Abstract Access Control Model for Dynamic RDF Datasets

Giorgos Flouris[1], Irini Fundulaki[1,2], and Vassilis Papakonstantinou[1,3]

[1] FORTH-ICS, Greece
[2] CWI, The Netherlands
[3] University of Crete, Greece
{fgeo,fundul,papv}@ics.forth.gr**

Abstract. Given the increasing amount of sensitive RDF data available on the Web, it becomes increasingly critical to guarantee secure access to this content. Access control is complicated when RDFS inference rules and other dependencies between access permissions of triples need to be considered; this is necessary, e.g., when we want to associate the access permissions of inferred triples with the ones that implied it. In this paper we advocate the use of abstract provenance models that are defined by means of *abstract tokens* and *operators* to support *fine-grained* access control for RDF graphs. The access label of a triple is a complex expression that encodes *how* said label was produced (i.e., the triples that contributed to its computation). This feature allows us to know exactly the effects of any possible change, thereby avoiding a complete recomputation of the labels when a change occurs. In addition, the same application can choose to enforce different access control policies or, different applications can enforce different policies on the same data, avoiding the recomputation of the label of a triple. Preliminary experiments have shown the applicability and benefits of our approach.

1 Introduction

RDF [5] has established itself as a widely used standard for representing data in the Semantic Web. The number of applications that publish and exchange possibly sensitive RDF data continuously increases in a large number of domains ranging from bioinformatics [16] to e-government[1]. In light of the sensitive nature of the available information, the issue of *ensuring the selective exposure of information* to different classes of users is becoming all the more important.

The problem becomes more complicated when the access permissions of triples are interrelated, i.e., when the access permission of a certain triple affects the access permission of other triples. This feature appears, e.g., when we consider *inference* and *propagation* of labels along the RDFS [3] *subclassOf* and *subpropertyOf* hierarchies.

** This work was partially supported by the European NoE PlanetData.

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

The majority of the state of the art approaches for RDF access control [2, 10, 11] use *annotation models* where each triple is assigned a *concrete value* as access label that determines whether the triple can be accessed. In these models the computation of the access label of a triple (via inference or propagation) is done in a fixed manner according to predefined semantics, and any change in the dataset, the access control authorizations, the way in which the labels of implied triples are computed or the access control policy in general would force the recomputation of the triple's access permission [12].

Our proposal to address this problem is based on *abstract access control models* inspired by similar models proposed for relational data provenance [8]. In contrast to annotation models, which store information on whether the triple can be accessed or not, abstract labels encode *how* this information was computed. Encoding such information allows one to quickly identify the labels affected by any given change, making the recomputation of the entire dataset's access permissions unnecessary.

Abstract access control labels *do not commit* to a specific assignment of values to access labels of triples and specific semantics for computing the labels obtained through inference and propagation. Instead, the model defines *abstract tokens and operators*, so the access label of a triple is an *abstract algebraic expression*. To determine the actual label of a triple associated with an abstract expression (i.e., to decide whether a triple can be accessed or not), one should use *concrete policies* that assign specific values to the abstract tokens and operators and are defined by the corresponding application. Using the policy semantics, one can then compute the value of the abstract expression, i.e., the concrete label associated with the triple, and decide whether the triple can be accessed.

The main benefit of the proposed model is that, in contrast to standard annotation models, changes in the dataset or the related authorizations do not force a complete recomputation of the access labels of all triples. Thus, various types of changes can be supported more efficiently. This can lead to important gains especially when large and dynamic datasets are considered. In addition, the use of concrete policies allows applications to easily experiment and/or dynamically adapt their policy according to their needs. It also allows different applications to use the same access control enhanced dataset, even if the applications choose to adopt different access control policies.

In summary, the main contributions of our work are: *(i)* the definition, formalization and use of an *abstract access control model* for storing the access permissions of RDF triples, which allows us to determine *how* the access label of a triple was computed when inference or propagation of labels is considered (Section 3); *(ii)* 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 (Section 3); *(iii)* the description of how abstract access control labels can be used to annotate triples (*annotation process*, Section 3), how concrete policies can be used to determine the accessibility of annotated triples (*evaluation process*, Section 4), and how different types of changes in the dataset and/or the information on whether a triple can be accessed can be handled without the need to recompute the access control labels of the entire dataset (Section 5); and, *(iv)* the description of preliminary experiments that quantify the overhead of our approach during annotation and evaluation, as well as its gains in the case of updates (Section 6).

The present paper describes ongoing work. Formal details are omitted in this version for reasons of brevity, but the reader can find in [13] a more detailed presentation of the work.

2 Preliminaries: RDF and SPARQL

An RDF triple [5] is of the form (*subject*, *predicate*, *object*) and asserts the fact that *subject* is associated with *object* through *property*. The RDF Schema (RDFS) language [3] provides a built-in vocabulary, which allows the assertion of *instanceOf* relationships of resources using the RDF predicate *type* (type), and *subsumption* relationships among classes and properties using the RDFS *subClassOf* (sc) and *subPropertyOf* (sp) properties respectively.

RDFS also defines a set of *inference rules* [9] which are used to infer new triples for the sp, sc and type relationships. An RDF *graph* is defined as a *set of* RDF *data* and *schema* triples. In this work we consider graphs in which the sc and sp relations are *acyclic*. Acyclicity holds in the large majority of real-world RDF datasets [19], and is a common assumption made for efficiency (e.g., query optimization [17]) in many RDF applications.

SPARQL [15] 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. In this paper, SPARQL will be used in *authorizations* to specify the triples to be associated with a given label.

SPARQL uses *triple patterns* which resemble an RDF triple, but may contain *variables*. Intuitively, a triple pattern denotes the triples in an RDF graph that have the form specified by the pattern. SPARQL *graph patterns* are produced by combining triple patterns through the *join*, *optional* and *union* operators. The SPARQL syntax follows the SQL select-from-where paradigm. CONSTRUCT SPARQL queries (that we use in our work) return an RDF graph (i.e., set of triples) specified by a graph pattern in the WHERE clause of the query. A more detailed description of the SPARQL language cab ne found in [14].

3 Annotation

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 through the *propagation rules*. RDF triples are either annotated with abstract tokens or with a complex algebraic expression involving abstract tokens and operators. To store access control information, we rely on *annotated RDF triples* represented as *quadruples* 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*.

	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

Fig. 1. RDF Triples

$\mathcal{A}_1$:	(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)

Fig. 2. Access Control Authorizations

The process of *annotation* amounts to associating all triples in an RDF graph with their corresponding *abstract label*. This is performed in three steps. First, *authorizations* are used to explicitly associate triples with labels; these authorizations are of the form (q, l) where q is a CONSTRUCT SPARQL query and l is an abstract access token to be associated to all triples that are in the scope of q. Second, inference rules are applied that generate new quadruples. Third, propagation rules are considered, which are used to give new labels to already existing triples, producing new quadruples (that correspond to existing triples). At the end of the process, each triple will have one or more (possibly complex) access labels associated with it. The exact meaning of these labels (whether the triple can be accessed or not) is determined during the *evaluation* process, described in Section 4 using the *concrete access control policies*.

More details on the annotation process, along with an illustrating example, are given below. For a more formal discussion of the annotation process and the abstract access control policies the reader is referred to [13].

Applying the Authorizations. Figure 1 shows (in tabular form) an RDF graph (inspired by the FOAF ontology[2]) that we use in this paper for illustration purposes, and Figure 2 shows a set of access authorizations defined for these triples. Figure 3 shows the RDF quadruples obtained by evaluating the authorizations of Figure 2 to the set of triples in Figure 1. For example, t_1, t_2 are in the scope of authorization $\mathcal{A}_2$, and are assigned token at_2 (resulting to the quadruples q_1, q_2); note that we do not store directly the abstract token associated with the triple, but the authorization responsible for said token ($\mathcal{A}_2$). This allows a more efficient recomputation of the labels when authorizations change (see Section 5). The rest of the quadruples in Figure 3 are similarly produced.

It is possible that a certain triple is in the scope of more than one authorizations and thus two or more different labels are assigned to it, in different quadruples, as in the case of q_1, q_7 in our example. Moreover, it is possible that a certain triple is not in the scope of an authorization, in which case we assign it a *default*

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

token, denoted by $\perp$; this is the case for quadruple q_5 (coming from t_5). Note that in our example each of the authorizations uses a different token. Nevertheless, our model does not forbid different authorizations from using the same token.

	s	p	o	l
q_1 :	*Student*	sc	*Person*	$\mathcal{A}_2$
q_2 :	*Person*	sc	*Agent*	$\mathcal{A}_2$
q_3 :	&a	type	*Student*	$\mathcal{A}_3$
q_4 :	&a	*firstName*	`Alice`	$\mathcal{A}_1$
q_5 :	&a	*lastName*	`Smith`	$\perp$
q_6 :	*Agent*	type	class	$\mathcal{A}_4$
q_7 :	*Student*	sc	*Person*	$\mathcal{A}_5$

Fig. 3. RDF Quadruples

Applying the Inference Rules. The inference rules that we consider in this work are shown in Figure 4, and extend those specified in [9] in a straightforward manner to take into account access labels. As shown in the table, an implied triple is annotated by a complex expression that involves the labels of its implying triples connected through the binary *inference operator* $\odot$, which reflects the fact that *both* implying triples were involved in the implication, and thus both should affect the label of the implied triple [7].

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

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

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

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

Fig. 4. Extended Inference Rules

	s	p	o	l
q_8 :	*Student*	sc	*Agent*	$q_1 \odot q_2$
q_9 :	*Student*	sc	*Agent*	$q_2 \odot q_7$
q_{10} :	&a	type	*Person*	$q_3 \odot q_1$
q_{11} :	&a	type	*Agent*	$(q_3 \odot q_1) \odot q_2$
q_{12} :	&a	type	*Agent*	$q_3 \odot (q_2 \odot q_7)$

Fig. 5. Implied Quadruples (Partial List)

The application of (some of) these rules in our running example is shown in Figure 5. Note that, again, we do not store the abstract annotation tokens directly, but the quadruples (*qid*) from where they came. Thus, e.g., in the case of q_{10} we store the label $q_3 \odot q_1$. This shows that the label of q_{10} is associated with the labels of q_1, q_3, a feature useful during changes (see Section 5).

Note that implicit quadruples may also be involved in inferences, resulting in more complex expressions; for example q_{11} is obtained from q_{10} and q_2, whereas q_{12} is obtained from q_3 and q_9. To improve efficiency during the evaluation process (see Section 4), the labels are recursively resolved, so the label of q_{12} is $q_3 \odot (q_2 \odot q_7)$ rather than $q_3 \odot q_9$.

Applying the Propagation Rules. The RDFS semantics associated with the class and property hierarchies has caused several authors to consider the *propagation* of access control labels along such hierarchies [11]. This is inspired by similar label propagation models employed in hierarchical models, like XML [6]. For example, an application may require that a triple that defines an instantiation relationship between an instance and a class, inherits the access label of the triple defining such class. This feature is modeled using *propagation rules.*

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

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

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

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

Fig. 6. Propagation Rules

The propagation rules *do not generate new triples* since this is the role of the inference rules discussed previously; instead, they 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 rules for the opposite direction. The propagation of labels is modeled with the *abstract propagation operator*, denoted by $\otimes$. The propagation rules that we consider in this work are shown in Figure 6; note that this set of rules can change to adapt to the application needs, or be omitted altogether.

The application of these rules in our running example would cause q_6 to propagate its label to all instances of $Agent$, thus obtaining the quadruple $q_{13} = (\&a, \mathsf{type}, Agent, \otimes q_6)$, which, in our running example, is the only propagated quadruple.

4 Evaluation of Access Control Labels

As explained above, our framework does not bound the abstract tokens (at_i), the default token ($\perp$) and the operators ($\odot$, $\otimes$) to concrete values. Instead, each application, depending on its needs, should define a *concrete policy* that *maps* each abstract access token to a concrete value and specifies the concrete operators that implement the abstract ones ($\odot$, $\otimes$). The concrete policy also specifies how ambiguous labels are resolved and how accessibility is determined. Using these tools one can determine whether a certain triple (which may be associated with multiple, possibly complex, abstract labels) can be accessed. This process is called *evaluation* and is described below using our running example.

To present the evaluation process, assume that the concrete policy determines 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 determines 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).

Using the above mappings and operators, we can easily compute the label of each quadruple containing abstract tokens. For example, the label of q_1 is *true* (because the label of $\mathcal{A}_2$ is at_2, mapped to *true*). Similarly, the label of q_8 is *true*; this occurs by computing the label of q_1 as above (*true*) and the label of q_2 (*true*) and using the definition of $\odot$ to "combine" these labels (giving *true*, by our definition above). Using similar arguments, the label of q_9 can be computed to $false$. Note that the label of q_5 is not computable because it contains the special label $\perp$. The operators $\odot$, $\otimes$ should define how this special label is used during inference and propagation respectively (e.g., we could state that $true \odot \perp = true$ and $\perp \odot \perp = \perp$), even though this is not necessary in the particular example. The labels of all quadruples considered in our running example are shown in Figure 7.

	s	p	o	l
q_1 :	*Student*	sc	*Person*	*true*
q_2 :	*Person*	sc	*Agent*	*true*
q_3 :	&a	type	*Student*	*true*
q_4 :	&a	*firstName*	`Alice`	*true*
q_5 :	&a	*lastName*	`Smith`	$\perp$
q_6 :	*Agent*	type	class	$false$
q_7 :	*Student*	sc	*Person*	$false$
q_8 :	*Student*	sc	*Agent*	*true*
q_9 :	*Student*	sc	*Agent*	$false$
q_{10} :	&a	type	*Person*	*true*
q_{11} :	&a	type	*Agent*	*true*
q_{12} :	&a	type	*Agent*	$false$
q_{13} :	&a	type	*Agent*	$false$

Fig. 7. Evaluated RDF Quadruples

Note that many triples are contained in several quadruples, so they are associated with different labels. To determine the concrete label of such triples, the concrete policy should specify how to disambiguate among the different labels, so that each triple is eventually associated with one, and only one concrete label. For example, a cautious policy would state that it favors quadruples with $false$ label (i.e., $false$ overrides *true* and $\perp$), and also favors *true* over $\perp$.

Finally, the concrete policy should specify whether the resulting label of a triple should be interpreted as allowing or denying access to a triple. In our example, we could state that a triple can be accessed if and only if its final label (after all the above manipulations have been applied) is *true*.

Under the above semantics, Figure 7 states that (&a, *lastName*, `Smith`) and (&a, type, *Agent*) cannot be accessed; this is true because the former is associated with $\perp$ (which indicates a non-accessible triple), whereas the latter is associated with four labels, namely, *true*, *true*, $false$, $false$, which are combined to $false$, thereby indicating a triple that cannot be accessed. On the other hand, (*Person*, sc, *Agent*) can be accessed because it is only associated with the label *true*, indicating an accessible triple.

5 Handling Changes in Our Framework

As already mentioned, the main advantage of our framework is that it allows an efficient recomputation of the access control labels whenever some kind of change occurs. In the following, we give the different types of changes that can occur, and describe how our framework addresses them.

Changes in the Dataset. The dataset can be changed by either adding or deleting triples from it. Due to space restrictions, we consider only the latter case.

When deleting a triple, we must first find all the quadruples that involve said triple and delete them. Then, we should recursively identify all quadruples whose label contains at least one of the IDs of the deleted quadruples and delete them also. The efficiency of our framework is evident here, as the abstract labels allow us to easily determine the quadruples that need to be deleted without having to re-apply the inference/propagation rules to recompute the labels.

Changes in the 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 authorizations; the latter case can be split in two subcases, namely modifying the SPARQL query of the authorization or modifying the associated token.

As before, the case of additions is easy and omitted due to space limitations. The deletion of authorizations is similar to the deletion of triples, except that now specific quadruples (namely those that were labeled using the deleted authorization), rather than all the quadruples associated with a triple, must be deleted. Then, the recursive deletion described in the case of triple deletions is applied to find all the quadruples affected by the change.

The only difference with the above case appears when a triple is associated with a label only through the deleted authorization. In this case, the corresponding quadruple should not be deleted, but rather have its label replaced with $\perp$. Note that in this case, no further changes (i.e., cascading deletions) are needed.

When the SPARQL query of an authorization $\mathcal{A}$ changes, then the triples that are in the scope of said authorization are affected. This will result to some triples losing their label acquired through $\mathcal{A}$ and some triples gaining a new authorization label through $\mathcal{A}$. This case can therefore be handled by adding and deleting the corresponding quadruples as indicated by the difference between the triples that are in the scope of the new and old queries and applying the cascading effects, as in the case of adding/deleting authorizations.

Finally, when the token associated with an authorization changes, no changes are required in the dataset, because triples are associated with their tokens only indirectly, through the authorization. Thus, we only need to update the token related to the authorization. This is not possible in standard annotation models.

Changes in the Access Control Policies. Even though this is the most complicated type of change, as it can take various different forms, it is the easiest type for our framework. In particular, changing the access control policy simply corresponds to changing the concrete policy. 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.

This fact provides a significant flexibility to applications to define their own access control semantics. 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.

6 Discussion

To evaluate our framework, we used an implementation based on the MonetDB column store[3]. The abstract access control labels were stored in a relational schema, the details of which are omitted due to space limitations, but can be found at [13]. All our algorithms (annotation, evaluation, and handling of the various change scenarios) were implemented using MonetDB's stored procedures.

Our evaluation included both real and synthetic datasets. The real datasets used were CIDOC [4] and GO [1], whereas the synthetic ones were produced using Powergen [18], which is a synthetic RDFS schema generator that takes into account the morphological features of real-world schemas to produce realistic ontologies. The sizes of the considered ontologies ranged from 1.000 to 265.000 triples, whereas the number of quadruples produced in these ontologies (after the annotation process) ranged from 5.000 to more than 6.200.000 quadruples.

The main conclusions drawn from our experiments are: *i)* annotation time is linear with respect to the size of the dataset, and is affected by the morphology of the RDF graph (e.g., annotation time for deeper subsumption hierarchies is higher) *ii)* the evaluation time increases linearly with respect to the number of triples being evaluated *iii)* the annotation time required in the case of updates, depends linearly on the number of quadruples that need to be changed, rather than the size of the dataset. This does not hold in the cases where the authorization token or the concrete policy changes, which run at constant time.

The overhead of the annotation time when compared to standard annotation approaches is negligible for datasets with less than 300.000 quadruples. Above this dataset size, there is a 40% to 60% overhead. The average evaluation time per triple (given a concrete policy) is in the order of few milliseconds.

It is straightforward that in the case of update scenarios, the time required for the re-annotation of affected triples was much smaller than the annotation time for the whole dataset. Depending on the dataset considered and the type of the change, the speedup achieved using our method is one order of magnitude higher, or more.

Finally, our preliminary experimental results showed that the storage overhead of our approach, when compared to standard annotation models, is approximately one order of magnitude higher; this is explained by the fact that abstract expressions can potentially get large, requiring much more space for their storage than the computed concrete labels.

7 Conclusion

In this paper we addressed the problem of controlling access to RDF graphs by providing an open and customizable framework that takes into account RDFS

[3] http://www.monetdb.org/Home

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 more efficient handling of updates for datasets as well as access control policies. The innovation of the method is that the abstract access labels encode the information of *how* each access control label was produced, thereby allowing the easy determination of the labels that are affected by an update. In addition, our framework is flexible enough to allow easy experimentation with different access control policies without requiring the recomputation of the accessibility information after each change.

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. CIDOC Conceptual Reference Model (CRM). http://www.cidoc-crm.org/, 2006. ISO 21127:2006.
5. B. McBride F. Manola, E. Miller. RDF Primer. `www.w3.org/TR/rdf-primer`, February 2004.
6. W. Fan, C. Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.
7. G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides. Coloring RDF Triples to Capture Provenance. In *ISWC*, 2009.
8. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
9. P. Hayes. RDF Semantics. `www.w3.org/TR/rdf-mt`, February 2004.
10. A. Jain and C. Farkas. Secure Resource Description Framework. In *SACMAT*, 2006.
11. J. Kim, K. Jung, and S. Park. An Introduction to Authorization Conflict Problem in RDF Access Control. In *KES*, 2008.
12. M. Knechtel and R. Peñaloza. A Generic Approach for Correcting Access Restrictions to a Consequence. In *ESWC*, 2010.
13. V. Papakonstantinou, M. Michou, I. Fundulaki, G. Flouris, and G. Antoniou. Access control for RDF graphs using abstract models. In *SACMAT*, 2012.
14. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.
15. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `www.w3.org/TR/rdf-sparql-query`, January 2008.
16. Gene Ontology. `www.geneontology.org`.
17. G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *ISWC*, 2005.
18. Y. Theoharis, G. Georgakopoulos, and V. Christophides. PoweRGen: A Power-Law Based Generator of RDFS Schemas. *Information Systems. Elsevier*, 2011.
19. Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On Graph Features of Semantic Web Schemas. *TKDE*, 20(5), 2008.