A Token-Based Access Control System for RDF Data in the Clouds

Arindam Khaled*
Computer Science Department
Mississippi State University
ak697@msstate.edu

Mohammad Farhan Husain[1]
Latifur Khan[2]
Kevin W. Hamlen[3]
Bhavani Thuraisingham[4]
Department of Computer Science
University of Texas at Dallas
{[1]mfh062000, [2]lkhan}@utdallas.edu
{[3]hamlen, [4]bhavani.thuraisingham}@utdallas.edu

Abstract

The Semantic Web is gaining immense popularity—and with it, the Resource Description Framework (RDF) broadly used to model Semantic Web content. However, access control on RDF stores used for single machines has been seldom discussed in the literature. One significant obstacle to using RDF stores defined for single machines is their scalability. Cloud computers, on the other hand, have proven useful for storing large RDF stores; but these systems lack access control on RDF data to our knowledge.

This work proposes a token-based access control system that is being implemented in Hadoop (an open source cloud computing framework). It defines six types of access levels and an enforcement strategy for the resulting access control policies. The enforcement strategy is implemented at three levels: Query Rewriting, Embedded Enforcement, and Post-processing Enforcement. In Embedded Enforcement, policies are enforced during data selection using MapReduce, whereas in Post-processing Enforcement they are enforced during the presentation of data to users. Experiments show that Embedded Enforcement consistently outperforms Post-processing Enforcement due to the reduced number of jobs required.

1 Introduction

The Semantic Web is becoming increasingly ubiquitous. More small and large businesses, such as Oracle, IBM, Adobe, Software AG, and many others, are actively using Semantic Web technologies [22], and broad application areas such as Health Care and Life Sciences are considering its possibilities for data integration [22]. Sir Tim Berners-Lee originally envisioned the Semantic Web as a machine-understandable web [3]. The power of the Semantic Web lies in its codification of relationships among web resources [22].

Semantic Web, along with ontologies, is one of the most robust ways to represent knowledge. An *ontology* formally describes the concepts or classes in a domain, various properties of the classes, the relationships between classes, and restrictions. A knowledge base can be constructed by an ontology and its various class instances. An example of a knowledge base (ontology and its instance) is presented in Figure 1.

Resource Description Framework (RDF) is widely used for Semantic Web due to its expressive power, semantic interoperability, and reusability. Most RDF stores in current use, including Joseki [15], Kowari [17], 3store [10], and Sesame [5], are not primarily concerned with security. Efforts have been made to incorporate security, especially in Jena [14, 20]; however, one drawback of Jena is that it lacks scalability. Its execution times can become quite slow with larger data-sets, making certain queries over large stores intractable (e.g., those with 10 million triples or more) [12, 13].

On the other hand, large RDF stores can be stored and retrieved from cloud computers due to their scalability, parallel processing ability, cost effectiveness, and availability. Hadoop (Apache) [1]—one of the most widely used cloud computing environments—uses Google's MapReduce framework. MapReduce splits large jobs into smaller jobs, and combines the results of these jobs to produce the final output once once the sub-jobs are complete. Prior work has demonstrated that large RDF graphs can be efficiently stored and queried in these clouds [6, 12, 13, 18]. To our knowledge, access control has not yet been implemented on RDF stores in Hadoop. Doing so is the subject of this work.

Our system implements a token-based access control system. System administrators grant access tokens for

*The work was done while the author was a graduate student at the University of Texas at Dallas.

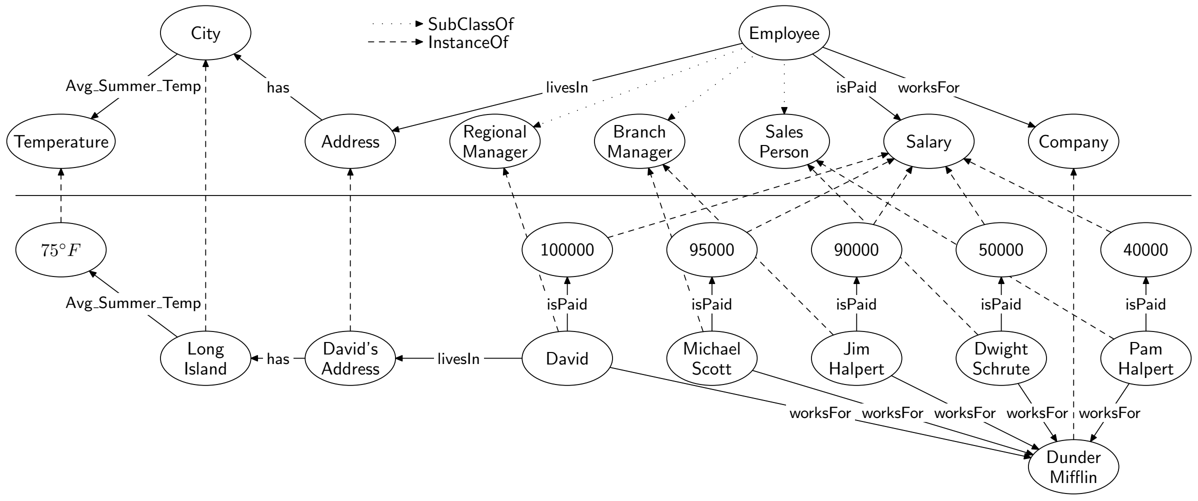

Figure 1. A sample RDF ontology and ontology instance

security-relevant data according to agents' needs and security levels. Conflicts that might arise due to the assignment of conflicting access tokens to the same agent are resolved using the timestamps of the access tokens. We use the Lehigh University Benchmark (LUBM) [9] test instances for experiments. A few sample scenarios have been generated and implemented in Hadoop.

We have several contributions. First, we propose an architecture that scales well to extremely large data-sets. Second, we address access control not only at the level of users but also at the level of subjects, objects, and predicates, making policies finer-grained and more expressive than past work. Third, a timestamp-based conflict detection and resolution algorithm is proposed. Fourth, the architecture has been implemented and tested on benchmark data in several alternative stages: Query Rewriting (preprocessing phase), Embedded Enforcement (MapReduce execution phase), and Post-processing Enforcement (data display phase). Finally, the whole system is being implemented on Hadoop—an open source cloud computing environment. We consider the work beneficial for others considering access control for RDF data in Hadoop.

The remainder of the paper is organized as follows. In Section 2, we present related work and a brief overview of Hadoop and MapReduce. Section 3 introduces access tokens, access token tuples, conflicts, and our conflict resolution algorithm. We describe the architecture of the our system in Section 4. In Section 5, we describe the impact of assigning access tokens to agents, including experiments and their running times. Finally, Section 6 concludes with a summary and suggestions for future work.

2 Background

2.1 Related Work

We begin by describing prior work on RDF Security for single machines. We then summarize some of the proposed Cloud Computing architectures that store RDF data. Finally, we provide a summary of our own prior work.

Although plenty of research has been undertaken on storing, representing, and reasoning about RDF knowledge, research on security and access control issues for RDF stores is comparatively sparse [20]. Reddivari et al. [20] have implemented access control based on a set of policy rules. They address insertion/deletion actions of triples, models, and sets in RDF stores, as well as see and use actions. Jain and Farkas [14] have described RDF protection objects as RDF patterns, and designed security requirements for them. They show that the security level of a subclass or sub-property should be at least as restricted as the supertype. The RDF triple-based access control model proposed in [16] considers explicit and implicit authorization propagation.

Most of these works are implemented in Jena. However, Jena scales poorly in that it runs on single machines and is unable to handle large amounts of data [12, 13]. Husain et al. [12, 13] propose and implement an architecture to store and query large RDF graphs. Mika and Tummarello [18] store RDF data in Hadoop. The SPIDER system [6] stores and processes huge RDF data-sets, but lacks an access control mechanism.

Our proposed architecture supports access control for large data-sets by including an access control layer in the architecture proposed in [13]. Instead of assigning access controls directly to users or agents, our proposed method generates tokens for specific access levels and assigns these tokens to agents, considering the business needs and security levels of the agents. Although tokens have been used by others for access control to manage XML documents [4] and digital information [11], these have not been used for RDF stores. One of the advantages of using tokens is that they can be reused if the needs and security requirements for multiple agents are identical.

2.2 Hadoop and MapReduce

In this section we provide a brief overview of Hadoop [1] and MapReduce. In Hadoop, the unit of computation is called a *job*. Users submit jobs to Hadoop's JobTracker component. Each job has two phases: Map and Reduce. The Map phase takes as input a key-value pair and may output zero or more key-value pairs. In the Reduce phase, the values for each key are grouped together into collections traversable by an iterator. These key-iterator pairs are then passed to the Reduce method, which also outputs zero or more key-value pairs. When a job is submitted to the JobTracker, Hadoop attempts to position the Map processes near to the input data in the cluster. Each Map process and Reduce process works independently without communicating. This lack of communication is advantageous for both speed and simplicity.

3 Access Control Levels

Definition 1. *Access Tokens (AT)* permit access to security-relevant data. An agent in possession of an AT may view the data permitted by that AT. We denote AT's by positive integers.

Definition 2. *Access Token Tuples (ATT)* have the form $\langle AccessToken, Element, ElementType, ElementName\rangle$, where $Element$ can be $Subject$, $Object$, or $Predicate$, and $ElementType$ can be described as URI, $DataType$, $Literal$, $Model$, or $BlankNode$. $Model$ is used to access Subject Models, and will be explained later in the section.

For example, in the ontology in Figure 1, $David$ is a subject and $\langle 1, Subject, URI, David\rangle$ is an ATT. Any agent having AT 1 may retrieve $David$'s information over all files (subject to any other security restrictions governing access to URI's, literals, etc., associated with $David$'s objects). While describing ATT's for $Predicates$, we leave the $ElementName$ blank (_).

Based on the record organization, we support six access levels along with a few sub-types described below. Agents may be assigned one or more of the following access levels. Access levels with a common AT combine conjunctively, while those with different AT's combine disjunctively.

1. **Predicate Data Access:** If an object type is defined for one particular predicate in an access level, then an agent having that access level may read the whole predicate file (subject to any other policy restrictions). For example, $\langle 1, Predicate, isPaid, \_\rangle$ is an ATT that permits its possessor to read the entire predicate file $isPaid$.

2. **Predicate and Subject Data Access:** Agents possessing a Subject ATT may access data associated with a particular subject, where the subject can be either a URI or a $DataType$. Combining one of these Subject ATT's with a Predicate data access ATT having the same AT grants the agent access to a specific subject of a specific predicate. For example:

   (a) **Predicate and Subject as URI's:** Combining ATT's $\langle 1, Predicate, isPaid, \_\rangle$ and $\langle 1, Subject, URI, MichaelScott\rangle$ (drawn from the ontology in Figure 1) permits an agent with AT 1 to access a subject with URI $MichaelScott$ of predicate $isPaid$.

   (b) **Predicate and Subject as DataTypes:** Similarly, Predicate and DataType ATT's can be combined to permit access to subjects of a specific data type over a specific predicate file.

   For brevity, we omit descriptions of the different Subject and Object variations of each of the remaining access levels.

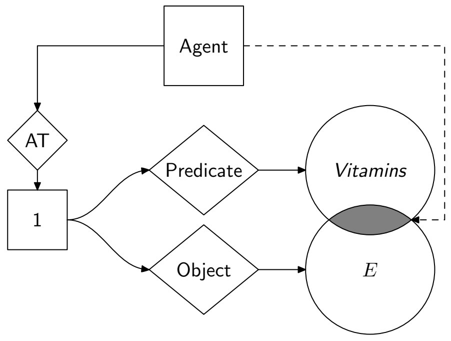

Figure 2. Conjunctive combination of ATT's with a common AT

3. **Predicate and Object:** This access level permits a principal to extract the names of subjects satisfying a particular predicate and object. For example, with ATT's $\langle 1, Predicate, hasVitamins, \_\rangle$ and $\langle 1, Object, URI, E\rangle$, an agent possessing AT 1 may view the names of subjects (e.g., foods) that have vitamin E. More generally, if X_1 and X_2 are the set of triples generated by Predicate and Object triples (respectively) describing an AT, then agents possessing the AT may view set $X_1 \cap X_2$ of triples. An illustration of this example is displayed in Figure 2.

4. **Subject Access:** With this access level an agent may read the subject's information over all the files. This is one of the less restrictive access levels. The subject can be a URI, $DataType$, or $BlankNode$.

5. **Object Access:** With this access level an agent may read the object's subjects over all the files. Like the

previous level, this is one of the less restrictive access levels The object can be a URI, $DataType$, $Literal$, or $BlankNode$.

6. **Subject Model Level Access:** Model level access permits an agent to read all necessary predicate files to obtain all objects of a given subject. Of these objects, the ones that are URI's are next treated as subjects to extract their respective predicates and objects. This process continues iteratively until all objects finally become literals or blank nodes. In this manner, agents possessing Model level access may generate models on a given subject.

   The following example drawn from Figure 1 illustrates. $David$ lives in $LongIsland$. $LongIsland$ is a subject with an Avg_Summer_Temp predicate having object $75°F$. An agent with Model level access of $David$ may therefore read the average summer temperature of $LongIsland$.

3.1 Access Token Assignment

Definition 3. An *Access Token List* (AT-list) is an array of one or more AT's granted to a given agent, along with a timestamp identifying the time at which each was granted. A separate AT-list is maintained for each agent.

When a system administrator decides to add an AT to an agent's AT-list, the AT and timestamp are first stored in a temporary variable $TempAT$. Before committing the change, the system must first detect potential conflicts in the new AT-list.

3.2 Final output of an Agent's ATs

Each AT permits access to a set of triples. We refer to this set as the AT's *result set*. The set of triples accessible by an agent is the union of the result sets of the AT's in the agent's AT-list. Formally, if $Y_1, Y_2, \dots, Y_n$ are the result sets of AT's $AT_1, AT_2, \dots, AT_n$ (respectively) in an agent's AT-list, then the agent may access the triples in set $Y_1 \cup Y_2 \cup \dots \cup Y_n$.

3.3 Security Level Defaults

An administrator's AT assignment burden can be considerably simplified by conservatively choosing default security levels for data in the system. In our implementation, all items in the data store have default security levels. Personal information of individuals is kept private by denying access to any URI of data type $Person$ by default. This prevents agents from making inferences about any individual to whom they have not been granted explicit permission. However, if an agent is granted explicit access to a particular type or property, the agent is also granted default access to the subtypes or sub-properties of that type or property.

As an example, consider a predicate file $Likes$ that lists elements that an individual likes. Assume further that Jim is a person who likes $Flying$, $SemanticWeb$, and $Jenny$, which are URI's of type $Hobby$, $ResearchInterest$, and $Person$, respectively, and 1 is an AT with ATTs $\langle 1, Subject, URI, Jim\rangle$ and $\langle 1, Likes, Predicate, \_\rangle$. By default, agent Ben having only AT 1 cannot learn that $Jenny$ is in Jim's $Likes$-list since $Jenny$'s data type is $Person$. However, if Ben also has AT 2 described by ATT $\langle 2, Object, URI, Jenny\rangle$, then Ben will be able to see $Jenny$ in Jim's $Likes$-list.

3.4 Conflicts

A conflict arises when the following three conditions occur: (1) An agent possesses two AT's 1 and 2, (2) the result set of AT 2 is a proper subset of AT 1, and (3) the timestamp of AT 1 is earlier than the timestamp of AT 2. In this case the later, more specific AT supersedes the former, so AT 1 is discarded from the AT-list to resolve the conflict. Such conflicts arise in two varieties, which we term *subset conflicts* and *subtype conflicts*.

A subset conflict occurs when AT 2 is a conjunction of ATT's that refines those of AT 1. For example, suppose AT 1 is defined by ATT $\langle 1, Subject, URI, Sam\rangle$, and AT 2 is defined by ATT's $\langle 2, Subject, URI, Sam\rangle$ and $\langle 2, Predicate, HasAccounts, \_\rangle$. In this case the result set of AT 2 is a subset of the result set of AT 1. A conflict will therefore occur if an agent possessing AT 1 is later assigned AT 2. When this occurs, AT 1 is discarded from the agent's AT-list to resolve the conflict.

Subtype conflicts occur when the ATT's in AT 2 involve data types that are subtypes of those in AT 1. The data types can be those of subjects, objects or both.

Conflict resolution is summarized by Algorithm 1. Here, $\text{Subset}(AT_1, AT_2)$ is a function that returns $true$ if the result set of AT_1 is a proper subset of the result set of AT_2, and $\text{SubjectSubType}(AT_1, AT_2)$ returns $true$ if the subject of AT_1 is a subtype of the subject of AT_2. Similarly, $\text{ObjectSubType}(AT_1, AT_2)$, decides subtyping relations for objects instead of subjects.

4 Proposed Architecture

Our architecture consists of two components. The upper part of Figure 3 depicts the data preprocessing component, and the lower part shows the components responsible for answering queries.

Three subcomponents perform data generation and preprocessing. We convert RDF/XML [2] to N-Triples serialization format [7] using our N-Triples Converter component. The Predicate Split (PS) component takes the N-Triples data and splits it into predicate files. These steps

```
Input: AT $newAT$ with timestamp $TS_{newAT}$
Result: Detect conflict and, if none exists, add
        $(newAT, TS_{newAT})$ to the agent's AT-list
1  $currentAT[\,]$ ← the AT's and their timestamps;
2  if (!Subset(newAT , tempATTS) AND
   !Subset(tempATTS , newAT) AND
   !SubjectSubType(newAT, tempATTS)) AND
   !SubjectSubType(tempATTS, newAT) AND
   !ObjectSubType(newAT, tempATTS)) AND
   !ObjectSubType(tempATTS, newAT)) then
3      $currentAT[length_{currentAT}].AT \leftarrow newAT$;
4      $currentAT[length_{currentAT}].TS \leftarrow TS_{newAT}$;
5  else
6      $count \leftarrow 0$;
7      while $count < length_{currentAT}$ do
8          AT $tempATTS \leftarrow currentAT[count].AT$;
9          $tempTS \leftarrow currentAT[count].TS$;
10         /* the timestamp during the AT assignment */
11         if (Subset(newAT , tempATTS) AND
           $(TS_{newAT} \geq tempTS)$) then
12             /* a conflict occurs */
13             $currentAT[count].AT \leftarrow newAT$;
14             $currentAT[count].TS \leftarrow TS_{newAT}$;
15         else if ((Subset(tempATTS, newAT)) AND
           $(tempTS < TS_{newAT})$) then
16             $currentAT[count].AT \leftarrow newAT$;
17             $currentAT[count].TS \leftarrow TS_{newAT}$;
18         else if ((SubjectSubType(newAT, tempATTS)
           OR ObjectSubType (newAT, tempATTS)) AND
           $TS_{newAT} \geq tempTS$) then
19             /* a conflict occurs */
20             $currentAT[count].AT \leftarrow newAT$;
21             $currentAT[count].TS \leftarrow TS_{newAT}$;
22         else if ((SubjectSubType(tempATTS, newAT)
           OR ObjectSubType (tempATTS, newAT)) AND
           $(tempATTS < TS_{newAT})$) then
23             $currentAT[count].AT \leftarrow newAT$;
24             $currentAT[count].TS \leftarrow TS_{newAT}$;
25         end
26         $count \leftarrow count + 1$;
27     end
28 end
```

Algorithm 1: Conflict detection and resolution

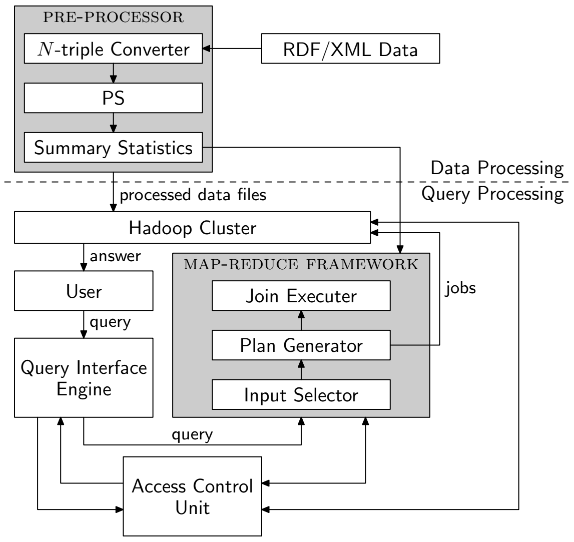

Figure 3. The system architecture

are described in Section 4.1. The output of the last component is then used to gather summary statistics, which are delivered to the Hadoop File System (HDFS).

The bottom part of the architecture shows the Access Control Unit and the MapReduce framework. The Access Control Unit takes part in different phases of query execution. When the user submits a query, the query is rewritten (if possible) to enforce one or more access control policies. The MapReduce framework has three sub-components. It takes the rewritten SPARQL query from the query interface engine and passes it to the Input Selector and Plan Generator. This component selects the input files, decides how many jobs are needed, and passes the information to the Job Executer component, which submits corresponding jobs to Hadoop. The Job Executer component communicates with the Access Control Unit to get the relevant policies to enforce, and runs jobs accordingly. It then relays the query answer from Hadoop to the user. To answer queries that require inferencing, we use the Pellet OWL Reasoner. The policies are stored in the HDFS and loaded by the Access Control Unit each time the framework loads.

4.1 Data Generation and Storage

We use the LUBM [9] dataset for our experiments. This benchmark dataset is widely used by researchers [8]. The LUBM data generator generates data in RDF/XML serialization format. This format is not suitable for our purpose because we store data in HDFS as flat files. If the data is in RDF/XML format, then to retrieve even a single triple we need to parse the entire file. Also, RDF/XML format is not suitable as an input for a MapReduce job. Therefore we store data as N-Triples, because with that format we have

Table 1. Sample data for an LUBM query

type		ub:advisor		ub:takesCourse		ub:teacherOf	
GS_1	Student	GS_2	A_2	GS_1	C_2	A_1	C_1
GS_2	Student	GS_1	A_1	GS_3	C_1	A_2	C_2
GS_3	Student	GS_3	A_3	GS_3	C_3	A_3	C_3
GS_4	Student	GS_4	A_4	GS_2	C_4	A_4	C_4
				GS_1	C_1	A_5	C_5
				GS_4	C_2		

a complete RDF triple (Subject, Predicate, and Object) in one file line, which is very convenient for MapReduce jobs. We therefore convert the data to N-Triple format, partitioning the data by predicate. This step is called *PS*. In real-world RDF datasets, the number of distinct predicates is no more than 10 or 20 [21]. This partitioning reduces the search space for any SPARQL query that does not contain a variable predicate [19]. For such a query, we can just pick a file for each predicate and run the query on those files only. We name the files by predicate for simplicity; e.g., all the triples containing predicate *p1:pred* are stored in a file named *p1-pred*. A more detailed description of this process is provided in [13].

4.2 Example Data

Table 1 shows sample data for three predicates. The leftmost column shows the type file for *student* objects after the PS step. It lists only the subjects of the triples having *rdf:type* predicate and *student* object. The rest of the columns show the *advisor*, *takesCourse*, and *teacherOf* predicate files after the PS step. Each row has a subject-object pair. In all cases, the predicate can be retrieved from the filename.

4.3 Policy Enforcement

Our MapReduce framework enforces policies in two phases. Some policies can be enforced by simply rewriting a SPARQL query during the query parsing phase. The remaining policies can be enforced in the query answering phase in two ways. First, we can enforce the policies as we run MapReduce jobs to answer a query. Second, we can run the jobs for a query as if there is no policy to enforce, and then take the output and run a set of jobs to enforce the policies. These post-processing jobs are called *filter* jobs. In both cases, we enforce predicate-level policies while we select the input files by the Input Selector. In the following sections we discuss these approaches in detail.

4.3.1 Query Rewriting

Policies involving predicates can be enforced by rewriting a SPARQL query. This involves replacing predicate variables by the predicates to which a user has access. An example illustrates. Suppose a user's AT-list consists of AT 1

```
SELECT ?o WHERE            SELECT ?o WHERE
{ A ?p ?o }        ⟹      { A takesCourse ?o }
```

Figure 4. A SPARQL query before and after rewriting

described by ATT $\langle 1, Predicate, takesCourses, \_\rangle$ (i.e., the user may only access predicate file $takesCourse$). If the user submits the query on the left of Figure 4, we can replace predicate variable $?p$ with *takesCourse*. The rewritten query is shown on the right of the figure.

After query is rewritten we can answer the query in two ways, detailed in the following two sections.

4.3.2 Embedded Enforcement

In this approach, we enforce the policies as we answer a query by Hadoop jobs. We leverage the query language's join mechanism to do this kind of enforcement. Policies involving URI's, literals, etc., can be enforced in this way. For example, suppose access to data for some confidential courses is restricted to only a few students. If an unprivileged user wishes to list the courses a student has taken, we can join the file listing the confidential courses with the file *takesCourse*, and thereby enforce the desired policy within the Reduce phase of a Hadoop job. Suppose courses C_3 and C_4 are confidential courses. If an unprivileged user wishes to list the courses taken by GS_3, then we can answer the query by the Map and Reduce code shown in Algorithms 2 and 3.

```
1: $splits \leftarrow value.split()$
2: if $Input\_file = sensitiveCourses$ then
3:     $output(splits[0], "S")$
4: else if $splits[0] = GS_3$ then
5:     $output(splits[1], "T")$
6: end if
```

Algorithm 2: Pseudo-code for EEMAP

```
1: $count \leftarrow 0$
2: $iter \leftarrow values.iterator()$
3: while $iter.hasNext()$ do
4:     $count$++
5:     $t \leftarrow iter.next()$
6: end while
7: if $count = 1$ AND $t = "T"$ then
8:     $output(key)$
9: end if
```

Algorithm 3: Pseudo-code for EEREDUCE

Algorithm 2 shows the code of the Map phase. It first splits each line into a key and a value. If the input is from a

Table 2. EEMap output and EEReduce input

EEMap Output		EEReduce Input	
Key	Value	Key	Values
C_1	T	C_1	T
C_3	S	C_3	S, T
C_3	T	C_4	S
C_4	S		

confidential course file, it outputs the course and a flag (`"S"` for "secret") denoting a confidential course as the output pair in line 3. If it is from the *takesCourse* file, it checks whether the subject is GS_3 in line 4. If so, it outputs the course as the key and a flag (`"T"` for "takes") indicating that the course is of student GS_3. The left half of Table 2 shows the output of Algorithm 2 on the example data.

Algorithm 3 shows the code of the Reduce phase. It gets a course as the key and the flag strings as the value. The input it gets while running on our example data is shown in the right half of Table 2. The code simply counts the number of flags in line 4. If the only flag indicates that the course is of student GS_3 (line 7), then it outputs the course (line 8). A confidential course that is taken by the student GS_3 has an additional flag, raising the count to 2, and preventing those courses from being reported. A confidential course not taken by the student will also have one flag indicating that it is a confidential course. The check whether the flag is the one for course taken by student GS_3 prevents such courses from being reported. These two checks together ensure that only non-confidential courses taken by GS_3 are divulged in the output. Hence, only course C_1 appears in the output.

4.3.3 Post-processing Enforcement

The second approach runs jobs as if there are no access controls, and then runs one or more additional jobs to filter the output in accordance with the policy. The advantage of this approach is that it is simple to implement, but it may take longer to answer the query. We can use the previous example to illustrate this approach. We first run the job as if there is no restriction on courses. Then we run one extra job to enforce the policy. The extra job takes two files as input: the output of the first job and the *confidentialCourses* file containing the URI's of confidential courses. In the Map phase we output the course as the key and, depending on the input file, a flag string. The Map code is largely the same as Algorithm 2. The only difference is that we do not need to check the URI identifying the student, since the output of the first job will contain the courses taken by only that student. The code for the Reduce phase remains the same. Hence, at the end of the second job we get output that does not contain any confidential courses.

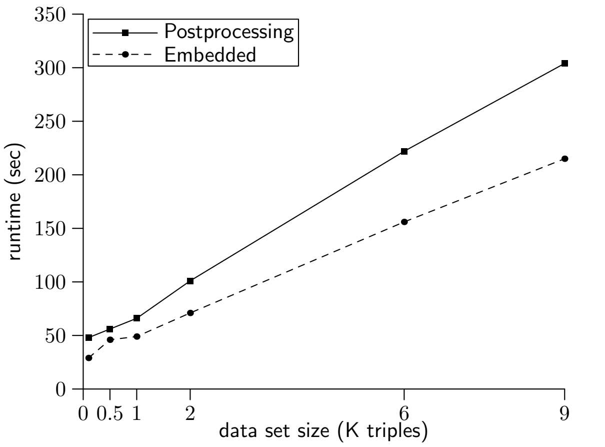

Figure 5. Performance measurements for the *takesCourse* scenario

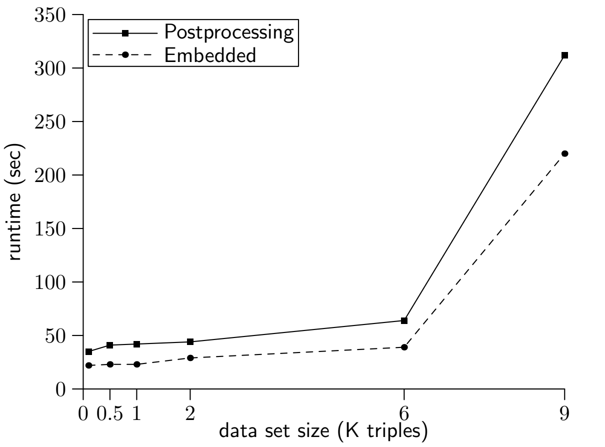

Figure 6. Performance measurements for the *displayTeachers* scenario

5 Experimental Setup and Results

We ran our experiments in a Hadoop cluster of 10 nodes. Each node had a Pentium IV 2.80 GHz processor, 4 GB main memory, and 640 GB disk space. The operating system was Ubuntu Linux 9.04. We compared our Embedded Enforcement approach with our Postprocessing Enforcement approach. We used the LUBM100, LUBM500, LUBM1000, LUBM2000, LUBM6000 and LUBM9000 datasets for the experiments.

We experimented with these approaches using two scenarios: *takeCourse* and *displayTeachers*. In the *takesCourse* scenario, a list of confidential courses cannot be viewed by an unprivileged user for any student. A query was submitted to display the courses taken by one particular student. Figure 5 shows the runtimes of the two different approaches.

In the *displayTeachers* scenario, an unprivileged user may view information about the lecturers only. A query was submitted to display the URI of people who are employed in a particular department. Even though professors, assistant professors, associate professors, etc., are employed in that department, only URI's of Lecturers are returned because of the policy. Figure 6 shows the runtimes we obtained from the two different approaches for this scenario.

We observe that Postprocessing Enforcement always takes 20–80% more time than the Embedded Enforcement approach. This can be easily explained by the extra job needed in Postprocessing. Hadoop takes roughly equal times to set up jobs regardless of the input and output data sizes of the jobs. The Postprocessing Enforcement approach runs more jobs than the Embedded Enforcement approach, yielding the observed overhead.

6 Conclusion and Future Improvements

Access controls for RDF data on single machines have been widely proposed in the literature, but these systems scale poorly to large data-sets. The amount of RDF data in the web is growing rapidly, so this is a serious limitation. One of the most efficient ways to handle this data is to store it in cloud computers. However, access control has not yet been adequately addressed for cloud-resident RDF data. Our implemented mechanism incorporates a token-based access control system where users of the system are granted tokens based on business needs and authorization levels. We are currently building a generic system that incorporates tokens and resolves policy conflicts. Our next goal is to implement Subject Model Level Access that recursively extracts objects of subjects and treats these objects as subjects as long as these objects are URI's. This will allow agents possessing Model level access to generate models on a given subject.

References

[1] Apache. Hadoop. `http://hadoop.apache.org/`.

[2] D. Beckett. RDF/XML syntax specification (revised). Technical report, W3C, February 2004.

[3] T. Berners-Lee. Semantic web road map. `http://www.w3.org/DesignIssues/Semantic.html`, 1998.

[4] L. Bouganim, F. D. Ngoc, and P. Pucheral. Client-based access control management for XML documents. In *Proc. 20émes Journées Bases de Données Avancées (BDA)*, pages 65–89, Montpellier, France, October 2004.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF. In *Proc. 1st International Semantic Web Conference (ISWC)*, pages 54–68, Sardinia, Italy, June 2002.

[6] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung. SPIDER: a system for scalable, parallel / distributed evaluation of large-scale RDF data. In *Proc. 18th ACM Conference on Information and Knowledge Management (CIKM)*, pages 2087–2088, Hong Kong, China, November 2009.

[7] J. Grant and D. Beckett. RDF test cases. Technical report, W3C, February 2004.

[8] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. In *In Proc. 3rd International Semantic Web Conference (ISWC)*, pages 274–288, Hiroshima, Japan, November 2004.

[9] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2–3):158–182, 2005.

[10] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *Proc. Web Information Systems Engineering (WISE) International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 235–244, New York, New York, November 2005.

[11] L. E. Holmquist, J. Redström, and P. Ljungstrand. Token-based access to digital information. In *Proc. 1st International Symposium on Handheld and Ubiquitous Computing (HUC)*, pages 234–245, Karlsruhe, Germany, September 1999.

[12] M. F. Husain, P. Doshi, L. Khan, and B. M. Thuraisingham. Storage and retrieval of large RDF graph using Hadoop and MapReduce. In *Proc. 1st International Conference on Cloud Computing (CloudCom)*, pages 680–686, Beijing, China, December 2009.

[13] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *Proc. IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 1–10, Miami, Florida, July 2010.

[14] A. Jain and C. Farkas. Secure resource description framework: an access control model. In *Proc. 11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 121–129, Lake Tahoe, California, June 2006.

[15] Joseki. `http://www.joseki.org`.

[16] J. Kim, K. Jung, and S. Park. An introduction to authorization conflict problem in RDF access control. In *Proc. 12th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES)*, pages 583–592, Zagreg, Croatia, September 2008.

[17] Kowari. `http://kowari.sourceforge.net`.

[18] P. Mika and G. Tummarello. Web semantics in the clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[19] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical report, W3C, January 2008.

[20] P. Reddivari, T. Finin, and A. Joshi. Policy based access control for an RDF store. In *Proc. Policy Management for the Web Workshop*, 2005.

[21] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proc. 17th International Conference on World Wide Web (WWW)*, pages 595–604, Beijing, China, April 2008.

[22] W3C. Semantic web frequently asked questions. `http://www.w3.org/RDF/FAQ`, 2009.