Access Control for RDF: Experimental Results

Giorgos Flouris[1], Irini Fundulaki[1], Maria Michou[1], and Grigoris Antoniou[1,2]

[1] Institute of Computer Science, FORTH, Greece
[2] Computer Science Department, University of Crete, Greece
{fgeo, fundul, michou, antoniou}@ics.forth.gr

Abstract. One of the current barriers towards realizing the huge potential of Future Internet is the protection of sensitive information, i.e., the ability to selectively expose (or hide) information to (from) users depending on their access privileges. In this work we discuss the experiments conducted with our repository independent, portable across platforms system that supports fine-grained enforcement of RDF access control.

1 Introduction

The potential of RDF as a data representation standard for the Future Internet is undermined by the lack of an effective mechanism for controlling access to RDF data. In light of the potentially sensitive nature of RDF information, the issue of *securing* RDF content and *ensuring the selective exposure of information* to different classes of users depending on their access privileges is an important issue. The building blocks of an access control system are the *specification language*, that allows the expression of access control permissions and policies, and the *enforcement mechanism*, responsible for applying the latter to the data, by denying access to data that the policy has deemed as non-accessible.

In this work, we enforce access control by proposing a solution which is repository independent, portable across platforms, and in which *fine-grained access control* (protection at the level of RDF triple) is enforced by a component built on top of the RDF repository. In this poster we report on experiments performed with our system; the full description and formal semantics of our language, as well as more details on the approach can be found in [6].

2 RDF Access Control Framework

We concentrate on *fine-grained* RDF access control for *read-only queries*. An *access control permission* is used to explicitly grant or deny to/from a given user the ability to access an RDF triple, or a set of RDF triples, and can be viewed as a query whose evaluation over an RDF graph results in a set of triples which are accordingly granted or denied access. Access control permissions are expressed using SPARQL [7] *triple patterns* and *value constraints*, and are of the form: $\mathcal{R} = \mathsf{include/exclude}\ (x,\ p,\ y)\ \mathsf{where}\ \mathcal{TP}, \mathcal{C}$ with $(x,\ p,\ y)$ a triple pattern, $\mathcal{TP}$ a conjunction of triple patterns and $\mathcal{C}$ a conjunction of value constraints on the variables appearing in the triple patterns. Explicit access rights are not set for all triples in an RDF graph, and permissions are not always unambiguous (i.e., a triple could be marked as both accessible and inaccessible). To determine whether such triples should be accessible, we use the notion of *access control policy*, which includes a set of positive and a set of negative permissions, as well

as two boolean flags (*default semantics* and *conflict resolution* – ds, cs resp.), which determine whether triples with missing (resp. ambiguous) permissions are accessible or not. A full description of the formal semantics of access control permissions and policies can be found in [6].

3 Implementation and Experiments

Architecture: We implemented a main memory platform which serves as an additional access control layer on top of an arbitrary RDF repository. Our goal was for our system to be portable across platforms, so it was designed in a repository-independent way. The system's architecture is shown in Fig. 1. It is comprised of the following modules, all implemented in Java: the RDF Dataset Loader, responsible for loading the complete RDF dataset in the underlying repositories, the RDF Access Control Policy Manager that loads in memory the access control policies and the RDF Access Control Enforcement Module, which translates the access control policies into the appropriate programs that compute the accessible triples of an RDF dataset and annotates accordingly the data in the repositories with accessibility information.

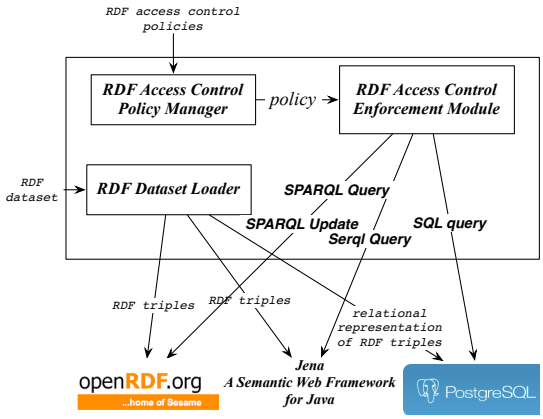
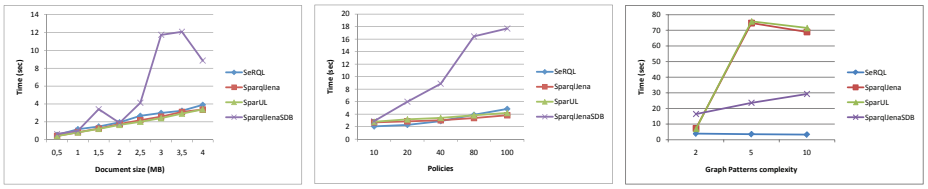


Fig. 1. System Architecture

To enforce an access control policy we produce a query which implements the semantics of the policy and is expressed in the language supported by the underlying repository. The triples in the result of the evaluation of this query on the RDF graph are exactly the accessible triples which are then *annotated* as such. Conceptually, annotations can be represented by adding a fourth column to an RDF triple (hence obtaining a quadruple), denoting whether the triple is accessible or not; if several different user roles need to be supported, one column per role should be added. Annotations can be stored using the *named graphs* mechanism of RDF repositories [5], or, in the case of a relational backend, by extending the triple table that stores the RDF triples with a fourth column.

Experiments: Our experiments measured the time required to annotate the set of RDF triples, using the above methodology, in state-of-the-art RDF repositories (Sesame [3], Jena [1]) or relational backends (Postgres [2]). We used the SP2Bench [8] data generator to obtain the input RDF graphs. We implemented our approach on top of Jena v2.6.2, Sesame v2.3.1 and Postgresql v8.4. For Jena we tested the SparqlJenaModule and SparqlJenaSDBModule (processing SPARQL queries) as well as the SPARULModule (processing SPARQL Update

queries) modules. SparqlJenaModule and SPARULModule load the datasets into main memory whereas the SparqlJenaSDBModule stores the datasets to a Postgresql database. For Sesame we used the SeRQLModule, which processes SPARQL [7] and SeRQL [4] queries in memory.



(a) $gp = 2$,$ps = 80$ (b) $gp = 2$,$doc = 4MB$ (c) $ps = 80$, $doc = 4MB$

Fig. 2. Experiments

We measured the time required for the annotation as a function of four different parameters: *(i) document size* (*doc*), i.e., the size of the input RDF graph (size ranging between 500KB-4MB with a 500KB increase); *(ii) policy size* (*ps*), i.e., the number of permissions in the access control policy (for sizes of 10, 20, 40, 80 and 100, with an equal share of positive/negative permissions in each case); *(iii) permission size* (*gp*), i.e., the number of triple patterns and constraints in the where clause of each access control permission (values considered: 2, 5, 10); and *(iv) policy parameters*, i.e., the values of the ds, cr parameters of the input policy (all 4 combinations considered).

Evaluation: Fig. 2 shows a subset of the results of our experiments: we run each experiment 5 times, and took the average time. In each graph, the annotation time is presented as a function of one of the parameters *(i)-(iv)*, for fixed values for the other parameters. We report here on policies with *"deny"* as default semantics (*ds*) and conflict resolution policy (*cr*) because it is the most common one. The results show that our approach scales along the considered parameters. All the platforms that we ran our experiments on demonstrated a linear behavior as document, policy sizes and permission complexity increased (except the Jena SPARUL and SPARQL Modules).

References

1. Jena A Semantic Web Framework for Java. `http://jena.sourceforge.net/`.
2. PostgreSQL. http://www.postgresql.org/.
3. Sesame: RDF Schema Querying and Storage. http://www.openrdf.org/.
4. J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *Workshop on Semantic Web Storage and Retrieval*, 2003.
5. J. J. Carroll, C. Bizer, P. J. Hayes, and P. Stickler. Named Graphs. *JWS*, 3(4), 2005.
6. G. Flouris, I. Fundulaki, M. Michou, and G. Antoniou. Controlling Access to RDF Graphs. In *FIS*, 2010.
7. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. `www.w3.org/TR/rdf-sparql-query`, January 2008.
8. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. Technical report, arXiv:0806.4627v1 cs.DB, 2008.