Enabling Advanced and Context-Dependent Access Control in RDF Stores

Fabian Abel[1], Juri Luca De Coi[2], Nicola Henze[1],
Arne Wolf Koesling[2], Daniel Krause[1], and Daniel Olmedilla[2]

[1] Distributed Systems Institute (KBS), University of Hannover, Hannover, Germany
{abel,henze,krause}@kbs.uni-hannover.de
[2] L3S Research Center and University of Hannover, Hannover, Germany
{decoi,koesling,olmedilla}@L3S.de

Abstract. Semantic Web databases allow efficient storage and access to RDF statements. Applications are able to use expressive query languages in order to retrieve relevant metadata to perform different tasks. However, access to metadata may not be public to just any application or service. Instead, powerful and flexible mechanisms for protecting sets of RDF statements are required for many Semantic Web applications. Unfortunately, current RDF stores do not provide fine-grained protection. This paper fills this gap and presents a mechanism by which complex and expressive policies can be specified in order to protect access to metadata in multi-service environments.

1 Introduction

The Semantic Web vision requires that existing data is provided with machine-understandable annotations. These annotations (commonly referred to as *metadata*) are meant to facilitate tasks such as data sharing and integration. However, it is often the case that information cannot be shared unconditionally: many Semantic Web applications require to control when, what and to whom information is disclosed. Nevertheless, existing metadata stores do not support access control, or their support is minimal (e.g., protection may apply only to the database as a whole and not to the data it contains). On the one hand, access control could be embedded within the metadata store: in this case the access control mechanism would be repository-dependent and not portable across different platforms. On the other hand, a more general solution would be adding a new component on top of the metadata store in charge of access control-related issues. Still this second approach requires to face problems which are not trivial, since the obvious solution of filtering out private triples from the results is not possible. The reason is that those triples may not be known in advance, as it happens when the result of the query consists of triples not previously available in the metadata store.

Furthermore, the Semantic Web envisions that interactions can be performed between any two entities, even if they did not carry out common transactions in the past, making unsuitable traditional identity-based access control mechanisms. Therefore, more advanced mechanisms (based e.g., on properties of the

K. Aberer et al. (Eds.): ISWC/ASWC 2007, LNCS 4825, pp. 1–14, 2007.
© Springer-Verlag Berlin Heidelberg 2007

requester) are required. Semantic policy languages lately emerged in order to address these requirements: they provide the ability to specify complex conditions such as time constraints and may even provide an interface to query external packages such as other repositories. However, evaluating such constraints for each triple to be potentially returned by the metadata store is not affordable, since it is too expensive in terms of time.

In this paper we present an architecture that integrates advanced access control mechanisms based on Semantic Web policies with different kinds of RDF metadata stores. Given an RDF query, our framework partially evaluates all applicable policies and constraints the query according to the result of such evaluation. The modified query is then sent to the RDF store which executes it like an usual RDF query. Our framework enforces fine-grained access control at triple level, i.e., all triples returned as a response to the query can be disclosed to the requester according to the policies in force.

The rest of the paper is organized as follows: §2 presents an scenario in order to motivate the need for flexible access control mechanisms over semantic data. Different approaches and related work are described in §3. In §4 we introduce how a policy engine can be integrated on top of an RDF store in order to restrict access at the RDF triple level. §5 presents our current implementation which is used in order to perform a set of experiments, which estimate the impact of this approach and are included in this section too. Finally, §6 concludes the paper and outlines our future work.

2 Access Control in RDF Stores

The Personal Reader [1] is a distributed multi-service and multi-user environment. In the Personal Reader Framework there exist many modular applications divided into (i) Personalization Services for accessing and personalizing Semantic Web data sources (ii) Syndication Services for aggregating and processing data provided by other services and (iii) User Interfaces which can be accessed by different users. A remarkable feature of the Personal Reader Framework is its plug-and-play nature, i.e., new Personalization and Syndication Services can be plugged into the system and immediately used by currently available services and users.

Enabling behavior and content adaptation in different applications requires the use of a shared user profile. Such a user profile is in charge of storing semantic data from different services, application domains, and users. RDF databases have been chosen to store these metadata, since they provide efficient access and high flexibility: arbitrary RDF data referring to various ontologies can be stored within the RDF database.

Different services may store in or require sensitive data from the user profile in the RDF repository. It is crucial for the user to be able to specify which (kinds of) services are allowed to access and retrieve which part of the data stored in the user profile. For example, Alice must be able to allow a recommendation service to access information about her friends but not her private information

(e.g., address or telephone number). Similarly, a means needs to be provided to allow Bob to restrict access to his health-related data only to his medical service. Health-related data may be defined as instances of a class `Health` in some ontology, and the medical service may need to identify itself by providing some credential.

The most part of current RDF databases provide none or very simple security mechanisms. For example, one of today's most widespread RDF database management systems, *Sesame* [2], allows to define access rights only for a whole database. Therefore, a more fine-grained solution is required, in which access to smaller fragments of data (e.g., RDF triples) can be checked at query time. Furthermore, the result of the check may depend on conditions unrelated to the data to be accessed (*contextual* conditions), such as properties of the requester (possibly to be certified by credentials) or environmental factors (e.g., time of the request). Ideally, a good solution should be expressive and flexible while at the same time not excessively affecting the response time.

3 Related Work

The problem we focus on in this paper is how to restrict access to RDF data. One way to address this problem is defining *a priori* which subsets of an RDF database can be accessed by some requester. For example, Named Graphs [3] can be used to evaluate SPARQL queries [4] based on allowed RDF graphs [5] or in combination with TriQL.P [6] which allows the formulation of trust-policies, in order to answer graph-based queries. Those queries describe conditions under which suiting data should be considered trustworthy. Access control based on identity could be performed if all requesters and their allowed graphs were known in advance. However, this is not the case in our scenario presented above and since access to data may be (not) allowed depending on contextual conditions, these approaches do not apply: on the one hand named graphs cannot be statically precomputed for each possible combination of environmental factors, since their number would be too big; on the other hand named graphs cannot be created at runtime, since the creation process would excessively slow down the response time. Furthermore, the plug-and-play nature of the Personal Reader Framework as well as the possibility that services dynamically change the RDF database itself by adding or removing data from the user profile would significantly complicate managing such named graphs.

[7] defines simple rule-based policies over the RDF database: such policies describe subgraphs on which actions like *read* and *update* can be executed: subgraphs are identified by specifying graph patterns. Some approaches also respect RDF Schema entailments [8]. However, all these approaches require to instantiate the graph patterns, i.e., to generate one graph for each policy and execute the given query on each graph, hence leading to longer response times. Furthermore, these approaches cannot be applied to contextual queries either.

Finally, many policy languages (e.g., KAOS [9], Rei [10], PeerTrust [11] or Protune [12]) allow to express access authorizations on the Semantic Web by

means of policies. However, none of them describes how to integrate policies in RDF databases.

4 Policy-Based Query Expansion

As shown in §3, existing work on RDF data protection does not suit the requirements of dynamic Semantic Web environments such as the Personal Reader Framework presented in §2. Available solutions do not handle contextual information in a proper way, as they either require a large amount of memory or unacceptably increase the response time. Filtering returned results is not an adequate solution, either: current RDF query languages allow to arbitrarily structure the results, as shown in the following example[1].

```
CONSTRUCT {CC} newns:isOwnedBy {User}
  FROM {User} ex:hasCreditCard {CC}; foaf:name {Name}
  WHERE Name = 'Alice'
```

Post-filtering the results of a query is hence not straightforward whenever their structure is not known in advance. It could be possible to break constructs queries into a select query and the generation of the returned graph (construct), therefore avoiding this problem. However, the query response time may be considerably too large since this approach cannot make use of repository optimizations and policies are enforced after all data (allowed and not allowed) has been retrieved. As an example, suppose an unauthorized requester submits a query asking for all available triples in the store. A post-filtering approach would retrieve all triples first and then filter them all out.

To address these problems we decided to enforce access control as a layer on top of RDF stores (also making our solution store independent). Our strategy is to pre-evaluate the contextual conditions of the policies, which do not depend on the content of the RDF store. Then, we expand the queries before they are sent to the database, therefore integrating the enforcement of the rest of (metadata) conditions with the query processing, thereby restricting the queries in such a way that they only utilize allowed RDF statements. This way, policies can hold a greater expressiveness and support both metadata and contextual conditions, while pushing part of its enforcement to the highly optimized query evaluation of the RDF store. This approach allows to include more complex conditions without dramatically increasing the overhead produced by policy evaluation, and while relying on the underlying RDF store to evaluate RDF Schema capabilities (as discussed in [8]).

[1] Our examples use SeRQL [13] syntax (and for simplicity we do not include the namespace definitions), since SeRQL is the language we exploit in our implementation. However the ideas behind our solution are language-independent and can be applied to other RDF query languages.

4.1 RDF Queries

We assume disjoint, infinite sets I, B, and L, which denote IRIs, blank nodes and literals. In addition, let $Pred$, $Const$ and Var be mutually disjoint sets of predicates, constants and variables such that $Const = I \cup B \cup L$. Then (using similar notation as in [14]) an RDF graph is a finite set of triples $I \cup B \times I \times Const$.

In the following we assume a query language with queries having the following structure (§6.19 in [15])[2,3]:

```
SELECT/CONSTRUCT RF FROM PE WHERE BE
```

where

- RF is the result form, either a set of variables (projection in select queries) or a set of triples (construct clause in construct queries).
- PE is a path expression as defined below.
- BE is a boolean expression, that is, a string[4] representing a set of constraints in the form of (in)equality binary predicates and numerical operators such us greater than or lower than, connected by boolean connectives (AND and OR).

and a query will be denoted as $q = (RF, PE, BE)$. As today's established RDF query languages like SeRQL [13] or SPARQL [4] do not support *insert* or *delete* operations yet, we focus on common *read* operations. An example query is provided in Figure 1. Without access control enforcement, this query would return an RDF graph containing all RDF triples matching the graph pattern defined in the FROM block, i.e., the query answer would include identifier and name of a person, her phone number(s) and the document(s) she is interested in.

We define a path expression as a triple (s, p, o) such that $s \in I \cup B \cup Var$, $p \in I \cup Var$ and $o \in Const \cup Var$. Hereafter we will use (s, p, o) and $triple(s, p, o)$ ($triple \in Pred$) as synonyms. In addition, given an expression E (result form, path or boolean expression), we will denote by $vars(E)$ the set of all unbound variables occurring in E.

Definition 1. *Given a path expression $e = (s, p, o)$ and a set of variable substitutions θ the function $disunify(e, \theta)$ returns the tuple (e', BE), where e' is a new pattern (s', p, o') and BE is a set of boolean expressions such that*

- $\begin{cases} s' = v_s \text{ and } be_s = (v_s = s) & \text{if } s \in I \cup B \\ s' = v_s \text{ and } be_s = (v_s = Value) & \text{if } s \in Var, [s = Value] \in \theta \\ s' = s \text{ and } be_s = \varepsilon & \text{otherwise} \end{cases}$

[2] Although our examples will use the syntax of the SeRQL query language, the results of this paper apply also to other languages with similar structure (e.g., SPARQL [4]).

[3] Extending our algorithm to support UNION and INTERSECTION operators is straightforward. The union (resp. intersection) of two queries would be expanded into the union (resp. intersection) of the two expanded queries.

[4] In the rest of the paper we also use BE to represent a set of boolean expressions. The exact meaning will be clear from the context.

$$- \begin{cases} o' = v_o \text{ and } be_o = (v_o = o) & \text{if } o \in Const \\ o' = v_o \text{ and } be_o = (v_o = Value) & \text{if } o \in Var, [o = Value] \in \theta \\ o' = o \text{ and } be_o = \varepsilon & \text{otherwise} \end{cases}$$

where v_s and v_o are fresh variables and $BE = \{be_s, be_o\}$. Intuitively, the variable substitutions for the subject and object of the pattern are extracted and converted into boolean expressions.

The purpose of this function is to extract variable substitutions in order to be able to reuse path expressions in the final RDF query, even if they are specified in different policies.

```
CONSTRUCT * FROM {Person} foaf:name {Name};
      foaf:phone {Phone}; foaf:interest {Document}
```

Fig. 1. Example RDF query

4.2 Specifying Policies over RDF Data

Using policies to restrict access to RDF statements requires to be able to specify graph patterns (path expressions and boolean expressions), such as one can do in an RDF query. In addition, it is desired to have the ability of checking contextual properties such as the ones of the requester (possibly to be certified by credentials) or time (in case access is allowed only in a certain period of time). Therefore, we consider a policy rule *pol* to be a rule of the form

$$pred(triple(s, p, o)) \leftarrow cp_1, \ldots, cp_n, pe_1, \ldots, pe_m, be_1, \ldots, be_p.$$

where $pred \in \{allow, disallow\}$, $triple(s, p, o)$ is a path expression as defined above, cp_i are contextual predicates (e.g., related to time, location, possession of credentials, etc.), pe_i are path expressions and be_i are boolean expressions. In the following we will refer to $H(pol)$ to the head of *pol*, $H^T(pol)$ to the triple in the head of *pol* and $B(pol)$ to the (possibly empty) body of *pol*.

Suppose that Alice specified the policies presented in Table 1[5]. Instead of choosing a specific language, our policies are expressed in a high level syntax, which can be mapped to existing policy languages[6]. Their intended meaning is as follows:

1. the *RecommenderService* is not allowed to access the phone number(s) of members of the REWERSE project
2. recognized trusted services (which have to provide a suitable credential) are allowed to access the phone number(s) of people Alice knows.

[5] Note that policies might also refer to named graphs, therefore allowing for approaches in which whole named graphs can be given access if the policy is satisfied.

[6] Although the final selection of the language will have an impact in the expressiveness and power of the kind of policies specified and contextual predicates supported.

Table 1. Example of high-level policies controlling access to RDF statements

No.	Policy

pol_1	`deny access to triples (X, foaf:phone, Z) IF`
	`(X, foaf:currentProject, l3s:rewerse) AND`
	`Requester = 'RecommenderService'.`

pol_2	`allow access to triples (X, foaf:phone, Z) IF`
	`Requester = Service AND`
	`Service is a trusted service AND`
	`(l3s:alice, foaf:knows, X).`

pol_3	`allow access to triples (l3s:alice, foaf:phone, Z).`

pol_4	`allow access to triples (X, Y, Z) IF`
	`Time is the current time AND`
	`09:00 < Time AND Time < 17:00 AND`
	`Y = foaf:name AND X != l3s:tom.`

pol_5	`allow access to triples (l3s:alice, foaf:interest, Z) IF`
	`(Z, rdf:type, foaf:Document) AND`
	`(X, foaf:currentProject, P) AND`
	`(Z, foaf:topic, T) AND (P, foaf:topic, T).`

3. anyone can receive Alice's phone number.
4. RDF statements containing *name* of entities different from Alice's boss Tom can be accessed during work time
5. the last policy controls access to Alice's interests. Only interests related to her current project(s) can be accessed.

Many algorithms could be exploited in order to evaluate policies and to handle conflicts which arise whenever two different policies allow and deny access to the same resource. However such algorithms are out of the scope of this paper. Therefore, in the following we assume a simple policy evaluation algorithm like the following one:

if a *deny* policy is applicable **then** access to the triple is denied
else if an *allow* policy is applicable **then** access to the triple is allowed
else access to the triple is denied (*deny by default*)

More advanced algorithms exploiting priorities or default precedences [10] among policies could be used as well.

4.3 Policy Evaluation and Query Expansion

Given an RDF query, each RDF statement matching a pattern specified in the FROM block is accessed and, if the policies in force allow it, returned. Our approach consists of analyzing the set of RDF statements to be accessed and restricting it according to the policies in force. Contextual conditions (e.g., time constraints and conditions on properties of the requester) are evaluated by some policy engine, whereas other constraints are added to the given query and enforced during query processing.

Definition 2 (Policy applicability). *Given a path expression e, a set of policies P and a time-dependent state Σ [12] (which in our case determines at each instant the extension of contextual predicates), we say that a policy $pol \in P$ is applicable to e (denoted by $\widehat{pol(e)}$) iff*

- $\sigma' = mgu(e, H^T(pol))$*, where mgu is the most general unifier*
- $\exists \sigma, \sigma'' : \sigma = \sigma'\sigma'' \wedge \forall cp_i \in B(pol), P \cup \Sigma \models \sigma cp_i$
- *if* $\exists be_i \in B(pol) : \forall pe_i \in B(pol), vars(\sigma be_i) \cap (vars(\sigma pe_i) \cup vars(\sigma e)) = \emptyset \Rightarrow P \cup \Sigma \models \sigma be_i$

and its application is a function $e, pol \xrightarrow{P,\Sigma} (PE, BE)$ *such that for all* pe_i, $disunify(pe_i, \theta) = (pe'_i, BE')$

- $PE = \{pe'_i | pe_i \in B(pol), pe'_i \neq pe_i\}$
- $\widetilde{BE} = \{\sigma be_i | be_i \in B(pol) \wedge \exists pe_i : vars(\sigma be_i) \cap (vars(\sigma pe_i) \cup vars(\sigma e)) \neq \emptyset\}$
- $BE = BE' \cup \widetilde{BE} \cup \{\sigma_i | \sigma_i = [X = Y] \wedge (X \in Const \vee Y \in Const)\}$

Intuitively, a policy *pol* is applicable to *e* if the triple the policy is protecting unifies with the path expression and all the contextual predicates and bound boolean expressions (or those not dependent of metadata expressions in the body of the policy) are satisfied. The return value is a set with the path expressions found in the body of the policy and all extracted boolean expressions which have not been evaluated and relate to the path expressions found.

Example 1. Following our example, assuming contextual predicates are satisfied, then pol_4 is applicable to $(Person, foaf : name, Name)$ and returns $(\emptyset, \{[Person! = l3s : tom]\})$. In addition, pol_1, pol_2 and pol_3 are applicable to $(Person, foaf : phone, Phone)$ and returns $(\{Var8, foaf : currentProject, Var9\}, \{[Var8 = Person], [Var9 = l3s : rewerse]\})$, $(\{(Var1, foaf : knows, Var2)\}, \{[Var1 = l3s : alice], [Var2 = Person]\})$ and $(\emptyset, [Person = l3s : alice])$ respectively.

Before we describe the query expansion algorithm, and for sake of clarity, we describe the conditions under a query does not need to be evaluated since the result is empty. Intuitively, a query fails if there does not exist any allowed triple to be returned according to both the query and the applicable policies, that is if there exist a path expression for which no allowed triples exist (disallow by default) or if there exist a path expression for which a policy (which does not depend on path expressions) specifies that no triple is allowed (explicit disallow).

Definition 3. *Given a query* $q = (RF, PE, BE)$*, a set of policy rules P and a state* Σ*, we say that q fails if either of the following two conditions hold:*

- $\exists e \in PE : \nexists pol \in P, H(pol) = allow(T) \wedge \widehat{pol(e)}$
- $\exists e \in PE : \exists pol \in P, H(pol) = disallow(T) \wedge \widehat{pol(e)} \wedge e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$

Let's denote by $append(BE, Conn)$ (resp. $prefix(BE, Conn)$) a function that given a set of boolean expressions BE and a connective (e.g., AND or OR) returns a new boolean expression in which all the elements of BE are enclosed by brackets and connected (resp. prefixed) by $Conn$. The pre-filtering algorithm is defined as follows:

Input:
a query $q = (RF, PE, BE)$, a set of policy rules P and a state Σ
Output:
$PE^{+}_{new} \equiv$ new optional path expressions (from allow policies)
$PE^{-}_{new} \equiv$ new optional path expressions (from disallow policies)
$BE^{+}_{new} \equiv$ conjunction of boolean expressions (from allow policies)
$BE^{-}_{new} \equiv$ conjunction of boolean expressions (from disallow policies)

$policy_prefiltering(q, P)$:
$BE^{+}_{or} \equiv$ disjunction of boolean expressions (from allow policies)
$BE^{-}_{or} \equiv$ disjunction of boolean expressions (from disallow policies)
$P_{app} \equiv$ a set of applicable policies

$PE^{+}_{new} = PE^{-}_{new} = \emptyset$
$\forall e \in PE$
$BE^{+}_{or} = BE^{-}_{or} = \emptyset$
// check allow policies
$P_{app} = \{pol | pol \in P \wedge H(pol) = allow(T) \wedge \widehat{pol(e)}\}$
if $P_{app} = \emptyset$
return query failure // no triples matching e are allowed
if $\exists pol \in P_{app} : e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$
// all triples matching e are allowed without restrictions
else
$\forall pol \in P_{app}$
$e, pol \xrightarrow{P,\Sigma} (PE', BE')$
if $PE' = \emptyset$
$BE^{+}_{or} \cup = append(BE', `AND')$
else if $\exists \theta, \widetilde{PE} \in PE^{+}_{new} : \theta = mgu(\widetilde{PE}, PE')$
$BE^{+}_{or} \cup = append(\theta BE', `AND')$
else
$PE^{+}_{new} \cup = PE'$
$BE^{+}_{or} \cup = append(BE', `AND')$
$BE+_{new} \cup = append(BE^{+}_{or}, `OR')$
// check disallow policies
$P_{app} = \{pol | pol \in P \wedge H(pol) = disallow(T) \wedge \widehat{pol(e)}\}$
if $\exists pol \in P_{app} : e, pol \xrightarrow{P,\Sigma} (\emptyset, \emptyset)$
return query failure // all triples matching e are denied
$\forall pol \in P_{app}$
$e, pol \xrightarrow{P,\Sigma} (PE', BE')$
if $PE' = \emptyset$
$BE^{-}_{or} \cup = append(BE', `AND')$
else if $\exists \theta, \widetilde{PE} \in PE^{-}_{new} : \theta = mgu(\widetilde{PE}, PE')$
$BE^{-}_{or} \cup = append(\theta BE', `AND')$
else
$PE^{-}_{new} \cup = PE'$
$BE^{-}_{or} \cup = append(BE', `AND')$

$BE^{-}_{new} \cup = append(BE^{-}_{or}, `OR')$

Let $copy(RF, PE)$ be a function that copies (replacing previous content) into RF either the variables (for SELECT queries) or the path expressions (for CONSTRUCT queries) from PE.

Input:
a query $q = (RF, PE, BE)$
$PE^{+}_{new} \equiv$ new optional path expressions (from allow policies)
$PE^{-}_{new} \equiv$ new optional path expressions (from disallow policies)
$BE^{+}_{new} \equiv$ conjunction of boolean expressions (from allow policies)
$BE^{-}_{new} \equiv$ conjunction of boolean expressions (from disallow policies)
Output:
an expanded query $q = (RF^{+}, PE^{+}, BE^{+})\ MINUS\ (RF^{-}, PE^{-}, BE^{-})$

$expandQuery(q, PE^{+}_{new}, PE^{-}_{new}, BE^{+}_{new}, BE^{-}_{new})$
$RF^{+} = RF^{-} = copy(RF, PE)$
$PE^{+} = PE \cup prefix(PE^{+}_{new}, `OPT')$
$PE^{-} = PE \cup prefix(PE^{-}_{new}, `OPT')$
$BE^{+} = BE \cup append(BE^{+}_{new}, `AND')$
$BE^{-} = BE \cup append(BE^{-}_{new}, `AND')$

where the connective 'OPT' represents the "optional path expression" modifier in the chosen query language (e.g., '[' and ']' in SeRQL[15]).

Briefly, the algorithm extracts the new path expressions found in the body of the policy rules. It extracts their variable bindings in order to reuse them in case they appear in more than one policy rule. However, if the same path expression is found in policies being applied to different from clauses, then they cannot be reused (since conditions on different expressions are connected conjunctively). After prefiltering each policy, a set of AND boolean expressions are extracted. The set of all boolean expressions from applicable allow policies to one from clause are connected by OR. The set of all boolean expressions applicable to different from clauses are connected by AND. From that query we have to remove the triples affected by disallow policies, which are specified in a similar fashion and added to the query using the MINUS operator.

Example 2. The result of applying the above algorithm to the query in Figure 1 and the policies in Table 1 (assuming that time is 15:00, the requester is 'RecommenderService' and it is trusted) is shown in Figure 2.

5 Implementation and Experiments

In this section we present the implementation of the ideas (and algorithm) presented in previous sections in order to transform an arbitrary RDF repository into an *access controlled RDF repository.* We first briefly describe the general architecture and evaluate the implementation in terms of scalability afterwards.

5.1 Architecture

Our implementation adds an additional layer on top of an arbitrary RDF repository (therefore being suitable for reusability among different ones). Incoming

```
  CONSTRUCT {Person} foaf:name {Name};
         foaf:phone {Phone}; foaf:interest {Document}
  FROM {Person} foaf:name {Name};
     foaf:phone {Phone}; foaf:interest {Document}
   [ {Var1} foaf:knows {Var2} ]
   [ {Var3} rdf:type {Var4}, {Var3} foaf:topic {Var5},
     {Var6} foaf:currentProject {Var7}, {Var7} foaf:topic {Var5} ]
  WHERE ( ( (Person != l3s:tom) ) ) AND
        ( ( (Var2 = Person) AND (Var1 = l3s:alice) ) OR
        ( (Person = l3s:alice) ) ) AND
        ( ( (Var3 = Document) AND (Var2 = Person) AND
          (Person = l3s:alice) AND (Var4 = foaf:Document) ) )
MINUS
  CONSTRUCT {Person} foaf:name {Name};
         foaf:phone {Phone}; foaf:interest {Document}
  FROM {Person} foaf:name {Name};
     foaf:phone {Phone}; foaf:interest {Document}
   [ {Var8} foaf:currentProject {Var9} ]
  WHERE ( ( (Var8 = Person) AND (Var9 = l3s:rewerse) ) )
```

Fig. 2. Expanded RDF query

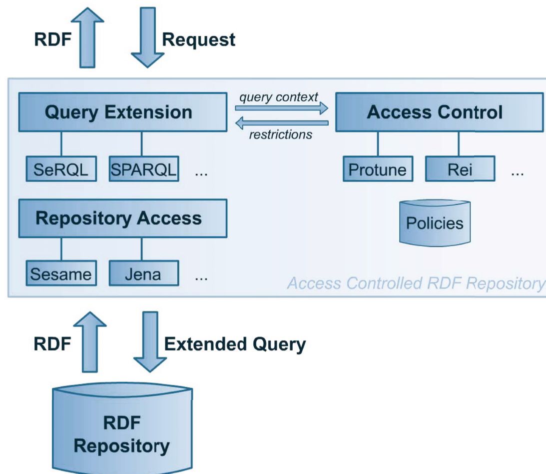

Fig. 3. Architecture - Access Controlled RDF Repository

queries are first processed and extended according to access control policies before they are directed to the underlying RDF repository.

The Architecture of our implementation, illustrated in Figure 3, is composed of three main modules: *Query Extension*, *Policy Engine* and *RDF Repository*.

Query Extension. The main task of this core module is to rewrite a given query in a way that only allowed RDF statements are accessed and returned.

It is in charge of querying the policy engine for each FROM clause of the original query in order and expand it with the extra path expressions and constraints (cf. §4.3). Our initial implementation provides query extension capabilities for the *SeRQL* [13] query language.

Policy Engine. This module is responsible for the policy (partial) evaluation. Input information (*query context*) such as the requester or disclosed credentials may be used as well. In the actual implementation we use the *Protune* policy language [12] and its framework.

RDF Repository. After extending a query the extended query can be passed to the underlying RDF repository. Since our solution is repository-independent, any store supporting SeRQL, such as *Sesame* [2] (which we integrated in our actual implementation), can be used. The result set returned contains only allowed statements and can be directly returned to the requester.

5.2 Experiments and Evaluation

We set up a Sesame database with more than 3,000,000 RDF statements about persons and their exchanged e-mails into an AMD Opteron 2.4GHz with 32GB memory and send the queries through the network from a Dual Pentium 3.00GHz with 2GB. We checked our approach in the "worst" scenario by setting an initial query returning a very large number of results (1,280,000 in this case). We then automatically generated extra path and boolean expressions, extracted from policies as described in §4.3, which were added to the original query. Since we wanted to test the impact of allow and disallow policies in our algorithm, we evaluated both expansion options: with and without the MINUS operator. All these queries were executed and we measured the time needed for its evaluation in order to see its impact. The results are shown in Figure 4.

Both graphs show how adding many WHERE clauses (extracted from allow policies) increases linearly the evaluation time. The reason for this increasement is a) each WHERE clause specifies new triples that are allowed to be accessed and as a consequence, the number of triples that will be returned and b) the new added clauses require time for its evaluation. The reason for the strong increase when adding 6 FROM clauses is that the new clauses produced a triplication in the number of triples to consider, even though none of the new ones were to be returned (so we believe that appropriate optimizations in the repository would help to reduce this impact too). We also made other experiments where the initial query was more selective and the addition of FROM clauses produced only linear increase on the evaluation time.

These results demonstrate that the approach described in this paper scales to a larger number of policies, especially when policies specify only boolean constraints or when the path expressions are selective, since the cost might well be accepted in order to provide fine-grained access control. For low selective queries, optimizations are required. It is also important to note that even if thousands policies are specified, not all will contribute to expand the original query with new expressions. Only those protecting the triples in the FROM

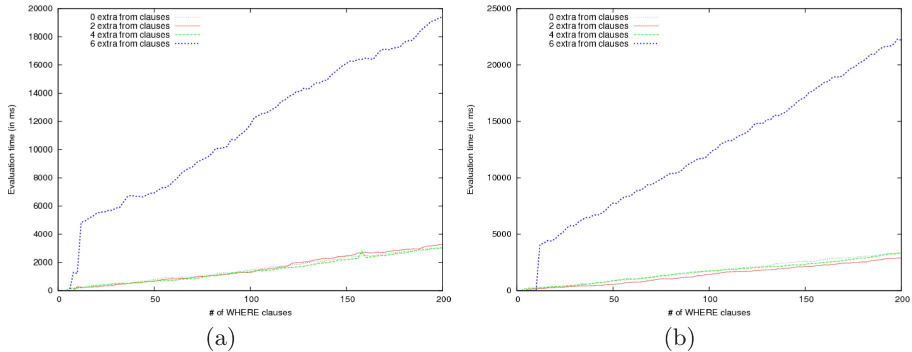

Fig. 4. (a) Response time when increasing the number of FROM and WHERE clauses (with allow policies), and (b) same as before including a MINUS between two queries (allow and disallow policies)

clause of the original query which have satisfied their contextual constraints will be taken into account, therefore reducing the number of applicable policies.

As main conclusions we would highlight that fine-grained access control comes with a cost. However, this cost may be acceptable for semantic web applications and services that must deal with sensitive data. For example, applications that retrieve personal data from the user, as in our Personal Reader scenario, and which use highly selective queries, may benefit from this solution and allow users to define their own policies over their data. In addition, optimizations on the generation of new queries and reordering of constraints as well as natively implemented optimizations in the RDF repository may help to further reduce the response query time.

6 Conclusions and Future Work

Semantic Web applications might require to store and access metadata while still preserving the sensitive nature of such data, especially in multi-service and multi-user environments. However, current RDF stores do not provide access control mechanisms that suit this requirement. In this paper we presented an approach (independent of the RDF store used) for the integration of expressive policies in order to provide a fine-grained access control mechanism for RDF repositories. These policies may state conditions on the RDF nature and content of the RDF store as well as other external (e.g., contextual) conditions. The evaluation of the process is divided in order to pre-evaluate conditions of the policy engine not depending on the RDF store and relying on the highly optimized query evaluation of semantic databases for RDF pattern and content constraints. We presented an implementation of those ideas and showed with our evaluation how the cost of this access control layer scales and might be acceptable for applications requiring fine-grained access control over their (possibly sensitive) data.

We are currently investigating other optimizations in order to improve the evaluation of our implementation such as reordering of constraints in a given query as well as caching techniques. In addition, we are applying and implementing this approach to the SPARQL query language.

Acknowledgements. We thank the anonymous reviewers, Axel Polleres and Piero Bonatti for their feedback and help to improve this paper.

References

1. Abel, F., Baumgartner, R., Brooks, A., Enzi, C., Gottlob, G., Henze, N., Herzog, M., Kriesell, M., Nejdl, W., Tomaschewski, K.: The Personal Publication Reader. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, Springer, Heidelberg (2005)
2. Broekstra, J., Kampman, A., Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, Springer, Heidelberg (2002)
3. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: WWW, NY, USA (2005)
4. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF, http://www.w3.org/TR/rdf-sparql-query/
5. Dietzold, S., Auer, S.: Access control on RDF triple stores from a semantic wiki perspective. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, Springer, Heidelberg (2006)
6. Bizer, C., Oldakowski, R.: Using context- and content-based trust policies on the semantic web. In: WWW, NY, USA (2004)
7. Reddivari, P., Finin, T., Joshi, A.: Policy based access control for a RDF store. In: Proceedings of the Policy Management for the Web Workshop. A WWW 2005 Workshop, W3C, May 2005, pp. 78–83 (2005)
8. Jain, A., Farkas, C.: Secure Resource Description Framework: an access control model. In: ACM SACMAT, CA, USA, ACM Press, New York (2006)
9. Uszok, A., Bradshaw, J.M., Jeffers, R., Suri, N., Hayes, P.J., Breedy, M.R., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In: POLICY (2003)
10. Kagal, L., Finin, T.W., Joshi, A.: A policy language for a pervasive computing environment. In: POLICY (June 2003)
11. Gavriloaie, R., Nejdl, W., Olmedilla, D., Seamons, K.E., Winslett, M.: No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In: Bussler, C.J., Davies, J., Fensel, D., Studer, R. (eds.) ESWS 2004. LNCS, vol. 3053, Springer, Heidelberg (2004)
12. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: POLICY 2005, Stockholm, Sweden (June 2005)
13. Broekstra, J., Kampman, A.: SeRQL: An RDF query and transformation language. (August 2004)
14. Polleres, A.: From SPARQL to rules (and ba ck). In: WWW, Banff, Canada (2007)
15. Aduna: The SeRQL query language (revision 1.2), http://www.openrdf.org/doc/sesame/users/ch06.html