s_2, \mathsf{sc}, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
13: **if** $o = s_2$ **then**
14: $S' = S' \cup \{ (s, type, o_2, (al_1 \odot al_2)) \}$
15: **end if**
16: **end for**
17: **end for**
18: **end if**
19: **return** $\mathcal{I} \cup S'$

Then, UPDT_PROPAGATION_CLASS_PROP (Algorithm 12) is applied to implement rule $\mathcal{QR}_5$. The input to $\mathcal{QR}_5$ is the set of explicit quadruples S that are going to be added and the set of implicit ones of $S\_cpTrans$ that were produced through $\mathcal{QR}_4$ rule application. The rule is applied to explicit quadruples

of the form of $s_1, \mathsf{type}, \mathsf{class}, al_1$ and $s_3, \mathsf{type}, \mathsf{class}, al_3$, provided that ones of the form of $q_2 = (s_3, \mathsf{sc}, s, al_2$ exist in S or $S\_cpTrans$, as shown in Lines $3 - 11$ of Algorithm 12.

Algorithm 12 UPDT_PROPAGATION_CLASS_PROP

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $G = (E, I)$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Set S of quadruples to be added
$S\_cpTrans$, a subset of implicit quadruples that produced through previous $\mathcal{QR}_4/\mathcal{QR}_1$ rule application
$pred$, one of sc, sp values
obj, one of class, prop values
Output: Set $\mathcal{I}$ of implicit quadruples

1: let $S' = \emptyset$
2: **if** $p = \mathsf{sc} \vee p = \mathsf{sp}$ **then**
3: **for all** $q_1 = (s_1, \mathsf{type}, obj, al_1)$ in $\mathcal{E}$ **do**
4: **for all** $q_2 = (s_2, pred, o_2, al_2)$ in $S \cup S\_cpTrans$ **do**
5: **for all** $q_3 = (s_3, \mathsf{type}, obj, al_3)$ in $\mathcal{E}$ **do**
6: **if** $s = o_2 \wedge s_2 = s_3$ **then**
7: $S' = S' \cup \{ (s_3, \mathsf{type}, obj, \otimes(al_1)) \}$
8: **end if**
9: **end for**
10: **end for**
11: **end for**
12: **else if** $p = \mathsf{type} \wedge (o = \mathsf{class} \vee o = \mathsf{prop})$ **then**
13: **for all** $q_1 = (s_1, \mathsf{type}, obj, al_1)$ in $\mathcal{E}$ **do**
14: **for all** $q_2 = (s_2, pred, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
15: **for all** $q = (s, \mathsf{type}, obj, al)$ in S **do**
16: **if** $s_1 = o_2 \wedge s_2 = s$ **then**
17: $S' = S' \cup \{ (s, \mathsf{type}, obj, \otimes(al_1)) \}$
18: **end if**
19: **if** $s = o_2 \wedge s_2 = s_1$ **then**
20: $S' = S' \cup \{ (s_1, \mathsf{type}, obj, \otimes(al)) \}$
21: **end if**
22: **end for**
23: **end for**
24: **end for**
25: **end if**
26: **return** $\mathcal{I} \cup S'$

The last relevant rule for triples of type 1 is $\mathcal{QR}_6$. This rule can be fired from quadruples previously produced either through $\mathcal{QR}_3$ or $\mathcal{QR}_5$ rule application. $\mathcal{QR}_6$ is implemented through UPDT_PROPAGATION_CLASS_INST procedure which is described in Algorithm 13. In more details, we assume that *S_rdfs9* and *S_prop1* are the sets of implicit quadruples that produced through application of $\mathcal{QR}_3$ and $\mathcal{QR}_5$ rules respectively. Firstly, the rule is applied to the set of explicit and implicit quadruples of the form of $(s_1, \mathsf{type}, \mathsf{class}, al_1)$, provided that ones of the form of $s_2, \mathsf{type}, s_1, al_2$ exist in *S_rdfs9* (Lines $3-9$). Then, it is also applied to implicit quadruples $s_1, \mathsf{type}, \mathsf{class}, al_1$ of *S_prop1*, provided that ones of the form of $s_2, \mathsf{type}, s_1, al_2$ exist in the dataset, as shown in Lines $10-16$ of Algorithm 13.

Algorithm 13 UPDT_PROPAGATION_CLASS_INST

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $G = (E, I)$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
S_rdfs9, *S_prop1* subset of implicit quadruples that produced through previous
$\mathcal{QR}_3$ and $\mathcal{QR}_5$ rule application respectively
Output: Set $\mathcal{I}$ of implicit quadruples

1: let $S' = \emptyset$
2: **if** $p = \mathsf{sc}$ **then**
3: **for all** $q_1 = (s_1, \mathsf{type}, \mathsf{class}, al_1)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
4: **for all** $q_2 = (s_2, \mathsf{type}, o_2, al_2)$ in *S_rdfs9* **do**
5: **if** $s_1 = o_2$ **then**
6: $S' = S' \cup \{ (s_2, \mathsf{type}, o_2, \otimes(al_1)) \}$
7: **end if**
8: **end for**
9: **end for**
10: **for all** $q_1 = (s_1, \mathsf{type}, \mathsf{class}, al_1)$ in *S_prop1* **do**
11: **for all** $q_2 = (s_2, \mathsf{type}, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
12: **if** $s_1 = o_2$ **then**
13: $S' = S' \cup \{ (s_2, \mathsf{type}, o_2, \otimes(al_1)) \}$
14: **end if**
15: **end for**
16: **end for**
17: **else if** $p = type \wedge o = \mathsf{class}$ **then**
18: **for all** $q_1 = (s_1, \mathsf{type}, o_1, al_1)$ in $S \cup S\_prop1$ **do**
19: **for all** $q_2 = (s_2, \mathsf{type}, o_2, al_2)$ in $\mathcal{E} \cup \mathcal{I}$ **do**
20: **if** $s_1 = o_2$ **then**
21: $S' = S' \cup \{ (s_2, type, o_2, \otimes(al_1)) \}$

```
22:             end if
23:         end for
24:     end for
25: else                                          // p = type ∧ o ≠ Prop ∧ o ≠ Class
26:     for all q_1 = (s_1, type, class, al_1) in E ∪ I do
27:         for all q_2 = (s_2, type, o_2, al_2) in S ∪ S_rdfs9 do
28:             if s_1 = o_2 then
29:                 S' = S' ∪ { (s_2, type, o_2, ⊗(al_1)) }
30:             end if
31:         end for
32:     end for
33: end if
34: return I ∪ S'
```

For the rest of the triple types, rules are applied in the same manner as described in case of type 1 and more details about them can be found in Algorithms 10 − 15.

Algorithm 14 UPDT_TRANS_PROP_INST

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Set S of quadruples to be added
S_rdfs5, a subset of implicit quadruples that produced through previous $\mathcal{QR}_1$ rule application
Output: Set I of implicit quadruples

```
1: let S' = ∅
2: if p = sp then
3:     for all q_1 = (s_1, p_1, o_1, al_1) | p_1 ≠ sc ∧ p_1 ≠ sp ∧ p_1 ≠ type in E do
4:         for all q_2 = (s_2, p_2, o_2, al_2) in S ∪ S_rdfs5 do
5:             if p_1 = s_2 then
6:                 S' = S' ∪ { (s_1, o_2, o_1, (at_1 ⊙ al_2)) }
7:             end if
8:         end for
9:     end for
10: else                                          // p = P
11:     for all q = (s, p, o, al) in S do
12:         for all q_2 = (s_2, sp, o_2, al_2) in E ∪ I do
13:             if p = s_2 then
14:                 S' = S' ∪ { (s, o_2, o, (al ⊙ al_2)) }
```

```
            end if
        end for
    end for
end if
return I ∪ S'
```

Algorithm 15 UPDT_PROPAGATION_PROP_INST

Input: Triple $t = (s, p, o)$ that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
S_rdfs7, *S_prop3* subset of implicit quadruples that produced through previous
$\mathcal{QR}_2$ adn $\mathcal{QR}_7$ rules application respectively
Output: Set I of implicit quadruples

```
let S' = ∅
if p = sp then
    for all q1 = (s1, type, class, al1) in E ∪ I do
        for all q2 = (s2, p2, o2, al2) in S_rdfs7 do
            if s1 = p2 then
                S' = S' ∪ { (s2, p2, o2, ⊗(al1)) }
            end if
        end for
    end for
    for all q1 = (s1, type, prop, al1) in S_prop3 do
        for all q2 = (s2, p2, o2, al2 | p2 ≠ sc ∧ p1 ≠ sp ∧ p1 ≠ type in E ∪ I do
            if s1 = p2 then
                S' = S' ∪ { (s2, p2, o2, ⊗(al1)) }
            end if
        end for
    end for
else if p = type ∧ o = prop then
    for all q1 = (s1, p1, o1, al1) in S ∪ S_prop3 do
        for all q2 = (s2, p2, o2, al2 | p2 ≠ sc ∧ p1 ≠ sp ∧ p1 ≠ type in E ∪ I do
            if s1 = p2 then
                S' = S' ∪ { (s2, p2, o2, ⊗(al1) )}
            end if
        end for
    end for
else                                                    // p = P
    for all q1 = (s1, p1, o1, all1) in E ∪ I do
        for all q2 = (s2, p2, o2, all2) in S ∪ S_rdfs7 do
            if s1 = p2 then
                S' = S' ∪ { (s2, p2, o2, ⊗(al1)) }
            end if
        end for
    end for
end if
return I ∪ S'
```

4.1.2 Deleting an existing triple

In this work, we only support deletion of explicit triples. Deleting inferred triples would require the deletion also of (some of) the triples that were used to infer said triple; in a different case, the inferred triple would re-emerge. This process is quite complicated and out of the scope of this work.

When deleting a triple, we must first find all the quadruples that involve said triple and delete them. Then, we should identify all quadruples whose label contains at least one of the IDs of the deleted quadruples and delete them also.

Consider that t is the triple to be deleted. We first find all explicit quadruples that correspond to t as shown in Line 1 of Algorithm 16. Assuming that these returned quadruples are stored in a set S we first delete them from already existing explicit ones (Line 2) and finally, as shown in Line 3, we find and delete those which contain at least one label of quadruples of S. (Algorithm 17).

The efficiency of our framework is evident in the case of triple deletion. The abstract label expressions as mentioned in previous Section store pointers to the quadruples that are used for the computation of the implied and propagated ones. Consequently, detecting the quadruples that are affected by a deletion of a set of quadruples can be done efficiently. This is not the case in state of the art annotation models where the label of the implied/propagated quadruples is simply a value.

Algorithm 16 DeleteTriple

Input: Triple $t = (s, p, o)$ that is going to be deleted
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Output: Updated Graph $G' = (E', I')$
1: $S = \{q \mid q \text{ in } \mathcal{E}, q = (s, p, o, at)\}$
2: $\mathcal{E}' = \mathcal{E} \setminus S$
3: $\mathcal{I}' =$ deleteImplicitQuads($\mathcal{G}$, S) // Algorithm 17
4: **return** $\mathcal{G}' = (\mathcal{E}', \mathcal{I}')$

4.2 Changes in Authorizations

Changes in the authorizations are more complicated, as they might affect more than one triples/quadruples at the same time. Changes in authorizations appear either as additions/deletions of authorizations, or as modifications of existing ones; the latter case can be split in two subcases, namely the modification of the authorization's *SPARQL query* or *access token*.

Algorithm 17 DeleteImplicitQuads

Input: RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Set S, quadruples that are going to be deleted
Output: Set $\mathcal{I}'$ of implicit quadruples

```
1: for all q1 = (s1, p1, o1, at1) in S do
2:     for all q2 = (s2, p2, o2, al2) in I do
3:         if at1 in al1 then
4:             I' = I \ {q2}
5:         end if
6:     end for
7: end for
8: return I
```

4.2.1 Adding a new authorization

When adding a new authorization, say $\mathcal{A} = (q, at)$, we need to, first, determine the triples that are in the scope of q and associate them with their new abstract token at (Line 1 of Algorithm 18), and second, update the labels of all the implicit or propagated quadruples that the affected triples participate in, so that the new label is also considered (Lines $6 - 12$).

The first step normally corresponds to the addition of new quadruples; however, if said triple was previously not in the scope of any authorization, then its existing label ($\perp$) should be deleted and replaced with the new one (at) as shown in Lines 4 and 14. In the latter case (i.e., when $\perp$is simply replaced with at), the second step is not necessary, because any applicable inference and propagation rules have already been applied.

In the former case, for each added quadruple, we should determine the new applications of inference and propagation rules that it causes, and add the corresponding inferred or propagated quadruples. This step can be easily implemented by "copying" existing labels, as shown in Line 9.

In particular, if there is a quadruple $q_1 = (t, at_1)$ and a new quadruple is added for the same triple t, say $q_2 = (t, at_2)$, then as shown in Lines $7 - 11$, all quadruples whose label involves at_1 must be "cloned" to replace at_1 with at_2; this would return the correct results and avoids the need to reconsider explicitly the inference and propagation rules.

Algorithm 18 AddAuth

Input: Authorization $\mathcal{A} = (q, at)$, the authorization that is going to be added
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Output: Updated Graph $\mathcal{G}' = (\mathcal{E}', \mathcal{I}')$

1: let $S_{new} = \{\ q \mid q = (s,\ p,\ o,\ at)$, where $(s,\ p,\ o)$ in the scope of $\mathcal{A}\ \}$
2: **for all** q in S_{new} **do**
3: **if** $q = (s, p, o, \perp)$ in $\mathcal{E}$ **then**
4: $\mathcal{E} = \mathcal{E} \setminus \{q\}$
5: **end if**
6: **if** $q = (s,\ p,\ o,\ at_1 \mid at_1 \neq \perp)$ in $\mathcal{E}$ **then**
7: **for all** $q_2 = (s_2,\ p_2,\ o_2,\ al_2)$ in $\mathcal{I}$ **do**
8: **if** $al_2 = (at_k \odot ... \odot at_1 \odot ... \odot at_m)$ **then**
9: $\mathcal{I}' = \mathcal{I} \cup \{s_1,\ p_1,\ o_1,\ ((at_k \odot ... \odot at \odot ... \odot at_m))\}$
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14: $\mathcal{E}' = \mathcal{E} \cup S_{new}$
15: **return** $\mathcal{G}' = (\mathcal{E}',\ \mathcal{I}')$

4.2.2 Deleting an existing authorization

When deleting an existing authorization, say $\mathcal{A} = (q, at)$, we need to, first, find all quadruples that got their token from authorization $\mathcal{A}$ (Line 2 of Algorithm 19) and possibly delete them (Line 4), and second, update the labels of the implicit or propagated quadruples that the returned quadruples participate in (Line 11).

The first step normally corresponds to the deletion of already existing quadruples; however, if these quadruples are associated with a label only through the authorization $\mathcal{A}$, they should not be deleted, but rather their label replaced with $\perp$, as shown in Lines 7 and 8.

On the other hand, that the quadruples are associated with labels also from another authorizations than $\mathcal{A}$ they should be deleted (Line 4), along with the implied ones that affected by their deletion (Algorithm 17).

Algorithm 19 DeleteAuth

Input: $A = (q, at)$, the authorization that is going to be deleted
RDF Graph $\mathcal{G} = (\mathcal{E}, \mathcal{I})$, $\mathcal{E}$ the set of Explicit Quadruples, $\mathcal{I}$ the set of Implicit Quadruples
Output: Updated Graph $\mathcal{G}' = (\mathcal{E}', \mathcal{I}')$

1: $S_{del} = \emptyset$
2: **for all** $q = (s, p, o, at)$ in E such that (s, p, o) is in the scope of $\mathcal{A}$ **do**
3: **if** there is some $at' \neq at$ such that $(s, p, o, at') \in E$ **then**
4: $\mathcal{E} = \mathcal{E} \setminus \{q\}$
5: $S_{del} = S_{del} \cup \{q\}$
6: **else**
7: $\mathcal{E} = \mathcal{E} \setminus \{q\}$
8: $\mathcal{E} = \mathcal{E} \cup (s, p, o, \perp)$
9: **end if**
10: **end for**
11: $\mathcal{I}' =$ deleteImplicitQuads$(\mathcal{G}, S_{del})$ // Algorithm 17
12: **return** $\mathcal{G}' = (\mathcal{E}', \mathcal{I}')$

4.2.3 Changing SPARQL query of an authorization

When the SPARQL query of an authorization $\mathcal{A}$ changes, the set of triples that are in the scope of the authorization change. This will result to some triples losing their label acquired through $\mathcal{A}$ and some triples gaining a new label through $\mathcal{A}$. This case can therefore be handled in a manner similar to the previous cases by adding and deleting the corresponding quadruples that are in the scope of the new query and not in the scope of the old one.

In more details, suppose that $\mathcal{A}_1 = (q_1, at_1)$ is the authorization with old SPARQL query and $\mathcal{A}_2 = (q_2, at_2)$ the authorization with the new one. We first get all quadruples returned by $\mathcal{A}_1$, say S_{old}, and all quadruples returned by $\mathcal{A}_2$, say S_{new}. The process of changing the SPARQL query can be divided into four cases that combine the processes of Delete_Auth and Add_Auth

The main idea is that for all the triples in $S_{old} \setminus S_{new}$ the process of Delete_Auth must be followed, with the only change that in Line 2 we have to iterate through the quadruples of $S_{old} \setminus S_{new}$. Correspondingly, for all the triples in $S_{new} \setminus S_{old}$ the process of Add_Auth must be followed, with the only change that in Line 1 we have to iterate through the quadruples of $S_{new} \setminus S_{old}$.

More specifically, the four cases are the following:

- In the first $S_old \setminus S_new = S_old$ and $S_new \setminus S_old = S_new$. So, we have to call the customized Delete_Auth procedure for all quadruples of S_old and customized Add_Auth for all quadruples of S_new

- In the second case in which $S_{new} \subset S_{old}$ we only have to delete the quadruples of $S_{old} \setminus S_{new}$ so only the customized process of DELETE_AUTH must be followed

- In the third case in which $S_{old} \subset S_{new}$ we only have to add the quadruples of $S_{new} \setminus S_{old}$ so we only follow the customized process of ADD_AUTH must be followed

- Finally, in case that $S_{old} \cap S_{new} \neq \emptyset$ we have to combine the last two cases. So, we have to delete the quadruples of $S_{old} \setminus S_{new}$ by following the customized process of DELETE_AUTH and add the quadruples of $S_{new} \setminus S_{old}$ by following a customized version of process ADD_AUTH.

4.2.4 Changing token of an authorization

Finally, when the token associated with an authorization changes, no changes are required in the dataset. Recall that triples are not directly associated with the token of the authorization, but with the identifier of the authorization, from which we can directly obtain its token and no changes are required in the associated quadruples. Thus, if the token changes, all we have to do is update the record related to the affected authorization. This is not possible in standard annotation models where we do not record how an access label was obtained (i.e., operations, authorizations, input quadruples involved).

4.3 Changes in the Access Control Policies

Changes in the access control policies is normally the most difficult of the different changes that can be performed on an access-control enhanced dataset. It may involve complicated change operations, such as changing the available tokens (e.g., from a true/false access control scheme to a more fine-grained one with levels of confidentiality), or changing the way that access labels are interpreted (e.g., by changing the confidentiality threshold below which a certain role can access information), of even changing the method for computing the accessibility of an inferred, propagated, or multiply-tagged triple.

Despite that, those types of changes are very easy to handle in our framework. In particular, changes in the access control policies, is the easiest type for our framework. In particular, changing the access control policy simply corresponds to changing the *concrete policy* as this is discussed in Section 3.2. As a result, no changes in the dataset or the related access labels are required, because the changes in the concrete policy will be automatically considered during future queries.

Apart from the obvious efficiency benefits of this feature, it also provides a significant flexibility for applications to define and experiment with different access control semantics.

In particular, different applications (with different requirements) can work seamlessly on the same access control enhanced dataset without having to make copies of the data for each application. Moreover, each application can easily experiment with different concrete policies and/or use different policies per role or user, or even dynamic policies, depending on its needs.

Chapter 5

Implementation

In this chapter we will discuss the relational schemas that we developed in order to implement our approach on controlling access to RDF data. The *Normalized Schema* is discussed in Section 5.1 and *Vertical Partitioning Schema* is presented in detail in Section 5.2. The former stores in a large *triple table* all the triples in the RDF graph, whereas the latter organizes the triples according to the *properties* that exist in the dataset.

5.1 Normalized Schema

The normalized schema is shown in Figure 5.1 that consists of the following tables:

- *UriMap(id,value)* stores the mapping between URIs and literals (*value* column) to unique identifiers (*id* column), in order to avoid managing long URIs and literals that add an overhead during query evaluation.

- *AuthMap(auth_ id,abs_ token)* stores the *authorizations*. For each authorization we store a *unique identifier* (*auth_ id* column) and its *abstract token* (*abs_ token* column).

- *ExplicitQuads(qid,tid,s,p,o,auth_ id)* table stores the *explicit quadruples* that are generated by evaluating the *authorizations* on the set of *explicit input triples*. Columns *s*, *p*, *o* store the identifier (*id*) from table *UriMap* to which the URI/literal of the triple's *subject (s)*, *predicate (p)* and *object (o)* are mapped to. Each quadruple is associated with a unique identifier (*qid* column). Column *tid* stores the identifier of the triple from which the quadruple was derived; it is computed using a hash function on the *subject, predicate* and *object* of the triple. We store both the triple's identifier *tid* and its *s, p, o* components in order to avoid any processing overhead in the case in which

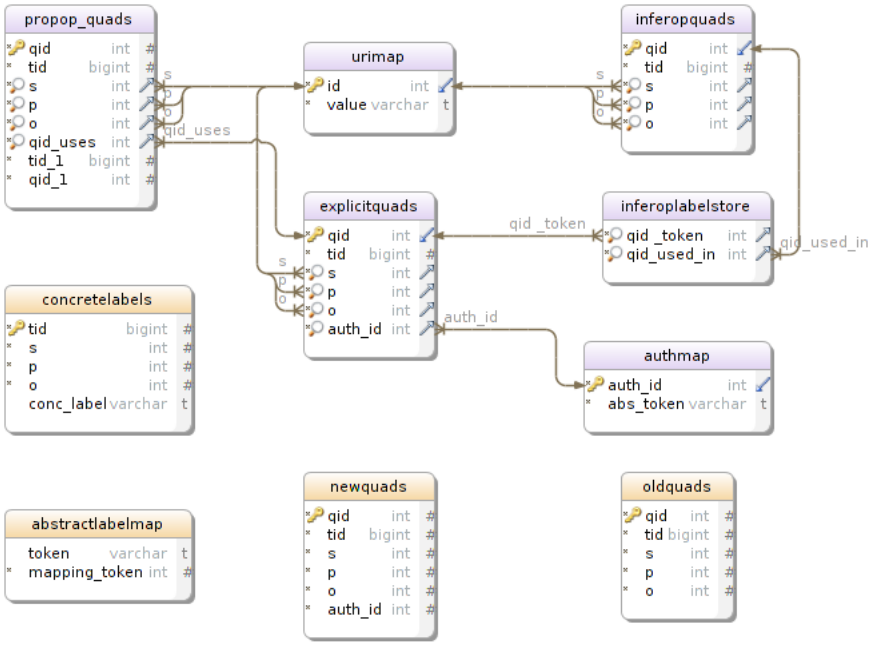


Figure 5.1: Normalized Schema

we need to compare two triples. Finally, *auth_id* stores the authorization that was used to annotate a given triple to produce said quadruple.

- *InferopQuads(qid,tid,s,p,o)* stores the implicit quadruples that were produced through the RDFS inference rules of Table 3.1. The meaning of the columns of this table is exactly the same as for the table *ExplicitQuads()*.

- *InferopLabelstore(qid_token,qid_used_in)* stores for each implied quadruple (*qid_used_in* column), the identifier of the quadruple (column *qid_token*) used in the implication of the former. We store the *quadruples* and not their access token since we want to have *direct access* to them in the case of updates. In addition, we store for each quadruple, *all the explicit quadruples* that participate in its implication. The reason is to avoid flatenning the complex inference expression at *query time*, something that would implicate a large number of joins. If we had followed a different approach in which we would record the identifiers of the contributing quadruples, then in order to retrieve all the labels (to decide on the accessibility of a triple) that contributed to the label of said triple, additional joins with the *InferopQuads* and *InferopLabelstore* tables would be necessary. The approach we follow resolves

this problem by eliminating this "nesting" of complex labels. This unnesting of labels occurs during the application of inference rules and helps avoiding recursive joins that would slow down the process of label (and query) evaluation. Table 5.2(e) shows the contents of *InferopLabelstore* for our motivating example.

- *PropopQuads(qid,tid,s,p,o,qid_ uses,tid_ 1,qid_ 1)* stores the quadruples that are obtained through *propagation*. The meaning of the columns of this table is exactly the same as for the tables *ExplicitQuads()* and *InferopQuads()*. Columns *qid_ uses* and *qid_ 1* are used to store the identifiers of the *explicit* and *implicit* quadruples respectively from which said quadruple obtains its label. Column *tid_ 1* stores the identifier of quadruple with *qid_ 1*. We explain below in detail the rational for this choice of representation.

Figures 5.2(a), 5.2(b) show the contents of tables *UriMap*() and *AuthMap* for our motivating example. Tables *ExplicitQuads*, *InferopQuads* and *PropopQuads* for the motivating example are shown in Figures 5.2 (c), 5.2 (d) and 5.2(f) respectively. In those figures, we write q_{id} to refer to the identifier of a quadruple and t_{id} to refer to the identifier of a triple.

- Tables *NewQuads* and *OldQuads* are used in our update scenarios and quadruples that have to be added or deleted stored in them. More specifically, *NewQuads* is used in cases of adding a triple, adding a new authorization or changing the query of an existing once and *OldQuads* in cases of deleting a triple, deleting an authorization or changing the query of an existing once. The meaning of the columns in these tables are exactly the same as in tables *ExplicitQuads* and *InferopQuads*.

Table *ConcreteLabels(tid, s, p, o, conc_ label)* stores the triples of the RDF graph after the *annotation process* is completed. It is used in evaluation process where given a concrete policy we have to compute for each triple its concrete label. The meaning of columns *tid*, *s*, *o* and *p* are as before. Column *conc_ label* stores the concrete label that is calculated during evaluation. This table is materialized at *query time*. Last, table *AbstractlabelMap(token, mapping_ token)* stores the *mapping* of the abstract token (column *token*) to the concrete one (column *mapping_ token*).

Below we discuss how the tables *ExplicitQuads*, *InferopQuads*, *PropopQuads* and *InferopLabelStore* are populated.

For doing this, recall that a label can be obtained as follows:

1. directly through an authorization, in which case it is an abstract token (e.g., quadruples q_1 to q_4 and q_6 to q_7, Figure 5.2 (c));

id	*Value*
$\mathbf{u_1}$	sc
$\mathbf{u_2}$	type
$\mathbf{u_3}$	*Student*
$\mathbf{u_4}$	*Person*
$\mathbf{u_5}$	*Agent*
$\mathbf{u_6}$	class
$\mathbf{u_7}$	*&a*
$\mathbf{u_8}$	*firstName*
$\mathbf{u_9}$	*lastName*
$\mathbf{u_{10}}$	Alice
$\mathbf{u_{11}}$	Smith

(a) *UriMap*

id	*Value*
$\mathbf{a_0}$	$\perp$
$\mathbf{a_1}$	at_1
$\mathbf{a_2}$	at_2
$\mathbf{a_3}$	at_3
$\mathbf{a_4}$	at_4
$\mathbf{a_5}$	at_5

(b) *AuthMap*

qid	*tid*	*s*	*p*	*o*	*auth_id*			
q_1	t_1	$\mathbf{u_3}$	$\mathbf{u_1}$	$\mathbf{u_4}$	$\mathbf{a_2}$	(*Student*, sc, *Person*)	$\rightarrow$	at_2
q_2	t_2	$\mathbf{u_4}$	$\mathbf{u_1}$	$\mathbf{u_5}$	$\mathbf{a_2}$	(*Person*, sc, *Agent*)	$\rightarrow$	at_2
q_3	t_3	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_3}$	$\mathbf{a_3}$	(*&a*, type, *Student*)	$\rightarrow$	at_3
q_4	t_4	$\mathbf{u_7}$	$\mathbf{u_8}$	$\mathbf{u_{10}}$	$\mathbf{a_1}$	(*&a*, *firstName*, *Alice*)	$\rightarrow$	at_1
q_5	t_5	$\mathbf{u_7}$	$\mathbf{u_9}$	$\mathbf{u_{11}}$	$\mathbf{a_0}$	(*&a*, *lastName*, *Smith*)	$\rightarrow$	$\perp$
q_6	t_6	$\mathbf{u_5}$	$\mathbf{u_2}$	$\mathbf{u_6}$	$\mathbf{a_4}$	(*Agent*, type, class)	$\rightarrow$	at_4
q_7	t_1	$\mathbf{u_3}$	$\mathbf{u_1}$	$\mathbf{u_4}$	$\mathbf{a_5}$	(*Student*, sc, *Person*)	$\rightarrow$	at_5

(c) *ExplicitQuads & Triples with their access token*

qid	*tid*	*s*	*p*	*o*	
q_8	t_7	$\mathbf{u_3}$	$\mathbf{u_1}$	$\mathbf{u_5}$	(*Student*, sc, *Agent*)
q_9	t_7	$\mathbf{u_3}$	$\mathbf{u_1}$	$\mathbf{u_5}$	(*Student*, sc, *Agent*)
q_{10}	t_8	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_4}$	(*&a*, type, *Person*)
q_{11}	t_9	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_5}$	(*&a*, type, *Agent*)
q_{12}	t_9	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_5}$	(*&a*, type, *Agent*)
q_{13}	t_8	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_4}$	(*&a*, type, *Person*)

(d) *InferopQuads*

qid_token	*qid_used_in*	
q_1	q_8	
q_2	q_8	$q_8 = q_1 \odot q_2$
q_7	q_9	
q_2	q_9	$q_9 = q_7 \odot q_2$
q_3	q_{10}	
q_1	q_{10}	$q_{10} = q_3 \odot q_1$
q_3	q_{11}	
q_1	q_{11}	$q_{11} = q_3 \odot q_8$
q_2	q_{11}	
q_3	q_{12}	
q_7	q_{12}	$q_{12} = q_3 \odot q_9$
q_2	q_{12}	
q_3	q_{13}	
q_7	q_{13}	$q_{13} = q_3 \odot q_7$

(e) *InferopLabelStore*

qid	*tid*	*s*	*p*	*o*	*qid_uses*	*tid_1*	*qid_1*	
q_{14}	t_9	$\mathbf{u_7}$	$\mathbf{u_2}$	$\mathbf{u_5}$	$\mathbf{u_6}$	$\mathbf{t_9}$	null	(*&a*, type, *Agent*), $q_{14} = \otimes q_6$

(f) *PropopQuads()*

Figure 5.2: *UriMap*, *AuthMap*, *ExplicitQuads*, *InferopQuads*, *InferopLabelStore* and *PropopQuads* Tables

2. indirectly, when the associated triple is not in the scope of any authorization, in which case it is the default access token $\perp$ (e.g., quadruple q_5, Figure 5.2 (c));

3. through an inference rule, in which case it is a complex expression of the form $al_1 \odot al_2$, where al_1, al_2 are the labels of the quadruples used to infer the said quadruple (quadruples q_8 to q_{13}, Figure 5.2 (d));

4. through a propagation rule, in which case it is $\otimes al$ (e.g., quadruple q_{14}, Figure 5.2(f)).

Consider now the case of implied quadruple (case (3) above). Implied quadruples are stored in the *InferopQuads* table (Figure 5.2(d)) and their complex expressions in *InferopLabelstore* table (Figure 5.2(e)). One tuple is added in *InferopLabelstore* for each of the quadruples that imply said quadruple. In *InferopLabelstore* we store the *explicit* quadruples that contribute to the implication of the implicit ones. For example, consider q_8 in Figure 5.2(d); q_8 is an inferred quadruple, occurring from the application of the inference rule $\mathcal{QR}_4$ of Table 3.1 upon quadruples q_1, q_2. Our approach is to store that q_8 results from applying the $\odot$ operator upon the identifiers of q_1, q_2. Note that this is different from storing the fact that q_8 results from applying the $\odot$ operator upon at_2, at_2 (the labels for q_1, q_2). The former is useful when updates are concerned since it allows us to find efficiently all the explicit quadruples that are involved in the production of an implicit one, and is especially useful when changes occur.

In our example *InferopLabelstore* contains the tuples $(q_1, q_8), (q_2, q_8)$. Consider now quadruple q_{11} in Figure 5.2(d) which is obtained by the application of the inference rule $\mathcal{QR}_3$ upon quadruples q_3, q_8. Since, as explained previously, in *InferopLabelstore* we store the identifiers of the *explicit quadruples* used to obtain the implicit ones, then we add tuples (q_3, q_{11}) (for q_3) and $(q_1, q_{11}), (q_2, q_{11})$ (for q_8).

Finally, in case (4) implicit quadruples that produced through propagation rules of Table 3.2 are stored in *PropopQuads* as shown in Figure 5.2(f).

[Irini Note: Must read this again] We store the complex label of a quadruple in table *PropopQuads* table and more specifically, we use column *qid_uses* for this purpose. The reason we do not need a separate table for storing the label is that there is an $1-1$ relation between an implied quadruple and the explicit one from it gets the label through propagation. Also as in case (3) only the identifiers of explicit quadruples used. Column (*tid_1*) is used in order to store the second triple (as show in rules of Table 3.2) that participated in production of an implied quadruple, except from the quadruple that propagated its label. This is necessary in the case that changes occur. For example, if the triple that will be deleted also participates in production of an implied quad, the implied quad has to be deleted

too. So, *tid_1* denotes the identifier t_{id} of the above triple. Finally, *qid_1* is used in cases that the quadruple from which label is propagated is an implied quad (first quadruple of $\mathcal{QR}_6$ or $\mathcal{QR}_8$ rule produced through $\mathcal{QR}_5$ or $\mathcal{QR}_7$ rule respectively - Table 3.2). As we said before, in *qid_uses* only identifiers of explicit quadruples used. By doing this, in case that the label of an implied quadruple have to be propagated to another implied one, we store the identifier of explicit quadruple that stored for the first implied quadruple. So, at this point we have lost the information that tell us which quadruples are contributed to production of new implied quadruples, which is necessary in cases of deletion, as we mentioned before. To avoid this information loss we use column *qid_1*, in which we store the identifier of implied quadruple from which label is propagated.

5.2 Vertical Partitioned Schema

The idea behind the vertical partioning schema is to define one table per RDFS *property*. More specifically, we define one table for the *subClassOf* (sc), *subPropertyOf* (sp), *type* (type) properties. We define one single table for all the remaining properties. For each such property, we define one table to store the *explicit* and *implicit* quadruples. In the case of *implicit* quadruples, we define one table (per property) to store the complex labels. Consequently, we have fourteen tables instead of six tables in the case of the normalized schema. By following this approach, we increase our performance as our tables become much smaller and because in rules application and update scenarios we know exactly in which table each time we have to look.

The schema contains tables *UriMap(id,value)*, *AuthMap(auth_id,abs_token)*, *NewQuads*, *OldQuads*, *ConcreteLabels(tid, s, p, o, conc_label)* and *AbstractlabelMap(token, mapping_token)* as for the previous schema. The tables used for the vertical partitioning schema are the following:

Storing Explicit Quads: In order to store the *explicit quadruples* we define tables *scExplicitQuads()*, *spExplicitQuads()*, *typeExplicitQuads()* and *pExplicitQuads()* for sc, sp, type for the remaining properties respectively. These tables share the same schema: column *qid* that stores the unique identifier of a quadruple, *tid* that stores the identifier of the triple from which the quadruple was produced. Finally, columns *s* and *o* that stand for the *subject* and *object* components of a triple. Figures 5.3(a), 5.3(b) and 5.3(c) how relational tables *scExplicitQuads*, *typeExplicitQuads* and *pExplicitQuads* respectively for our motivating example.

Storing Implicit Quads: To store implicit quads, we use tables *scInferopQuads()*, *spInferopQuads()*, *typeExplicitQuads()* and *pInferopQuads()* for sc, sp, type for the remaining properties respectively. These tables have the same schema as the tables that store the explicit quadruples. Figures 5.4(a), 5.4(b), 5.4(b) shows the contents

qid	*tid*	*s*	*o*	*auth_id*				
q_1	t_1	$\mathbf{u_3}$	$\mathbf{u_4}$	$\mathbf{a_2}$	(*Student*, sc, *Person*)	→	at_2
q_2	t_2	$\mathbf{u_4}$	$\mathbf{u_5}$	$\mathbf{a_2}$	(*Person*, sc, *Agent*)	→	at_2
q_7	t_1	$\mathbf{u_3}$	$\mathbf{u_4}$	$\mathbf{a_5}$	(*Student*, sc, *Person*)	→	at_5

(a) *scExplicitQuads & Triples with their access token*

qid	*tid*	*s*	*o*	*auth_id*			
q_3	t_3	$\mathbf{u_7}$	$\mathbf{u_3}$	$\mathbf{a_3}$	(&*a*, type, *Student*)	→	at_3
q_6	t_6	$\mathbf{u_5}$	$\mathbf{u_6}$	$\mathbf{a_4}$	(*Agent*, type, class)	→	at_4

(b) *typeExplicitQuads & Triples with their access token*

qid	*tid*	*s*	*p*	*o*	*auth_id*			
q_4	t_4	$\mathbf{u_7}$	$\mathbf{u_8}$	$\mathbf{u_{10}}$	$\mathbf{a_1}$	(&*a*, *firstName*, *Alice*)	→	at_1
q_5	t_5	$\mathbf{u_7}$	$\mathbf{u_9}$	$\mathbf{u_{11}}$	$\mathbf{a_0}$	(&*a*, *lastName*, *Smith*)	→	$\perp$

(c) *pExplicitQuads & Triples with their access token*

Figure 5.3: *scExplicitQuads*, *typeExplicitQuads* and *pExplicitQuads* Tables

of tables *scInferopQuads()* and *typeExplicitQuads()* for our motivating example.

qid	*tid*	*s*	*o*	
q_8	t_7	$\mathbf{u_3}$	$\mathbf{u_5}$	(*Student*, sc, *Agent*)
q_9	t_7	$\mathbf{u_3}$	$\mathbf{u_5}$	(*Student*, sc, *Agent*)

(a) *scInferopQuads*

qid	*tid*	*s*	*o*	
q_{10}	t_8	$\mathbf{u_7}$	$\mathbf{u_4}$	(&*a*, type, *Person*)
q_{11}	t_9	$\mathbf{u_7}$	$\mathbf{u_5}$	(&*a*, type, *Agent*)
q_{12}	t_9	$\mathbf{u_7}$	$\mathbf{u_5}$	(&*a*, type, *Agent*)
q_{13}	t_8	$\mathbf{u_7}$	$\mathbf{u_4}$	(&*a*, type, *Person*)

(b) *typeInferopQuads*

Figure 5.4: *scInferopQuads* and *typeInferopQuads* tables

Storing access labels: To store the access labels of the implicit quadruples we use the *scInferopLabelstore*, *spInferopLabelstore*, *typeInferopLabelstore* and *pInferopLabelstore* tables for sc, sp, type for the remaining properties respectively. All tables share the same schema: *qid_ used_ in* that stores the identifier of the implied quadruple and *qid_ token* that stores the identifiers of the *explicit* quadruples that contributed to the implication of the former. Finally, figures 5.5(a) and 5.5(b)

show tables *scInferopLabelStore()* and *typeInferopLabelStore()* for our motivating example.

qid_token	*qid_used_in*	
q_1	q_8	
q_2	q_8	$q_8 = q_1 \odot q_2$
q_7	q_9	
q_2	q_9	$q_9 = q_7 \odot q_2$

(a) *scInferopLabelstore*

qid_token	*qid_used_in*	
q_3	q_{10}	
q_1	q_{10}	$q_{10} = q_3 \odot q_1$
q_3	q_{11}	
q_1	q_{11}	$q_{11} = q_3 \odot q_8$
q_2	q_{11}	
q_3	q_{12}	
q_7	q_{12}	$q_{12} = q_3 \odot q_9$
q_2	q_{12}	
q_3	q_{13}	
q_7	q_{13}	$q_{13} = q_3 \odot q_7$

(b) *typeInferopLabelstore*

Figure 5.5: *scInferopQuads* and *typeInferopQuads* tables

The implicit quadruples that are produced through the propagation rules of Table 3.2 are stored in *typePropopQuads* and *pPropopQuads* tables. The schema is the same as for the *PropopQuads* table discussed earlier. Figure 5.1 shows table *typePropopQuads* for our motivating example.

qid	*tid*	*s*	*o*	*qid_uses*	*tid_1*	*qid_1*	
q_{14}	t_9	$\mathbf{u_7}$	$\mathbf{u_5}$	q_6	t_9	null	$(\&a,\ \mathsf{type},\ Agent),\ q_{14} = \otimes q_6$

Table 5.1: *PropopQuads()* table

Chapter 6

Evaluation

6.1 Datasets and Access Policies

We used both real and synthetic datasets in our experiments.

6.1.1 Real Datasets

We used datasets GADM-RDF [32] and GeoSpecies [33] enhanced with links from DBPedia using LDSpider [34] from the PlanetData Network of Excellence[1]. We also used, the CIDOC [35] and GO [36] ontologies in our experiments. To obtain statistical information about the datasets we used LODStats[2]. LDSpider provides a web crawling framework for the Linked Data web and more specifically, it traverses the Web of Linked Data by following RDF links between data items.

More specifically:

- GADM-RDF is a spatial database that stores all the administrative areas. Administrative areas in this database are countries and lower level subdivisions. GADM provides for each area some attributes, foremost being the name and variant names. In order to enhance GADM-RDF with links from DBPedia we used LDSpider. After this enhancement GADM consist of 11.303.686 million triples that define 26 classes, 40 properties, 25 sc, and approximately 2 million type relations.

- The GeoSpecies Knowledge Base provides information on Biological Orders, Families, Species as well as species occurrence records and related data. As in the case of GADM-RDF, we used LDSpider in order to follow the 11.805 links from said dataset to DBPedia. The resulting GeoSpecies dataset contained

[1] PlanetData: http://www.planet-data.eu/

[2] LODStats: http://wiki.aksw.org/Projects/LODStats?show_comments=1

approximately 2.591.011 million triples that define 111 classes, 40 properties, 97 sc, 144 sp and approx. 2 million type relations.

- The CIDOC Conceptual Reference Model (CRM) provides definitions and a formal structure for describing the concepts and relations used in cultural heritage documentation. CIDOC consists of 3282 triples that define 82 classes structured in a hierarchy of 5 levels (depth 4), 260 properties structured in a hierarchy of 4 levels (depth 3), no class or property instances, 94 sc, 130 sp and 342 type relations.
- The Gene Ontology (GO) project is a bioinformatics initiative with the aim of standardizing the representation of genes and gene product attributes. GO consists of a controlled vocabulary of terms describing gene product characteristics and gene product annotation data. It contains 265.355 explicit triples that define 35.451 classes structured in a hierarchy of 10 levels (depth 9) and no properties. Also it has 35.451 class instances and no property instances. Finally, it defines 55.169 sc, no sp and 35.451 type relations.

	GADM-RDF	GeoSpecies	CIDOC	GO
classes:	26	111	82	35451
properties:	40	40	260	0
class instances:	2915	88868	0	35451
property instances:	41387	48145	0	0
sc relations:	25	97	94	55169
sp relations:	0	0	130	0
type relations:	1856187	127587	342	35451
class hierarchy depth:	-	-	4	9
property hierarchy depth:	-	-	3	0
explicit triples:	11303686	2591011	3263	265355
implied triples:	9	496028	451	395517
implied quads:	88630	1174110	2325	5581414

Table 6.1: Characteristics of real datasets

The characteristics of the aforementioned datasets are shown in Figure 6.1.

6.1.2 Synthetic Datasets

To produce the synthetic schemas for our experiments we used Powergen [37], which is the first synthetic RDFS schema generator that takes into account the

morphological features that schemas frequently exhibit in reality [17] to produce realistic ontologies. We used the parameters of PowerGen to obtain ontologies with an increasing number of implied triples and different characteristics, in order to test the applicability of our approach in various cases.

In particular, we produced 38 synthetic ontologies containing 100-1000 classes, 113-1635 properties, 124-50.295 class instances and 70-18.242 property instances and 110-1321 sc relations. We also experimented with different depths for the sc and sp hierarchies of the input RDF schema (ranging from depth 4 to 8).

6.1.3 Access Policies

Since there is no standard benchmark for access control, we used our own custom set of authorizations. In a sense, all the authorizations and concrete policies we experimented with are synthetic. Our authorizations assigned access tokens to the sc, sp and type relations, and were defined in such a way that featured the assignment of different access tokens to the same triple (setting the basis for the application of conflict resolution), as well as the assignment of the same token to different triples.

For our experiments, we produce *explicit quadruples* (i.e. labeled triples) from a given set of input RDF triples in a random manner. More specifically, we consider three abstract tokens at_1, at_2 and at_3 which are assigned to triples with a rate of *20%*, *50%* and *5%* respectively. By doing this, a triple may annotated with one, two or three tokens. In details, based on above rates, one triple will be annotated with exactly one abstract token with a rate of *49.5%*, with exactly two tokens with a rate of *12%* and with all three tokens with a rate of *0.5%* . The remaining *38%* of triples is not annotated with any abstract token, so $\perp$ is assigned to them. To decide from which authorization each quadruple got its abstract token we suppose that we have a set of 15 authorizations. Authorizations $A_1...A_5$ assign to triples abstract token at_1, $A_6...A_{10}$ abstract token at_2 and $A_{11}...A_{15}$ abstract token at_3. So, we choose one of them with a rate of *20%* from each group.

The concrete policy that we use for our experiments uses concrete tokens *Low*, *Medium* and *High* as confidentiality levels, which we map to 1, 2 and 3. The default token $\perp$ is mapped to 0. Abstract access tokens at_1, at_2 and at_3 are mapped to concrete tokens *Low*, *Medium* and *High* respectively. The inference operator is $min()$ the propagation operator is the identity function and the conflict resolution operator is $max()$.

6.2 Experimental Settings

All experiments were conducted on a Dell OptiPlex 755 desktop with CPU Intel® Core^{TM}2 Duo CPU E8400 at 3.00GHz, 8 GB of memory and running Linux Ubuntu

10.04.4 LTS release with 2.6.35-31-generic x86_64 Kernel release.

We used MonetDB column store DBMS, version v11.11.7-Jul2012-SP1 as our relational backend. The implementation of entailment and propagation rules shown in Tables 3.1 and 3.2, as well as the implementation of our update scenarios done using MonetDB's *stored procedures*. We performed only *cold-cache* experiments. More specifically, before running each of the experiments that we will discuss in the following, we first flushed out MonetDB's buffers by executing a query that returns all the tuples in all tables of the database.

The experiments that we conducted for both the *normalized* and the *vertical partitioning* schemas discussed in Section 5 are the following:

EXPERIMENT 1 measures the *annotation time*, i.e., the time required to compute all the *implicit* and *propagated* quadruples.

EXPERIMENT 2 measures the *evaluation time*, that is the time needed to compute for *a concrete policy* and for *all the triples* in a specific dataset, the concrete labels of triples.

EXPERIMENT 3 measures the *evaluation time* for a concrete policy and for a *subset* of the annotated quadruples which correspond to a specific *type* of triples (as specified in Table 4.1 of Section 4.1.1).

EXPERIMENT 4 measures the time required to *delete* a randomly selected *explicit triple* of a specific type. All the implied that were produced through this explicit one must also be deleted. The number of *implicit* quadruples that produced through an explicit triple t depend on the number of *explicit* quadruples in which t corresponds to, as the more explicit quadruples of one triple exist, the more implied ones will be produced. So, for each type of triple (as defined in Table 4.1 of Section 4.1.1) we select three random ones that have been annotated with one, two and three tokens respectively. The time required to delete a triple of a specific type is computed as the average of the three chosen triples.

EXPERIMENT 5 measures the time required to *add* an *explicit triple* of specific type and to recompute the implied ones with their abstract labels, that are produced through this addition. For testing purposes we consider as our input the (a) triple that we chose to delete in EXPERIMENT 4 and (b) its corresponding quadruples.

EXPERIMENT 6 compares the *annotation time* and the average time required to *delete* or *add* an *explicit triple* as this was computed from the EXPERIMENT 5 and EXPERIMENT 4.

EXPERIMENT 7 considers the difference between *annotation time* and the average time required to *delete* or *add* a random *authorization*. To achieve the randomness in selecting the authorization that will be deleted, we select one authorization from

each of the groups we discussed in Section 6.1.3. So, we select one from $A_1...A_5$, one from $A_6...A_{10}$ and one from $A_{10}...A_{15}$. In the same manner as in case of triple addition, for testing purposes, the authorization that selected for addition is this one that previously was deleted.

6.3 Experimental Results

The general observation from our experiments is that the annotation, evaluation and update times are roughly linear to the number of triples or quadruples considered, i.e., the number of implied quadruples of the input for EXPERIMENT 1 and total quadruples for EXPERIMENT 2 to EXPERIMENT 7. Other factors affecting the time for all experiments are (a) the structure of the ontology and (b) the storage schemes employed. More details on the various experiments and the related results follow below.

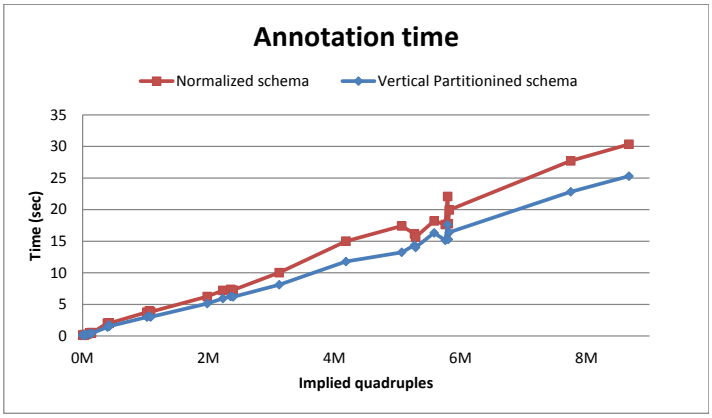


Figure 6.1: EXPERIMENT 1 - Annotation time

Experiment 1

EXPERIMENT 1 measured the *annotation time*, that is the time needed to apply the inference and propagation rules of Tables 3.1 and 3.2 on the initial set of quadruples. The quadruples were obtained from the application of the authorizations to the initial set of RDF triples; note that the time needed to assign these initial access tokens (using the authorizations) upon the existing triples is an offline process and so we are not reporting this time.

Figure 6.1 shows the *annotation time* for the synthetic datasets for both normalized and vertical partitioned schemas. The graph shows that the annotation time increases as the number of implied quadruples increases. This is an expected

result since when the number of implied quadruples increases, the number of times the inference and propagation rules are applied also increases. The observed peak in the point of 6M of implied quadruples is due to changes in the structure of the ontology and more specifically to the increase of the depth of the class hierarchy from 4 to 6 (a PowerGen parameter). The implication of such a increase is that even though the number of implied quadruples remains the same, there are more applications of inference rules. As shown in this graph, the vertical partitioned schema outperforms the normalized one for all synthetic datasets, since in the second one, the database operations are applied on smaller tables.

Regarding real datasets Table 6.2 shows the *annotation time* for both normalized and vertical partitioned schemas. As we can see for both schemas annotation time increases along with the number of implied quadruples which depends on the complexity of the ontology's schema. For example, consider the GO and GADM-RDF ontologies. GO is much smaller than GADM-RDF, but its structure is much more complex. More specifically, as we can see in Table 6.1, GO has a very deep class hierarchy (9), as well as a large number of sc, sp and type relations. This complexity causes more applications of inference and propagations rules, and, consequently, leads to a larger number of implied quadruples.

Dataset	Normalized	Vertical Partitioned
GADM-RDF	6,189	1,471
GeoSpecies	5,778	4,015
CIDOC	0,151	0,162
GO	57,620	57,063

Table 6.2: Experiment 1 - *Annotation time* for real datasets in seconds

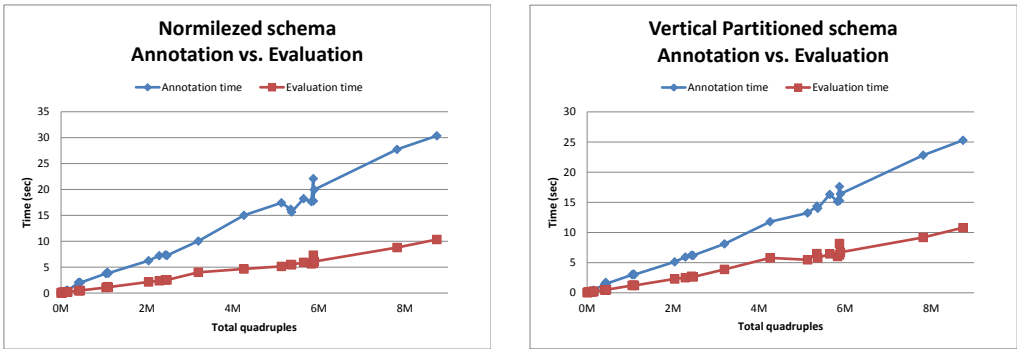
Comparing *annotation time* between the two schemas we can see that *vertical partitioned* schema outperforms the *normalized* one for all datasets and especially for GADM-RDF. The reason why the improvement in GADM-RDF is greater, is that the majority of the rules are applied on tables much more smaller than the one that stores all the explicit triples in the case of the normalized schema. For example, the rule that computes the transitive closure of sc relation in the case of the vertical partitioned are applied on a table containing 56 rows, in contrast to the normalized schema, where the table contains all the 13.043.954 explicit quadruples.

Experiment 2

Experiment 2 compares *annotation time* and the time required to compute the concrete labels for all RDF quadruples of each dataset, for the concrete policy

discussed in Section 6.1.3.

Figure 6.2 shows the related *evaluation time* for the synthetic datasets for both schemas. In both schemas we note, as expected, that the evaluation time increases linearly with respect to the total quadruples in the dataset. The small peak of *evaluation time* observed in the point of 6M quadruples is due to the fact that the depth of the ontology was increased at these point (from 4 to 6), causing a sudden increase in the number of applications of the inference and propagation rules;Also as shown in the figure, the annotation time is much larger than the time required to compute the concrete label of a triple for both schemas, for real as well as synthetic datasets.



(a) Normalized schema (b) Vertical Partitioned schema

Figure 6.2: EXPERIMENT 2 - Comparison between *annotation* and *evaluation time*

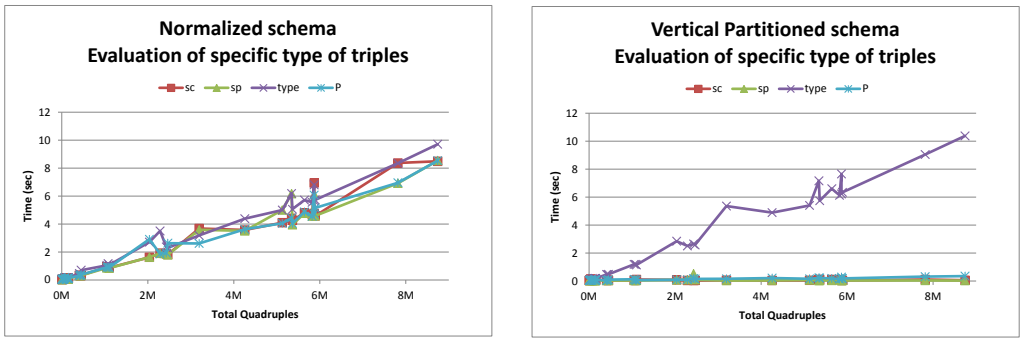
The comparison of *annotation* and *evaluation time* for real datasets is shown in Table 6.3. Note that in this experiment we evaluate the concrete labels for *all* the triples in the dataset. Looking at the results we can say that it is not always the case that evaluating the concrete labels is more preferrable than reannotating the whole graph. This is evident in the case of the GADM-RDF ontology that has a simple schema but a very large number of instances. In this case, it is more preferable to re-annotate the whole graph than to compute the concrete labels for all triples. However, in a real-world scenario, a users does not query for all triples of the dataset, and as we will see in the next experiments our model is very beneficial.

Dataset	Normalized		Vertical Partitioned	
	Evaluation	Annotation	Evaluation	Annotation
GADM-RDF	25,381	6,189	26,645	1,471
GeoSpecies	7,927	5,778	7,654	4,015
CIDOC	0,060	0,151	0,051	0,162
GO	19,331	57,620	20,564	57,063

Table 6.3: EXPERIMENT 2 - *Annotation* vs. *Evaluation time* for all triples of real datasets in seconds

Experiment 3

EXPERIMENT 3 measures the time required to compute the concrete labels for a subset of annotated quadruples, under the concrete policy we used for our experiments. This set of annotated quadruples corresponds to a specific type of triple as defined in Table 4.1 of Section 4.1.1. More specifically from the defined triple types we compute *evaluation time* for sc, sp, type and P (some user defined property).



(a) Normalized schema

(b) Vertical Partitioned schema

Figure 6.3: EXPERIMENT 3 - *Evaluation time* for a specific type of triples

As shown in Figure 6.3, *evaluation* in the case of the *vertical partioned* schema is much more efficient than in the case of the *normalized* one, for all types of properties. This is an expected outcome since in the case of the vertical partitioned schema, as described in Section 5, quadruples are stored in specific tables according to their predicate. Following this storage technique we have a benefit on computing the concrete labels of specific type of triple, as we know exactly the tables we have to look at. In addition, the tables we have to look at, are much smaller than the ones in the case of the normalized schema. The above can be shown in our

experiments by looking i.e. the *evaluation time* of triples of category type and sc. The synthetic datasets that we use, mostly consist of instance triples (type), so in the case of the vertical partitioned schema, the tables that store these types of triples are much bigger than the ones that store scones. This, implies more time for computing the concrete label for a triple of category type than for one of category sc. On the other hand, the evaluation time for type and sc for the normalized schema, is the same since these triples are stored in the same table. The same observations can also be made for real datasets for which the *evaluation time* for a specific type of triples shown in Table 6.4.

Type of triple	Schemas & Info	GADM-RDF	GeoSpecies	CIDOC	GO
sc	Normalized	7,956	3,745	0,037	24,155
	Vertical Partitioned	0,057	0,041	0,096	21,830
	# of evaluated triples	31	147	483	448868
sp	Normalized	7,824	3,557	0,026	18,425
	Vertical Partitioned	0,034	0,024	0,042	0,034
	# of evaluated triples	0	16	192	0
type	Normalized	12,845	4,017	0,034	18,519
	Vertical Partitioned	4,054	0,595	0,042	0,131
	# of evaluated triples	1856188	237726	342	35451
P	Normalized	25,383	7,345	0,035	18,734
	Vertical Partitioned	22,519	5,312	0,042	0,413
	# of evaluated triples	9447475	2829389	2697	174735

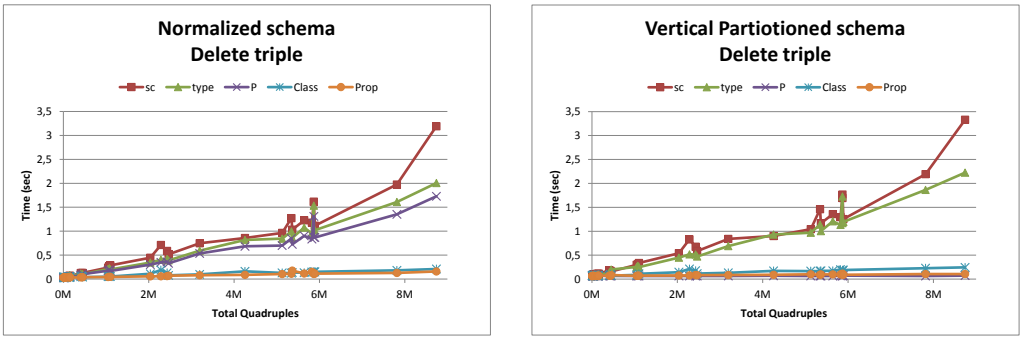
Table 6.4: EXPERIMENT 3 - *Evaluation time* for a specific type of triples of real datasets (in seconds)

Experiment 4

EXPERIMENT 4 considers the time required to *delete* a random explicit triple of a specific type, and all the implied that were produced from this explicit one (through implication).

Figure 6.4 shows the time needed for performing triple deletion for both the normalized and vertical partitioned schemas. The graph shows that the time increases as the number of total quadruples increases. This is an expected result since when the number of total quadruple increase, the size of tables that stores them increase, so the database operations become slower. There are no differences for the triple deletion time of the two schemas, except in the case of triples with predicate P. The deletion time is much faster in this case, since the size of the

table that stores triples of type P is smaller that the one of the normalized schema (for the same reason that described in EXPERIMENT 3). The same observation holds for the real datasets (see Figure 6.5).



(a) Normalized schema

(b) Vertical Partitioned schema

Figure 6.4: EXPERIMENT 4 - Triple Deletion, Synthetic Datasets

Type of triple	Schemas & Info	GADM-RDF	GeoSpecies	CIDOC	GO
1: sc	Normalized	0,629	0,300	0,054	5,030
	Vertical Partitioned	0,089	0,087	0,080	5,231
2: sp	Normalized	-	0,724	0,032	-
	Vertical Partitioned	-	0,766	0,072	-
3: type	Normalized	0,567	0,287	-	4,237
	Vertical Partitioned	0,139	0,087	-	0,084
4: P	Normalized	0,452	0,244	0,029	4,254
	Vertical Partitioned	0,475	0,387	0,076	0,085
5: class	Normalized	0,131	0,069	0,034	-
	Vertical Partitioned	0,083	0,090	0,078	-
6: prop	Normalized	0,136	0,062	0,035	-
	Vertical Partitioned	0,084	0,077	0,073	-

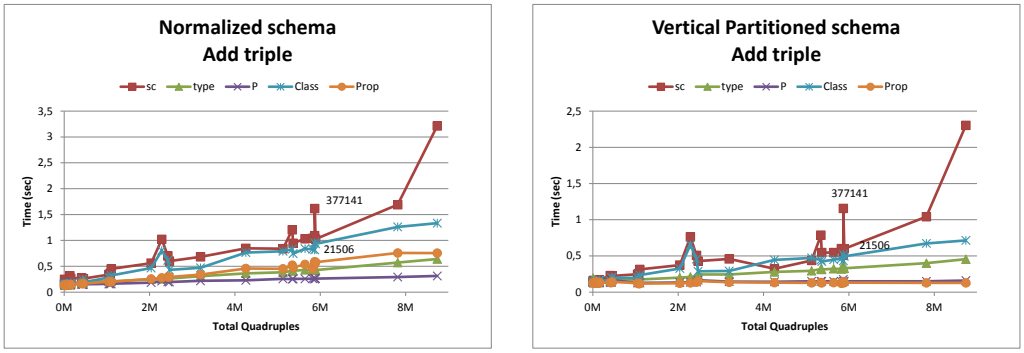
Table 6.5: EXPERIMENT 4 - Triple Deletion, Real Datasets (in seconds)

For the real datasets shown in Table 6.5 the benefit of the vertical partitioned schema over the normalized one, depends on the type of the triple to be deleted. For example, consider the GO ontology and a sc triple to be deleted. Recall that the GO ontology contains of a very large number of sc properties. We can see that the vertical partitioned schema does not provide any benefits as the size of the table that stores the sc property is the same as the the size of the table that stores

the implicit quadruples.

Experiment 5

Experiment 5 measures the time required to *add* an explicit triple of specific type and to recompute the new implicit ones with their abstract labels for synthetic datasets.



(a) Normalized schema

(b) Vertical Partitioned schema

Figure 6.5: Experiment 5 - Triple Addition, Synthetic Datasets

As an be seen in Figure 6.5, the time required for triple addition increases as the number of total quadruples increase for the same reason that described in Experiment 4. In addition, can be shown that the time also depends to a lesser degree on the number of total quadruples that a triple addition will cause.

For example, consider two of the synthetic datasets with 6M total quadruples. We can see see that the addition of a sc triple differs between the two datasets. More specifically, for the first dataset, the addition of a triple causes the addition of 21.506 total quadruples and the time required is 1 sec. For the second dataset, the triple addition causes addition of 377.141 total quadruples, time is about 1,6 seconds. From our experimental results we can see that the time needed for triple addition depends on the type of the triple to be added. As shown in the graphs, there are type of triples, as the sc type that require more time to be added, since their addition will fire more rules, and other ones, like P which needed less time, as their addition fires only two rules.

Finally, comparing the normalized with the vertical partitioned schema, we can see that the former one outperforms the later, as the database operations needed to add the implied triples operate on smaller tables.

The same observations as for Experiment 4 hold for real datasets (see Table 6.6) in this case.

Type of triple	Schemas & Info	GADM-RDF	GeoSpecies	CIDOC	GO
1: sc	Normalized	6,024	1,264	0,210	4,150
	Vertical Partitioned	0,337	0,173	0,162	3,246
2: sp	Normalized	-	2,497	0,146	-
	Vertical Partitioned	-	1,359	0,147	-
3: type	Normalized	1,776	0,535	-	1,215
	Vertical Partitioned	0,181	0,152	-	1,012
4: P	Normalized	1,774	0,478	0,129	0,164
	Vertical Partitioned	0,344	0,183	0,141	0,139
5: class	Normalized	1,643	0,418	0,136	-
	Vertical Partitioned	0,432	0,201	0,148	-
6: prop	Normalized	2,919	0,741	0,131	-
	Vertical Partitioned	1,772	0,633	0,133	-

Table 6.6: EXPERIMENT 5 - Triple Addition, Real datasets (in seconds)

Experiment 6

EXPERIMENT 6 compares the *annotation time* and the *average* time required for the *deletion* or *addition* of an *explicit* triple for the vertical partitioned and normalized schemas and for synthetic datasets. Figure 6.6 shows that the the time needed for adding or deleting a triple with our model is negligible when compared with the time required to annotate the whole graph. The benefit becomes more apparent as the size of the dataset grows.

The same observations can also be made for real datasets by taking into consideration the results presented in Table 6.7.

Dataset	Average Deletion time	Average Addition time	Annotation time
GADM-RDF	0,145	0,511	1,471
GeoSpecies	0,238	0,450	4,015
CIDOC	0,063	0,122	0,162
GO	1,115	0,733	57,063

Table 6.7: EXPERIMENT 6 - Annotation vs Average Triple Addition/Deletion Time, Real Datasets

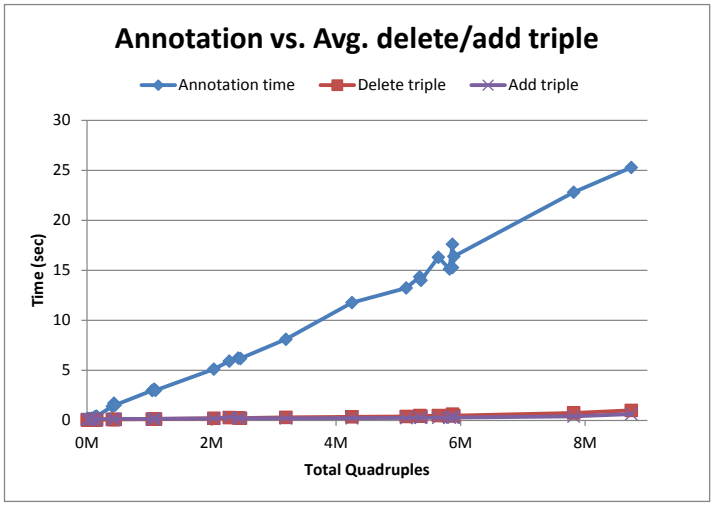


Figure 6.6: EXPERIMENT 6 - Annotation vs Average Triple Deletion/Addition Time, Synthetic Datasets

Experiment 7

Finally, EXPERIMENT 7 compares the *annotation time* and the average time required for the *deletion* or *addition* of an *authorization*. Figure 6.7 shows that the time required for both operations increase linearly as the total number of quadruples increases. We can see the benefits of our approach when comparing the time required for re-annotating the whole graph with the time required to modify only the necessary triples in the graph in the case of an authorization insertion or deletion.

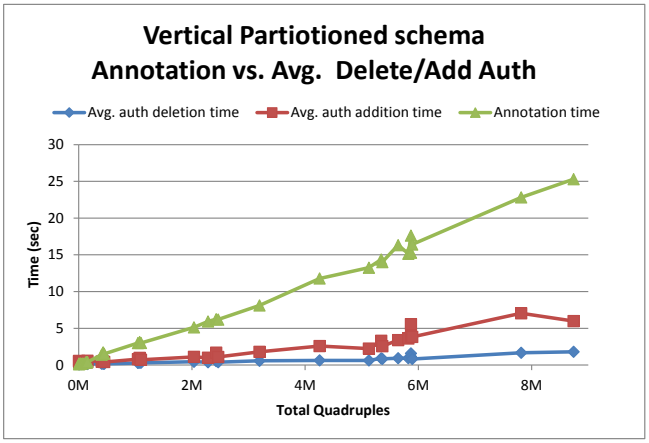


Figure 6.7: EXPERIMENT 7 - Annotation vs Average Authorization Addition/Deletion Time, Synthetic Datasets

Chapter 7

Related Work

In general, there are three approaches to access control enforcement: (a) *materialization*, (b) *query rewriting* and (c) *static analysis*. In the case of materialization, the inferred triples and their labels are *pre-computed* and the user queries are directly evaluated on this dataset. This is the case of *annotation models* that associate specific concrete access labels to each triple. The drawback of this approach is that it does not scale well in the case of updates. The work that we propose in this thesis tackles exactly this problem: the implied triples are annotated with *abstract* expressions that denote the triples that contributed to the implication of the implied triple. Hence updates can be handled easily, by eliminating the need to recompute the closure of the dataset, and changing only the labels of the triples that are affected by the update.

The idea behind *query rewriting* is to change the query by *injecting* the authorization queries in the user query. For instance, consider a user query requests *"return the names of all persons"* and the authorization that specifies that only the names of adults (expressed by a condition of the form *"age >= 18"*) are accessible. The user query that will be evaluated by the engine will be *"return the names of all persons, with age >= 18"*, that is the condition specified in the authorization will be included in the user query.

Last, in *static analysis* approaches, we do not check the data, but only whether the user query is *subsumed* by the authorizations. Towards this end, SPARQL query *containment* approaches should be considered. Static analysis is not always successful, especially when complex filters on data are involved.

Despite the importance of the problem, there have been only a few works dealing with the problem of access control for RDF data and schema graphs, and most of them do not adequately consider the RDFS semantics. Moreover, to the best of our knowledge, none of the existing works consider abstract access control models; instead, they use an access control policy with fixed semantics, which implies that

all access labels have to be recomputed following any change in the authorizations, in the dataset, or in the access control policy itself.

Jain et al. [9] use the notions of RDF security object (that is equivalent to our notion of quadruple) and access control authorizations (which is similar to our respective authorizations, but with limited expressivity, as they use simple RDF triple patterns rather than SPARQL queries). RDFS inference is supported but the approach does not consider propagation rules. Given a set of security objects, the authors produce a *security cover* which essentially amounts to resolving conflicts caused by ambiguous security labels. The computation of the labels of the implied triples is hard-coded (takes the least upper bound of the labels of its implying triples). In our work, we model the label of an implied triple by means of the *abstract inference operator*.

The specification of the semantics of the operator are left to be specified by the concrete policy. The conflict resolution strategy is also hard-coded and always resolves the conflict in favor of the security label which is the least upper bound of the involved ones. This comes in contrast with our approach where both strategies are defined by customizable concrete policies.

In Kim et al. [10], access control is defined at the level of nodes in the RDF graph (URIs and literals), rather than at the level of RDF triples as in our work and in [9]. The authors focus on how to resolve conflicts that arise due to the implicit and explicit propagation of RDF authorizations for the RDFS class and property hierarchies and propose an efficient algorithm to identify conflicts and resolve them according to specific (fixed) semantics.

The work that we present in this thesis and that has been published in [38] generalizes in a straightforward manner the access control models that have been proposed in the literature. This is done by the novel concept of *concrete policies* that encode the different access annotations and semantics for computing the labels of the implied triples.

Buneman et al. [39] discuss RDFS inference in the spirit of [9]. In this work, the access labels of triples are *logical expressions* involving *conjunction* (for RDFS inference) and *disjunction* (for conflict resolution). This approach is closer to ours since the access labels are logical expressions and not concrete values. Nevertheless, the expressions are again formulated with concrete, rather than abstract operators.

Our work can also be compared to works related to *annotated RDFS* [40, 41], where algebraic models are used. Authors in [40] annotate triples with algebraic terms following the line of research done in the context of provenance for relational databases [13] and develop an algebra of annotations for RDFS. They show that the proposed annotation algebra can be used for computing annotations on inferred triples that provide information on belief, trust and temporal aspects of data as well as for providing a framework for default reasoning. Work in [40] generalizes the work in [41], whose goal is to apply fuzzy logic to RDFS via an abstract model

similar to [13]. In our work we also model the propagation of labels along the RDFS class and property hierarchies, and propose a unified framework for modeling access control for RDF graphs. Flouris et al. in [6] propose a commutative semiring to model the provenance of implied RDF triples. The approach advocated by the authors in that work is similar to ours as far as modeling the labels of implied triples. In [16] authors showed that Named Graphs [15] alone cannot model the labels of implied quadruples, but a higher level constructs are needed.

Chapter 8

Conclusions and Future Work

As more and more (potentially sensitive) data are being made available through the Web (e.g., as part of the Linked Open Data initiative [2]), the need for controlling access to such data becomes all the more important. This work addresses this problem for RDF graphs, by providing an open and customizable framework for defining access control policies which takes into account RDFS inference rules and propagation of access labels along the RDFS class and property hierarchies.

The main contribution, and the distinguishing feature of this work compared to existing frameworks for RDF access control, is the use of *abstract access control models*, which allows for more effcient handling of changes in the dataset or in the access control authorizations associated with the dataset. The model uses *abstract tokens* and *operators*, the latter encode *inference* and *propagation* of labels along the RDFS *subclassOf* and *subpropertyOf* relations. The proposed model provides full support for RDFS inference rules (unlike existing frameworks that provide only partial or inadequate support) and propagation rules which are missing in most existing access control frameworks.

In our work, abstract tokens are used by authorizations to tag triples, and abstract operators express how the access label of a triple was constructed, e.g., through the application of inference or propagation rules. To compute the value of a label (which is a complex expression), each application provides its own definition of a *concrete policy* (concretization for said tokens and operators) and semantics. The main advantage of our approach is its *flexibility*, caused by the fact that the access control label contains the information on *how* each access control label was produced, thereby supporting the efficient discovery of the labels that are affected by a change in the dataset or the access control policy.

We implemented our ideas on top of the MonetDB column store DBMS and we conducted a set of experiments with real and synthetic datasets. For our implementation we used the state of the art storage schemes for storing RDF data:

schema agnostic and *vertical partitioning* modified to encode *access control information*. Our experimental evaluation showed the benefits of an abstract access control model in the case of updates for both storage schemes.

In the future we plan to work with different schemas for storing more efficiently the complex access control labels, and experiment with engines that (a) provide indexing schemes (not the case with MonetDB) and (b) efficient implementations for computing the transitive closure of a graph.

In addition, we plan to consider in our framework multiple roles and purposes, thereby working towards a *privacy-aware access control framework*. More specifically, we intend to support hierarchies of roles and purposes and study how we can integrate both in our work. We envision a solution based on extending the novel concept of *concrete policies* that we have proposed in our work by considering the dimension of *purpose*. In this context, we plan to experiment with more complex and realistic concrete policies and real world applications (health care, telecommunications etc.).

Bibliography

[1] B. M. F. Manola, E. Miller, "RDF Primer," www.w3.org/TR/rdf-primer, February 2004.

[2] "W3C Linking Open Data," esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData.

[3] D. Brickley and R. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," www.w3.org/TR/2004/REC-rdf-schema-20040210, 2004.

[4] P. Hayes, "RDF Semantics," www.w3.org/TR/rdf-mt, February 2004.

[5] "Gene Ontology," www.geneontology.org.

[6] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides, "Coloring RDF Triples to Capture Provenance," in *ISWC*, 2009.

[7] F. Abel, J. L. D. Coi, N. Henze, A. W. Koesling, D. Krause, and D. Olmedilla, "Enabling Advanced and Context-Dependent Access Control in RDF Stores," in *ISWC/ASWC*, 2007.

[8] S. Dietzold and S. Auer, "Access Control on RDF Triple Store from a Semantic Wiki Perspective," in *ESWC Workshop on Scripting for the Semantic Web*, 2006.

[9] A. Jain and C. Farkas, "Secure Resource Description Framework," in *SACMAT*, 2006.

[10] J. Kim, K. Jung, and S. Park, "An Introduction to Authorization Conflict Problem in RDF Access Control," in *KES*, 2008.

[11] 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.

[12] M. Knechtel and R. Peñaloza, "A Generic Approach for Correcting Access Restrictions to a Consequence," in *ESWC*, 2010.

[13] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *PODS*, 2007.

[14] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," www.w3.org/TR/rdf-sparql-query, January 2008.

[15] J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *WWW*, 2005.

[16] P. Pediaditis, G. Flouris, I. Fundulaki, and V. Christophides, "On Explicit Provenance Management in RDF/S Graphs," in *TaPP*, 2009.

[17] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides, "On Graph Features of Semantic Web Schemas," *TKDE*, vol. 20, no. 5, 2008.

[18] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen, "Containment and minimization of rdf/s query patterns," in *ISWC*, 2005.

[19] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM TODS*, vol. 34, no. 3, 2009.

[20] Y. Theoharis, V. Christophides, and G. Karvounarakis, "Benchmarking Database Representations of RDF/S Stores," in *ISWC*, 2005, pp. 685–701.

[21] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[22] ——, "The RDF-3X engine for scalable management of RDF data," *VLDB Journal*, August 2009.

[23] ——, "Scalable join processing on very large rdf graphs," in *SIGMOD Conference*, June 2009, pp. 627–640.

[24] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold, "Column-store support for RDF data management: not all swans are white," *PVLDB*, vol. 1, no. 2, 2008.

[25] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An efficient SQL-based RDF querying scheme," in *VLDB*, 2005, pp. 1216–1227.

[26] L. Baolin and H. Bo, "HPRD: A High Performance RDF Database," in *Network and Parallel Computing*, 2007, pp. 364–374.

[27] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[28] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[29] S. M. D. J. Abadi, A. Marcus and K. Hollenbach, "Sw-store: a vertically partitioned dbms for semantic web data management," *VLDB Journal*, vol. 18, no. 2, pp. 385–406, 2009.

[30] M. G. W. Fan, C.-Y. Chan, "Secure XML Querying with Security Views," in *SIGMOD*, 2004.

[31] G. Dong, L. Libkin, J. Su, and L. Wong, "Maintaining transitive closure of graphs in SQL," *Int. Journal of Information Technology*. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.53

[32] "GADM-RDF," http://gadm.geovocab.org/.

[33] "GeoSpecies Knowledge Base," http://lod.geospecies.org/.

[34] R. Isele, J. Umbrich, C. Bizer, and A. Harth, "LDSpider: An open-source crawling framework for the web of linked data," in *Proceedings of 9th International Semantic Web Conference (ISWC 2010) Posters and Demos*, 2010. [Online]. Available: http://iswc2010.semanticweb.org/pdf/495.pdf

[35] "CIDOC Conceptual Reference Model (CRM)," http://www.cidoc-crm.org/, 2006, iSO 21127:2006.

[36] "The Gene Ontology (GO)," http://www.geneontology.org/.

[37] Y. Theoharis, G. Georgakopoulos, and V. Christophides, "PoweRGen: A Power-Law Based Generator of RDFS Schemas," *Information Systems. Elsevier*, 2011.

[38] I. F. V. Papakonstantinou, M Michou G. Flouris and G. Antoniou, "Access control for RDF graphs using abstract models." in *SACMAT*, 2012.

[39] J. Lu, J. Wang, Y. Zhang, B. Zhou, Y. Li, and Z. Miao, "An Inference Control Algorithm for RDF(S) Repository," in *PAISI*, 2007.

[40] P. Buneman and E. V. Kostylev, "Annotation algebras for RDFS," in *SWPM*, 2010.

[41] U. Straccia, N. Lopes, G. Lukacsy, and A. Polleres, "A General Framework for Representing and Reasoning with Annotated Semantic Web Data," in *AAAI*, 2010.