Access Control for RDF Graphs using Abstract Models

Vassilis Papakonstantinou
FORTH-ICS and University of Crete
papv@ics.forth.gr

Maria Michou
maria.michou@gmail.com

Irini Fundulaki
FORTH-ICS
fundul@ics.forth.gr

Giorgos Flouris
FORTH-ICS
fgeo@ics.forth.gr

Grigoris Antoniou
FORTH-ICS and University of Huddersfield
antoniou@ics.forth.gr

ABSTRACT

The Resource Description Framework (RDF) has become the defacto standard for representing information in the Semantic Web. Given the increasing amount of sensitive RDF data available on the Web, it becomes increasingly critical to guarantee secure access to this content. In this paper we advocate the use of an *abstract access control model* to ensure the selective exposure of RDF information. The model is defined by a set of *abstract operators* and *tokens*. Tokens are used to label RDF triples with access information. Abstract operators model RDF Schema inference rules and propagation of labels along the RDF Schema (RDFS) class and property hierarchies. In this way, the access label of a triple is a complex expression that involves the labels of the triples and the operators applied to obtain said label. Different applications can then adopt different *concrete access policies* that encode an assignment of the abstract tokens and operators to concrete (specific) values. Following this approach, changes in the interpretation of abstract tokens and operators can be easily implemented resulting in a very flexible mechanism that allows one to easily experiment with different concrete access policies (defined per context or user). To demonstrate the feasibility of the approach, we implemented our ideas on top of the MonetDB and PostgreSQL open source database systems. We conducted an initial set of experiments which showed that the overhead for using abstract expressions is roughly linear to the number of triples considered; performance is also affected by the characteristics of the dataset, such as the size and depth of class and property hierarchies as well as the considered concrete policy.

Categories and Subject Descriptors

H.1 [**Information Systems Applications**]; H.2.8 [**Database Applications**]

SACMAT'12 June 20-22, 2012, New Jersey, Newark, USA

General Terms

Access Control Models

Keywords

Access Control, Abstract Model, RDF, Inference

1. INTRODUCTION

RDF [8] 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 [19], target the development of RDF datasets. The popularity of the RDF data model [8] 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*. A set of triples form an *RDF graph* that can be represented by a node and edge labeled directed graph. Nodes of the graph represent the *subject* and *object* components of triples (*resources*) connected by edges that model the *predicates* of triples. RDFS is used to add semantics to RDF triples, through the *inference rules* specified in the RDF Schema language [11] that entail new *implicit* triples.

The number of applications that publish and exchange possibly sensitive RDF data continuously increases in a large number of domains. 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 this paper we focus 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 a possible scenario is to consider that the label of an implied triple depends on the labels of its implying triples [9]. Propagated labels are useful when one wants to enforce the inheritance of labels along the RDFS class and property hierarchies (e.g., an application may consider that an instance should inherit the label of its class). This is a common approach in XML access control where the labels are inherited from a node to its children nodes in an XML tree.

The majority of the state of the art approaches for RDF access control [2, 7, 12, 13, 20] 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 in a fixed manner according to predefined semantics. 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 its implying triples are labeled as "public". Consequently, when the initial assignment of the access labels to triples 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 *(i)* a specific assignment of values to access labels of triples and *(ii)* predefined semantics for computing the access labels of the triples obtained through inference and propagation. Instead, we use an *abstract access control model* defined by *abstract tokens and operators*. In this way, the access label of a triple is an *abstract expression*, consisting of tokens and operators. This expression describes *how the access label of said triple is computed*. This is inspired by the approach advocated in *how provenance models* [10] for relational data provenance.

To determine the actual label of a triple associated with an abstract expression (i.e., in order to decide whether a triple is accessible or not), one should concretize the respective 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 discussed below:

• When abstract models are considered, the abstract labels are computed only once at *annotation* time. 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 [14]. 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 that can lead to important gains especially when large datasets are considered. The present paper only lays the foundations for our abstract access control model and does not consider the problem of dynamically adapting the abstract labels in response to an update.

• 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.

The main contributions of our work are:

1. 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. 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 their interconnections and relationships as expressed in triples. Thus, triples are the "first-class citizens" that comprise an RDF graph; triples can describe 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).

2. The *extension* of the standard *RDFS inference rules* 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.

3. A first implementation of our ideas on top of the MonetDB and PostgreSQL open source RDBMSs and a preliminary set of experiments that show the computational properties of our approach.

The rest of this paper is structured as follows: in Section 2, we provide a motivating example that is used in the remainder of the paper. In Section 3, we describe the RDF, RDFS and SPARQL languages. In Section 4 we define the abstract access control model. Section 5 discusses how concrete policies can be defined. In Section 6 we discuss implementation issues and the conducted experiments. In Section 7 we describe related work, and conclude in Section 8.

2. MOTIVATING EXAMPLE

Figure 1 shows (in tabular form) a set of RDF triples (inspired by the FOAF ontology[1]) that we will use in this paper 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*. In this paper we rely on authorizations that make use of the SPARQL [18] language to determine the RDF triples concerned by it.

Figure 2 shows a set of access authorizations defined for the set of RDF triples of Figure 1. In our work, we use *quadruples* to encode the access label of an RDF triple. We write (s, p, o, l) to denote that l is the access label assigned to triple (s, p, o). Quadruples are used to represent information such as time, trust, provenance among others. The approach we follow is similar to the named graphs concept discussed in [5].

[1] 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

Figure 1: RDF Triples

Figure 3 shows the RDF quadruples obtained by evaluating the authorizations of Figure 2 to the set of triples in Figure 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 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).

$\mathcal{A}_1$:	(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*$\}$, $at_5)$

Figure 2: Access Control Authorizations

	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

Figure 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 [11] 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 Figure 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. We model the composite label of such an implied quadruple using the *abstract inference 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 Figure 4).

By applying the RDFS inference rules on the type and sc hierarchies in all possible ways, we obtain several quadruples, some of which are shown in Figure 4. Implicit quadruples may also be involved in inferences, resulting in more complex expressions. For example q_{11} is obtained from quadruples q_{10} and q_2. Quadruple q_{12} results 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*	$at_5 \odot at_2$
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*	$at_3 \odot (at_5 \odot at_2)$

Figure 4: Implied Quadruples (Partial List)

The RDFS semantics associated with the class and property hierarchies cause several authors to consider the *propagation* of labels along such hierarchies [13]. 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 $q_{13} = (\&a, \text{type}, \textit{Agent}, \otimes at_4)$, which, in our example is the only propagated quadruple.

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 and how the default access token should

be treated by the concrete operators. 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* ($\odot$ is mapped to conjunction), and that the labels are propagated as such ($\otimes$ 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_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 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).

3. PRELIMINARIES

3.1 RDF and RDF Schema

An RDF triple [8] is of the form (*subject*, *predicate*, *object*) and asserts the fact that *subject* is associated with *object* through *property*. We assume two disjoint and infinite sets $\mathbb{U}$, $\mathbb{L}$, denoting the URIs and literals respectively. Then set $\mathcal{T} = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ is the set of all RDF triples.

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)[2] can be used to specify *class* and *property* types. These can be used as the object of an RDF triple. 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.

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 data used in real applications [24], and is a common assumption made for efficiency (e.g., query optimization [21]) in many RDF applications.

RDFS defines a set of *inference rules* [11] depicted in Table 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.

3.2 SPARQL

SPARQL [18] 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 of the RDF graph that matches this pattern.

[2]In parenthesis are the terms we use in this paper 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 1: RDFS Inference Rules

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 a specific form. 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 specifies conditions on the triple patterns.

The SPARQL syntax follows the SQL select-from-where paradigm. The SPARQL queries we consider 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. We choose this form of queries since in our framework we assign access labels to *triples* and not to nodes in the RDF graph. The graph pattern specified in the CONSTRUCT clause should also be included in the query's WHERE clause. For readability purposes only we have not included it in our authorization queries.

The interested reader can find a more detailed description of the semantics of the SPARQL language in [17]. 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 Table 1) or evaluate it on the fly. In our 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.

4. 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 *authorization 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 tokens or with a complex expression that involves the 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* of the triple and l is an *abstract access control expression*. Now we are ready to define the notion of *access control model*.

DEFINITION 4.1. *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*
- *$\bot$ is the default access token that is assigned to triples that are not in the scope of some authorization rule*
- *$\odot$ is the binary inference operator*

- $\otimes$ *is the unary propagation operator*

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

4.1 Inference Operator

Implied triples are obtained from the application of RDFS inference rules [11] on an RDF graph. The inference rules shown in Table 2 extend those specified in Table 1 in a straightforward manner to take into account access labels. An implied triple is annotated by a complex expression that involves the labels of its implying triples associated through the binary *inference operator* $\odot$.

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

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

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

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

Table 2: 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 4.2. *The* abstract inference operator*, denoted by $\odot$, is a binary operator defined over abstract tokens from $\mathcal{L} \cup \{\bot\}$ with the following properties:*

$$al_1 \odot al_2 = al_2 \odot al_1 \quad \textit{(Commutativity)}$$
$$(al_1 \odot al_2) \odot al_3 = al_1 \odot (al_2 \odot al_3) \quad \textit{(Associativity)}$$

EXAMPLE 4.1. *Consider rule $\mathcal{QR}_3$ from Table 2. 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 3, we obtain quadruple $q_{10} = (\&a, \mathsf{type}, Person, (at_3 \odot at_2))$ shown in Figure 4.*

4.2 Propagation Operator

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 [25]. 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 to *existing* triples new labels hence producing *new quadruples*. In this work we focus on *"downward"* propagation rules along the sc, sp and type hierarchies: 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.

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

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

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

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

Table 3: Propagation Rules

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 shows the propagation rules that we consider in our work. Note that this set of rules can change and adapt to the application needs, or be omitted altogether.

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: $(t, \otimes al)$, $(t, \otimes \otimes al)$, ..., $(t, \otimes \ldots \otimes al)$, ... We will hence obtain an infinite number of quadruples for the same triple.

DEFINITION 4.3. *The* abstract propagation operator*, denoted by $\otimes$, is a unary operator defined over labels, with the property:*

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

EXAMPLE 4.2. *$\mathcal{QR}_6$ shown in Table 3 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 4 and $q_6 = (Agent,$ type,class, $at_4)$ from Figure 3 we obtain quadruple $q_{13} = (\&a,$ type, $Agent,$ $\otimes at_4)$.*

4.3 Computing Abstract Expressions

Now consider an RDF graph $\mathcal{G}$ (i.e., set of triples) and a set of *authorization rules* that assign abstract access tokens to triples by means of SPARQL queries. In order to obtain the set of quadruples *(i)* we first evaluate the authorization rules on the triples in the input RDF graph; after this step, the triples that did not receive any label are annotated with the default access token ($\bot$); *(ii)* we then apply the *inference rules* on the resulting quadruples to obtain the *implied* ones along with their labels. Inference rules are applied until no new quadruples are produced; *(iii)* last the *propagation rules* are applied on the resulting quadruples to get the quadruples with propagated labels.

At the end of this process, the access label of any quadruple is of the form:

$$proplabel := \otimes aclabel \mid at_i$$
$$aclabel := at_i \mid aclabel \odot aclabel$$

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

5. CONCRETE POLICIES

As discussed in Section 2, 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 is achieved through a *concrete policy* $\mathcal{P}$ that specifies *i)* the *concrete tokens*, *ii*) a mapping between the concrete and abstract tokens *iii)* the *concrete operators iv)* the *conflict resolution operator* and *v)* the *access function*.

A concrete policy is bound to a *fixed* set of *abstract tokens* from $\mathcal{L}$. The set of *concrete tokens* is denoted by $\mathcal{L}_\mathcal{P}$; the *mapping* specifies how the abstract tokens from $\mathcal{L}$ are mapped to the concrete tokens in $\mathcal{L}_\mathcal{P}$. Note that $\perp$ is a special token that does not belong in $\mathcal{L}$.

The *concrete operators* implement the abstract inference ($\odot$) and propagation ($\otimes$) operators and handle the default access token. The proposed implementations must respect the properties defined for the their abstract counterparts. The concrete operators are defined over the set $\mathcal{L}_\mathcal{P} \cup \{\perp\}$ (i.e., they 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. The *conflict resolution operator*, denoted by $\oplus$, is used to select the final concrete token of said triple. If $\perp$ is included in the set of tokens assigned to the triple, then, by definition, it should be ignored in the presence of other concrete tokens from $\mathcal{L}_\mathcal{P}$. Formally, $\oplus$ is defined as a selection function over subsets of $\mathcal{L}_\mathcal{P} \cup \{\perp\}$, such that, for any given $\mathcal{X} \in 2^{\mathcal{L}_\mathcal{P} \cup \{\perp\}}$ it holds that:

- $\oplus\mathcal{X} \in \mathcal{X}$ if and only if $\mathcal{X} \neq \emptyset$.
- $\oplus\mathcal{X} = \perp$ if and only if $\mathcal{X} = \emptyset$ or $\mathcal{X} = \{\perp\}$.

The *access function*, denoted by *access*() is the final component of a concrete policy and determines whether a triple associated with a certain token 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 as well as $\perp$, to *allow* or *deny*, indicating that the corresponding triple is accessible or not 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 $\mathcal{L}_\mathcal{P}$ (any mapping would do – details are omitted). The abstract operator $\odot$ is mapped to the following operator:

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

where $\wedge$ is the standard boolean conjunction. The propagation operator is simply defined as $\otimes al = al$. The conflict resolution operator resolves conflicts in favor of *false* values:

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

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 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 mapping, as before, is irrelevant (any mapping from $\mathcal{L}$ to $\mathcal{L}_\mathcal{P}$ would do). The operator $\odot$ is mapped to addition, ignoring $\perp$, i.e., $n \odot m = n + m$, $n \odot \perp = n$. The propagation operator is the identity: $\otimes al = al$. The conflict resolution operator selects the most restrictive (maximum) token, ignoring $\perp$: $\oplus\mathcal{X} = max(\mathcal{X} \setminus \{\perp\})$ when $\mathcal{X} \setminus \{\perp\} \neq \emptyset$, and $\oplus\mathcal{X} = \perp$ otherwise. The access function specifies that all quadruples whose confidentiality token is lower than a certain number (say *2*) are accessible (i.e., $access(n) = allow$ iff $n \leq 2$), and inaccessible otherwise (thus, e.g., $access(\perp) = deny$).

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.

6. IMPLEMENTATION AND EVALUATION

6.1 Storage (Relational Schema)

The quadruples and the related access labels are stored in a relational schema, which uses three tables: $Map(value, id)$, $Quad(qid, s, p, o, prop, iop, label)$ and the auxiliary table $LabelStore(qid, qid\_uses)$. Table $Map(value, id)$ is used to map URIs and literals (*value* column) to unique identifiers (*id* column), in order to avoid an overhead during query evaluation.

Table $Quad(qid, s, p, o, prop, iop, label)$ stores the quadruples. More specifically, column *qid* stores the quadruple's unique identifier; columns s, p, o store the *id* from table *Map* to which the URI/literal of the subject (s), predicate (p) and object (o) of the corresponding triple are mapped to. Columns *prop* and *iop* take boolean values and are used to indicate whether $\otimes$ or $\odot$ respectively are used in the computation of the abstract label of the quad; *label* stores the access token of the quadruple (in case it is explicit), or is set to *null* to indicate that the label is a complex expression. In this case, the auxiliary table $LabelStore(qid, qid\_uses)$ is used to store the access label (discussed in detail below). In this paper we reported our experiences with a non-normalized schema where all quadruples are stored in a single table. We are studying a normalized version of the proposed schema that considers three tables: one for the explicit, one for the implied and one for the propagated quadruples. This schema takes into consideration the foreign and primary key constraints in the schema and preliminary experiments have shown that we can achieve better performance results since the underlying query optimizer takes into account the above constraints and optimizes the produced plans.

To understand the intuition behind the representation of labels, recall that a label can be obtained as follows:

1. directly through an authorization, in which case it is an abstract token;

2. indirectly, when the associated triple is not in the scope of any authorization, in which case it is the default access token $\perp$;

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 said quadruple;

4. through a propagation rule, in which case it is $\otimes al$ (al is the propagated label);

In Case (1) we set *prop* and *iop* to *false*, and store the access token in *label*. Case (2) is similar to the previous one except from the fact that *label* is set to $\perp$. In Case (3) the label of the quadruple is a complex expression, so we set $label = null$, and the *iop* flag to the value *true* (to indicate that $\odot$ is used in the expression). The labels involved in the inference (al_1,al_2) are stored using the table *LabelStore*: the first column (*qid*) keeps the *qid* of the quadruple whose complex label we want to store. *qid_uses* stores the *qid* of the quadruple that contributed to the label of the implied quadruple. More specifically, one tuple is added in *LabelStore* for each of the quadruples that imply said quadruple.

In *LabelStore* we store the *explicit* quadruples that contribute to the implication of the implicit ones. For example, consider q_8 in Figure 4; q_8 is an inferred quadruple, occurring from the application of the inference rule $\mathcal{QR}_4$ upon q_1, q_2. We store the fact 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 because it allows finding in constant time all the explicit quadruples that are involved in an implicit one. We plan to exploit this feature in the future in order to improve performance when authorizations change (i.e., one or more authorizations are added, deleted or modified). In our example *LabelStore* would contain the tuples $(8, 1), (8, 2)$ where x is the identifier for quadruple q_x. Consider now quadruple q_{11} from Figure 4, which results from the application of inference rule $\mathcal{QR}_3$ upon q_3, q_8. Recall that in *LabelStore* we store the identifiers of the *explicit quadruples* used to obtain the implied quadruples. In this manner *LabelStore* will contain tuples $(11, 1), (11, 2)$ (for q_8) and $11, 3$ (for q_3). If we had followed a different approach in which we would record the identifiers of the contributing *implied* quadruples, then in order to retrieve all the labels that contributed to the label of said quadruple, additional joins with the *Quad* and *LabelStore* tables would be necessary. The approach we follow resolves this problem by eliminating this "nesting" of complex labels. The "unnesting" of labels occurs during the application of inference and propagation rules and helps avoiding recursive joins that would slow down the process of label (and query) evaluation.

Case (4) is handled in a manner similar to Case (3). Again, we store the operation used ($prop = true$, $iop = false$) and the fact that the label of the quad is complex (by setting $label = null$). *LabelStore* stores the quadruple whose label was propagated (as above).

6.2 Datasets and Access Policies

We used both real and synthetic datasets in our experiments. The real datasets used were the CIDOC [6] and the GO [1] ontologies. The CIDOC Conceptual Reference Model (CRM) provides definitions and a formal structure for describing the concepts and relations used in cultural heritage documentation. CIDOC contains 3282 triples that define 82 classes, 262 properties, 94 sc, 130 sp relations and 342 type relations. The ontology does not include any class or property instances. After the application of the inference and propagation rules, we obtain in total 3733 implicit and explicit quadruples. 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 265355 explicit triples that define 35451 classes and only 5 properties. Similarly to CIDOC, GO does not have any class or property instances; it defines 55169 sc, no sp relations and 35451 type relations.

To produce the synthetic schemas for our experiments we used Powergen [23], which is the first synthetic RDFS schema generator that takes into account the morphological features that schemas frequently exhibit in reality [24] 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 152 synthetic ontologies containing 100-1000 classes, 113-1635 properties, 124-50295 class instances and 110-1321 property instances. We also experimented with different depth for the sc and sp hierarchies of the input RDF schema (ranging from depth 4 to 8).

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 multiple access tokens to the same triple, as well as the assignment of the same token to multiple triples.

6.3 Experiments

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 2.6.35-31-generic x86_64. We used PostgreSQL 8.4.9 (a row-store) and MonetDB v11.3.3-Apr2011-SP1 (a column-store) as our RDBMS backends. MonetDB does not support recursive SQL and hence the implementation of entailment and propagation rules shown in Tables 2 and 3 is done with the use of stored procedures. To avoid issues with performance varying between different runs, we performed only cold-cache experiments.

For our work we conducted three experiments. Experiment 1 measures the time required to compute all the inferred and propagated quadruples (*annotation time*). Experiment 2 and Experiment 3 respectively consider the time required to compute for a concrete policy and a set of annotated quadruples, the concrete labels from the abstract ones (*evaluation time*). Experiment 2 and Experiment 3 differ in the following aspect: the former measures the time needed to compute the concrete values for all the datasets used in our experiments and for their integrality whereas the latter reports the time needed to compute the concrete values for a fraction of triples in a specific dataset. We set a timeout of 60 minutes, i.e., any experiment that ran for

more than 60 minutes was stopped and is not included in the reported times.

The general observation from our experiments is that the annotation and evaluation times are roughly linear to the number of triples considered: the number of implied triples of the input for EXPERIMENT 1 and total triples for EXPERIMENT 2, and the fraction of the dataset for EXPERIMENT 3. Other factors affecting the time are the ontology's structure and the nature of the concrete policy used (for EXPERIMENTS 2 and 3), and, of course, the RDBMS backend (MonetDB is generally faster). However, some experiments (especially those involving PostgreSQL) gave poor results. This indicates that, despite the encouraging (linear) scaling properties of our approach, there is much room for improvement in terms of efficiency, using more efficient storage schemes for our abstract labels; this effort will be part of our future work. More details on the various experiments and the related results follow.

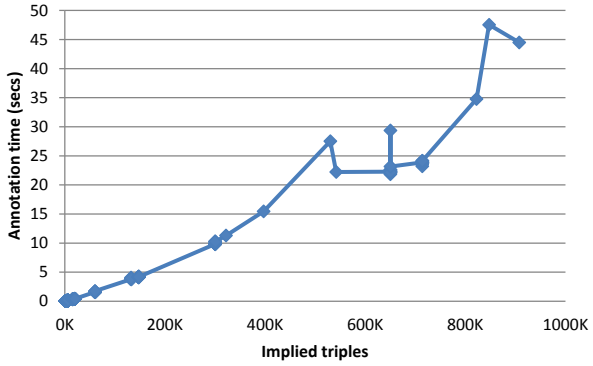

Figure 5a: EXPERIMENT 1 **(MonetDB)**

EXPERIMENT 1 measured the *annotation time*, that is the time needed to apply the inference and propagation rules in Tables 2 and 3 on a set of quadruples. This set was 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, so we omit it from our experiments.

Figures 5a and 5b show the annotation time for the synthetic schemas for MonetDB and PostgreSQL respectively and for the synthetic datasets only. The graphs show that the annotation time increases as the number of implied triples increases. This is an expected result since when the number of implied triples increases, the number of times the inference and propagation rules are applied also increases. The observed plunges are due to changes in the structure of the ontology and more specifically to the reduction of the depth of the class and property hierarchies (a PowerGen parameter). The implication of such a decrease is that even though the number of implicit triples increases, there are fewer recursive applications of the inference rules. With MonetDB we were able to annotate all synthetic schemas that we produced. However, PostgreSQL managed to compute the abstract expressions for schemas with up to 1000 classes, 1635 properties, 50167 class instances and 95 property instances before reaching the timeout.

Regarding the real datasets, in the case of CIDOC ontology, the annotation process required 69ms and 4000ms for MonetDB and PostgreSQL respectively. On the other hand, the GO ontology required approximately 32sec and 844sec for MonetDB and PostgreSQL respectively. This result is due to the increased size of the GO ontology, as well as its large number of sc, sp and type relations that are involved in the inference rules that lead to a large number of inferred quadruples. More specifically, the CIDOC ontology produces only 549 new quadruples (using the inference and propagation rules), whereas GO produces 393699 inferred quadruples (almost twice the size of the original GO RDF graph).

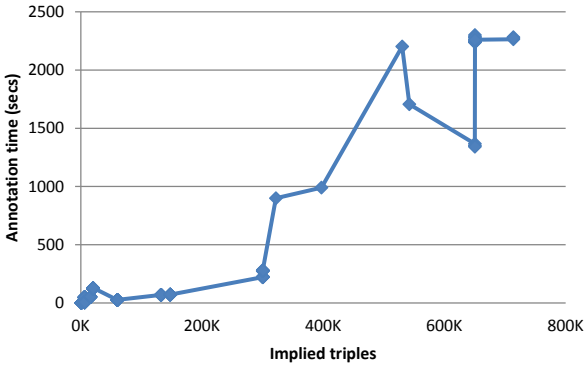

Figure 5b: EXPERIMENT 1 **(PostgreSQL)**

EXPERIMENT 2 measured the time required to compute the concrete labels for all RDF quadruples of each dataset, under concrete policies similar to the ones discussed in Section 5. Figure 6 shows the related evaluation time for the synthetic datasets. We note, as expected, that the evaluation time increases linearly with respect to the total triples in the dataset and that MonetDB outperforms PostgreSQL for all synthetic datasets. Another interesting observation is that concrete policy **C2** is faster for both RDBMSs; this is explained by the fact that **C2** uses arithmetic operations which are implemented as built-in SQL functions, contrary to **C1**, which consists of logical operations that are implemented using custom SQL code.

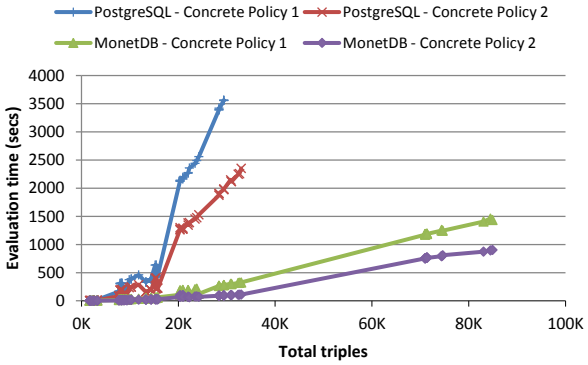

Figure 6: EXPERIMENT 2 **(MonetDB/PostgreSQL)**

The same observations can also be made for CIDOC, for which we need about 7775ms to evaluate the concrete access

values for **C1** and 3923ms for **C2**. Unfortunately, we failed to compute the concrete access values for the quadruples in the case of the GO ontology, as the process exceeded our set timeout.

EXPERIMENT 3 measured the evaluation time for some of the quadruples in the chosen dataset, as a function of the fraction of the dataset considered. This is useful at query time, when the retrieved triples must be filtered to exclude non-accessible triples, and this includes the process of determining the concrete access token for said triples. In our experiments we consider the 10%, 30% and 50% of the total triples in the dataset. This experiment was done for the largest synthetic dataset for which all concrete labels were computed successfully (without a timeout) in EXPERIMENT 2. For MonetDB we used the dataset that contained 100 classes, 113 properties, 10028 class instances and 8706 property instances. The dataset we used for the experiment for PostgreSQL contained 1000 classes, 1635 properties, 1244 class and 704 property instances.

The results are shown in Figures 7a, 7b for MonetDB and PostgreSQL respectively.

Finally, our preliminary experimental results showed that our approach requires up to one order of magnitude more space to store triples with their final abstract label, compared to the standard approach of storing the computed concrete label.

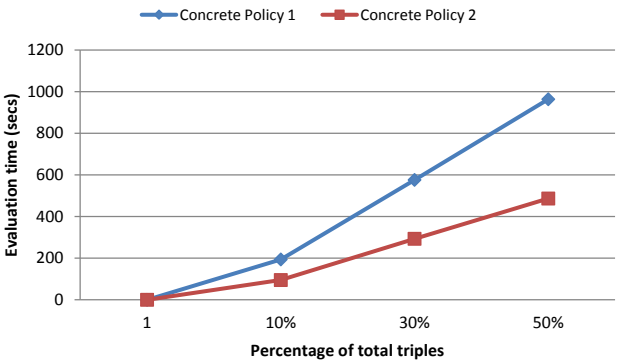

Figure 7a: EXPERIMENT 3 **(MonetDB)**

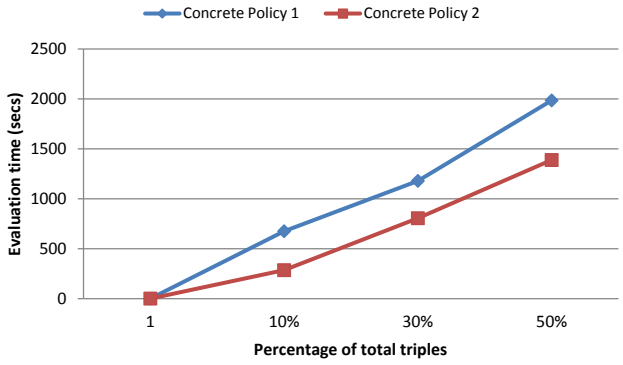

Figure 7b: EXPERIMENT 3 **(PostgreSQL)**

7. RELATED WORK

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 [12] use the notions of RDF security object that is equivalent to our notion of quadruple and access control 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 (the label of the implied triple is the least upper bound of the labels of its implying triples). 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 [13], 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 Jain et. al. [12]. 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.

Authors in [15] discuss RDFS inference in the spirit of [12]. 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* [4, 22], where algebraic models are used. Authors in [4] annotate triples with algebraic terms following the line of research done in the context of provenance for relational databases [10] and develop an algebra of annotations for RDFS. They show that the proposed annotation algebra can be used for computing annotations of inferred triples that store information on belief, trust and temporal aspects of data among others. Work in [4] generalizes the work in [22], whose goal is to apply fuzzy logic to RDFS via an abstract model similar to [10]. 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 [9] propose a commutative semiring to model the provenance of implied RDF triples. The approach we advocate in our work is similar to [9] as far as modeling the labels of implied triples is concerned. Authors in [16] showed that Named Graphs [5] alone cannot model the labels of implied quadruples, but a higher level construct is needed.

8. CONCLUSION

As more and more (potentially sensitive) data are being made available through the Web (e.g., as part of the Linked Open Data initiative [19]), 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*. 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 state of the art access control frameworks.

In our work, abstract tokens are used by authorizations to tag triples, and abstract operators express how the label was constructed, e.g., through the application of inference or propagation rules. To evaluate the label, each application provides its own *concrete policy* (concretization for said tokens and operators) and semantics which allows the application to decide whether a triple is accessible or not. The main advantage of our approach is its *flexibility*, caused by the fact that the access control label contains the information on how it was generated. We plan to exploit this fact as part of our future work, in order to support dynamic datasets, authorizations, requirements and access control semantics and simplify the maintenance of an access control-enhanced dataset. We believe that in a dynamic setting the efficiency of adapting the access labels during changes will justify the overhead imposed by the use of abstract labels.

We implemented our ideas on top of the MonetDB and PostgreSQL open source RDBMS. We also conducted a preliminary set of experiments that show the need for more efficient storage and indexing schemes to handle abstract expressions. This also involves identifying ways to address the problem of ontologies producing a large number of quads (through the inference and propagation rules).

Acknowledgments

The authors would like to thank E. Kostylev, Y. Theoharis, and G. Karvounarakis for helpful discussions on earlier versions of this work. We would also like to thank Stefan Manegold for his help with MonetDB. This work was partially supported by the European NoE PlanetData.

9. REFERENCES

[1] The Gene Ontology (GO). http://www.geneontology.org/.

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

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

[4] P. Buneman and E. V. Kostylev. Annotation algebras for RDFS. In *SWPM*, 2010.

[5] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *WWW*, 2005.

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

[7] 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.

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

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

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

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

[12] A. Jain and C. Farkas. Secure Resource Description Framework. In *SACMAT*, 2006.

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

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

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

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

[17] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.

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

[19] W3C Linking Open Data. `esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`.

[20] 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.

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

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

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

[24] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On Graph Features of Semantic Web Schemas. *TKDE*, 20(5), 2008.

[25] M. Garofalakis W. Fan, C.-Y. Chan. Secure XML Querying with Security Views. In *SIGMOD*, 2004.