Access Control, Triggers and Versioning over SPARQL Endpoint

Sergey Gorshkov

TriniData, Mashinnaya 40-21,
620089 Ekaterinburg, Russia
serge@trinidata.ru

Abstract. Industrial use of RDF triple stores is facing lack of supplementary functionality such as fine-grained access control, changes approval, triggers and versioning. We have faced industrial use case in which this functionality is essential. The solution is the transparent proxy middleware implemented over SPARQL endpoint. It allows usage of the standard application interface, not requiring any changes in third-party software working with the triple store. It provides all the required functionality by using metadata stored outside of the model, leaving triple store content intact. The general middleware algorithm and some particular workaround are described. Performance slowdown factor is reduced by implementing internal caching for frequently used queries.

Keywords: Access control, SPARQL endpoint, RDF triple store, triggers, versioning, collaborative authoring.

1 Introduction

Expressiveness of semantic models leads to an idea of replacement of relational databases (RDBMS) by RDF triple stores as the data storage for corporate software. One of the most promising areas for such upgrade is the Master Data Management (MDM). Semantically expressed master data, available through SPARQL endpoint, are significantly richer by classification features, attributes model (including multiple values for each object/attribute pair), and methods of use, than any RDBMS-based MDM solution could be.

However, most of the currently available RDF triple stores are lacking of functionality which is standard for RDBMS: access control and triggers. This causes difficulties in implementation of such features as versioning and changes approval, required for MDM. We've enriched advantages of RDF triple stores by implementing RDMBS-like features such as access control, triggers, changes approval and versioning, by building the middleware layer, serving as proxy for SPARQL queries.

To proceed with discussion of our solution, let's look first at the existing experience in triple store access control, triggers implementation and ontology versioning.

P. Klinov and D. Mouromtsev (Eds.): KESW 2014, CCIS 468, pp. 67–75, 2014.

2 Related Work

Since the functionality similar to `GRANT/REVOKE` SQL queries is not yet contained in SPARQL specification [15], we cannot expect the unified implementation of security control at the endpoint level. Existing particular implementations have very limited functionality. The very basic way of securing RDF triple stores is restricting access to specific named graphs [1].

The more flexible method, and as far as we know – the only native implementation, is used in Oracle Triple store. There are two ways of model security control. One of them is label-based: each triple is associated with security label. The same set of labels may be assigned to the user or session, and used for filtering ontology content. The other way of security control assigns security labels to the subjects and predicates, which allows more pragmatic use; however, using this way is not recommended by Oracle [2].

Some specific approaches are developed for particular applications of semantic computing. For example, LiMDAC framework provides complex access rights control mechanism for medical data organized in cubes. However, this platform is generating SPARQL queries by itself, while user requests are formulated by other ways [3]. RAP framework stores access rules using special ontology, which allows very flexible rules definition, but it is also implemented as Web service having its own set of methods [4].

Finally, some authors propose not to develop custom (and non-standard) query interface, but to perform security check over usual SPARQL queries and its results [5]. In this case, the proxy layer between end-user and actual SPARQL endpoint is implemented. The end-user works with it using standard SPARQL endpoint interface, but it requires transfer of some authorization and/or context information along with query. These identification markers are used to determine appropriate access level. It is usual to rely on external authorization methods in this case, such as WebID [6].

An especially interesting and promising approach is proposed by S. Kirrane in [16]. Extrapolating the DAC concept which is accepted as a standard for relational databases, author presents a framework providing similar functionality for RDF data storages, including query language extension with `GRANT/REVOKE`-like operations. However, at the current implementation stage this framework processes only simple queries (no subqueries and aggregates support); also, the rights definition mechanism is extensive, but is not exactly suitable for our practical task.

We might conclude that RDF/SPARQL access control engines could be classified by:

- The level of protection: only output filtration [7], output and `INSERT/DELETE`, or all methods including bulk graph import;
- Protection granularity: named graphs, whole triples, subjects, predicates;
- The way of implementation: built in SPARQL endpoint, framework with custom program interface (often Web-service), or SPARQL proxy;
- Rules evaluation type: use of ACL [8], or defining rules as SPARQL statements, evaluated using ASK query which involves user identification information [9].

The task of ontology versioning is also may be resolved by implementing middleware [10]. In the case mentioned above, middleware combines versioning with access control functionality. Kiryakov and Ognyanov propose to perceive ontology evolution as a set of states, each of them characterized by an identifier and the full ontology image. In their practical implementation of KCS (Knowledge Control System) they propose to extend the schema of underlying database, over which RDF store is implemented. We cannot act this way in our case, not being bound with any particular triple store implementation, and being unable to do any changes at triple store level.

It is useful to implement logical grouping of ontology changes, because a group of triple-level operations may refer to the single logical action, such as editing one object [11]. Some authors are considering Version Control system-like approach (SVN etc) applicable and useful for ontology versioning [12].

We should state that most researchers are focused on ontology comparison problem, rather of tracking versions of the one specific ontology.

Some implementations of trigger-like functionality over RDF stores are developed, although there are a significantly lesser number of such solutions comparing to access control and versioning. A good example is OUL, a standalone application which allows defining handlers for ontology update events [13]. However, in this case handlers can only perform cascading updates within the ontology. We are interested in handlers which could perform external actions, such as firing events to external applications.

We have concluded that there is no single solution that can fulfill all of our functional requirements “out of the box”; various implementations are having their strong and weak points, but none of them have the balance required for our use case. Especially, we’ve strongly needed the triggers implementation, while there is no single product offering this functionality at the satisfactory level, in conjunction with access control/versioning. The task of ontology changes approval/moderation isn’t resolved in the solutions we’ve reviewed. So we’ve chosen the way of creating our own implementation from the scratch, keeping in mind the task of facilitating further development and providing necessary level of functional flexibility of the solution.

3 Motivation

Our task is to implement all-in-one middleware covering all the tasks mentioned above. The main ideas of this middleware are:

- It should allow third-party software to work with the endpoint not being aware of proxy existence – that is, it should implement standard SPARQL interface.
- It should not affect the model itself.

Our development has been conducted in the context of industrial Master Data Management system implementation. The Triple Store (Apache Jena / Fuseki) is considered as the master data storage. It is surrounded by a number of program components and interfaces providing master data query and update functionality. Ontology management software (Onto.pro, web-based ontology editor, which access ontology using SPARQL queries) is used by a number of users having different roles. Model update policies require that:

- Users could be granted access for read, propose updates, or update without confirmation instances of particular classes, and/or classes and attributes definitions.
- Some users may approve or reject changes proposed by another, in case if they have sufficient rights to perform proposed change.
- Every change in the model should be registered in the log, which is accessible by third-party software in a programmatic way.
- Even accepted changes could be reviewed later and rolled back if necessary.
- A wide number of external program components should access the model by performing SPARQL queries over the endpoint, or by using special wrapper implemented as SOAP web-service. These programs might be granted rights for requesting instances of particular classes, and their properties.
- The master data storage should immediately notify external applications on ontology elements update. Those applications might subscribe on notifications using list of classes of interest.

All these policies have produced the following functional requirements:

- A middleware should be created to wrap the standard endpoint SPARQL interface to provide all the required functionality.
- Master data storage (triple store) should allow access control both for read and update, depending on rights defined for accessing user or program component.
- If user trying to perform some update has the right only to propose changes – the actual model should not be modified, but proposed change should be placed in the queue for review by the users who have approval rights.
- All the changes made in the model should be written in the log, which will allow to review and rollback every change.
- Trigger-like notification mechanism should be implemented.
- External applications not aware of accessing secure parts of the master data, or performing updates, should work with the middleware without supplying any authorization information and work with it like with standard SPARQL endpoint interface.

4 Implementation

We see the only way of implementation of all the mentioned requirements by developing a middleware which will rewrite SPARQL queries. List of the actions need to be performed on each type of query should look as shown in the table 1.

Other query types were not considered due to conditions of particular use case.

As we have mentioned, we're assigning access rules for the user group / class pairs. It means that each rule is telling that the users of a particular group may (or may not) perform some operations with the objects belonging to some class. List of the operations is limited to: read, update with confirmation (moderation), update. As we have only one level of rights definition, the conflict resolution policy is simple: the strictest of the applicable rules is always selected. However, such a model of rights assignment is the result of our practical task conditions, not of some built-in restrictions, so it easily can be extended.

Table 1. Types of query

Query type	Query processing	Results processing
`SELECT`, `ASK`, `CONSTRUCT`		Filter results by removing information on all objects that cannot be accessed
`INSERT`, `DELETE`, `UPLOAD`	Check for URIs of objects that cannot be updated (at the subject position), reject query if found. Place proposed changes into queue for approval, if needed. Keep previous version of the data and log action. Send notifications on query complete.	Notify application if query was not executed due to access rights restrictions.

Filtering results of `SELECT` query is split into two parts: query and results processing. Query processing, at first glance, implies checking access rights to all the objects mentioned in it. The object rights checking, in its turn, is a process of finding all its classifications (including inferred ones), and checking limitations for the current user against these classes. If a prohibited object is found, the whole query has to be cancelled. Subqueries are processed separately at the query processing stage: they are extracted at the first step, and processed recursively.

Response processing includes filtering result set line-by-line, removing the objects which cannot be read, and then rebuilding the whole response structure.

In some cases, filtering requires a special workaround. For example, the query result might not contain URI of prohibited object, but contain its properties. Consider the following query:

```
SELECT ?prop WHERE { ?object <has_property_1> "some val-
ue".
      ?object <has_property_2> ?prop }
```

Imagine that this pattern will match some prohibited object for the "object" variable. But in this case, URI of the prohibited object will not be contained in query nor result. We are rewriting such queries by setting the projection to all variables – it means replacing variables list with the *, so object URI will be present in the result acquired by middleware. After results filtering, the set of returning variables is reduced to the one defined in the query (the projection requested by the initial query).

Similar workaround is applied for `COUNT(*)` queries. `COUNT(*)` is replaced with the *, returned rows are cleared, and the size of the resulting rows set is returned.

All these algorithms are increasing the retrieved result set and thus reducing performance, but this can be overrun in general only by simplifying the functional requirements, which was unacceptable in our case. So we were trying to find another solution to compensate performance loss, which we will describe further.

Almost the same logic is applied for `DELETE WHERE` queries: such queries are first rewritten to extract all the triples going to be deleted, and if access check control does not pass for any of affected triples – the whole operation is rejected. We should note that data retrieve operations might be performed partially (the returned results are filtered), but ontology update operations are approved or rejected only in whole. Moreover, due to absence of `UPDATE` query in SPARQL 1.1 standard, model updates are always performed by the pairs of consequent `DELETE / INSERT` queries. These queries should be approved, placed for moderation or rejected only in pairs. Such paired queried needs to be identified and processed by other way than pure `INSERT` or `DELETE` operations. All these algorithms have been implemented in our middleware.

To be able to control access rights, we need to identify user/application account. Non-identified user may access only elements of ontology for which access rules are not defined (non-secured objects). The application pretending to access secured objects should pass authentication information as session parameters. List of user accounts, groups and applicable restrictions is stored in the meta-data database.

Because middleware shouldn't store metadata in the model, it will need additional data storage for it. We have used relational database for this purpose. The interface of Onto.pro, which becomes administration application for the middleware, was modified to work with this metadata (access rights and notifications settings, model changes approval, change log review and rollback).

SPARQL queries rewrite is a challenging task in general [14], so it is useful to take some assumptions, which should decrease rewritten queries size and complexity, and simplify rewrite process to reduce computational cost. In our case, these assumptions are derived from the access rules definition logic.

As our primary task was to control access to the instances of particular classes, access level calculation may be represented as the following simple algorithm:

- Identify all the classes that the processed object belongs to, including standard-defined types such as `owl:Class` or `owl:Property` (restrictions might be defined for these types too). List of classes should be built taking into account indirect classifications caused by model rules (`rdfs:subClassOf` etc).
- Find the most strict access level for the classes of this list, and apply it to the requested operation.

One of pragmatic uses of notifications feature is connecting semantic MDM with the Enterprise Service Bus (ESB). In our use case, MDM should fire events to ESB on every update of the ontology, to notify applications that are possibly using changed objects. Implementation of this feature is rather obvious: when the `INSERT` or `DELETE` queue affecting the object of the monitored type is executed, the middleware is placing this event to the internal notifications queue. The paired `DELETE+INSERT` queries are recognized and merged into the single event. The separate notifications handler process is sending events from this queue as the data packages over the bus, or simply writes some information to the external RDBMS. The data package describing an event contains its type (create/update or delete), affected object identification and classification, assigned values of the attributes. The bus can then route this package according to the business rules.

The operations log is written to the RDBMS. It allows to quickly find out all the events affecting some object, and thus to display its changes history (including author of every change), and restore any of the previous states, if necessary.

5 Benchmark

Obviously, middleware layer is reducing performance. We have considered that slowdown of ontology update operations is not critical because of rare changes in the master data. But data retrieval speed, in general, should not be significantly affected by the middleware. To reduce middleware impact on the speed, we have performed frequency analysis of the queries performed over the model by the consuming applications. This analysis had shown that more than 50% of the queries are of two types: "A is subclass of B", and "C is a member of class A". This led us to the idea of implementing caching of those queries. Cache is used for both answering client application's queries, and for internal use by the access level computation algorithm. Necessary procedures for cache renewal on ontology or access rights update were developed. Caching has allowed to almost eliminate slowdown impact of the middleware. It is interesting to compare middleware impact on queries execution speed both with and without caching.

Because our middleware was created for the specific task and specific environment, our benchmarking program is closely related with the supposed way of its use. We have recorded actual query log for two most typical scenarios of our ontology usage: one for model update, and another for data retrieval by the average user session. The update operation have consisted of several `DELETE+INSERT` queue pairs, reflecting the object editing operation in the Onto.pro editor. The `SELECT` operations were taken from the Wiki page generation procedure, which queries all the properties and classifications of the displayed object, and all the objects it is related to. The mentioned logs then were passed through the middleware several times to record execution time.

The results are presented in the table 2. The cells are containing execution slowdown, in percent relative to the direct SPARQL endpoint request.

Table 2. Middleware layer performance

Operation	Using Middleware	Using Middleware (caching is off)
`DELETE+INSERT`	+595%	+750%
`SELECT`	**+13%**	+115%

We have not considered another query types in the experiment, but it is clear that `ASK/CONSTRUCT` queries will show the performance similar to the `SELECT`, as they are processed by the same code.

The relative slowdown factor we've showed above does not significantly depends on the data set size, at least in the conditions of our tests (up to 100 000 triples).

Update operations are expectedly slowed down in several times, due to necessity of extensive security checks, logging, triggers firing and cache update. The interesting fact, however, is that the `SELECT` query performance is reduced only by 13%. Comparison to the right table column proves that introducing cache has allowed us to almost compensate impact of security check algorithm which is performed over each `SELECT` query.

However, these results were obtained on the particular industrial ontology example with the use-case specific set of business rules. As the number and complexity of the rules are significant for security check duration, in the other cases performance may vary.

6 Conclusions

We have developed the middleware framework for transparent SPARQL queries processing. It provides functionality absent in current triple store implementations, but required by the business processes. The middleware does not place any metadata in the model storage, and does not require non-standard applications interface (if the application does not pretend accessing secured objects). Performance reduce for `SELECT` operations is almost eliminated by implementing caching of the most frequently used queries, and those required for the access rights computation algorithm.

Further work includes improving performance of `INSERT/DELETE` operations, and extending access rights rules definition logic.

References

1. Costabello, L., Villata, S., Gandon, F.: Context-Aware Access Control for RDF Graph Stores. In: 20th European Conference on Artificial Intelligence (2012)
2. Fine-Grained Access Control for RDF Data, `http://docs.oracle.com/cd/E16655_01/appdev.121/e17895/fine_grained_acc.htm`
3. Kamateri, E., Kalampokis, E., Tambouris, E., Tarabanis, K.: The Linked Data Access Control Framework. Journal of Biomedical Informatics. Special Issue on Informatics Methods in Medical Privacy (2014)
4. Reddivari, P., Finin, T., Joshi, A.: Policy-Based Access Control for an RDF Store. In: Proceedings of the Policy Management for the Web Workshop, A WWW 2005 Workshop (2005)
5. Abel, F., De Coi, J.L., Henze, N., Koesling, A.W., Krause, D., Olmedilla, D.: Enabling Advanced and Context-Dependent Access Control in RDF Stores. In: Aberer, K., et al. (eds.) ISWC/ASWC 2007. LNCS, vol. 4825, pp. 1–14. Springer, Heidelberg (2007)
6. Sacco, O., Passant, A., Decker, S.: An Access Control Framework for the Web of Data. In: International Joint Conference of IEEE TrustCom 2011/IEEE ICESS 2011/FCST 2011 (2011)
7. Flouris, G., Fundulaki, I., Michou, M., Antoniou, G.: Controlling Access to RDF Graphs. In: Proceedings of the Third Future Internet Conference on Future Internet, Berlin, Germany, pp. 107–117 (2010)

8. Hollenbach, J., Presbrey, J., Berners-Lee, T.: Using RDF Metadata to Enable Access Control on the Social Semantic Web. In: Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (2009)
9. Costabello, L., Villata, S., Delaforge, N., Gandon, F.: Ubiquitous Access Control for SPARQL Endpoints: Lessons Learned and Future Challenges. In: Proceedings of the 21st International Conference Companion on World Wide Web, Lyon, France (2012)
10. Ognyanov, D., Kiryakov, A.: Tracking Changes in RDF(S) Repositories. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS (LNAI), vol. 2473, pp. 373–378. Springer, Heidelberg (2002)
11. Auer, S., Herre, H.: A Versioning and Evolution Framework for RDF Knowledge Bases. In: Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics, Novosibirsk, Russia (2006)
12. Noy, N., Musen, M.: Ontology Versioning in an Ontology Management Framework. IEEE Intelligent Systems 19(4), 6–13 (2004)
13. Sangers, J., Hogenboom, F., Frasincar, F.: Event-Driven Ontology Updating. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 44–57. Springer, Heidelberg (2012)
14. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting Queries on SPARQL Views. In: Proceedings of the 20th International Conference on World Wide Web, Hyderabad, India (2011)
15. Kirrane, S., Mileo, A., Decker, S.: Applying DAC Principles to the RDF Graph Data Model. In: 28th IFIP TC-11 SEC International Information Security and Privacy Conference (2013)
16. Kirrane, S., Abdelrahman, A., Mileo, A., Decker, S.: Secure Manipulation of Linked Data. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 248–263. Springer, Heidelberg (2013)