International Journal of Software Engineering
and Knowledge Engineering
Vol. 22, No. 1 (2012) 85–106
© World Scientific Publishing Company
DOI: 10.1142/S0218194012500040

A VERSION MANAGEMENT FRAMEWORK FOR RDF TRIPLE STORES

DONG-HYUK IM
School of Computer Science and Engineering
Seoul National University
Seoul, 151-742, Korea
dhim@idb.snu.ac.kr

SANG-WON LEE
School of Information and Communication Engineering
Sungkyunkwan University
Suwon, 440-746, Korea
wonlee@ece.skku.ac.kr

HYOUNG-JOO KIM
School of Computer Science and Engineering
Seoul National University
Seoul, 151-742, Korea
hjk@snu.ac.kr

Received 19 March 2011
Revised 17 May 2011
Accepted 6 September 2011

RDF is widely used as an ontology language for representing the metadata in Semantic Web, knowledge management system and E-commerce. Since ontologies model the knowledge in a particular domain, they may change over time. Furthermore, ontologies are usually developed and controlled in a distributed and collaborative way. Thus, it is very important to be able to manage multiple versions for RDF data. Earlier studies on RDF versions have focused on providing the accesses to different versions (i.e. snapshots) and computing the differences between those two versions. However, the existing approaches suffer from the space overhead for large scale data, since all snapshots should be redundantly kept in a repository. Moreover, it is very time consuming to compute the delta between two specific versions, which is very common in RDF applications. In this paper, we propose a framework for RDF version management in relational databases. It stores the original version and the deltas between two consecutive versions, thereby reducing the space requirement considerably. The other benefit of our approach is appropriate for change queries. On the flip side, in order to answer a query on a specific logical version, version should be constructed on the fly by applying the deltas between the original version and the logical version. This can slow down query performance. In order to overcome this, we propose a compression technique for deltas, called Aggregated Delta, to create a logical version directly rather than executing the sequence of deltas. An experimental study with real life RDF data sets shows our framework maintains multiple versions efficiently.

Keywords: RDF; ontology; versioning system; ontology evolution; relational database.

1. Introduction

The Semantic Web has been garnering much recent attention since it allows software agents to automatically handle the web resources marked with semantic information. Among several alternative data models for the Semantic Web, ontologies are regarded to be most promising, because they can represent the concepts in a particular domain and support the various relationships between the concepts. Many websites populate their data using ontology languages such as RDF (Resource Description Framework) [1] and OWL (Web Ontology Language) [2] format. Since the RDF ontology language has a simple data model with well-defined, formal semantics and provable inference, we focus on RDF models and RDF syntax on behalf of ontology language in this paper. However, these ontologies model the real world ever-changing over time [3, 4]. Thus, we need to manage and control ontology changes (i.e. RDF deltas). Often ontologies are developed and managed in a distributed environment, where several users can create and develop the concepts in a collaborative and synchronized manner. In such cases, we need to support ontology versions [5]. For example, the community of life science has widely used Gene Ontology [6] and NCI Cancer Ontology [7]. Each of these provides a controlled vocabulary and its numerous members can update and revise its vocabulary. In order to answer which concept has changed or to trace the history of concept changes, it is necessary to keep all the previous versions, as well as the most recent version. And, in order to synchronize two or more versions correctly, it is also necessary to keep track of the changes between versions.

Recent work on the ontology versioning system, including SemVersion [8] and PromptDiff [9], has focused on how to support accesses to various versions or how to compute the differences between versions. However, they do not consider the space overhead in supporting versions. That is, if we redundantly store every version in a separate storage space, the space requirement would be enormous, especially in a large scale ontology system. Furthermore, this approach also has a limitation, in that it should recalculate the changes between versions whenever the users' query changes. The computation of the differences is the most important function of a versioning framework. Based on these changes, we can figure out what have changed and how they have changed. That is, the tracking of the changes provides the necessary basis for further analysis in versioning systems [10]. Hence, there is a strong need for delta-based version framework for RDF data that is space-efficient to retain the deltas.

RCS (Revision Control System) [11] which is popular in software development and open source projects, stores revisions in the form of deltas. RCS stores only edit commands (i.e. insertion or deletion of line). In RCS, only the most recent revision is stored intact and all edit instructions, which describe how to go to the previous versions, are stored. RCS maintains a complete history of changes and resolves conflicts in communication by managing the revisions. In addition, several works have been done on handling XML version management, including [12–14]. These works focus on managing deltas in a XML repository and minimizing the number of pages that must be accessed to reconstruct a specific version of XML data. However, RDF version model, which is a set of RDF triples, cannot be effectively managed by RCS or XML version management system. In fact, the line-edit script in RCS and the tree structure of XML document are not adequate to support the management of RDF-based information (i.e. RDF data, RDF deltas, RDF storage, and RDF query processing). Thus, in RDF versioning systems, it is necessary to manage multiple versions using deltas which consider the structure and the semantic in the RDF model. In this paper, we propose a version management framework for RDF data. The contribution of this paper is as follows.

(1) We identify the problems of the existing version management that store all snapshots in a triple store. One of the problems is the storage overhead for all versions and another problem is the computation time for the delta between two versions.
(2) We propose a version framework for RDF data model based on relational databases. This scheme stores the original RDF version data and the deltas between the two consecutive versions. We store the deltas separately in a delete table and an insert table, and construct a logical version on the fly using a SQL statement that joins the version from the original version and the relevant delta tables. With this scheme, we can store the versions with minimum storage space and support the multiple logical versions with nominal overhead of logical version construction.
(3) We also introduce an optimized storage scheme for deltas, called *Aggregated Delta*. By compressing the delta between two specific versions, which has more than one in-between delta, and storing it in advance (i.e. a form of materialization), we can avoid the applications of the in-between deltas in sequence. In particular, it can eliminate a considerable amount of duplicates that exist in the in-between deltas.
(4) We compare our scheme to existing ones in terms of space, version construction time and query processing time. Experimental results show that our scheme is very promising for RDF version management.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes basic concepts, such as the data model and prior work related to version management. In Sec. 4, we propose a framework for RDF version management based on relational database. Section 5 presents the experimental results comparing our scheme to other approaches in terms of space and time overhead. Finally, Sec. 6 concludes this paper.

2. Related Work

This section reviews the existing works on evolution and version functionality in ontology. We also review the versioning schemes in other application areas, such as software version management systems, object-oriented databases, XML versioning and data warehouses.

2.1. *Evolution and versioning in ontology*

Ontology evolution and versioning. Some studies on ontology evolution [15, 16] describe the process of modifying the ontology whilst maintaining the consistency of the ontology. However, while ontology evolution focuses on the validity of the newest version, ontology versioning focuses on the validity, interoperability and management of all versions [3]. Much work has been done in the area of ontology versioning. OntoView [17] allows ontology engineers to compare versions of ontology and to specify how the ontology concepts in two versions are related. SemVersion [8] is an RDF-based ontology versioning system that supports query answering across multiple versions and the differences between arbitrary versions. PromptDiff [9] compares different versions of ontologies using heuristics, and provides the user with their deltas. However, all these versioning systems store all snapshots in a repository so that the deltas between versions must be recomputed on the fly whenever the change information is necessary. On the other hand, Tzitzikas *et al.* [18] focused on the storage space in RDF repositories and proposed storage index, called POI (Partial Order Index), which provides an efficient RDF version insertion algorithm in main memory. Since this storage scheme is based on partial orders of triple sets, it is the most efficient in storage space when the new version is a subset or superset of the existing versions. However, the new version cannot be a subset or superset when it has both the added and removed triples from the existing versions. In addition, in order to construct a specific version, it needs to traverse all the ancestor elements of the given element in the POI. Thus, it is not scalable as the data size increases. In contrast, we focus on a scalable data management in a relational database.

2.2. *Versioning in other application areas*

Software versioning. CVS (Concurrent Versioning System) [19] and SCCS (Source Code Control System) [20] allow collaborative development of software. First, the developers can download the source code from the repository and make a version copy (check-out). Then, they revise the version and create a new one. Finally, they upload their source code to the repository (check-in). These systems allow two or more developers to work on the same source code concurrently. However, in contrast to our work, they store a complete copy of all the files.

Object-oriented database. Schema evolution and versioning in object-oriented database systems, which emerged in the mid 80s, influence issues in ontology evolution and versioning. In particular, semantics of change operation and set of invariants proposed in [21] can also be applied to the change operation in ontology evolution. Furthermore, version management for dynamic schema changes in OODB applications (i.e. CAD/CAM) had been studied [22]. However, there is significant difference between our framework and OODB from the view point of the target. While version management in OODB focuses on the schema change operation, our work considers the triple change at instance level, as well as the triple change at schema level.

XML versioning. A variety of version management schemes has been developed for XML documents. UBCC (Usefulness-based copy control) [13] clusters the document objects separately in order to avoid accessing unrelated objects at version reconstruction. Marian *et al.* [14] proposes completed deltas for XML and an efficient storage policy for the deltas. The work in [12] introduces an adaptive XML version management scheme taking into account both performance (access time) and storage space requirements. However, since these schemes mainly deal with methods for managing the XML tree model (e.g. DOM structure), they cannot be directly applied to RDF data (RDF graph).

Data warehousing. In data warehousing applications, a set of materialized views should be kept consistently with the ever-changing underlying base tables. When any base table changes, we need to maintain the views consistent in an incremental way, rather than to rebuild the relevant views from scratch. This technique is called incremental view maintenance. In order to incrementally maintain the views, we first capture the changes in base tables, and calculate how those changes can be propagated to the relevant views. According to the point of propagation time, there are two approaches for incremental view maintenance. The first method is eager maintenance [23]: that is, incremental update is performed as soon as the change occurs. The second method is lazy maintenance [24], where view maintenance is deferred to a later time. Unlike our work, these approaches can be classified as updating methods rather than managing multiple versions.

3. Basic Concept

In this section, we present the basic RDF data model and the change operations generally used in RDF. Also, we describe the version model of RDF data assumed in this paper.

3.1. *RDF data model*

An RDF is modeled as a DAG (Directed Acyclic Graph) where each node and each arc represents a resource and a relationship, respectively. An RDF graph is a set of triples that represent binary relationships between two resources. In turn, each triple consists of three parts: subject, property and object [1]. Resource (i.e. subject) is usually named by URIs (Uniform Resource Identifier), the property is a type of resource and the object is the value of the resource property type for the specific subject. The value of object could be either URIs or character strings called literals.

3.2. *RDF change operation*

We can model the changes in RDF data only with triple insertions and triple deletions, not triple updates [10]. A triple update could be modeled as a deletion of the old triple, followed by an insertion of the new value of the triple (We only need to consider insertion and deletion on a triple, and do not need to consider updates explicitly.) In general, these operations are quite common in the RDF change detection field [4, 8, 25].

Definition 1. An RDF version V_i consists of a set of RDF triples whose every triple represents a statement of the form (subject, property, object), denoted by $t \in V_i$.

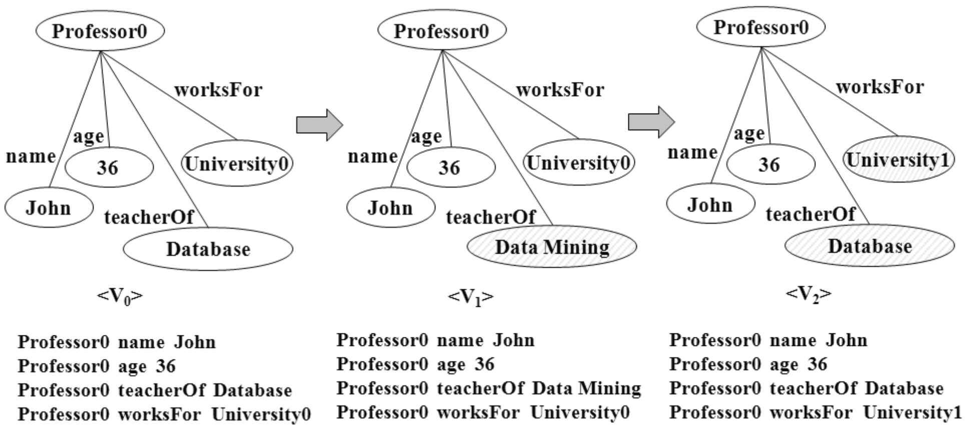
Figure 1 illustrates three RDF versions and the corresponding set of triples of each RDF version. We define the delta between two RDF versions as follows.

Definition 2. Given two RDF versions V_i and V_{i+1}, the delta between V_i and V_{i+1}, $\Delta_{i,i+1}$ is as follows. Let delete(t) denote a deleted triple t and insert(t) denote an inserted triple t.

(1) $\Delta_{i,i+1} = \Delta^-_{i,i+1} \cup \Delta^+_{i,i+1}$
(2) $\Delta^-_{i,i+1} = \{\text{delete}(t) \mid t \in V_i \text{ and } t \notin V_{i+1}\}$
(3) $\Delta^+_{i,i+1} = \{\text{insert}(t) \mid t \in V_{i+1} \text{ and } t \notin V_i\}$

Example 1. The following delta examples are found in Fig. 1.

$$\Delta^-_{0,1} = \{\text{delete}(\text{Professor}\mathit{0}\ \text{teacher of Database})\}$$
$$\Delta^+_{0,1} = \{\text{insert}(\text{Professor}\mathit{0}\ \text{teacher of Data Mining})\}$$



Professor0 name John	Professor0 name John	Professor0 name John
Professor0 age 36	Professor0 age 36	Professor0 age 36
Professor0 teacherOf Database	Professor0 teacherOf Data Mining	Professor0 teacherOf Database
Professor0 worksFor University0	Professor0 worksFor University0	Professor0 worksFor University1

Fig. 1. A sequence of snapshots with RDF graphs and triples.

$$\begin{aligned}\Delta^-_{1,2} &= \{\text{delete(Professor 0 teacher of Data Mining)},\\ &\quad\ \text{delete(Professor 0 works for University 0)}\}\\ \Delta^+_{1,2} &= \{\text{insert(Professor 0 teacher of Database)},\\ &\quad\ \text{insert(Professor 0 works for University1)}\}\end{aligned}$$

In RDF change operations, several works [4, 8] have been done on minimizing the RDF deltas using the inference (i.e. computing the closure of RDF). In this paper, we only consider the explicit differences between triples sets.

3.3. *RDF version*

An RDF versioning system should provide the mechanism to store various RDF versions and to highlight the changes between them. In addition, it should allow the user to access different versions of the RDF ontology easily [5]. Most existing version control systems use the tree model.

Definition 3. A RDF version tree is modeled as a tree $T = (V, E)$, where each node in V represents a version and each edge in E is a set of RDF change operations (deltas) from the source node to the destination node.

The "ancestor-descendant" relationship in a version tree represents a "derived-from" relationship. In a tree version model, a version can be derived from only one parent version, and a parent version cannot be further changed if at least one child version is derived from it. Figure 2 shows an example of a version tree in RCS [11]. A trunk in Fig. 2 is the main body of the version tree and a branch is created to isolate large changes onto a separate line of version tree. However, the main goal of the ontology versioning system in a particular domain is to support the collaborative creation of ontologies, and therefore, in most cases, there are few branches. For example, in SemVersion for MarcOnt [8], any community member can suggest changes and the ontology builder should manage versions of suggestions and generate snapshots of the main ontology with suggestions applied. In this paper, we assume a similar approach. That is, although any user can change the ontology version, only the ontology builder can manage the changes and release a new version.

In most existing RDF versioning systems, each version is assumed to have its own separate storage. This All Snapshots approach enables us to access each version

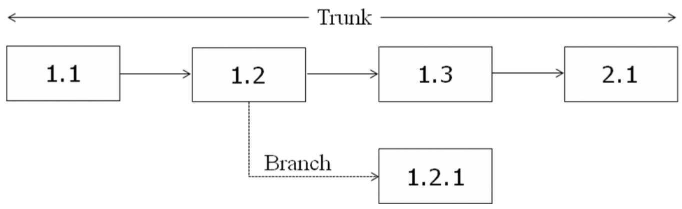


Fig. 2. RCS version tree.

quickly. However, it has a serious drawback that it requires an excessive storage space, as the number of versions increases. Generally, two consecutive versions of a document are not significantly different. This is also applicable to RDF version management (the result reported in [10] shows that 97.3% of the entire data in each version remains unchanged). Thus, with the All Snapshots approach, more than 90 percent of the entire storage is used to store redundant data. Obviously, the All Snapshots approach is very inefficient in RDF versions. For example, consider the three versions in Fig. 1. If we store each of three versions separately in independent storage, triples (*Professor0 name John*) and (*Professor0 age 36*) are redundantly stored in all snapshot of each version. Thus, this approach is inadequate for large scale RDF data that change frequently. The history of changes provides the users with the change information (i.e. version id, the modification time) and makes them easier to trace the version. This lineage trace of versions is one of the most common functionalities in RDF data processing. However, with the All Snapshots approach, the tracking of history change is also very inefficient, because we should calculate the change between versions whenever users ask the difference. Therefore, we propose a delta-based version storage scheme on top of a relational database to overcome two problems in the All Snapshots, space overhead and change detection. Our approach stores only an original version and the sequence of deltas between versions, instead of the snapshots of all versions.

4. Framework for RDF Version Management

In this section, we present our framework for managing the RDF version. We do not propose a RDF versioning system, but rather the basis for storing RDF versions in relational databases.

4.1. *Delta-based version management*

To solve the excessive storage space requirements in the All Snapshots approach, we suggest delta-based version management as a possible alternative for storing the versions. For example, traditional document version management schemes, such as RCS, SCCS and XML version management system [13, 14], stores the most recent version while other older versions are stored as deltas (i.e. reverse editing scripts). In these schemes, the particular version should be constructed on the fly by applying the reverse editing scripts. However, these techniques are not efficient in handling RDF versions because they do not fully leverage the structure and the semantics contained in RDF data. In XML version management system, edit-scripts deal with changes of nodes and subtrees in XML tree model. On the other hand, RDF data consists of a set of triples which are the smallest manageable entities in RDF repository [10]. Thus, it is necessary to keep stored only the deltas which consider the RDF data.

In this paper, we use a triple store method for an original version and deltas. It stores triples in a single table with three columns: the subject, the property and the object [26, 27]. In general, we term this approach the triple store. Although there are alternatives for storing RDF data in relational databases, such as vertical partitioning [28] and property table [27], the goal of our work is to propose a framework for RDF versions. Therefore, we assume the simple triple store scheme for RDF version management. Of course, we believe that it is necessary to investigate other storage schemes for managing RDF versions efficiently. Fortunately, since the unit of change operation is a triple, this triple-based storage scheme offers the foundation that makes it easy to construct a new version from a delta. This means that we create a version using the relational-algebra operators such as *union* and *set-difference*. Figure 3 shows the result of delta-based storage scheme that is applied to the RDF versions in Fig. 1. We maintain the delta of each version separately in two tables, a delete table and an insert table.

Definition 4. Given the original version V_i, let V_{i+1} be the logical version and $\Delta_{i,i+1}$ be the set of change operations between V_i and V_{i+1}. Then V_{i+1} is represented as follows.

$$V_{i+1} = \Delta_{i,i+1}(V_i)$$

By Definition 4, we can easily construct the logical version V_1 in Fig. 3. The following SQL statement creates the logical version V_1 from the relational database:

(((select $*$ from V_0) minus (select $*$ from Delete_V_0_V_1)) union all (select $*$ from Insert_V_0_V_1))

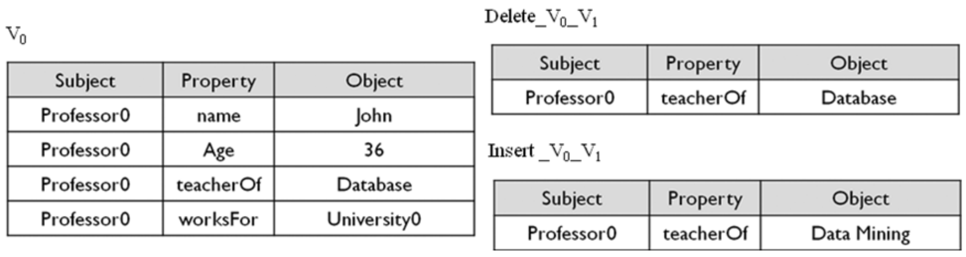
If there is a sequence of versions, we can obtain each logical version by executing its Sequential Delta, defined as follows.

Definition 5. A sequence $\Delta_{i,i+1}, \Delta_{i+1,i+2}, \ldots, \Delta_{j-1,j}$ of change operations is Sequential Delta if there is the original version V_i and the logical version V_j such that $\Delta_{i,i+1}, \Delta_{i+1,i+2}, \ldots, \Delta_{j-1,j}$ is the set of change operations that transforms V_i to V_j. Then we have:

$$V_{i+1} = \Delta_{i,i+1}(V_i); \quad V_{i+2} = \Delta_{i+1,i+2}(V_{i+1}); \ldots; \quad V_j = \Delta_{j-1,j}(V_{j-1}).$$

4.2. *Aggregated delta*

In order to access a specific logical version, we must construct the logical version on the fly by applying the deltas between the original version and the logical version.



V_0

Subject	Property	Object
Professor0	name	John
Professor0	Age	36
Professor0	teacherOf	Database
Professor0	worksFor	University0

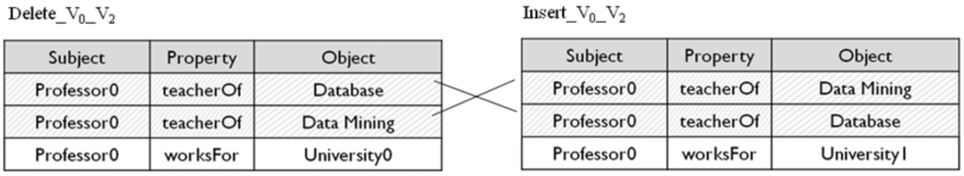
Delete_V_0_V_1

Subject	Property	Object
Professor0	teacherOf	Database

Insert _V_0_V_1

Subject	Property	Object
Professor0	teacherOf	Data Mining

Fig. 3. Delta-based RDF version storage.



Delete_V$_0$_V$_2$

Subject	Property	Object
Professor0	teacherOf	Database
Professor0	teacherOf	Data Mining
Professor0	worksFor	University0

Insert_V$_0$_V$_2$

Subject	Property	Object
Professor0	teacherOf	Data Mining
Professor0	teacherOf	Database
Professor0	worksFor	University1

Fig. 4. The duplicate triples in aggregated tables.

The problem with this Sequential Delta approach is that as the number of deltas between the original version and the logical version increases, the version construction time also increases, because we need to use the set operators such as *union* and *minus* in SQL statements in proportion to the number of deltas. Moreover, consider the versions in Fig. 4. If we make a logical version V_2 by applying the deltas $\Delta_{0,1}$, $\Delta_{1,2}$ to the original version V_0, we insert the triple (*Professor0 teacherOf Data Mining*) in V_1 and delete it from V_2. Likewise, we delete the triple (*Professor0 teacherOf Database*) from V_1 and insert it in V_2. These changes are unnecessary to construct the logical version V_2 from the original version V_0. This unnecessary change is one of the major factors that causes performance overhead and the space requirement.

Based on this observation, we propose an optimized scheme, called Aggregated Delta, which can create a logical version directly by storing all of the possible deltas in advance, instead of executing all the in-between deltas in sequence.

While the Sequential Delta approach takes a kind of lazy maintenance in computing the transitive closure of the "derived-from" relation on the fly, the Aggregated Delta approach uses a form of eager maintenance. As some RDF management systems such as Sesame [26, 29] pre-compute the complete closure of the RDF data and then store them in the same repository, we compute and store the closure of all the "derived-from" relations in advance when new RDF version is added. Moreover, in the Aggregated Delta, we can reduce the number of triples using a compression algorithm that removes the duplicates. This compression allows us to avoid accessing unrelated triples at version reconstruction and to save storage space. The Aggregated Delta can be formally defined as follows.

Definition 6. Given the Sequential Delta $\Delta_{i,i+1}, \Delta_{i+1,i+2}, \ldots, \Delta_{j-1,j}$ between V_i and V_j, an Aggregated Delta is as follows. Let C_t be the set of change operations with overlapped triples in both delta tables.

$$\sum_{n=i}^{j-1} \Delta_{n,n+1} = \sum_{n=i}^{j-1} \Delta^{-}_{n,n+1} \cup \sum_{n=i}^{j-1} \Delta^{+}_{n,n+1}, \quad (i < j)$$

$$\Delta_{g(i,j)} = \sum_{n=i}^{j-1} \Delta_{n,n+1} - C_t.$$

Consider the duplicate triples (*Professor0 teacherOf Data Mining*) and (*Professor0 teacherOf Database*) between two aggregated delta tables, as shown in

```
Input : insert_table $T_i$ and delete_table $T_d$ in aggregated delta
Output : insert_table $T_i$ and delete_table $T_d$ in compressed delta
Initialize: Sorting each table by subject, property, object
        Set cursor1 to beginning of table $T_i$, Set cursor2 to beginning of table $T_d$
DO {
    if (cursor1.triple = cursor2.triple)
        delete the triple in each table
        cursor1++; cursor2++;
    else if (cursor1.triple < cursor2.triple)
        cursor1++;
    else if (cursor1.triple > cursor2.triple)
        cursor2++;
} While (cursor1 $\neq$ end of $T_i$ and cursor2 $\neq$ end of $T_d$)
```

Fig. 5. Compression algorithm.

Fig. 4. Aggregated Delta can eliminate the duplicates using the compression algorithm. The pseudocode of the compression algorithm is given in Fig. 5. First, it sorts each aggregated table in lexicographic order. Next, it gets the pointers that point initially to the first triple of the respective table. Then, it compares two triples to which the pointers point. If two triples have the same value, each of the triples is deleted from its table and the pointer in each table is set to the next triple. If a triple in one aggregated table is less than a triple in the other aggregated table in lexicographic order, the pointer in the first table is set to the next triple. This process is repeated until the pointer in either table reaches the end of the table just like the merge-sort algorithm. The compression algorithm would be more effective when one triple is repetitively inserted and deleted along the version path. We can observe the frequent in-and-out of the same triple along the versions in many RDF data sets. For example, FOAF (Friend of a Friend) project supports distributed descriptions of people and their relationships. If we model people in FOAF, specific properties such as *name* are invariant. Conversely, we can change *interest* or *currentProject* frequently. Thus, the compression algorithm can be useful in the version management system.

Example 2. We can obtain the result set of Aggregated Delta $\Delta_{g(0,2)}$ in Fig. 1.

$$\begin{aligned}\Delta_{g(0,2)} = \{&\text{delete (Professor 0 worksFor University 0)},\\ &\text{insert (Professor 0 worksFor University 1)}\}\end{aligned}$$

We can access the logical version V_j directly from the original version V_i using the following Aggregated Delta.

$$V_j = \Delta_{g(i,j)}(V_i)$$

The properties of Aggregated Deltas are described.

— Suppose there are n distinct versions in a trunk. Then the number of the deltas is $n-1$ and Aggregated Delta increases the size of $(n-2)(n-1)/2$ deltas for additional storage overhead. However, we can minimize the space requirement with the compression algorithm. Obviously, this size is significantly smaller than storing all versions.

— If there are no overlapping triples, the size of Aggregated Delta is always the same as the sum of Sequential Delta (Worst case). However, if C_t exists in both tables, the property described above is satisfied, because duplicates are removed:

$$|\Delta_{g(i,j)}| \leq \sum_{n=i}^{j-1} \Delta_{n,n+1}$$

The principle behind the compression algorithm in the Aggregated Delta is similar to aggregation scheme in XML version management [14] and refreshing materialized view in data warehouses [24]. However, there are critical differences between them. First, the aggregation scheme for XML deltas [14] aggregates the consecutive deltas for a document in the period of time (i.e. weekly versions, monthly versions). Thus, this approach requires less space than delta-based approach (note that it reduces the number of the consecutive deltas). In contrast, our proposed method aggregates all possible deltas between versions that have the "derived-from" relationship and stores them (note that this scheme increases the number of deltas and requires more space than delta-based version management). Moreover, unlike [14] which is based on XML trees, we exploit the structural and the semantic feature of the RDF (i.e. DAG model, triple-based diff algorithm, RDF triple repository). Second, in the context of data warehouses, Zhou *et al.* propose a compression scheme, called the condense operator [24], it also skips unnecessary intermediate changes for lazy view maintenance. However, they applied this operator to the tuples with the same key value in a single table for stream deltas. In contrast, our compression approach removes the triples whose subject, property and object can match those in another delta table. That is, our compression scheme is designed to reflect the semantics of the RDF change model (i.e. triple-based change operation).

4.3. *Version management policy*

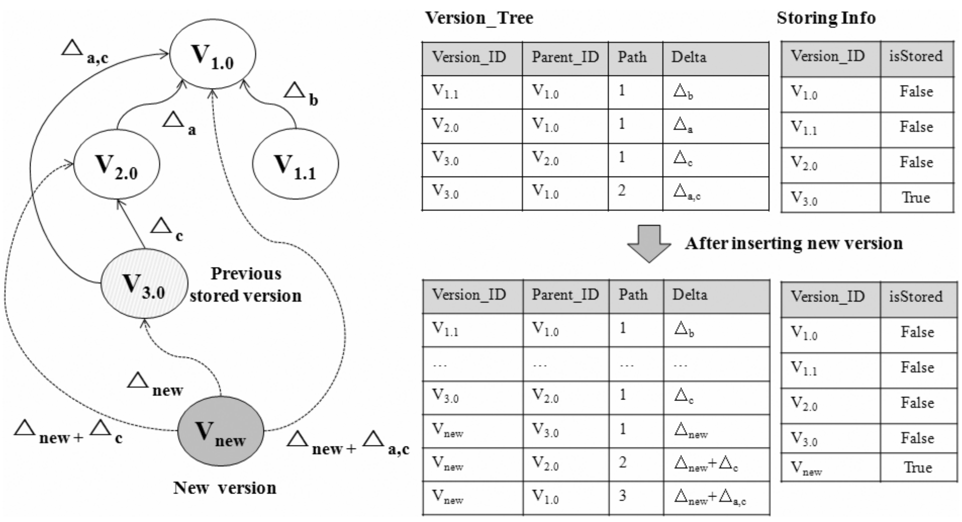
In version management, we need to store the derived-from relationships between versions, as well as each version and the corresponding deltas. From these relationships, we know which deltas should be applied to the original version when we construct a logical version. We store the relationship in the version tree in separate tables. The following shows the relational table structure for a version tree.

Version_Tree (VersionID, ParentID, Path, Delta)
StoringInfo (VersionID, isStored)

Version tree in Definition 3 can be mapped into *Version_Tree* table, which represents the derived-from relationship containing the transitive closure. *StoringInfo* table represents the version we store as a snapshot. With regard to how to store the original version, two approaches exist: the first version and the last version. Version control systems generally store the last version [14, 20]. In this paper, we also choose to store the last version as the original version. In this case, we should use a backward delta, which enables us to go to the previous version. We can easily compute a backward delta, applying the delta in reverse. The tree model for versions would increase the space requirement, because there can be many leaves in the tree. However, since our version tree is mainly based on the trunk (with only a few branches) as described in Sec. 3.3, the space overhead would not be so onerous. When a new version is created, we maintain the versions using the following steps (we assume that only the ontology builder releases the new version. Thus, a new version is created only by a set of changes, rather than every data change):

(1) When a new version is inserted, we compute the deltas between its parent version and the new version itself, then store the deltas D in two delta tables.
(2) Drop the previous original version (the new snapshot becomes the original version).
(3) Merge the backward deltas D with all Aggregated Delta in previous version respectively and create the delta table for new Aggregated Delta using our compression.
(4) Update the value in the "isStored" column in *StoringInfo* table.

Example 3. Suppose we insert the new version V_{new} in the version tree as shown in Fig. 6. First, Δ_{new}, the deltas between $V_{3.0}$ and V_{new} are merged into all deltas Δ_a and $\Delta_{a,c}$ in $V_{3.0}$ respectively. Note that we can access version $V_{1.0}$ applying the delta



Version_ID	Parent_ID	Path	Delta
$V_{1.1}$	$V_{1.0}$	1	Δ_b
$V_{2.0}$	$V_{1.0}$	1	Δ_a
$V_{3.0}$	$V_{2.0}$	1	Δ_c
$V_{3.0}$	$V_{1.0}$	2	$\Delta_{a,c}$

Version_ID	isStored
$V_{1.0}$	False
$V_{1.1}$	False
$V_{2.0}$	False
$V_{3.0}$	True

Version_ID	Parent_ID	Path	Delta
$V_{1.1}$	$V_{1.0}$	1	Δ_b
...	...	...	...
$V_{3.0}$	$V_{2.0}$	1	Δ_c
V_{new}	$V_{3.0}$	1	Δ_{new}
V_{new}	$V_{2.0}$	2	$\Delta_{new}+\Delta_c$
V_{new}	$V_{1.0}$	3	$\Delta_{new}+\Delta_{a,c}$

Version_ID	isStored
$V_{1.0}$	False
$V_{1.1}$	False
$V_{2.0}$	False
$V_{3.0}$	False
V_{new}	True

Fig. 6. The version tree and the corresponding relational schema.

$\Delta_{\text{new}} + \Delta_{a,c}$. Then, "*isStored*" of $V_{3.0}$ is changed from true to false and "*isStored*" of V_{new} is true. If the approach to store the first version as an original version is taken, it would be time consuming to compute the latest version from the first version, especially when recent versions are accessed more frequently than old versions. Thus, it is more efficient to store the last version and compute Aggregated Delta incrementally.

5. Experimental Results

In this section, we compare the performance of the three methods in RDF version management: the All Snapshots approach (used in SemVersion [8]), the Sequential Delta (based on the change detection between consecutive versions [4]) and our Aggregated Delta. The All Snapshots and the Sequential Delta approaches have been very popular in version management area, and our work is mainly motivated by their shortcomings. Thus, we include these techniques in our experiments. In particular, these three approaches are evaluated in terms of storage overhead, logical version construction time, computation time, compression ratio, and query processing time.

5.1. *Experimental setting and test data set*

All experiments were performed on Intel Xeon CPU 3GHz PC with 16GB memory. We implemented all the version schemes using Java with the RDF parser Rio,[a] and used Oracle 11g Enterprise edition as the relational database to store versions. We store each RDF data set in a corresponding triple table, and create a B$^+$ tree index on each table with the three columns of triple as its key. For the experiment, we use the dataset from Uniprot Taxonomy RDF.[b] Uniprot is the integrated database that provides the life science community with the information about proteins. Every two weeks a new version of the Uniprot data set is published; each version is maintained by the ontology builder, as explained in Sec. 3.3. Thus, all versions constitute a linear version model without branch. Table 1 summarizes the characteristics of our real data set.

Table 1. Uniprot Taxonomy RDF.

Version	U1	U2	U3	U4	U5	U6	U7	U8	U9
ID	13.6	14.0	14.1	14.2	14.3	14.4	14.5	14.6	14.7
# of triples	2829630	2862424	2923007	2969302	2989544	3019495	3019511	3084444	3113008
Size (MB)	196	198	203	206	207	209	209	214	216

[a]Available at http://www.openrdf.org.
[b]Available at http://dev.isb-sib.ch/projects/uniprot-rdf/.

5.2. *Storage space overhead*

Figure 7 shows the capacity of the storage space required in each RDF version scheme. The storage space of each scheme includes the total size of disk space in all the versions and, if any, all the deltas in the relational database. In Sequential Delta and Aggregated Delta, we store version U9 (last version) as the original version. With the All Snapshots approach, as shown in Fig. 7, the storage space increases linearly as the number of versions increases, since this scheme stores all version snapshots in a repository. In contrast, the Sequential Delta approach requires less space than the All Snapshots, because it stored only an original version and the deltas. However, the problem with this approach is that the construction time of versions is much slower than the others. This will be discussed in the next section. On the other hand, the Aggregated Delta requires more space than the Sequential Delta. This is because the Aggregated Delta increases the size of $(n-1)(n-2)/2$ deltas for additional storage overhead. However, its total storage requirement is still quite small (30%) compared to the All Snapshots. If the number of version increases dramatically, it can be useful to store intermediate versions physically. These intermediate versions enable us to restart to compute and store the additional deltas. Obviously, keeping several intermediate versions physically are also smaller than storing all versions physically. In addition, the Aggregated Delta has reduced the size of aggregated deltas using compression technique. The effects of compression technique will be analyzed in Sec. 5.4.

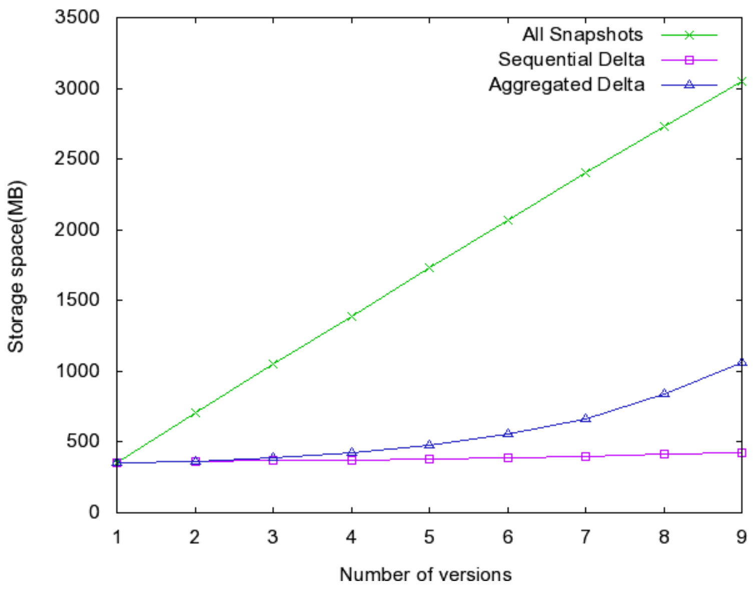


Fig. 7. Storage space for version management.

5.3. *Computation time and version construction time*

In order to assess the overhead of the Aggregated Delta, we measured the time to compute the deltas. Figure 8 shows the computation time for the Sequential Delta and the Aggregated Delta, where the x-axis represents the version to be inserted. Obviously, the Sequential Delta is quite faster than the Aggregated Delta and this performance gap between the two methods is widening as the number of versions increases. This is because the Sequential Delta computes the difference between the current version and the new version. On the other hand, the Aggregated Delta needs $(n - 1)$ deltas when the new version is inserted. However, we need not compute the differences again between all versions. As explained in Sec. 4.3, the Aggregated Delta is computed by adding the new delta to each previous delta and applying the compression method to each aggregated delta. Thus, the performance ratios between the Aggregated Delta and the Sequential Delta are usually less than two.

Next, Fig. 9 shows the construction time of versions in the Sequential Delta and Aggregated Delta. The y-axis represents the construction time of versions in seconds, and the x-axis denotes the specific versions to be constructed. In order to generate versions which are not stored physically, we need to construct logical versions on the fly from the original version (i.e. U9 in this example). As shown in Fig. 9, while the construction time in the Sequential Delta is proportional to the number of versions we need to trace backwards, the Aggregated Delta can compute any specific version almost at a constant time. This is because the Aggregated Delta creates any version just by applying only a corresponding aggregated delta to the original version U9. Thus, under the Aggregate Delta, we can access any version in a

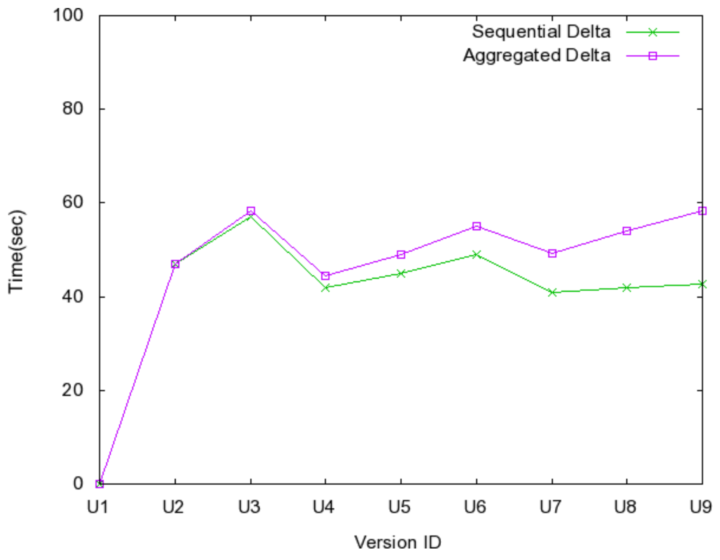


Fig. 8. Computation time for Aggregated Delta.

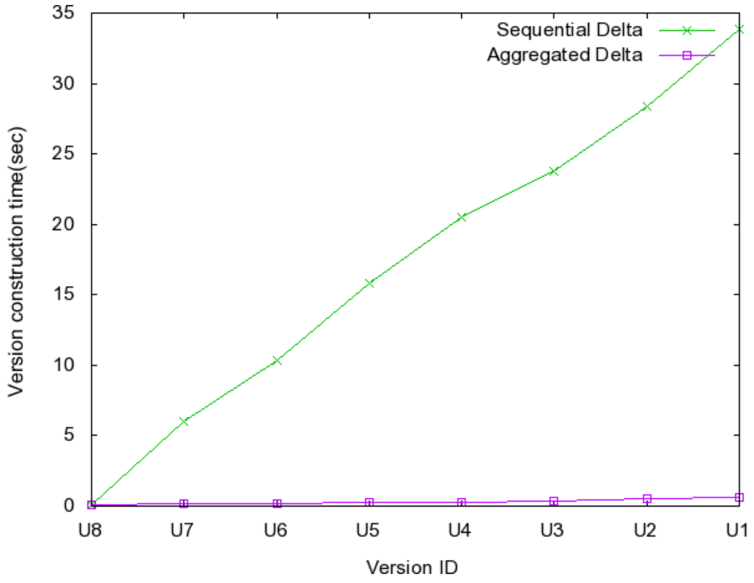


Fig. 9. Version construction time.

constant time by reducing the construction time that is unavoidable in the Sequential Delta.

Since the relevant version need to be constructed on the fly for a given query and this version construction occurs very frequently, the version construction performance is very critical. In this respect, the benefits of our method to efficiently construct version far outweigh its marginal overhead in computation time.

5.4. *Compression performance*

Figure 10 shows the effect of compression technique in the Aggregated Delta approach, where the x-axis represents each pair of versions for Aggregated Delta and the y-axis represents the number of triples in deltas. From Fig. 10, we know that a number of triples being removed by our compression technique tends to increase as the number of in-between deltas between two versions increases. Specifically, more than half of all the triples are deleted due to the compression in the last Aggregated Delta. We can infer that a considerable portion of triples changes frequently during the RDF evolution. In fact, the Uniprot has been developed to provide the biologist with a controlled vocabulary. For this reason, any scientific community member can change and review a specific concept in Uniprot until consensus is reached. This frequent change between version characteristic can be also found in many other ontology domains that are developed to share concepts. Consequently, our compression technique suggested in this paper would be generically applicable to any RDF data version scheme.

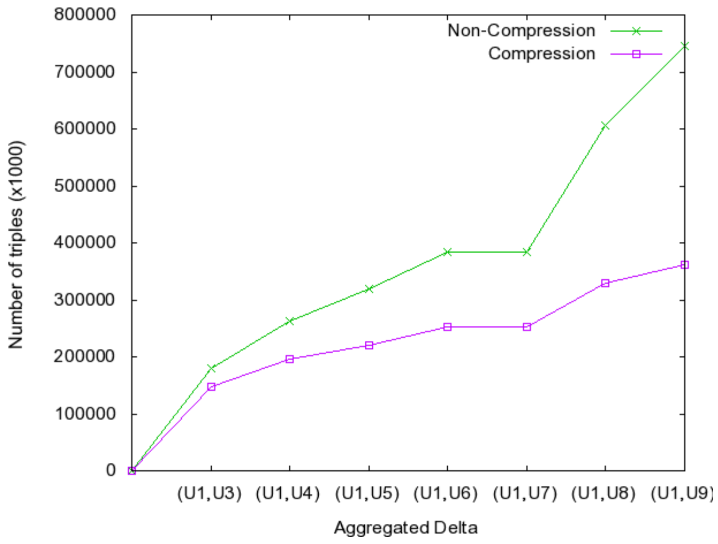


Fig. 10. The effect of compression.

5.5. *Query performance*

Finally, we evaluate the query performance of various version storage schemes using the queries in Table 2. The queries in RDF version management can be broadly classified as two types: one type of query accesses a specific version and the other type traces the changes between versions. The queries used in the experiment are classified and described in Table 2. Q1 is a simple query to find the subclasses of a specific class. Each of Q2, Q3 and Q4 involves a self-join operation (i.e. Subject-Subject join, Subject-Object join, Object-Object join). Q2 is to find which other properties are related for all subjects and calculate the frequency of these properties. Q3 is to calculate the path expression that can be connected with a subject-object join (i.e. transitive closure). Q4 is to find the siblings in the hierarchy. Each of Q5 and Q7 is to calculate the changes between two versions and analyze the difference.

Table 2. Sample queries.

Type	Query	Characteristics	Description
Single Version	Q1	Simple	List the subclasses of a specific class
	Q2	Sub-Sub Join	Find what other properties are defined for these subjects and calculate the frequency
	Q3	Sub-Obj Join	List the properties that can be connected
	Q4	Obj-Obj Join	List the classes that have a common parent
Cross Version	Q5	Change Analysis	What is the most changed property? (hot property)
	Q6	History	Tracking specific changes during evolution (lineage)
	Q7	Change Detection	Display the differences between the first and the last version

In particular, Q6 is known to be used in data warehousing as the lineage trace of a data item [30]. It is possible to trace the data back to the source data from which it is derived with this query, and thus to find how the data has evolved. Thus, Q5, Q6 and Q7 are commonly used in managing versions.

In Q1, Q2, Q3 and Q4, we use version U1 as the version that we would access, because we store the last version, U9, as described in Sec. 4.3. That is, we consider the worst case in these queries. Each query was represented as an SQL statement, as exemplified for Table 3 using Q2, and was run in a commercial RDBMS. In both the Sequential Delta approach and the Aggregated Delta approach, we use the WITH clause when executing the self-join operation for efficiency. u1, u2, . . . ,u8 and u9 are

Table 3. SQL representation of Q2.

Approach	SQL expression (Q2)
All Snapshot	Select t.pred, count(*) from u1 s, u1 t where s.subj = t.subj and group by t.pred
Sequential Delta	with subquery as (((((((((((((select * from u9) minus (select * from delete_u9_u8) union all (select * from insert_u9_u8)) . . . minus (select * from delete_u2_u1) union all (select * from insert_u2_u1)) select t.pred, count(*) from subquery *s*, subquery *t* where s.subj = t.subj group by t.pred
Aggregated Delta	with subquery as ((select * from u9) minus (select * from delete_u9_u1) union all (select * from insert_u9_u1)) select t.pred, count(*) from subguery *s*, subquery *t* where s.subj = t.subj group by t.pred

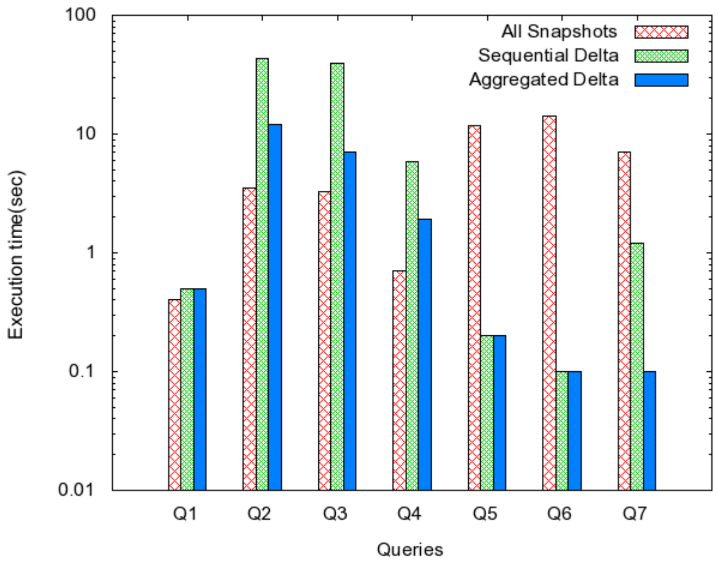


Fig. 11. Performance analysis for sample queries.

the tables that store the corresponding versions. The delete_u9_u8 table stores the deleted triples between U9 and U8. Likewise, the insert_u9_u8 table stores the inserted triples between U9 and U8.

Figure 11 shows the response time of the queries in Table 2. The All Snapshot approach is superior in queries Q1, Q2, Q3 and Q4, all of which require the computation in a specific version, since the All Snapshots approach stores all versions. Conversely, with the delta-based approaches, we first need to construct the version U1 on the fly and then query against it. In Q5, Q6 and Q7, however, the All Snapshots approach performs worst. While the All Snapshots approach imposes a query after computing the differences, the delta-based approaches can query the changes immediately from the delta tables that are already computed and materialized. The Aggregated Delta approach outperforms the Sequential Delta approach in all cases. The queries for changes are of greater importance in versioning systems, because these provide change information. In general, users access a specific version after determining what happened in each version.

Moreover, if we query the last version, the performance for queries Q1, Q2, Q3 and Q4 in the Aggregated Delta approach is the same as in the All Snapshots approach.

6. Conclusion and Discussions

In this paper, we proposed a framework for version management on RDF triple stores. It stores one snapshot of the latest version and the deltas between the versions. With this delta-based approach, we can considerably reduce the storage space for versions than the All Snapshots approach. In addition, we proposed the Aggregated Delta approach that can construct the versions by using a compression algorithm. Since the proposed scheme can avoid unnecessary computation for redundant data during version construction, the Aggregated Delta is superior to the Sequential Delta approach in terms of version construction time. Although the proposed method requires computing and storing the additional deltas, we proved that the Aggregated Delta is a flexible method which can reduce the storage space for version and create a version efficiently.

In addition, the Aggregated Delta can process various types of queries efficiently. Although the All Snapshot shows the best performance with some queries, it performs worst for Q5, Q6 and Q7. Note that these queries are widely used in managing versions. On the other hand, the Aggregated Delta outperforms the Sequential Delta in most cases. In particular, the Aggregated Delta shows an outstanding performance for queries for Q5, Q6 and Q7. As a result, our approach can be sufficiently used in implementing RDF versioning system based on relational database system.

In future, the scheme will be extended to be able to handle the OWL model which has a more powerful and more complicated structure and inference rules. And as we mentioned in Sec. 5.2, it may be useful to store intermediate versions physically when the number of versions is increased dramatically. Thus, we need to investigate which version we should keep physically and how many intermediate versions we need. In this paper, a simple triple store scheme for the RDF data was assumed. However, there are a number of different physical organization techniques for RDF data [26–28]. Alternative storage schemes for RDF data would be advantageous to the performance of version control. Thus, we need to further investigate query optimization techniques in constructing the versions, such as indexing technique.

Acknowledgment

This research was supported by MIKE, Korea under ITRC NIPA-2011-(C1090-1121-0008) and the Brain Korea 21 Project.

References

1. G. Klyne and J. J. Carroll, Resource Description Framework (RDF): Concepts and abstract syntax, W3C Recommendation, http://www.w3.org/TR/rdf-concepts.
2. D. L. McGuinness and F. V. Harmelen, OWL Web Ontology Language overview, W3C proposed recommendation, 2003.
3. G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis and G. Antoniou, Ontology change: Classification and survey, *The Knowledge Engineering Review* **23**(2) (2008) 117–152.
4. D. Zeginiz, Y. Tzitikas and V. Christophides, On the foundations of computing deltas between RDF models, in *Proc. 6th International Semantic Web Conference*, 2007.
5. N. F. Noy and M. A. Musen, Ontology versioning in an ontology management framework, *IEEE Intelligent Systems* **19**(4) (2004) 6–13.
6. M. Ashburner *et al.*, Gene ontology: Tool for the unification of biology, *Nature Genetics* **25** (2000) 25–29.
7. J. Golbeck, G. Fragoso, F. Hartel, J. Hendler, J. Oberthaler and B. Parsia, The National Institute's thesaurus and ontology, *Journal of Web Semantics* **1** (2003) 75–80.
8. M. Volkel and T. Groza, SemVersion: An RDF-based ontology versioning system, in *Proc. 5th IADIS International Conference on WWW/Internet*, 2006.
9. N. F. Noy and M. A. Musen, PromptDiff: A fixed-point algorithm for comparing ontology versions, in *Proc. 18th International Conference on Artificial Intelligence*, 2002.
10. D. Ognyanov and A. Kiryakov, Tracking changes in RDF(S) repositories, in *Proc. 13th International Conference on Knowledge Engineering and Knowledge Management*, 2002.
11. W. F. Tichy, RCS — a system for version control, *Software: Practice & Experience* **15**(7) (1985) 637–654.
12. B. Benatallah, M. Mahdavi, P. Nguyen, Q. Z. Sheng, L. Port and B. McIver, An adaptive document version management scheme, in *Proc. of 15th International Conference on Advanced Information Systems Engineering*, 2003.
13. S. Chien, V. J. Tsotras and C. Zaniolo, Version management of XML documents, in *Proc. of 3rd International Workshop WebDB*, 2000.
14. A. Marian, S. Abiteboul, G. Cobena and L. Mignet, Change-centric management of versions in an XML warehouse, in *Proc. 27th International Conference on Very Large Data Bases*, 2001.

15. N. F. Noy, S. Kunnatur, M. Klein and M. A. Musen, Tracking changes during ontology evolution, in *Proc. 3rd International Semantic Web Conference*, 2004.
16. P. Plessers, O. Troyer and S. Castelyan, Understanding ontology evolution: A change detection approach, *Journal of Web Semantics* **5**(1) (2007) 39–49.
17. M. Klein, A. Kiryakov, D. Ognyanov and D. Fensel, Ontology versioning and change detection on the web, in *Proc. of 13th International Conference on Knowledge Engineering and Knowledge Management*, 2002.
18. Y. Tzitzikas, Y. Theoharis and D. Andreou, On storage policies for Semantic web repositories that support version, in *Proc. of 5th European Semantic Web Conference*, 2008.
19. Concurrent Versions System, Free Software Foundation, http://nongnu.org/cvs/, 2006.
20. M. J. Rochkind, The source code control system, *IEEE Trans. Software Engineering* **1**(4) (1975) 364–370.
21. J. Banerjee, W. Kim, H. J. Kim and H. F. Korth, Semantics and implementation of schema evolution in object-oriented database, in *Proc. of ACM SIGMOD Conference*, 1987.
22. W. Kim and H. T. Chou, Versions of schema for object-oriented databases, in *Proc. 14th International Conference on Very Large Data Bases*, 1988.
23. Y. Zhuge, H. Garcia-Molina, J. Hammer and J. Widom, View maintenance in a warehousing environment, in *Proc. of ACM SIGMOD Conference*, 1995.
24. J. Zhou, P. Larson and H. G. Elmongui, Lazy maintenance of materialized views, in *Proc. 33rd International Conference on Very Large Data Bases*, 2007.
25. T. Berners-Lee and D. Connolly, Delta: An ontology for the distribution of differences between RDF graphs, http://www.w3.org/DesignIssues/Diff, 2004.
26. J. Broekstra, A. Kampman and F. V. Harmelen, Sesame: A generic architecture for storing and querying RDF and RDF schema, in *Proc. 1st International Semantic Web Conference*, 2002.
27. K. Wilkinson *et al.*, Efficient RDF storage and retrieval in Jena2, in *Proc. 1st Workshop on Semantic Web and Databases*, 2003.
28. D. J. Abadi, A. Marcus, S. R. Madden and K. Hollenbach, Scalable semantic web data management using vertical partitioning, in *Proc. 33rd International Conference on Very Large Data Bases*, 2007.
29. J. Broekstra and A. Kampman, Inferencing and truth maintenance in RDF schema, in *Proc. Workshop on Practical and Scalable Semantic System*, 2003.
30. Y. Cui, J. Widom and J. Wiener, Tracing the lineage of view data in a warehousing environment, *ACM Trans. Database Systems* **25**(2) (2000) 179–227.