A Change Detection Technique for RDF Documents Containing Nested Blank Nodes

Dong-Hee Lee, Dong-Hyuk Im, and Hyoung-Joo Kim
Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea
{leedh, dhim, hjk}@idb.snu.ac.kr

Abstract. RDF is recommended by W3C for web-based information. Recently, many web sites distribute RDF-formatted information. Since the online information is updated frequently, a change detection tool is needed. However, the current research on detecting changes still remains insufficient. With blank nodes, detecting changes between two RDF graphs is difficult. In this paper, we propose a change detection technique supporting nested blank nodes of RDF documents by using a labeling scheme for blank nodes, predicate-grouping and triple partitioning. The experimental results show that our approach is more efficient and accurate than previous works which compare all blank nodes.

Keywords: RDF, change detection, document update

1 Introduction

RDF (Resource Description Framework) is recommended by W3C (World Wide Web Consortium) as the language for representing the meta-data on the web. RDF enables software agent to facilitate automated processing of web resources [1]. As aiming at the semantic web, many web sites provides the data as the form of RDF and many storages and query languages for RDF are developed [2], [3]. These RDF documents are currently used in a variety of area. Especially, in bioinformatics, they use RDF for storing genetic information. Since there are many updates in data on the web, we need a change detection tool that finds the differences between two versions. Generally, changed part is relatively small compared with the whole document. Therefore, it is more efficient to update only changes. If data is very large, we can get only updated parts and apply them to previously stored data. For example, Gene Ontology [4] has changed daily and Uniprot data [5] has changed per two weeks. However, there are no distributions about changes. Since the most web sites provide new data, we do not know what has changed between two versions. A change detection can be useful the following cases:

- **Versioning and Querying the past**. We have need of management for versioning documents while using documents. We make a record each version of document changed continuously like history information. RDF is widely used to carry out the tasks such as creating ontology for annotations. Therefore it is important to store

each version of data annotated and to allow queries about individual versions. If we use a change detection tool, we can manage each version efficiently by using only changes.

- **Profit in coordinated activities**. In collaborative environments, there are many cases that each group member should update and revise the same data. When we integrate individual work, conflicts may happen. GNU diff utility [6] comparing two documents is widely used as the part of management system for version control. Using a tool like this, we can find changes and manage versioning using them.

RDF data model consists of triple statements. Three parts of a triple are called, respectively, the subject, the predicate, and the object. RDF graph is the set of these triples. Fig. 1 shows an example of RDF graph. In RDF statements, the subjects may be URIs, the predicates may be URIs and the objects may be either URIs or character strings called literal. Literals may not be used as subjects or predicates in RDF statements [7]. If statements consist of URIs and literals, computing the differences between two triples by a comparison of two texts is easy. However, if documents contain blank nodes such as _:1 in Fig. 1, matching two documents is very difficult because system gives blank nodes arbitrary labels for the purpose of treating them as named nodes. It makes the same blank nodes to get the different labels. Since blank nodes are used in practice, it is important to compute the equality between blank nodes.

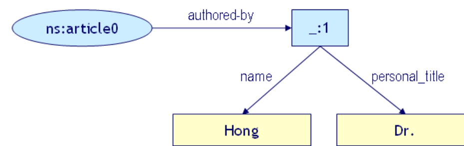

Fig. 1. A small RDF model with blank node

In this paper, we propose a change detection technique supporting blank nodes by analyzing the patterns of blank nodes in practical RDF documents. In addition, we propose a change detection using the patterns of the subjects, the predicates and the objects in statements. In our approaches, blank nodes in practical RDF are expressed as being nested by subject which has the first URI. Thus subgraphs which are connected to blank nodes can be represented in a tree structure which has the first subject as the root. We can get the difference by minimum-cost comparing two trees which have the same root node using a labeling scheme. Additionally, we can find that the kind of the predicates in statement is limited in comparison to that of the subjects or the objects. Therefore we compute the differences between two RDF graphs by not comparing all triples but comparing only the group partitioned by the predicates. The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 presents the characteristics of RDF for change detection. Section 4 proposes the detail of our approaches. Section 5 gives the performance results. The conclusion is contained in Section 6.

2 Related Work

Many change detection techniques are widely introduced not only in text files but also in HTML and in XML. The GNU diff utility [6] using the LCS (Longest Common Subsequence) algorithm is frequently used. However, this utility can compare only two text files. This cannot be applied to structured data such as XML and RDF.

Delta [8] is a text diff for RDF graphs. It gives update ontology for RDF and proposes the distribution containing update information. It describes a limited method for matching blank nodes that have not identified numbers. It can support the matching only when the predicate is a *functionaproperty* and an *inverseFunctional property*. Because of this restriction, it cannot be applied to practical RDF documents.

[9] presents a canonicalization algorithm that serializes an RDF graph. First, this method works by sorting the triples of each RDF graph with blank node identifier lexicographically. Then for matching blank nodes, it replaces the initial identifiers with the new identifier. If the blank node identifiers in each document are the same, these blank nodes are considered the same. Comparing two canonical forms is more efficient than comparing two RDF graphs using the text diff. However, as the triple is added or deleted, the blank node identifier is also changed. This makes more differences in comparing triples.

Jena [3] computes only the differences of triples not considering blank nodes. Although two blank nodes compared for matching are the same, the identifier given by the RDF parser can be different. Therefore these same nodes are regarded as different. Problem with this approach is that the matching result is larger than exact result.

X-Diff [10] is a change detection algorithm for XML. XML documents can be represented in a tree structure. Thus, the methods using the relationship parents and children for matching the trees are widely used. X-Diff uses the signature as this method. However, RDF document is a graph structure. We can not use this method as it is.

3 Characteristics in Practical RDF Documents

A blank node in RDF is used when a resource URI is not meaningful [11]. To distinguish blank nodes from other resources, the blank node identifier is given by the RDF parser. Fig. 2 shows RDF graph of RDF document. In Fig. 2, Since the triple “John ns:brother Jack” consists of the URI, We can find easily whether this triple has been changed or not. However, in the triples containing blank nodes such as _:1, _:2, _:3, _:4 in Fig. 2, computing the differences is very difficult. Since their identifiers are created arbitrarily, a blank node labeled with _:1 can be labeled with _:2 in another document. In addition, we find that each blank node has many outgoing edges and only one incoming edge in most practical RDF documents. This happens when a blank node is used only once in a document and described as being nested. For

example, a blank node _:1 in Fig. 2 has one incoming edge and the form of being nested by John which has labeled with URI.

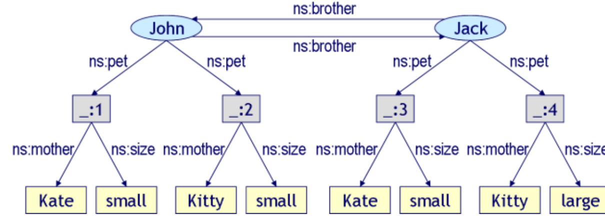

Fig. 2. An Example for RDF graph

If writing the RDF document using rdf:nodeID, a blank node can be referenced more than once. In this case, a blank node has many incoming edges and is not nested by unique node with URI. However, using a URI is appropriate for referencing a blank node many times [11]. Thus, practical RDF document has a nested form.

To detect the changes efficiently, we use the characteristics of nested blank nodes in RDF documents which are widely used in real application. Since a nested blank node has one incoming edge, the subject and the predicate in a triple that has blank node as the object position are important. Furthermore, the subgraph containing a blank node can be represented as a small tree that has the first URI as the root and the URI or literal as the leaf. Fig. 3 shows trees after applying this rule to RDF graph in Fig. 2 and trees after updating RDF graph.

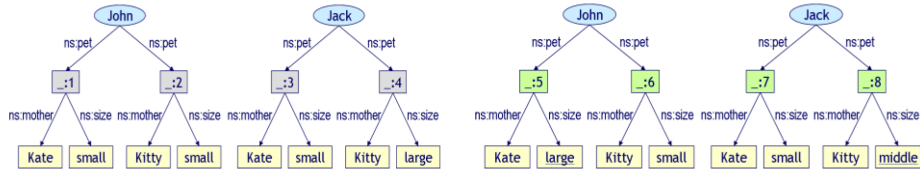

Fig. 3. Trees for RDF graph (left), and trees after update (right)

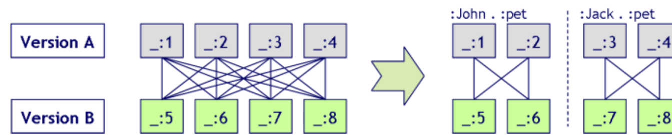

Fig. 4. The minimum-cost matching

Intuitively, we can easily get the differences between triples which do not contain a blank node by a comparison of two texts. However, we have to take the blank nodes into account. With respect to the improved comparison, it is not a good idea to match every blank node in the old version to every blank node in the new version. If we use

this property of nested blank nodes, we can compare only blank nodes that have the same root. Fig. 4 shows the minimum-cost matching between blank nodes.

4 Change Detection Technique

Our approach is introduced in this section. Section 4.1 describes the outline of our change detection technique. Section 4.2 proposes labeling scheme for blank nodes. Section 4.3 presents the matching between two blank nodes and section 4.4 describes the improved matching between the triples which do not contain blank nodes using predicate-grouping and triple partitioning.

4.1 Overview of Change Detection

First, given two RDF documents, D_1 and D_2, we verify if D_1 is different from D_2. If so, we make the patch file containing changes. In the case in which triples contain blank nodes in D_1, comparing them to the triples containing blank nodes in D_2. If the subject, the predicate and the object of triple are URI or literal, comparing them to the remainder in D_2. Steps in change detection are as follows.

1. Parsing and Labeling
2. Matching between Blank Nodes
3. Predicate-grouping and Triple Partitioning

4.2 Parsing and Labeling

Returning to the properties of blank nodes, if a blank node is nested, then the subgraph containing a blank node can be represented as a small tree that has the first URI as the root which is the ancestor of the blank node. Thus, computing the differences between two blank nodes can be easily obtained by comparing only two trees with the same root node. This reduces unnecessary comparisons. To support comparing the blank nodes, we propose a labeling scheme for blank nodes. The following defines the method of labeling.

Definition 4.1. Suppose x is a blank node in a tree T. Label(x) = Name(x_1).Name(x_2). … .Name(x_n), where x_1 is the root of T which has the first URI, (x_2, x_3, … , x_n) is the predicate with URI in the path from root to x_n.

Note that a blank node may be used as the object in triple. However, since a blank node is used in order to describe the resource with URI, finally, the leaf node of tree is URI or literal. Fig. 5 shows an example of labeling scheme. In Fig. 5, a blank node _:1 is labeled with “John.ns:pet”, a blank node _:3 is labeled with “Jack.ns:pet”. As

the depth of tree is increased, we add the predicate to a label. For instance, if _:1 in Fig. 5 is not literal but a blank node, this label is "John.ns:pet.ns:mother".

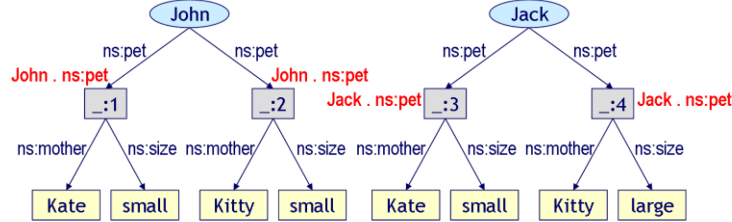

Fig. 5. RDF graph after labeling blank nodes

4.3 Matching between Blank Nodes

To detect changes in RDF documents D_1, D_2 containing blank nodes, we must verify which blank nodes can be matched. As stated in previous section, this problem is matching two trees with the same root. Thus, we compute the minimum-cost matching two trees T_1, T_2. Since RDF graph contains a lot of trees, we must compare a pair of trees repetitively. Matching from T_1 to T_2 is widely used in change detection for XML. We consider only insert operation and delete operation. To simplify the computation of the minimal cost, we assume that update operation do not happen in RDF statement [12]. This means that RDF statement can only be added and removed.

Definition 4.2. Suppose both x and y are blank nodes, label(x) = label(y) and $x \in D_1$, $y \in D_2$. Let sub be the subject, pred be the predicate and obj be the object in triple. When we transform D1 into D2, the set of deleted triples, D, and the set of inserted triple, I, are as follows.
– D = {(pred, obj) | Triple(x, pred, obj) $\in D_1$ and Triple(y, pred, obj) $\notin D_2$}
– I = {(pred, obj) | Triple(x, pred, obj) $\notin D_1$ and Triple(y, pred, obj) $\in D_2$}
And the cost of transforming is Dist(x, y) = n(I) + n(D).

Definition 4.3. If there is no matching node, the cost operation is that all nodes in tree are deleted or inserted. This notation is defined as follows.
– $Dist(T_x, \varphi) = Cost(Delete(T_x))$, $Dist(\varphi, T_y) = Cost(Insert(T_y))$

The cost of transforming two trees, $Dist(T_1, T_2)$, is computed by dynamic programming method. First, we compute the cost of two blank nodes in the bottom-up fashion. We can save the cost of subtrees we have already computed. If computation contains two blank nodes, in order to get the minimum-cost matching between two blank nodes in each subtree, we use the minimum-cost maximum flow algorithm for minimum-cost bipartite mapping. Among several implementations of this, we select Hungarian method [13], [14], [15].

Using the minimum-cost bipartite mapping, computing the differences between two trees is optimal solution. This proof is described in X-Diff [10].

4.4 Predicate-grouping and Triple Partitioning

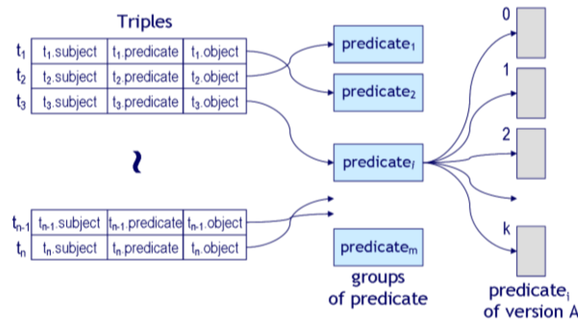

Fig. 6. Predicate-grouping and partitioning method

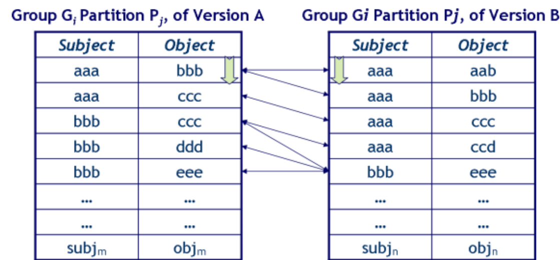

Fig. 7. The comparing procedures in partition

For the triples which do not contain a blank node, we can easily compute the differences by sorting the triples. However, to deal with large data, we have to consider high efficiency. We find that the kind of the predicates in statement is limited in comparison to the subjects and the objects. Thus, we propose predicate-grouping and triple partitioning. Fig. 6 describes predicate-grouping and partitioning method. We use the hash function as the partitioning method. The hash key is the subject and the object in triple. Since triples are sorted in each partition, we compute the differences between two RDF graphs by not comparing all triples but comparing only the group partitioned by the predicates. Fig. 7 shows the comparing procedures in partition. Comparing two partitions is performed just like merge-sort.

5 Performance Evaluation

Our change detection tool is implemented in Java using SDK 1.5.0. We use Rio 1.0.8 in Sesame 1.2.4 as RDF parser. We ran the experiments on Pentium 4 3.2GHz PC with 2GB memory. We used two data sets for experiments. One data set is Gene Ontology Term DB [4] and eco data in KEGG Pathway [16]. These data contain nested blank nodes. The size of GO RDF document is 23MB. We transform pathway data in XML to RDF using XSLT [17]. The size of this eco RDF is 9MB. The other data set is UniProt Taxonomy [18]. UniProt do not have blank nodes and its size is 108MB. Thus, this set was used for predicate-grouping and partitioning experiment.

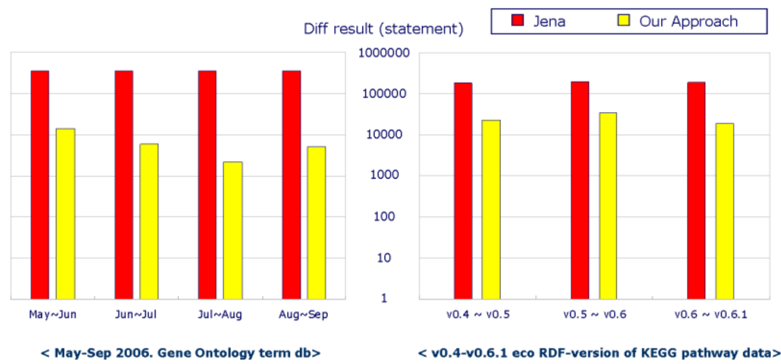

Fig. 8. Quality of change detection result

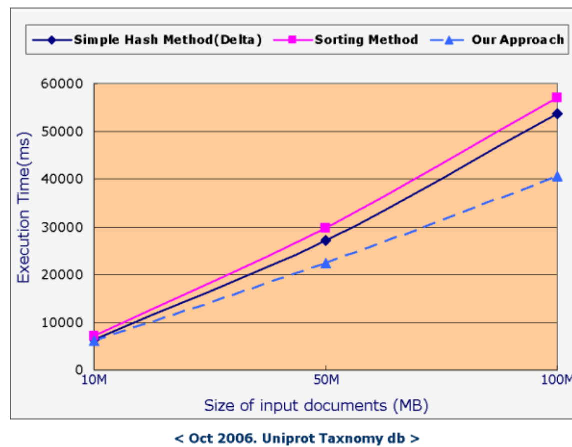

Fig. 9. Execution time of change detection

First we tested the difference result for our approach and Jena with the first data set in various versions. Fig. 8 shows the comparison result of change detection. Jena does

not consider blank nodes. Although two blank nodes are the same, each blank node is regarded as different. On the other hand, our approach finds out the optimal difference.

The next experiment is to test the performance of the change detection with the UniProt data increasing the data size. We evaluate the execution time of the three methods, simple hash method, sorting method and grouping-partitioning method. Sorting method is the naïve algorithm and simple hash method in Delta use a hash function when comparing triples. Fig. 9 illustrates the change detection time in three methods. As the data is large, our grouping-partitioning has the best performance.

6 Conclusion

As RDF documents are changed frequently, change management for RDF Documents is important issue. In this paper, we propose a change detection technique for RDF documents. Blank nodes cause the problem in computing the differences between two RDF graphs. Thus, the matching between blank nodes is needed. We find that blank nodes exist in a nested form in RDF. This is practically applicable to most RDF Documents. Since RDF graphs can be divided into several trees, we can use the minimum-cost matching between two trees with the same root. We also propose predicate-grouping and partitioning method for high-efficiency when triples do not contain blank nodes. The experiments show that our technique generates more accurate results and runs faster than previous works.

Acknowledgments. This work was supported by the Brain Korea 21 Project in 2007 and the Ministry of Information and Communication, Korea, under the College Information Technology Research Center Support Program, grant number IITA-2005-C1090-0502-0016.

References

1. Ora Lassila, Ralph, R., Swick.: Resource Description Framework (RDF) Model and Syntax Specification. W3C. http://www.w3c.org/TR/1999/REC-rdf-syntax-19990222/
2. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In proceedings of the International Semantic Web Conference. (2002)
3. Carroll, Jeremy J., Dickinson, Ian., Dollin, Chris, Reynolds, Dave, Seaborne, Andy, Wilkinson, Kevin.: Jena: Implementing the Semantic Web Recommendations. International World Wide Web Conference (2004)
4. Gene Ontology Consortium. http://geneontology.org
5. UniProt :The Universal Protein Resource. http://www.pir.uniprot.org
6. CVS : Concurrent Versions System(CVS) . Free Software Foundation. http://ww.gnu.org/manual/cvs-1.9

7. Frank Manola et al. RDF Primer. Technical report, W3C, http://www.w3.org/TR/rdf-primer/, 2004.
8. Tim Berners-Lee, Dan Connolly: Delta:An Ontology for the Distribution of Differences Between RDF Graphs. http://www.w3.org/DesignIssues/Diff
9. Carroll, Jeremy J.: Signing RDF Graphs. In Proceedings of the International Semantic Web Conference (2003)
10. Wang, Y., DeWitt, D., Cai J. Y.: X-Diff: An Effective Change Detection Algorithm for XML Documents. In 19th International Conference on Data Engineering (2003)
11. Shelly Powers. : Practical RDF. O'Reilly & Associates. (2003)
12. Ognyanov, D., Kirakov, A.: Tracking Changes in RDF(S) Repositories. In the Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management. (2002)
13. Harold W. Kuhn.: The Hungarian Method for the assignment problem. Naval Research Logistic Quarterly (1955)
14. James Munkres.: Algorithm for the Assignment and Transportation Problems. Journal of the Society of Industrial and Applied Mathematics. 5(1) (1957) 32-38
15. Hungarian algorithm, http://en.wikipedia.org/wiki/Hungarian_algorithm#Algorithm
16. Kanehisa, M., Goto, S.: KEGG: Kyoto Encyclopedia of Genes and Genomes. Nucleic Acids Research. (2000)
17. W3C. KEGG RDF Mapping, http://www.w3c.org/2005/02/13-KEGG/
18. UniProt RDF, http://expasy3.isb-sib.ch/~ejain/rdf/