Article

JIS

Journal of Information Science
39(2) 238–255

DOI: 10.1177/0165551512463650
jis.sagepub.com
SAGE

Backward inference and pruning for RDF change detection using RDBMS

Dong-Hyuk Im
School of Computer Science and Engineering, Seoul National University, Seoul, Korea

Sang-Won Lee
School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea

Hyoung-Joo Kim
School of Computer Science and Engineering, Seoul National University, Seoul, Korea

Abstract
Recent studies on change detection for RDF data have focused on minimizing the delta size and, as a way to exploit the semantics of RDF models in reducing the delta size, the forward-chaining inferences have been widely employed. However, since the forward-chaining inferences should pre-compute the entire closure of the RDF model, the existing approaches are not scalable to large RDF data sets. In this paper, we propose a scalable change detection scheme for RDF data, which is based on backward-chaining inference and pruning. Our scheme, instead of pre-computing the full closure, computes only the necessary closure on the fly, thus achieving fast and scalable change detection. In addition, for any two RDF data input files to be compared, the delta obtained from our scheme is always equivalent to the one from the existing forward-chaining inferences. In addition, in order to handle RDF data sets too large to fit in the available RAM, we present an SQL-based implementation of our scheme. Our experimental results show that our scheme, in comparison to the existing schemes, can reduce the number of inference triples for RDF change detection by 10–60%.

Keywords
backward-chaining inference; change detection; ontology; pruning; RDF

1. Introduction

RDF (Resource Description Framework) is recommended by W3C (World Wide Web Consortium) as the language to represent information about web resources in the semantic web. Many web sites publish their data in the RDF format, as RDF has a simple data model, well-defined formal semantics and provable inference [1]. In addition, RDF is known to be useful for integrated query processing because it can easily integrate information from diverse data sources. Therefore, users can find meaningful information using the RDF query language (i.e. SPARQL), irrespective of the data format.

RDF data can change over time for various reasons [2, 3]. First, as the information itself can change in the domain of interest, RDF that models the knowledge of that domain should also change accordingly. Second, RDF data can change according to the user's needs. Since RDF data is, in general, created and evolved in a distributed environment, multiple users can manage the RDF data in a collaborative manner. Changes made by a user should be propagated to the other users of the domain to maintain consistency in the RDF data. Therefore, detecting and representing RDF changes (i.e. RDF deltas) are crucial in evolution process [4], synchronization system [5] and versioning system [6–9] to trace the history of changes in an RDF data set to synchronize two or more versions correctly, and compare different versions.

Corresponding author:
Dong-Hyuk Im, School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea.
Email: dhim@idb.snu.ac.kr

The main role of change detection tools is to determine what has been changed between two models. The change detection between RDF data is similar to that in flat files or structured data (e.g. HTML and XML). For example, the GNU Diff utility is a popular tool to detect changes in program source files and text files. Several change detection algorithms based on the tree model have been proposed for structured documents such as XML [10, 11]. However, these traditional algorithms cannot be generalized to handle the RDF model, because they do not fully utilize the semantic information in RDF data. Therefore, Tim Berners-Lee emphasizes that it is necessary and important to develop efficient change detection schemes for RDF graphs to exchange and distribute RDF deltas [12]. In addition, as RDF deltas would be actively used in the synchronization system and versioning system, it is necessary to minimize the delta size for the fast exchange over the network, as well as storage efficiency [5, 7, 13, 14]. Thus, several approaches have been proposed to minimize the size of RDF deltas by exploiting the semantics of the RDF model [9, 15, 16].

Figure 1 illustrates an example of change in the RDF model. In Figure 1, we represent each RDF model (K and K') as the set of triples, and detect changes between two RDF triple sets using the set-difference operation. RDF change detection, such as Delta [12], has taken this approach, since it is very simple to compute RDF deltas using set-difference operation. However, by applying the RDF inference rules under the RDFS (RDF Schema) specification to the triple set [17], we can derive new triples from the existing ones; that is, the closure of the RDF model (the RDF closure). This result of the closure enables us to reduce the size of the differences between two RDF triple sets. For example, when we compute the differences between K and K', even if a triple (TA subClassOf Univ_Person) is deleted from K in Figure 1, we can infer the triple (TA subClassOf Univ_Person) by computing the transitive closure of the triples in K'. That is, although this triple is explicitly deleted from one set (syntactic level), there is no semantic difference between two sets if we can infer the deleted triple in the other set (semantic level). Therefore, this semantic property of the RDF delta allows us to minimize the size of differences by exploiting the semantics of the RDF model.

Existing work, such as SemVersion [9], pre-computes the complete closures of both the RDF models and then performs the set-difference operation between two closures. In this respect, we term this approach the forward-chaining inference strategy. However, it is very time-consuming to compute and store the entire closures of both the RDF models in advance. In addition, forward-chaining inference-based change detections must cover all derived triples in computing the differences between two sets of triples. To make it worse, in general only a small fraction of the entire data in RDF applications changes. (The result reported in PromptDiff [8] shows that 97.9% of the data in each version remains unchanged.) Thus, it should compute the redundant closures of more than 90% of the entire data that are irrelevant to the result of changes. On the other hand, Delta function [15, 16] computes the closure on-the-fly using a labelling scheme (e.g. interval-labelling scheme [18]). Thus, it efficiently checks the transitive closure without computing all the derived triples. In Delta function, the label information itself should be constructed by performing two plain traversals

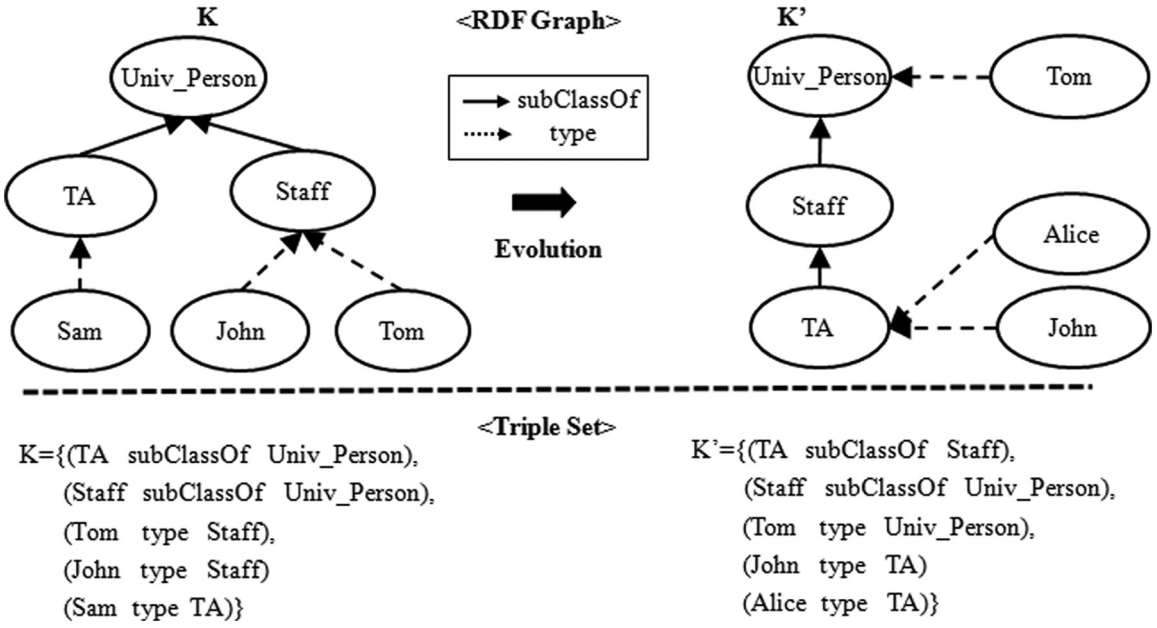

Figure 1. An example of change in the RDF model.

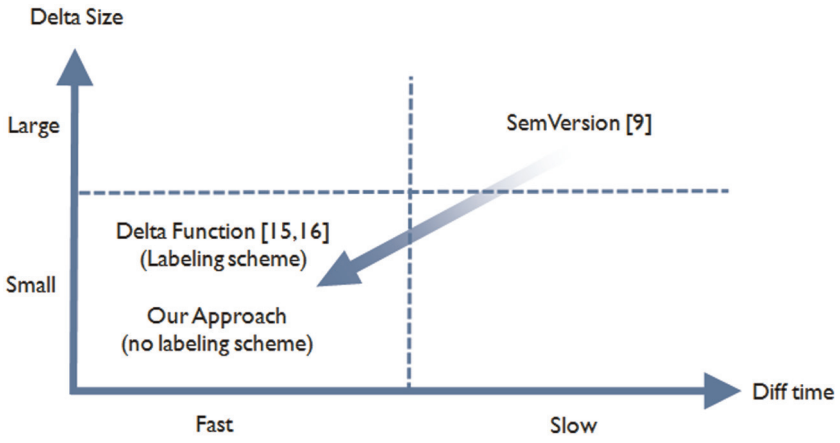

Figure 2. Classification of RDF change detection techniques.

of the explicit graph by the subClassOf and subPropertyOf relations. However, since we assume the DRAM memory is not large enough to keep the two RDF triple sets, it requires additional overhead (e.g. construction time for the labels and additional space to store the label information). Thus, the existing change detection approaches are neither efficient nor scalable.

These observations lead us to develop scalable solutions to compute differences between the large data efficiently. This issue is critical in terms of incremental maintenance. For example, when a large RDF data set changes frequently, we need a change detection tool that can efficiently find the differences between two consecutive versions to update from the old version to the new one. In this paper, we propose a scalable change detection technique for RDF data. It computes only the relevant closure of the RDF model using a backward-chaining inference strategy. In addition, we propose a new pruning method, which skips unnecessary backward-chaining inference, to improve the performance of change detection using the backward-chaining inference. While previous work on RDF change detection mainly focused on minimizing the delta size, our backward-chaining inference strategy, in combination with the pruning method, can drastically improve change detection performance, as well as minimizing the delta size. Figure 2 shows the generic comparisons of our scheme to existing ones.

The contributions of this paper can be summarized as follows:

- **Pruning method based on backward-chaining inference**. We propose a change detection technique using the backward-chaining inference strategy and pruning method. When we detect RDF changes using the inference scheme, we can compute a small part of the RDF closure, not the entire closure. Although some studies have dealt with inference issues in RDF query processing [19–21], no work has tried to detect RDF changes efficiently. Our pruning scheme-based change detection is faster than change detection using forward-chaining inference [9] by a factor of 10–80 and faster than any existing change detection technique using only backward-chaining inference for real data sets [20, 21] by a factor of 1.5–4.
- **Correctness of change detection**. We show that our change detection generates the same result of RDF deltas as that produced by the forward-chaining inference strategy.
- **Change detection based on relational database**. We propose the change detection for RDF data using relational databases. Memory-based change detection is not suitable for large RDF data sets, as it requires much memory to retain two RDF triple sets. In contrast, in our implementation, we store large RDF data sets in a relational database and compute the delta using SQL queries against the database, which embodied our backward-chaining inference and pruning scheme. As far as we know, this is the first implementation of backward inference-based RDF change detection using relational database.

The remainder of this paper is organized as follows. Section 2 reviews the RDF data model, background concepts on RDF inference and five well-known change detections for RDF. Section 3 proposes a new RDF change detection based on the backward-chaining inference and pruning method. Section 4 explains how we implement our change detection in a relational database. Section 5 presents the experimental results. Section 6 concludes the paper.

2. Background

In this section, we explain the RDF data model and the basic concept of inference used throughout this paper. In addition, we describe two RDF inference strategies (forward-chaining and backward-chaining). We then review five existing RDF change detection techniques and explain their limitations.

2.1. RDF preliminaries

RDF is a language to represent metadata about resources in the World Wide Web [1]. RDF data is modelled as a directed labelled graph, where each node and each arc represent a resource and a relationship between two resources, respectively. In general, an RDF graph is represented as a set of triples that represent binary relationships between two resources, and can be formally defined as follows:

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

The RDF inference (i.e. RDF closure calculation) is based on the RDFS entailment rules provided by the RDFS semantics specification [17]. However, we do not consider all of RDFS entailment rules when computing the inferential closure of an RDF Schema model. That is, we ignore some of entailment rules because they do not affect minimizing the delta size. For example, although we apply the rules (U A Y $\Rightarrow$ A rdf:type rdf:Property) and (U A Y $\Rightarrow$ U rdf:type rdfs:Resource) from the RDFS entailment rules to RDF data, we cannot reduce RDF deltas. The closure rules for our change detection among the RDFS entailment rules are defined as follows. These closure rules play crucial roles in limiting the coverage of inference in our change detection method.

Definition 2 (Closure Rule). The definitions of closure rules for change detection are as follows:

Rule 1. (U rdfs:subPropertyOf V) $\wedge$ (V rdfs:subPropertyOf X) $\Rightarrow$ (U rdfs:subPropertyOf X).

Rule 2. ($U\ A\ Y$) $\wedge$ (A rdfs:subPropertyOf B) $\Rightarrow$ ($U\ B\ Y$).

Rule 3. (V rdf:type U) $\wedge$ (U rdfs:subClassOf X) $\Rightarrow$ (V rdf:type X).

Rule 4. (U rdfs:subClassOf V) $\wedge$ (V rdfs:subClassOf X) $\Rightarrow$ (U rdfs:subClassOf X).

Definition 3. Given an RDF model K, the closure of K, denoted by $C(K)$, is defined by recursively applying the closure rules to K and adding the derived triples to K.

New triples can be inferred from existing ones using the closure rules in Definition 2. For example, we can obtain $C(K)$ by adding the derived triples (John type Univ_Person, Rule 3), (Sam type Univ_Person, Rule 3) and (Tom type Univ_Person, Rule 3) to K. Likewise, we compute $C(K')$ by adding the inferred triples (TA subClassOf Univ_Person, Rule 4), (John type Univ_Person, Rule 3), (John type Staff, Rule 3), (Alice type Staff, Rule 3) and (Alice type Univ_Person, Rule 3) to K'.

2.2. RDF inference strategy

In this section, we provide background on RDF inference strategy and show how RDF inference is used. Existing RDF inference strategies can be classified into two approaches based on the direction of inference: forward-chaining inference strategy and backward-chaining inference strategy [19]. Forward-chaining inference derives all new triples from the existing ones with the closure rules, until no more triples can be derived. This forward-chaining is preferred when data are uploaded (load-time inference), but it can often result in a long load time and excessive space overhead. However, its main advantage is that it can boost query processing time using the pre-computed triples in query processing, instead of computing the triples on the fly. In contrast, the backward-chaining approach computes the relevant triples among all the closures on the fly during query processing (run-time inference) [21]. Therefore, it can quickly load data and requires little storage overhead, but it can increase the query response time. Since the query performance is of more concern than storage overhead, most RDF storage systems choose the forward-chaining strategy. For example, Sesame [22] uses the exhaustive forward-chaining algorithm to pre-compute the closure and stores all derived triples in the database. In addition, Broekstra and Kampman [19] present how to maintain consistency in RDF storage: if we need to insert (or delete) a triple in forward-based system, we should also insert (or delete) the triples derived by inference. Conversely, Shen and Qu [20] and Stuckenschmidt and Broekstra [21] propose a flexible inference strategy (combining forward-chaining and

backward-chaining) in RDF management systems to mitigate excessive space overhead. For ease of presentation, we may use forward-chaining inference strategy and forward inference interchangeably (respectively, backward-chaining inference strategy and backward inference).

2.3. Change detection techniques for RDF data

In this section, we overview the syntax and semantics of existing change detection techniques for RDF data. We term them ΔE, ΔC, ΔD, ΔDC and ΔED, respectively [9, 12, 15, 16]. We model the change operations in the RDF data only with triple insertions and triple deletions [23]. A triple update is assumed to be modelled as a triple deletion-then-insertion. In general, these change operations are common in the RDF change detection field. Given two RDF triple sets K and K', let $C(K)$ and $C(K')$ denote the closure of K and K', respectively. Given a triple $t(S\ P\ O)$, let $\mathrm{Del}(t(S\ P\ O))$ denote triple deletion, and $\mathrm{Ins}(t(S\ P\ O))$ denote triple insertion.

The first method, ΔE (Explicit Delta), computes the differences using set arithmetic for triple sets. Since this method is intuitive and simple, many change detection tools [12, 24–26] choose this approach, and we categorize ΔE as the syntactic level. In ΔE, Del denotes $K - K'$ and Ins denotes $K' - K$.

$$\Delta E = \{\mathrm{Ins}(t(S\,P\,O)) \,|t(S\,P\,O) \in K' - K\} \cup \{\mathrm{Del}(t(S\,P\,O)) \ |t(S\ P\ O) \in K - K'\}$$

The second method, ΔC (Closure Delta), exploits the semantics of the RDF model. It uses the RDF schema entailment rules on K and K'. It first calculates $C(K)$ and $C(K')$ by applying the closure rules to K and K', and then calculates the differences between $C(K)$ and $C(K')$. Consider an example where a triple t in K is not in K'. Even though the triple t is not in K', it might not be included in ΔC where $t \in C(K')$. This means that, when computing differences, we can remove the triples that can be inferred in the other set from the deltas. Although ΔC is different from ΔE, we can obtain the same $C(K')$ if we compute the RDF closure after applying each RDF delta (ΔE, ΔC) to K. Some RDF management systems store RDF triple set and the inferred triples together, and support query processing using the closure of the RDF model. Since the same RDF closure can be generated by ΔC and ΔE, ΔC is useful for managing and exchanging the deltas [9, 15, 16]. ΔC is computed as follows:

$$\Delta C = \{\mathrm{Ins}(t(S\,P\,O)) \,|t(S\,P\,O) \in C(K') - C(K)\} \cup \{\mathrm{Del}(t(S\,P\,O)) \ |t(S\ P\ O) \in C(K) - C(K')\}$$

However, since ΔC computes the closure of both sets and then performs a set-difference between two closures, ΔC is not necessarily smaller than ΔE. In fact, the delta size of ΔC could be larger than that of ΔE, in case one set has many inferred triples in one set that are not inferred in the other set and thus belong to ΔC.

To cope with this problem, Delta Function proposes ΔD (Dense Delta), ΔDC (Dense&Closure Delta) and ΔED (Explicit&Dense Delta) [15, 16]. They also exploit the semantics of the RDF model for change detection like ΔC. However, these differ from ΔC, in that they do not perform the set-difference between both closures. Although ΔD generates the smallest delta of the change detection techniques using RDF inference, ΔD is not always correct. Thus, only ΔDC and ΔED can be used for synchronization in the general case. In most practical cases, ΔED produces smaller deltas than ΔDC [16]. ΔD, ΔDC and ΔED are computed as follows:

$$\Delta D = \{\mathrm{Ins}(t(S\ P\ O)) \ |t(S\,P\,O) \in K' - C(K)\} \cup \{\mathrm{Del}(t(S\,P\,O)) \ |t(S\,P\,O) \in K - C(K')\}$$

$$\Delta DC = \{\mathrm{Ins}(t(S\,P\,O)) \ |t(S\,P\,O) \in K' - C(K)\} \cup \{\mathrm{Del}(t(S\,P\,O)) \ |t(S\,P\,O) \in C(K) - C(K')\}$$

$$\Delta ED = \{\mathrm{Ins}(t(S\,P\,O)) \ |t(S\,P\,O) \in K' - K\} \cup \{\mathrm{Del}(t(S\,P\,O)) \ |t(S\,P\,O) \in K - C(K')\}$$

By applying the above five different change detection schemes against the example in Figure 1, we can obtain the following results of change detections.

$$\Delta E = \{\mathrm{Del}(\text{TA subClassOf Univ\_Person}),\ \mathrm{Del}(\text{Sam type TA}),\ \mathrm{Del}(\text{Tom type Staff}),\ \mathrm{Del}(\text{John type Staff}),\ \mathrm{Ins}(\text{TA subClassOf Staff}),\ \mathrm{Ins}(\text{John type TA}),\ \mathrm{Ins}(\text{Tom type Univ\_Person}),\ \mathrm{Ins}(\text{Alice type TA})\}$$

$$\Delta C = \{\mathrm{Del}(\text{Sam type TA}),\ \mathrm{Del}(\text{Tom type Staff}),\ \mathrm{Del}(\text{Sam type Univ\_Person}),\ \mathrm{Ins}(\text{John type TA}),\ \mathrm{Ins}(\text{Alice type TA}),\ \mathrm{Ins}(\text{Alice type Staff}),\ \mathrm{Ins}(\text{Alice type Univ\_Person})\}$$

$$\Delta D = \{\mathrm{Del}(\text{Sam type TA}),\ \mathrm{Del}(\text{Tom type Staff}),\ \mathrm{Ins}(\text{TA SubClassOf Staff}),\ \mathrm{Ins}(\text{John type TA}),\ \mathrm{Ins}(\text{Alice type TA})\}$$

$$\Delta DC = \{\text{Del}(\text{Sam type TA}),\ \text{Del}(\text{Tom type Staff}),\ \text{Del}(\text{Sam type Univ\_Person}),\ \text{Ins}(\text{TA subClassOf Staff}),\ \text{Ins}(\text{John type TA}),\ \text{Ins}(\text{Alice type TA})\}$$

$$\Delta ED = \{\text{Del}(\text{Sam type TA}),\ \text{Del}(\text{Tom type Staff}),\ \text{Ins}(\text{TA subClass Of Staff}),\ \text{Ins}(\text{John type TA}),\ \text{Ins}(\text{Tom type Univ\_Person}),\ \text{Ins}(\text{Alice type TA})\}$$

In summary, conventional change detection tools have developed to minimize the delta size by exploiting the semantics of the RDF model. In this paper, we consider only inference-based change detection. In particular, we focus on the change detection which produces the smaller deltas and the correct deltas (i.e. ΔED). Other issues on executing the RDF deltas, such as the order of change operations and the execution semantics of change operations [15, 16], are beyond the scope of this paper.

2.4. Limitations of existing change detection methods

As explained above, the basic operations in RDF change detection are the set-difference operation and the inference operation. In particular, since all the inferable triples from an RDF data set are collected by applying the deduction rules recursively, the overhead of the inference operation is large. Let us consider the inference strategy in existing change detection. SemVersion [9] uses forward inference for change detection. SemVersion materializes the complete closure in advance and then performs set-difference. Consider the change detection that produces the smaller size changes ΔED using forward inference. Given two RDF triple sets K and K', the change detection is computed as follows. First, the closure of K', denoted as $C(K')$, is computed and stored. Then, the delta ΔED is computed using two set-difference operations, $(K - C(K'))$ and $(K' - K)$. However, it is too costly to compute the entire closure. Moreover, the set-difference operations are performed by comparing sets that contain all derived triples, as well as the existing triples. Therefore, it is desirable to reduce the time required for these two steps in the case of large RDF data sets. Note that we do not consider a labelling scheme because the label information itself should be pre-computed, just like in forward inference.

Incidentally, since change detection algorithms, unlike the query processing case, use only a small part of the closure, it may not be necessary to compute the entire closure of RDF models. Therefore, from the perspective of change detection, forward inference, which computes all the closure in advance, could be quite inefficient, in that it pre-computes the closures of triples that are irrelevant to the change detection. This observation leads us to apply the backward inference to the change detection technique. Although Broekstra and Kampman [19], Shen and Qu [20] and Stuckenschmidt and Broekstra [21] discuss the backward-inference strategy in RDF management (i.e. query processing, update processing), no work uses backward inference (i.e. not a labelling scheme) in RDF change detection. Furthermore, we propose a new pruning method that skips unnecessary backward inference for efficient change detection.

3. Change detection based on backward inference and pruning method

As mentioned earlier, the main idea of our method is based on the observation that inferring only relevant triples is critical to change detection performance. In this section, we first present a method that infers only the necessary triples for change detection using a backward-chaining inference, instead of pre-computing all the triples in closures. This method improves the time and space complexity in change detection. Note that backward-chaining begins with a goal and checks if there are known facts that can satisfy the goal. That is, we need to check if the specific triple will be supported by some triples without computing the entire closure of triples.

Even with this backward-inference, however, some triples might be unnecessarily computed; this is still a time-consuming part of change detection. Therefore, we propose a pruning method in backward inference to avoid this unnecessary inference. The proposed pruning method allows us to skip some irrelevant derived triples during change detection. Then we check if each of the remaining triples can be inferred in the other set.

3.1. Outline of the proposed change detection technique

Figure 3 shows the overview of the three approaches (forward inference (case 1), backward inference (case 2) and pruning and backward inference (case 3)). The change detections based on forward-chaining produce ΔED by computing the entire closure of all triples in K' first and calculating the set-differences. In this respect, we term this approach the *inference-then-difference* strategy. In contrast, backward-inference-based change detection computes $((K - K') \cup (K' - K), \Delta E)$ using set-difference operations between two sets of triples, and then checks if every triple $t \in (K - K')$ can be included in the closure of the other set (i.e. delete t from $(K - K')$ if $t \in C(K')$). In this sense, this approach takes the

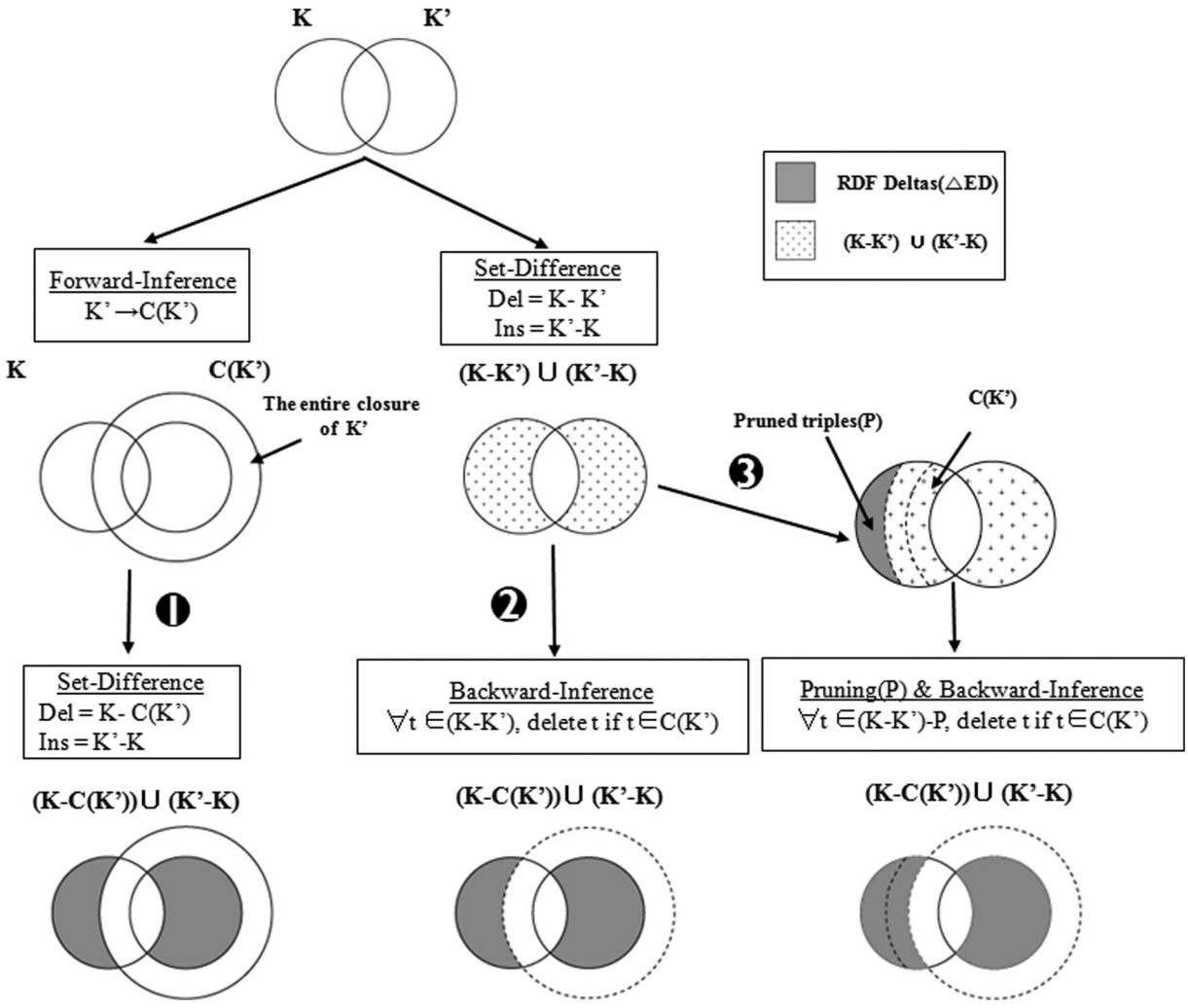

Figure 3. Overview of our change detection and comparison to existing methods.

difference-then-inference strategy. For example, triple (TA subClassOf Univ_Person) in Figure 1 is in $(K - K')$. Therefore, we are able to use backward inference to check if (TA subClassOf Univ_Person) can be derived from the set of triples in K'.

Moreover, prior to applying backward inference, our pruning method is employed to compute the closure of the relevant triples, thus avoiding the inference paths from non-relevant triples. Note that this pruning technique is applicable only to backward inference, not to forward inference. Since forward chaining calculates all the inferable triples from the entire triples, we cannot determine which triples should be pruned in advance. However, the backward-chaining approach takes the *difference-then-inference* strategy. Thus, we can apply the closure rules to only a small number of triples in $(K - K')$ (only 2–3% of data changed explicitly). In summary, backward inference in combination with pruning optimization enables us to detect changes more efficiently.

3.2. Correctness of the proposed change detection

In this section, we deal with two theoretical topics of our change detection method. First, we show that the result of change detection by backward inference is equivalent to that by forward inference. Then, we prove that some triples need not be inferred for correct change detection and could be safely pruned in backward inference, and propose the pruning condition to stop further inference.

Unlike forward inference, backward inference does not compute the entire closure in advance. Instead, it computes only the relevant closure at query time. Therefore, we must determine which triple is relevant to the delta. That is, we check if the triples can be included in the closure of the other set by backward inference. Since Delta Function proved that ΔED is a subset of ΔE, we can get ΔED from ΔE. The following Propositions show the correctness of backward inference-based change detection [16]. Given two triple sets K and K', let Del in ΔE denote $(K - K')$, Ins in ΔE denote $(K' - K)$, Del in ΔED denote $(K - C(K'))$, and Ins in ΔED denote $(K' - K)$. Since the inserted sets in ΔED and ΔE are the same triple set, we consider the deleted set.

Proposition 1. Given an RDF triple set K and its closure $C(K)$, the following condition holds: $K \subseteqq C(K)$.

Proposition 2. Given two RDF sets K and K', $(K - C(K'))$ is a subset of $(K - K')$:

$$((K - C(K')) \cap (K - K')) = (K - C(K'))$$

Proposition 2 guarantees that $(K - C(K'))$ is a subset of $(K - K')$. That is, $(K - K')$ contains all triples in $(K - C(K'))$, and we can obtain the results $(K - C(K'))$ from $(K - K')$ by finding and deleting the triples in $(K - K')$ that are inferable from the other set. These inferable triples are defined as follows:

Definition 4. Given $(K - K')$, ΔI (Inferable) is a set of deleted triples from $(K - K')$ to $(K - C(K'))$ by the backward inference process $(\Delta I = K \cap (C(K') - K'))$.

Note that, although the triples in ΔI do not exist explicitly in the other triple set, they can be inferred in the compared set. Therefore, the goal of our change detection is to find ΔI using backward inference. However, in addition to this difference-then-inference approach, there is still another chance to avoid applying backward inference to all triples in $(K - K')$. For this, we propose a pruning method that skips unnecessary inferences as early as possible in computing the closure. The following Proposition and Theorem present some properties of ΔI to show the correctness of our pruning method.

Proposition 3. Given ΔI, the following relationships hold:

$$\text{If } \mathrm{Del}(t(S\,P\,O)) \in \Delta I, \text{ then } t(S\,P\,O) \in C(K')$$

Theorem 1. Let A and B denote rdf:Property and T' denote rdf:Resource. Given a triple $t(S\ P\ O)$, the following conditions hold:

(1) If Del(t(S subClassOf O)) $\in$ ΔI, then t(S subClassOf T') $\in$ K' and t(T' subClassOf O) $\in$ K'.

(2) If Del(t(S subPropertyOf O)) $\in$ ΔI, then t(S subPropertyOf T') $\in$ K' and t(T' subPropertyOf O) $\in$ K'.

(3) If Del(t(S B O)) $\in$ ΔI, then t(S A O) $\in$ K' and t(A subPropertyOf B) $\in$ K'.

(4) If Del(t(S type O)) $\in$ ΔI, then t(S type T') $\in$ K' and t(T' subClassOf O) $\in$ K'.

Proof. First, we prove condition (1). If Del(t(S subClassOf O)) $\in$ ΔI, then t(S subClassOf O) $\in$ K and t(S subClassOf O) $\notin$ K'. According to Proposition 3, obviously t(S subClassOf O) $\in$ $C(K')$. If t(S subClassOf O) $\in$ $C(K')$ and t(S subClassOf O) $\notin$ K', then t(S subClassOf O) is a derived triple in K' by the closure rules in Definition 2. Therefore, there exist triples that satisfy both t(S subClassOf T') and t(T' subClassOf O) in K'. Likewise, we can prove conditions (2)–(4).

Based on Theorem 1, given the triple of ΔI where $t(S\ P\ O) \notin K'$ and $t(S\ P\ O) \in K$, the triple $t(S\ P\ O)$ can be created in $C(K')$ by inferring the triples that contain t(S * *) $\in$ K' and t(* * O) $\in$ K' (* is a wildcard that matches any resource). However, it is difficult to choose the exact triples for ΔI, because we cannot determine which triples can be inferred in the other set, until computing the closures of triples using backward inference. That is, not all the triples whose subject and object exist in the other set are necessarily included in ΔI. Instead, we propose a pruning method that removes the triples unrelated to ΔI from ΔE. The following Theorem describes our pruning method under the backward-inference process.

Theorem 2. Given a triple $t(S\ P\ O)$ in $(K - K')$, the following conditions hold:

(1) When Del(t(S subClassOf O)): if (t(S subClassOf T')$\notin K'$ or t(T' subClassOf O)$\notin K'$), then Del(t(S subClassOf O)) $\in$ ΔED.

(2) When Del(t(S subPropertyOf O)): if (t(S subPropertyOf T')$\notin K'$ or t(T' subPropertyOf O)$\notin K'$), then Del(t(S subPropertyOf O)) $\in$ ΔED.

(3) When Del(t(S B O)): If (t(S A O)$\notin K'$ or t(A subPropertyOf B)$\notin K'$), then Del(t(S subClassOf O => S B O)) $\in \Delta ED$.

(4) When Del(t(S type O)): If (t(S type T')$\notin K'$ or t(T' subClassOf O)$\notin K'$), then Del(t(S type O)) $\in \Delta ED$.

Proof. Theorem 2 is proved by Definition 4 and the contraposition of Theorem 1. For a given triple $t(S\ P\ O)$ where $t(S\ P\ O) \in K - K'$, if the subject or the object of triple does not exist in K', then Del($t(S\ P\ O)$) $\notin \Delta I$. Thus, by $\Delta I = \Delta E - \Delta E$D, Del($t(S\ P\ O)$) $\in \Delta E$D. Since each of conditions (2)–(4) can be similar way, we omit its proof here.

Theorem 2 is the key point of our pruning method. Since the goal of the backward inference is to generate ΔI from ΔE, the triples whose subject or object does not appear in the other set can be safely pruned from the backward-inference process. That is, these triples must be included in ΔED irrespective of using backward inference. Meanwhile, each of the remaining triples in ΔE that pass the pruning step needs to check if it is included in ΔI using backward inference.

Our pruning method considerably reduces the number of triples to be checked, in case they cannot be inferred in the other set. As mentioned in the Introduction, RDF describes information (e.g. class and instance) about resources in a specific domain. Therefore, the existing concepts in a released version might disappear in a subsequent version, or new concepts can be introduced. Since those concepts are also represented as the subject or the object in a triple, we can prune a number of triples in $(K - K')$.

4. Change detection using relational database

As the size of the RDF data set grows exponentially, scalable change detection algorithms are required. However, existing main-memory-based change detections are not scalable. That is, all the sets of triples in RDF models should be memory-resident to compute the differences between RDF models using those schemes. Therefore, for large-scale RDF data, the memory should be sufficiently large to keep the two RDF triple sets. To make matters worse, in change detection based on forward-chaining, the number of triples inferred by the closure rules increases exponentially and thus the temporary RAM does too. To solve this excessive RAM requirement in the existing approaches, we designed and implemented our change detection scheme using relational databases. Thus, as the change detection for the large XML data set in [27] uses a relational database, we present a relational approach to detect the changes in RDF data using the pruning method. Moreover, change detection based on relational database would be reliable, as well as scalable, because the technology of commercial DBMSs has been developed for more than 30 years. In addition, we need not parse the RDF document repeatedly whenever we need to compare it with another RDF model, since these two RDF triple sets are already stored in a database.

4.1. Relational schema for change detection

In order to detect changes in efficient and scalable way, we store RDF data in a relational database. Several approaches have recently been proposed for storing RDF data in relational databases [22, 28, 29]. In this paper, we use a triple store method (subject, property and object) [25], popularly used in RDF storage. In particular, we partition the triple store into several separate tables, so that change detection can be computed by examining only the small tables. Figure 4 shows the relational schema used in this paper for RDF change detection. The principle in designing the relational schema composed of four tables (SubClass, SubProperty, Type and Triple) in Figure 4 is to model the type of property in closure rules in a natural way. For example, we store the triple whose property is a 'subClassOf' to the SubClass table. Likewise, all the triples whose properties are 'type' and 'subProperty' are stored in the Type table and the SubProperty table, respectively. The remaining triples (except subClassOf, subProperty and type property) are stored in the Triple table.

This storage scheme has two advantages. First, this scheme makes it easier to compute the set-difference using SQL operators. Second, when computing the closure through backward inference, we need to access only those tables that correspond to the closure rules. For example, when computing the set-difference between 'subClassOf' relationships, we

Triple (Sub, Prop, Obj)	Ins (Sub, Prop, Obj)
SubClass (Sub, Prop, Obj)	Del (Sub, Prop, Obj)
SubProperty (Sub, Prop, Obj)	
Type (Sub, Prop, Obj)	

Figure 4. Relational schemas for change detection.

need to access only two SubClass tables from the two triple sets being compared. Similarly, the class subsumption hierarchies can be computed by accessing only the SubClass tables. In addition, since the unit of change operation is a triple, the results of change detection can be represented as the triple representation (Del table and Ins table).

4.2. Relational implementation of change detection

In this section, we present the implementation details of our change detection algorithm using a commercial relational DBMS. We then give an illustrative example for its execution.

Algorithm 1. Change detection technique using pruning method

```
01: Input: K = Set of triples in source table
02:       K′ = Set of triples in target table
03: Output: Set of change operations (ΔED: Del table & Ins table)
04: //Step 1: compute differences between K and K′
05:       Del = Set-Difference(K − K′)
06:       Ins = Set-Difference(K′ − K )
07: //Step 2: Prune unnecessary triples for backward inference
08: // I_k′: triples that take part in the backward-inference process (inferable triples)
09: For every triple x ∈ Del
10:   if ((x.subject is in K′.subject ) && (x.object is in K′.object)) then
11:   put x into I_k′
12: //Step 3: Backward-chaining inference
13: For every triple x ∈ I_k′
14:   if (Backward_Infer(x, K′))
15: //Step 4: if x can be inferred in K′, delete x from Del
16:         remove x from Del
```

Algorithm 1 shows the change detection algorithm using the pruning method in Section 3. Suppose we have two versions of the RDF triple sets, K and K'. First, the algorithm starts to compute the differences at the syntactic level between K and K' using the set-difference. Each result (i.e. $(K - K')$, $(K' - K)$) is stored in the Del table and the Ins table respectively (lines 5 and 6). Note that, for each property in closure rules, the set-difference operation is executed against the two tables corresponding to the property. Next, for every triple in the Del table, the algorithm checks if it is inferable from the other set using backward inference. However, the algorithm uses the pruning method, which is based on Theorem 2 in Section 3.2 (lines 9–11) to reduce the number of the triples to check whether if they are inferable from other set (i.e. check if they are included in ΔI). This process prunes unnecessary triples using the conditions in Theorem 2 and returns the set of the triples that participate in backward inference (we denote these inferable triples $I_{k'}$). Next, the algorithm applies the closure rules to every triple in $I_{k'}$. The function Backward_Infer(t, K') returns a true value if the triple $t \in C(K')$ using backward inference (lines 13 and 14). We need not use all closure rules in the function Backward_Infer. If the triple has the 'subProperty' property, rule 1 in Definition 2 is applied to only the subProperty table and checks if this triple can be inferred within the subProperty table. Likewise, we can apply each closure rule to its corresponding tables (e.g. rule 2 is applied to both the Triples table and the subProperty table, and rule 3 is applied to both the subClass and the Type tables (rule 3 uses two properties 'type' and 'subClassOf')). Then, if the function Backward_Infer returns the true value for a triple, we remove the triple from the Del table (line 16). This process is repeated until all the triples in $I_{k'}$ have been checked. Finally, the remaining triples in the Del table and the Ins table are the delta (ΔED).

Figure 5 illustrates how Algorithm 1 works for 'subClassOf' closure rule. We first compute differences between the K.Subclass table and the K'.Subclass table using the following queries (step 1 in Figure 5) based on the algorithm.

*INSERT INTO Del ((SELECT * FROM K.SubClass) MINUS (SELECT * FROM K′.SubClass))*

INSERT INTO Ins ((SELECT * FROM K′.SubClass) MINUS (SELECT * FROM K.SubClass))

Then, we prune unnecessary triples that cannot be inferred in the Del table before the inference process begins. For example, consider step 2 of Figure 5, subject *C*, subject D, object A and object B in the Del table appear in the

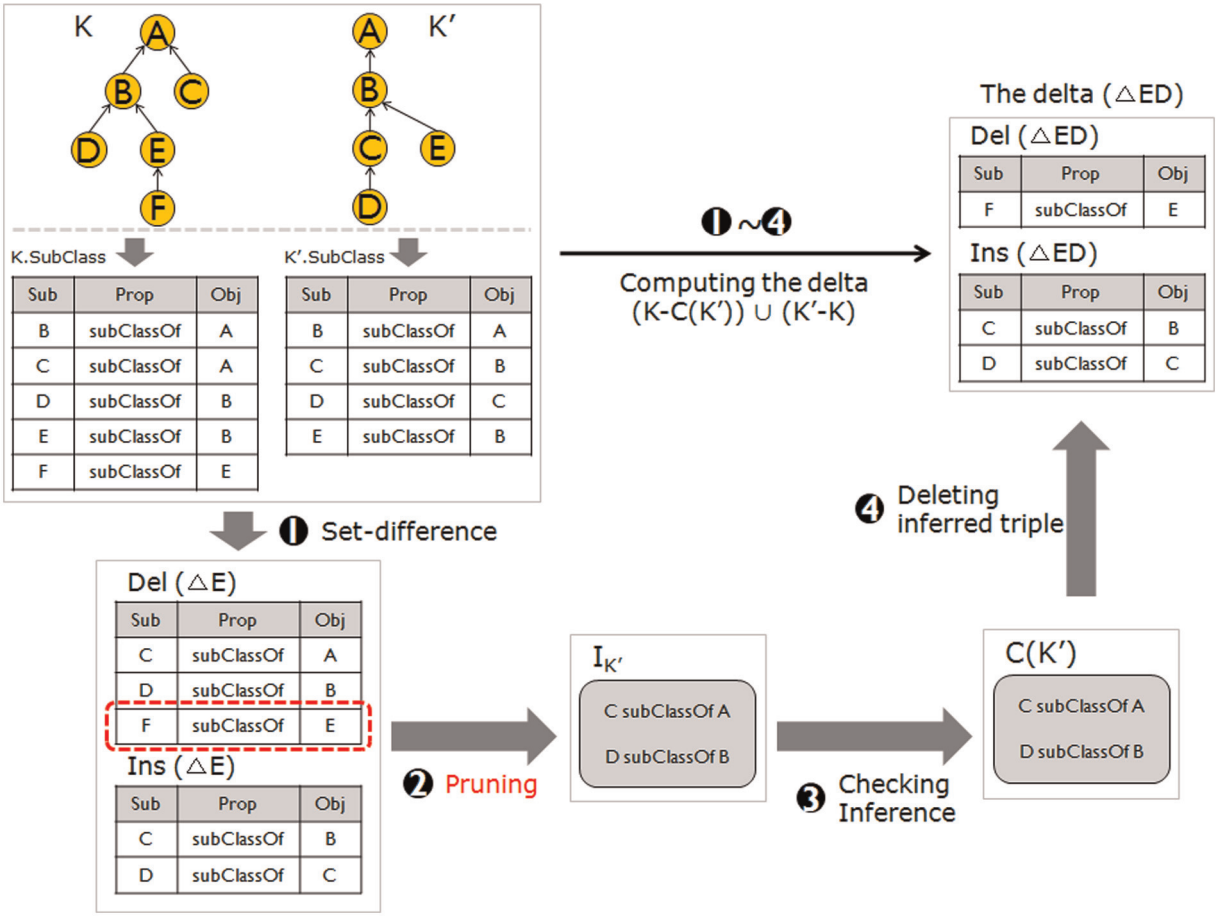

Figure 5. Execution process of change detection.

K'.SubClass table. However, subject F and object E in the Del table do not exist in the K'.SubClass table. Therefore, we prune a triple (F subClassOf E) from Del, because it cannot be inferred in K'.SubClass. In order to remove unnecessary inference from Del, we use the following SQL queries before the inference process begins.

*SELECT * FROM Del* t_1

WHERE t_1.sub IN (SELECT sub FROM K′.SubClass) AND t.obj IN (SELECT obj FROM K′.SubClass)

In step 3, we temporarily store the resultant triples ($I_{k'}$) from step 2 (e.g. ResultSet) and apply the function Backward_Infer to the inferable triples directly. Unlike Sesame, which uses the repetitive self-joins of the triple table for backward inference [4], we use the recursive query feature of the standard SQL language [30, 31]. Although self-join operations would work for backward inference, this is expensive in terms of both program development and query processing, because we have to write a recursive procedure. More importantly, we do not know in advance how many self-join operations would be required for inference processing. Conversely, we can use the recursive WITH clause in the ANSI SQL standard for the recursive query; this allows us to query by specifying what we want, rather than how to perform the recursive procedure. Several commercial DBMSs, including IBM DB2 and Oracle, support recursive query processing (e.g. In case of Oracle RDBMS, its own proprietary syntax, based on START WITH and CONNECT BY clauses, has been supported, but its recent version Oracle 11gR2 has started to support the ANSI Standard Recursive WITH clause). For example, the following SQL computes if a triple (*C* subClassOf *A*) in the Del table can be inferred in the K'SubClass table by backward inference (rule 4 in Definition 2). If it is inferable from the K'.SubClass, the function Backward_Infer returns the true value. In the case of the example in Figure 5, since there exist (*C* subClassOf *B*) and (*B* subClassOf *A*) in the K'.SubClass table, the inference result of a triple (*C* subClassOf *A*) is true.

```
WITH temp (sub, obj) AS
(SELECT sub, obj FROM K′.SubClass WHERE sub='C' UNION ALL
SELECT t1.sub, t1.obj FROM K′SubClass t1, temp t2 WHERE t1.sub=t2.obj)
SELECT * FROM temp WHERE obj='A'
```

This SQL statement constructs an initial result set with the first query (e.g. select the triples whose subject are '*C*' as in above example), and then adds rows to the previous result set with the second query (e.g. select the triples where subjects and objects are connected (subject–object join)). Next, we know that the triple (*C* subClassOf *A*) is inferable by selecting the triple satisfying the specific conditions (e.g. the triple whose object is '*A*') among the result set obtained.

Finally, using the inference result, we remove the corresponding triples from the Del table (step 4 in Figure 5). For example, since a triple (*C* subClassOf *A*) in Figure 5 is included in the inference result, it is deleted from the Del table. Similarly, a triple (*D* subClassOf *B*) is removed from the Del table, because it is inferable from the *K′*.Subclass table. Thus, we obtain the smaller RDF deltas from the Del table. In addition, although only the transitive closure of the subclass relationships in the class hierarchy (rule 4 in Definition 2) is exemplified in Figure 5, all the other types of rules can be handled using the SQL language. For example, suppose that *e* is an instance of class *E* (i.e. *e* type *E*) in triple set *K* and this triple is stored in the *K*.Type table, the result inference of a triple (*e* type *A*) can be computed using the following SQL statement.

```
WITH temp (sub, obj) AS
(SELECT sub, obj FROM K′.SubClass WHERE sub=(SELECT obj FROM K.TYPE WHERE SUBJ='e') UNION ALL
SELECT t1.sub, t1.obj FROM K′SubClass t1, temp t2 WHERE t1.sub=t2.obj)
SELECT * FROM temp WHERE obj='A'
```

Complexity analysis. Let N_1 and N_2 be the numbers of triples in K and K'. For the brevity of discussion, we assume that all K and K' are already stored in a repository and there are no labels for the transitive RDFS relations. Since the forward inference takes $O(n^2)$ time, the forward-inference-based change detection runs in time $O({N_2}^2)$. Conversely, let $O(\mu)$ be the complexity of the backward inference (generally, μ can be much less than linear in model size). The total run time in backward-inference-based change detection will be $O(N_2\mu)$. Finally, let p be the size of pruned triples in each data and $(N_2 - p)$ be M. The complexity of pruning-based change detection will be $O(M\mu)$. Obviously, the amount of time saved depends on the number of pruned triples. The worst case for pruning is when all subjects and objects in each triple set exist in another triple set. In this case, the time requirement equals to the backward-inference-based change detection. Conversely, the best case for the pruning method is when every triple in $(K - K')$ is pruned. In that case, there is no need to use backward inference.

5. Performance analysis

In this section, we compare our change detection technique to other change detection methods in terms of delta size and performance. First, for the comparison of the delta size, Explicit (ΔE), Closure (ΔC) and Explicit&Dense (ΔED) are used. Next, for the comparison of execution time, three methods for the minimum size delta are evaluated: forward-chaining used in SemVersion [9] (denoted as FC), backward-chaining (denoted as BC) and our pruning-then-backward-chaining (denoted as PBC). Note that we do not compare our approach against Delta Function [15, 16], since it is highly beneficial only when the labels are already constructed.

5.1. Experimental setting and test data set

All experiments were performed on a Pentium 4–3.2GHz PC with 1 GB memory. We implemented change detection using Java with the RDF parser Rio,[1] and used Oracle 11gR2 as the relational database to store each version and deltas. The buffer cache of the Oracle database was set to 512M and the database block size was set to 8K, the default size in Oracle. Since there is no benchmark for RDF change detection, we used one synthetic RDF data set generated using our RDF generator and a few real RDF sets from the bioinformatics domain, including Gene Ontology termDB RDF[2] (GO) and Uniprot Taxonomy RDF[3] (Uniprot). Tables 1 and 2 show the characteristics of our real data sets, each of which consists of nine versions (GO monthly version) and six versions (Uniprot versions from 14.8 to 15.3). We do not consider other properties, such as 'type' and 'subPropertyOf', since these real data contain only the 'subClassOf' relationship. Conversely, the synthetic data were generated by Jena [25] and the number of triples ranges up to 1,000,000 (note that,

Table 1. Gene Ontology TermDB RDF

	G1	G2	G3	G4	G5	G6	G7	G8	G9
Number of triples	419,995	426,235	431,540	432,705	456,793	459,225	462,589	490,488	467,654
Number of subClassOf	44,301	38,024	39,339	39,575	39,739	39,943	40,360	40,703	41,309
Size (MB)	33	33	35	35	35	35	35	35	36

Table 2. Uniprot taxonomy RDF

	U1	U2	U3	U4	U5	U6
Number of triples	3,163,163	3,198,383	3,230,870	3,256,174	3,282,876	3,327,128
Number of subClassOf	481,736	484,656	488,465	492,376	496,597	501,745
Size (MB)	220	222	224	226	228	231

```
…
<rdfs:Class rdf:about="http://example.org/Class99">
   <rdfs:subClassOf>
     <rdfs:Class rdf:about="http://example.org /Class34"/>
   </rdfs:subClassOf>
</rdfs:Class>
<rdfs:Class rdf:about="http://example.org /Class10">
…
<Class34 rdf:about="http://example.org /Instance34"/>
<Class35 rdf:about="http://example.org /Instance27"/>
…
```

Figure 6. Sample RDF in synthetic set.

Table 3. Quality of change detection in GO (number of subClassOf relationships)

Pairs of RDF								
	(G1, G2)	(G2, G3)	(G3, G4)	(G4, G5)	(G5, G6)	(G6, G7)	(G7, G8)	(G8, G9)
ΔE	873	1955	588	254	436	609	533	887
ΔC	9,038	17,002	3,750	2,617	4,170	5,029	3,401	5,568
ΔED	794	1716	517	226	399	578	479	731

if the data sets contain the inferred triples, the number of triples is about 9 million). Figure 6 shows an example of synthetic RDF data. Unlike real data, it contains all kinds of properties (e.g. subClassOf, type and subProperty). Changes between the two triple sets being compared were randomly generated by varying the change ratio. Since only a small fraction of the entire data changes and we focus on the practical support for the change detection, two specific change ratios (i.e. 5 and 10%) are used in the experimental setting.

5.2. Performance analysis of various change detection methods on real data

Tables 3 and 4 show the delta sizes of three deltas for GO and Uniprot, respectively. The delta sizes are defined as the sum of deleted triples and inserted triples. As expected, ΔED achieves the better performance (with fewest triples). The three change detection techniques for ΔED (FC, BC and PBC) always generate the same result.

Figure 7 compares the performance of the three techniques for ΔED using real data sets, GO and Uniprot. First, for the three alternative techniques for ΔED, Figure 7(a and b) shows the elapsed times to compute the changes between

Table 4. Quality of change detection in Uniprot (number of subClassOf relationships)

Pairs of RDF	(U1, U2)	(U2, U3)	(U3, U4)	(U4, U5)	(U5, U6)
ΔE	6,538	4,335	4,635	5,080	6,503
ΔC	73,015	46,313	49,383	62,524	67,952
ΔED	6,516	4,303	4,566	4,873	6,279

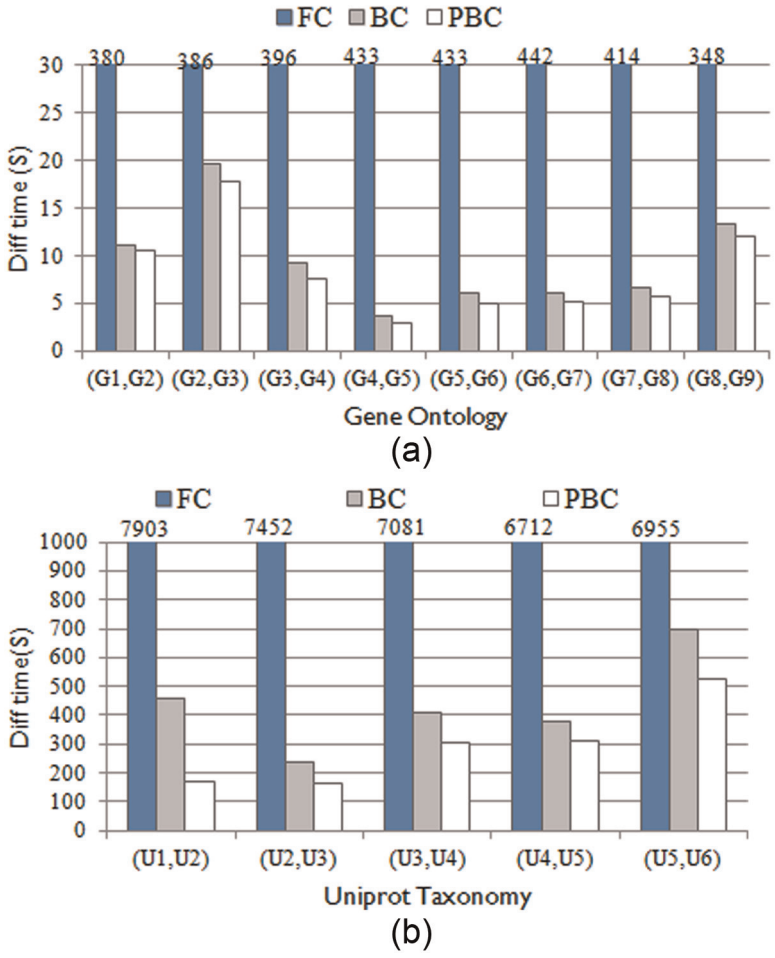

Figure 7. Elapsed time on real data sets. (a) Elapsed time on GO; (b) elapsed time on Uniprot.

two consecutive versions in each real data set. From Figure 7(a and b), we can observe that PBC significantly outperforms both FC and BC. PBC is about 10–80 times faster than FC and about 1.5–4 times faster than BC. We separately measured the execution time of each core operations in change detection, set-difference and inference to understand why PBC can outperform both FC and BC significantly. Table 5 shows the elapsed times of set-difference and inference operations when each change detection technique computes ΔED for (G6, G7) from GO and (U4, U5) from Uniprot. The results in Table 5 show that the inference operation takes most of the time in change detection. Consequently, FC performs worst, as it must compute the closure of all triples in each version in advance, while both BC and PBC calculate the closure of triples that are already computed by a set-difference between the two RDF triple sets. However, although both BC and PBC use backward-chaining inference for change detection, PBC outperforms BC considerably in all cases, because PBC prunes a number of triples for backward inference.

Table 5. Elapsed time analysis: (G6, G7) and (U4, U5) (s)

	(G6, G7)		(U4, U5)	
	Inference time	Set-operation time	Inference time	Set-operation time
FC	280 (63%)	162 (37%)	6585 (98%)	127 (2%)
BC	3 (50%)	3 (50%)	377 (96%)	15 (4%)
PBC	2 (40%)	3 (60%)	309 (95%)	15 (5%)

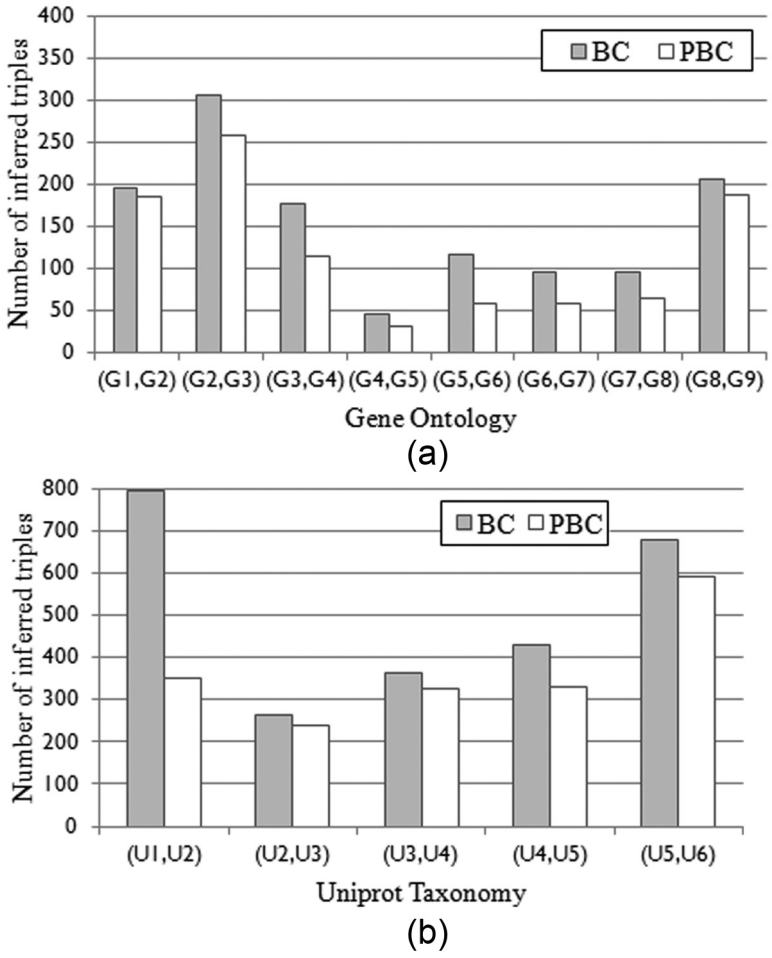

Figure 8. Pruning effect on real data sets. (a) Pruning effect on GO; (b) pruning effect on Uniprot.

We counted the number of triples that participated in the backward-chaining process in BC and PBC to investigate the effect of the pruning scheme in PBC. Figure 8(a and b) shows the results for GO and Uniprot data sets. As is illustrated in Figure 8(a and b), PBC can prune 10–60% of triples as early as possible prior to backward inference. We also observe that the elapsed times in Figure 7(a and b) are approximately in proportion to the number of triples that participate in the backward inference in Figure 8(a and b). In addition, we know from Figure 8(a and b) that the pruning effect of PBC is greater in Uniprot than in GO. This is due to the characteristics of each data set. In the Uniprot data set, numerous existing concepts are removed and many new concepts are introduced in the subclass hierarchy between two consecutive versions. This type of change matches our pruning conditions (recall the subject–object condition in Section 3.2). As shown in Figure 8(b), most triples, particularly for (U1, U2), are pruned by PBC. Meanwhile, numerous concepts in GO data set are, rather than being deleted, moved from one position to another position in the hierarchy between two consecutive

Table 6. Delta size of change detection in synthetic data (number of all relationships)

	Change ratio: 5%				Change ratio: 10%			
	1,000	10,000	100,000	1,000,000	1,000	10,000	100,000	1,000,000
ΔE	44	455	4,576	45,720	90	911	9,109	91,261
ΔC	170	2,365	20,255	347,099	218	3,354	42,533	508,774
ΔED	36	373	3,746	37,443	74	747	7,471	74,797

Table 7. Elapsed time on synthetic data sets (s)

	Change ratio: 5%				Change ratio: 10%			
	1,000	10,000	100,000	1,000,000	1,000	10,000	100,000	1,000,000
FC	7.8	64	978	13,102	6.5	53.1	1072	15,087
BC	0.1	2.2	22.1	6,426	0.6	4.4	63.7	13,099
PBC	0.1	1.4	17.7	4,007	0.5	3.5	33.5	8,108

Table 8. Pruning effect on synthetic data sets (number of triples)

	Change ratio: 5%				Change ratio: 10%			
	1,000	10,000	100,000	1,000,000	1,000	10,000	100,000	1,000,000
BC	6	62	625	6,208	12	125	1,229	12,342
PBC	3	48	482	4,681	7	81	947	9,437

versions, and thus they are not pruned. As shown in Figure 8(a), the ratio of pruned triples in GO by PBC is much less than in Uniprot.

5.3. Scalability on synthetic data

In this section, in order to show that our change detection scheme is scalable, we present the delta size and the elapsed time of each of three change detection schemes by varying the number of triples in the synthetic data set from 1000 to 1,000,000. Table 6 plots the delta size of the deltas generated by the three schemes with the ratio of changed triples as 5 and 10%, respectively. Unlike the real data sets, the result shows the sum of deleted triples and inserted triples that contain not only 'subClassOf' relationships but also all the properties, such as 'type' and 'subProperty'. We know from the table that ΔC generates much larger deltas and ΔED yields the fewest triples. In addition, we notice that all of FC, BC and PBC produce the same ΔED.

Table 7 plots the elapsed time of three change detection techniques in calculating ΔED. As expected, FC performs worst and PBC performs better than BC. However, the pruning effect of PBC is, unlike in real data sets, not so large, because each change operation in the synthetic data, such as triple deletion, triple insertion and triple deletion-then-insertion (i.e. moving from one position to another position in the hierarchy) was generated uniformly. In addition, we can observe that the percentage of changes have some influence on the performance trends of FC, BC and PBC. PBC performs better than BC when we set the ratio of changed triples to 10%. In particular, it should be noted that the performance gap between BC and PBC is widening as the number of triples exceeds 100,000. This is because, as the number of triples is increased, the pruning effect of PBC allows us to skip the inference step for the pruned triples while BC should carry out the backward inference against more triples. In addition, we also counted the number of triples that participated in the backward-chaining process in BC and PBC. Table 8 shows the results for two data sets. As illustrated in Table 8, we can also observe that most triples are pruned in PBC. Consequently, we confirm that the proposed change detection is more suitable for large RDF data sets.

6. Conclusion

Previous work on the RDF change detection technique mainly focused on minimizing the delta size using the semantics of RDF. However, no existing approaches are scalable. We need to develop scalable schemes to compute the closure of the triples to detect changes for the large RDF data. In this paper, we propose an efficient change detection technique for RDF. It generates the smaller delta using backward-chaining inference and a pruning method. Our proposed scheme first stores two RDF models in a relational database. It then computes the difference using the set-difference operation. Next, it prunes many triples that are not necessary to apply backward inference. Finally, we apply the closure rules only to the relevant triples for the deltas. Our experiments prove that our method generates the same size of RDF deltas that previous work produced, but performs better.

We have two areas for future work. First, we need to improve the performance of our change detection schemes. For this, we will investigate alternative storage schemes and will develop new query processing for RDF triples for change detection. Next, we will extend our scheme to be able to handle the OWL model that has more powerful but complicated inference rules.

Notes

1. Available at http://openrdf.org
2. Available at http://geneontology.org
3. Available at http://dev.isb-sib.ch/projects/uniprot-rdf/

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MEST) (No. 20120005695) and also supported by the MKE, Korea under ITRC NIPA-2012-(H0301-12-3001).

References

[1] Klyne G, Carroll JJ and McBride B. Resource description framwork (RDF): Concepts and abstract syntax. *W3C Recommendation*, 2004.
[2] Flouris G, Manakanatas D, Kondylakis H, Plexousakis D and Antoniou G. Ontology change: classification and survey. *The Knowledge Engineering Review* 2008; 23(2): 117–152.
[3] Noy NF and Klein M. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems* 2004; 6(4): 428–440.
[4] Plessers P, Troyer O and Castelyan S. Understanding ontology evolution: A change detection approach. *Journal of Web Semantics* 2007; 5(1): 39–49.
[5] Tummarello G, Morbidoni C, Bachmann-Gmur R and Erling O. RDFSync: Efficient remote synchronization of RDF models. In: *Proceedings of international semantic web conference*, 2007, pp. 537–551.
[6] Im D, Lee S and Kim H. A version management framework for RDF triple stores. *International Journal of Software Engineering and Knowledge Engineering* 2012; 22(1): 85–106.
[7] Klein M, Kiryakov A, Ognyanov D and Fensel D. Ontology versioning and change detection on the Web. In: *Proceedings of international conference on knowledge engineering and knowledge management*, 2002, pp. 197–212.
[8] Noy NF and Musen MA. Ontology versioning in an ontology management framework. *IEEE Intelligent Systems* 2004; 19(4): 6–13.
[9] Volkel M and Groza T. SemVersion: An RDF-based ontology versioning system. In: *Proceedings of IADIS international conference on WWW/internet*, 2006, pp. 195–202.
[10] Leonardi E, Hoai TT, Bhowmick SS and Madria S. DTD-DIFF: A change detection algorithm for DTDs. *Data & Knowledge Engineering* 2006; 61(2): 384–402.
[11] Wang Y, DeWitt DJ and Cai J. X-Diff: An effective change detection algorithm for XML documents. In: *Proceedings of international conference on data engineering*, 2003, pp. 519–530.
[12] Berners-Lee T and Connolly D. Delta: An ontology for the distribution of differences between RDF graphs, http://www.w3.org/DesignIssues/Diff (2004).
[13] Papavassiliou V, Flouris G, Fundulaki I, Kotzinos D and Christophides V. On detecting high-level changes in RDF/S KBs. In: *Proceedings of international semantic Web conference*, 2009, pp. 473–488.
[14] Tzitzikas Y, Theoharis Y and Andreou D. On storage policies for semantic Web repositories that support versioning. In: *Proceedings of European Semantic Web conference*, 2008, pp. 705–719.
[15] Zeginis D, Tzitzikas Y and Christophides V. On the foundations of computing deltas between RDF models. In: *Proceedings of international Semantic Web conference*, 2007, pp. 637–651.

[16] Zeginis D, Tzitzikas Y and Christophides V. On computing deltas of RDF/S knowledge bases. *ACM Transactions on the Web* 2011; 5(3): 14.
[17] Hayes P and McBride B. RDF semantics. *Technical Report, W3C Recommendation*, 2004.
[18] Christophides V, Plexousakis D, Scholl M and Tourtounis S. On labeling schemes for the Semantic Web. In: *Proceedings of international World Wide Web conference*, 2003, pp. 544–555.
[19] Broekstra J and Kampman A. Inferencing and Truth maintenance in RDF Schema. In: *Proceedings of workshop on practical and scalable semantic system*, 2003.
[20] Shen W and Qu Y. An RDF storage and query framework with flexible inference strategy. In: *Proceedings of Asia–Pacific Web conference*, 2006, pp. 166–175.
[21] Stuckenschmidt H and Broekstra J. Time–space trade-offs in scaling up RDF schema reasoning. . In: *Proceedings of workshop on scalable Semantic Web knowledge base system*, 2005, pp. 172–181.
[22] Broekstra J, Kampman A and Harmelen FV. Sesame: A generic architecture for storing and querying RDF and RDF schema. In: *Proceedings of international Semantic Web conference*, 2002, pp. 54–68.
[23] Ognyanov D and Kiryakov A. Tracking changes in RDF(S) repositories. In: *Proceedings of international conference on knowledge engineering and knowledge management*, 2002, pp. 373–378.
[24] Carroll JJ. Signing RDF graphs. In: *Proceedings of international Semantic Web conference*, 2003, pp. 369–384.
[25] Carroll JJ, Dickinson I, Dollin C, Reynolds D, Seaborne A and Wilkinson K. Jena: Implementing the Semantic Web recommendation. In: *Proceedings of international World Wide Web conference*, 2004, pp. 74–83.
[26] Eder J and Wiggisser K. Change detection in ontologies using DAG comparison. In: *Proceedings of international conference on advanced information systems engineering*, 2007, pp. 21–35.
[27] Leonardi E, Bhowmick SS and Madria S. XANDY: Detecting changes on large unordered XML documents using relational databases. In: *Proceedings of international conference on database systems for advanced applications*, 2005, pp. 711–723.
[28] Abadi DJ, Marcus A, Madden SR and Hollenbach K. Scalable semantic web data management using vertical partitioning. In: *Proceedings of international conference on very large data bases*, 2007, pp. 411–422.
[29] Wilkinson K, Sayers C, Kuno H and Reynolds D. Efficient RDF storage and retrieval in Jena2. In: *Proceedings of workshop on Semantic Web and databases*, 2003, pp. 131–150.
[30] Chong EI, Das S, Eadon G and Srinivasan J. An efficient SQL-based RDF querying scheme. In: *Proceedings of international conference on very large data bases*, 2005, pp. 1216–1227.
[31] Eisenberg A and Melton J. SQL: 1999, formerly known as SQL 3. *sigmod record* 1999; 28(1): 131–138.