(SEMANTIC WEB) EVOLUTION THROUGH CHANGE LOGS: PROBLEMS AND SOLUTIONS

Yannis Tzitzikas and Dimitris Kotzinos
Computer Science Department, University of Crete, GREECE, and
Institute of Computer Science - FORTH, GREECE
email: (tzitzik | kotzino)@ics.forth.gr

ABSTRACT
Knowledge evolution is currently a hot research topic within the Semantic Web community. This paper investigates a *change*-based Semantic Web, according to which any modification applied to an ontology should be logged. The merits of this approach for supporting the process of evolution are discussed. Subsequently, a number of basic problems concerning the management of such change logs are introduced, discussed and analyzed. Finally specific techniques and efficient algorithms for managing change logs are provided.

1 Introduction

The statement of Heraclitus ”Everything flows, nothing stands still” [1], is true also in the SWeb (Semantic Web) as everything changes and evolves over time: resources themselves, ontologies, resource annotations and application programs. It is not hard to realize that even if we had a wide arsenal of ontologies for every application domain and machine readable semantics according to this set, in a short time a major part of the whole information infrastructure could become obsolete. Recall that according to recent surveys the world produces between 1 and 2 exabytes of unique information per year, 90% of which is digital and with a 50% annual growth rate[1]. It is therefore evident that supporting evolution is very critical for the realization of the SWeb (and for digital information preservation) and this justifies the recent interest around this topic.

This paper introduces a ”delta” view of the SWeb and discusses its merits for supporting evolution. Specifically it proposes logging every change operation and devising methods and tools for managing such log files. Specifically, the contributions of this paper are: (a) it motivates the need for logging change operations, (b) it formulates the requirements for managing change logs, and (c) it gives a number of techniques for managing change logs, including algorithms for deciding equivalence and computing reductions, as well as data structures for efficiently constructing ontology versions from change logs. Although several works have identified the need for logging, this is (to the best of our knowledge) the first paper that tries to identify and formulate explicitly the problems related to change log management, and to provide specific techniques and algorithms. In particular, this paper is organized as follows: Section 2 discusses in brief some aspects of knowledge evolution that are closely related to our purposes. Subsequently, Section 3 introduces the term ”delta” Semantic Web and motivates this line of research. Subsequently Section 4 introduces some fundamental definitions and problems regarding the management of change operations, and Section 5 provides techniques for managing change logs. Finally, Section 6 summarizes and concludes the paper.

[1] http://www2.sims.berkeley.edu/research/projects/how-much-info-2003

2 Background

Knowledge evolution has several aspects. Assuming a knowledge management system for the SWeb, supporting knowledge evolution may amongst others include evolution strategies [27], declarative languages for bulk updates (like RUL [20]), ontology changes at run time, graphical ontology editors and powerful versioning services (for more see [15, 22]). Versioning is a task closely related to the theme of this paper. Versioning (and deltas) are traditionally studied in the context of software engineering, and text file management (e.g. the classical CVS system). Versioning systems commonly employ deltas because it is more space efficient to keep deltas than keeping all states of an artifact. The need for ontology versioning and for computing and exchanging deltas of RDF [21] graphs has been identified and motivated multiple times the recent years, e.g. in [16], and firstly at [13]. For instance, deltas can be exploited for reducing the data that have to be transmitted over the network in order to update and synchronize Semantic Web data [3, 6]. In general, and as stated in [22], ontology-versioning environments should allow users to: (1) examine the changes between versions visually, (2) understand the potential effects of changes on applications, and (3) accept or reject changes (when an ontology is being developed in a collaborative setting).

Concerning ontology comparison, RDF graphs can be serialized and used with traditional line-oriented tools. If graphs are not serialized in a standard method, then a line-oriented delta can be as large as the data itself even between files representing the same graph. So such methods are serialization dependent and this is true also for pretty-printed RDF (for more see [3]). Non text-based approaches for comparing RDF graphs are described in SemVersion [28], PromptDiff [22] and [3]. Specifically, [28] proposes two

Diff operations: (a) one returning a triple-set-based difference, and (b) one semantic-aware which takes into account also the semantically inferred triples. PromptDiff [23, 24, 22] is another ontology-versioning environment, that includes a version-comparison algorithm (based on heuristic matchers [23, 24]), while the visualization of the computed difference between two ontologies is discussed in [22]. There are also quite a lot of works that deal with the problem of comparing XML documents (e.g. see [30, 5, 7, 17]).

Concerning the management of change logs, there are only very few works which only touch the problem and do not provide any specific technique or insight. For instance, [26] specifies a set of change operations for controlled vocabularies and describes mechanisms for synchronizing different versions based on change logs and user input, while [25] suggests that log files should keep only additions and deletions of triples (not "modify" statements).

3 Towards a "Delta" Semantic Web

Here we discuss a view according to which the central point of the SWeb is not the notion of ontology, nor the notion of triple, but the notion of *change operation*, specifically the notion of *sequence of change operations*. At first note that in every case, an ontology (or knowledge base in general) is the result of applying a sequence of change operations. It follows that if we have stored this sequence then we can reconstruct the ontology at any point, so there is no need to have stored the ontology, apart from reasons of efficiency (i.e. for avoiding the cost of reconstructing an ontology multiple times). Such an approach, which we could call "delta" approach, has two main advantages:

Reconstructiveness

At any point in time (now or in the future) we could execute the sequence of change operations that led to the current state of the ontology by adopting different assumptions. For instance, different belief revision approaches (e.g. [8, 9, 29, 11]) lead to different results. The application of such rules in ontology evolution is elaborated in [10, 18]. One can easily see that if the sequence of change operations is available, then we can "reexecute" it according to various different assumptions and rules. For example consider the case where a user (or agent) A submits the triple (myCar, color, Green) and a user B who at later time submits the triple (myCar, color, Blue). The resulting information base could keep only the latest information, i.e. (myCar, color, Blue), or the triple (myCar, color, Green $\vee$ Blue), or the triple (myCar, color, Green $\wedge$ Blue), e.g. for the case that the car is actually turquoise-colored. A modal logic-based repository could store also the provenance of the above statements (i.e. $\mathbf{K}$(A, (myCar, color, Green)), and $\mathbf{K}$(B,(myCar, color, Blue))). For instance, RDF reification could be used or the more recently emerged named graphs [4]. Furthermore, recall the difference between *update* and *revision* as stated in [14], i.e. the difference between assuming a dynamic world and a static world, which is also discussed in [12] for the case of RDF. The availability of the change log would allow constructing a different knowledge base for each one of the above assumptions. For instance, suppose an ontology O derived by the application of a sequence of change operations $\langle u_1, \ldots, u_n \rangle$ assuming a static world (this means that the change operations express changes of our knowledge about the world, not of the world itself). If in the future we realize that at time i the world started changing, then we could reexecute this sequence by considering a static world for those change operations issued before time i, and a dynamic world for the rest. This functionality is not possible according to the current, ontology-based, semantic web as the notion of time and sequence is lost and cannot by recovered.

Efficient Version Management

If change operations are logged, then the deltas between two different versions of an ontology are almost ready-made: each position in the sequence of change operations of the log file actually specifies an ontology version, so the delta between two versions is simply the sequence of change operations between the corresponding positions. According to this approach, we do not have to employ any structural Diff algorithm for computing the delta between two versions. Recall that the general problem of computing the differences between two graphs becomes a special case of the graph isomorphism problem which cannot be solved in polynomial time (but has not been shown to be NP-complete either).

Apart from the above, logging changes is useful for organizational reasons, e.g. for enabling other developers and tools to process and understand the evolution history of an ontology [19]. For all these reasons it is worth investigating this "delta semantic web" approach.

4 Change Logs Management

Here we introduce some basic definitions and then we formulate a number of fundamental problems for supporting evolution effectively and efficiently.

We shall use B to denote an *information base* and $\mathcal{B}$ to denote the set of all possible information bases. We use the term information base to capture both ontologies and ontology-based repositories. An RDF-based information base B is actually a graph, but it is also equivalent to consider it as a set of RDF triples.

Let UT be the set of primitive *change operation types*, and U the (infinite) set of all possible *change operations* with type in UT. Now let US be the set of all possible *sequences* of elements in U.

If B is an information base and $u \in U$, then we will denote by $u(B)$ the information base obtained by applying u on B. If us is an element of US, then we will denote by $us(B)$ the information base obtained by applying each u of us on B in the specified order, e.g. if $us = \langle u_1, u_2, u_3 \rangle$ then $us(B) = u_3(u_2(u_1(B)))$.

Now we will introduce a form of constraints that we call *conditions*. A condition C may be True (hold) or

False (not hold) on an information base. Given two conditions $C1$ and $C2$, we will write $C1 \leq C2$ if the satisfaction of $C1$ implies the satisfaction of $C2$, in other words, if $C1$ is stronger (stricter) than $C2$. Now let $\mathcal{B}^C$ denote the elements of $\mathcal{B}$ that satisfy condition C, i.e. $\mathcal{B}^C = \{ B \in \mathcal{B} \mid B \text{ satisfies } C\}$. Now we will introduce a simple method to formalize the notion of conditions. Let Ω denote the set of all possible triples. We can consider a condition C as a pair of disjoint subsets of Ω, i.e. $C = (C^+, C^-)$ where $C^+ \cap C^- = \emptyset$. An information base B satisfies a condition $C = (C^+, C^-)$, if $C^+ \subseteq B$ and $C^- \cap B = \emptyset$, i.e. if it contains every element of C^+ and does not contain any element of C^-. According to this view, $(C_1^+, C_1^-) \leq (C_2^+, C_2^-)$ iff $C_1^+ \supseteq C_2^+$ and $C_1^- \supseteq C_2^-$. So one advantage of expressing conditions in this form is that we can decide condition containment efficiently (with two set containment operations).

4.1 Equivalence and Reduction

Let B denote an information base, i.e. a set of triples. Suppose that UT comprises only two types of change operations; one for adding and one for deleting triples, i.e. $UT = \{add(< triple >), del(< triple >)\}$. Now U is the set of all possible operations that add or delete triples, plus the *null operation* which is denoted by ϵ. Three relations of equivalence over US are defined next.

Def 4.1 Two sequences us and us' are:
(a) *universally equivalent*, denoted by $us \equiv us'$, if $us(B) = us'(B)$, for every $B \in \mathcal{B}$,
(b) *conditionally equivalent (assuming C)*, denoted by $us \equiv^C us'$, if $us(B) = us'(B)$ for every $B \in \mathcal{B}$ that satisfies the condition C, and
(c) *equivalent over an information base B*, denoted by $us \equiv_B us'$, if $us(B) = us'(B)$.

Clearly, $us \equiv us'$ implies $us \equiv_B us'$ for every $B \in \mathcal{B}$, and $us \equiv us'$ implies $us \equiv^C us'$ for every possible condition C. For example, let t_1 and t_2 be two RDF triples, and let B be a set of triples. One can easily see that $\langle add(t_1), add(t_1)\rangle \equiv add(t_1)$ and that $\langle add(t_1), add(t_2)\rangle \equiv \langle add(t_2), add(t_1)\rangle$. These were two examples of universal equivalence relationships. Note that $\langle add(t_1), del(t_1)\rangle$ is not universally equivalent to the null operation ϵ, i.e. $\langle add(t_1), del(t_1)\rangle \not\equiv \epsilon$. For example, if $B = \{t_2\}$ then $\langle add(t_1), del(t_1)\rangle(B) = B$. However, if $B = \{t_1\}$ then $\langle add(t_1), del(t_1)\rangle(B) = \emptyset$. This is why $\langle add(t_1), del(t_1)\rangle \not\equiv \epsilon$. It is not hard to see that if an information base B does not contain the triple t_1, then it holds $\langle add(t_1), del(t_1)\rangle \equiv_B \epsilon$. This can be expressed using conditional equivalence, i.e. with Def. 4.1(b). Specifically, we can write $\langle add(t_1), del(t_1)\rangle \equiv^C \epsilon$, where in our case the condition C could be $C = \not\exists t_1$, i.e. $C^- = \{t_1\}$.

For each equivalence relation we can define the corresponding decision problem, i.e. $us \stackrel{?}{\equiv} us'$, $us \stackrel{?}{\equiv}_B us'$ and $us \stackrel{?}{\equiv}^C us'$, which are quite important as it will be made clear later on. Concerning conditional equivalence, note that since $C1 \leq C2 \Leftrightarrow \mathcal{B}^{C1} \subseteq \mathcal{B}^{C2}$, it follows that if $C1 \leq C2$ and $us \equiv^{C2} us'$, then $us \equiv^{C1} us'$. So if we want to prove equivalence given a condition C, we can try proving it using a condition less strict than C. It is not hard to see that equivalence assuming C, i.e. $\equiv^{(C^+, C^-)}$, can capture equivalence over information base, i.e. $\equiv_B$. Specifically, $\equiv_B$ can be written as $\equiv^{(B,\emptyset)}$. It follows that Def. 4.1.(b) is more general than Def. 4.1.(c). So we can hereafter consider two kinds of equivalence: universal and conditional.

We will now define the notion of *reduction* which is important for optimization. Let $size(us)$ denote the number of change operations in a sequence us.

Def 4.2 Given a $us \in US$ we define as *universal reduction* of us, denoted by $r(us)$, the smallest in size element (or elements) of US that is universally equivalent to us.

This means that for all $B \in \mathcal{B}$ it holds $r(us)(B) = us(B)$, and there is no $us' \in US - \{r(us)\}$, such that $us'(B) = us(B)$ and $size(us') < size(r(us))$. (In this paper we use "$-$" to denote set difference.) Of course, there may be more than one equivalent sequences all having the same minimum size. So the reduction is not necessarily unique, however its size is unique. Analogously, we can define the reduction of a sequence us "over B" or "given C" (i.e. with respect to $\equiv_B$ or $\equiv^C$).

5 Techniques for Change Logs Management

5.1 Deciding Equivalence

Below we present an algorithm that takes as input a sequence of change operations us and returns two sets denoted by A and D. The first contains the triples that certainly belong to the outcome of the execution of us, while the second contains the triples that certainly do not belong to the outcome of the execution of us. Clearly, $A \cap D = \emptyset$. The algorithm AD computes these sets by scanning us once. For each triple that appears in us (either in an Add or in a Del operation) the algorithm keeps a variable X that ranges $\{-1, 1\}$.

Alg. AD
Input: a sequence of change operations $us = \langle u_1, \ldots, u_n\rangle$
Output: the sets A and D of the sequence us

(1) $A := D := Visited := \emptyset$
(2) for $i = 1$ to n do
(3) t = operand triple of u_i
(4) if $t \notin Visited$ then
(5) $Visited := Visited \cup \{t\}$;
(6) if $u_i = Add(t)$ then
(7) $t.X$:=1
(8) elseif $u_i = Del(t)$ then
(9) $t.X$:=-1
(10) for each $t \in Visited$ do
(11) if $t.X = 1$ then $A := A \cup \{t\}$
(12) if $t.X = -1$ then $D := D \cup \{t\}$
(13) return (A, D)

Now suppose that we want to decide whether us and us' are universally equivalent. For doing so we can apply AD on each sequence in order to compute the sets $A(us), D(us)$ and $A(us'), D(us')$. The motivation for computing these sets is evident from the following proposition.

Prop. 1 $us \equiv us'$ iff $A(us) = A(us')$ and $D(us) = D(us')$.

Proof: The proof follows easily from set theory.
($\Rightarrow$) We should prove that if $us \equiv us'$, then both equalities hold. Let's suppose that this is not true.

At first suppose that $A(us) \neq A(us')$, specifically suppose that there is a $t \in A(us) - A(us')$. In this case us and us' cannot be equivalent because their application on a B that does not contain t would be different.

Now suppose that $D(us) \neq D(us')$, e.g. suppose that $t \in D(us) - D(us')$. In this case us and us' cannot be equivalent because their application on a B that contains t would be different.
($\Leftarrow$) We should prove that if $A(us) = A(us')$ and $D(us) = D(us')$, then us and us' are universally equivalent. Suppose they are not, i.e. suppose there exist a B such that $us(B) \neq us'(B)$. Specifically suppose that there exist a triple t such that $t \in us(B) - us'(B)$. We can distinguish the following two cases:
(a) $t \in B$. As $t \in B$ and $t \notin us'(B)$, it follows that $t \in D(us')$. This contradicts with our hypotheses that $D(us) = D(us')$ and that $t \in us(B)$.
(b) $t \notin B$. As $t \notin B$ and $t \in us(B)$, it follows that $t \in A(us)$. This contradicts with our hypotheses that $A(us) = A(us')$ and that $t \notin us'(B)$. $\diamond$

Recall that the time complexity of algorithm AD is $\mathcal{O}(n)$ as it scans us once. It follows that the time complexity for deciding whether $us \equiv us'$ is $\mathcal{O}(N)$ where $N = \max(size(us), size(us'))$. Now assume that we want to decide whether us and us' are *conditionally equivalent*, specifically suppose that we want to decide whether $us \equiv^{(C+,C-)} us'$. Again, we first apply AD on each sequence. Subsequently, we can exploit the following proposition.

Prop. 2 $us \equiv^{(C+,C-)} us'$ iff: (a) $A(us) - C^+ = A(us') - C^+$, and (b) $D(us) - C^- = D(us') - C^-$.

Proof: The proof follows easily from set theory. For reasons of space it is omitted.

For example, the role of equality (a) is evident in the following example: $\langle Add(t_1), Add(t_2)\rangle \equiv^{(t_1,)} \langle Add(t_2)\rangle$. The role of equality (b) is evident in the example: $\langle Del(t_1), Del(t_2)\rangle \equiv^{(,t_1)} \langle Del(t_2)\rangle$. Both (a) and (b) are needed to yield the equivalence: $\langle Del(t_1), Add(t_2), Add(t_3)\rangle \equiv^{(t_2,t_1)} \langle Add(t_3)\rangle$.

This section described an algorithm for deciding universal and conditional equivalence whose complexity is linear with respect to the size of the sequences.

5.2 On Computing Reductions

In this section we give an efficient method that derives the universal reduction of any update sequence. At first we run the algorithm AD(us) for computing the sets A and D. Based on these sets we can produce a sequence of update operations us' such that $A(us') = A$ and $D(us') = D$. Finding such a us' is not so difficult. In fact, for any given pair (A, D) there are several possible ways to derive a sequence us' such that $A(us') = A$ and $D(us') = D$.

Prop. 3 The minimum size of a sequence us' such that $A(us') = A$ and $D(us') = D$ is equal to $|A| + |D|$.

An algorithm that takes as input any sequence of change operations and returns an equivalent sequence with the minimum size (i.e. the size stated at Prop 3) is given next.

Alg. REDUCE
Input: a sequence of change operations us
Output: the reduction of us, i.e. $r(us)$

(1) $(A, D) := AD(us)$
(2) for each $t \in D$ do
(3) out("Del(t)")
(4) for each $t \in A$ do
(5) out("Add(t)")

The algorithm uses a command out for writing the output file. The time complexity of REDUCE is again linear. Let's now discuss *conditional reduction*. Consider a sequence us and suppose that we want to find the shortest in size sequence us', such that $us \equiv^{(C+,C-)} us'$. We can find this reduction by exploiting Prop. 2 and the algorithm REDUCE. Specifically, at first we run the algorithm AD(us) for computing the sets A and D. Then we set $A' = A - C^+$ and $D' = D - C^-$. The final step is to produce a sequence of update operations us' such that $A(us') = A'$ and $D(us') = D'$. This can be done by algorithm REDUCE if we delete line (1) and we use (A', D') instead of (A, D) at the rest of the algorithm. The correctness of this method follows easily from Prop. 2.

5.3 Semantics-aware Change Log Management

So far we have treated triples as independent abstract elements. However, triples are interrelated and they have semantics. For instance, we may have a triple of the form $Sb(B, A)$ meaning that B is a subclass of A, or triples of the form $In(o, C)$ meaning that resource o is an instance of the class C. Below we give two examples of semantically equivalent sequences:

$$\langle Add(Sb(C, B)), Add(Sb(B, A)), Add(Sb(C, A))\rangle \equiv \quad (1)$$
$$\langle Add(Sb(C, B)), Add(Sb(B, A))\rangle \quad (2)$$
$$\langle Add(In(o, C)), Add(Sb(C, D)), Add(In(o, D))\rangle \equiv \quad (3)$$
$$\langle Add(In(o, C)), Add(Sb(C, D))\rangle \quad (4)$$

The techniques we have described so far will fail to identify the above equivalence relationships. Below we discuss three approaches to tackle this problem.
(a) Avoid logging redundant change operations.
One approach is to avoid logging change operations that

do not actually have any effect on the information base. For instance, $Add(Sb(C, A))$ at line (1) is redundant. The same is true for $Add(In(o, D))$ at line (3). If such change operations are not logged, then we can bypass the problem. Although we could have ontology editors that follow this policy, in general it is not easy to enforce this policy.

(b) Complete-Reduce change logs.

Before checking whether two sequences are equivalent we could "complete" each sequence by adding all semantically inferred triples. Alternatively, and preferably, we could "reduce" each sequence by deleting all redundant triples. For instance, this can be achieved by applying an algorithm that computes the reflexive and transitive reduction of the binary relation that is obtained if we take the union of the $subclassOf$ relation and the $instanceOf$ of relation. After this completion or reduction we can check equivalence and compute reductions using the techniques that we provided earlier.

(c) Late Completion/Reduction.

A more realistic and more efficient approach is to exploit Prop. 1 and Prop. 2. Specifically, instead of "completing" (or "reducing") the entire sequences, we could "complete" (or "reduce") only the sets A and D which are used for deciding equivalence. Obviously, this approach is more efficient than approach (b), because in this way we avoid repetitions and compensating operations. An even more efficient approach is sketched below. We first check whether the equalities hold (i.e. whether $A(us) = A(us')$, etc). If both equalities hold then we conclude that the sequences are equivalent and there is no need for anything else. If not then we start checking in more detail each pair of sets that is not equal. For instance, suppose that $A(us) \neq A(us')$. If an element t belongs to $A(us)$ but does not belong to $A(us')$, then we check if it is inferred by us', i.e. if $t \in Cons(A(us'))$ where $Cons$ denotes the consequence operator that completes a set of triples with those that are semantically inferred. If this is not true then we conclude that the sequences are not equivalent, otherwise we proceed analogously for all elements of the symmetric difference of $A(us)$ and $A(us')$ and then for the pair $D(us)$ and $D(us')$. An alternative approach would be to use reduction instead of completion, i.e. instead of checking whether $t \in Cons(A(us'))$, to check whether $t \in Red(A(us))$ where Red is an operator that eliminates the semantically inferred triples.

5.4 Constructing Ontology Versions from Logs

An important question is whether from the log file we can compute the current (or a past) version of the ontology efficiently. The cost of constructing an ontology version O_i, where i corresponds to the time when operation u_i was issued ($1 \leq i \leq n$), is $\mathcal{O}(i)$, so for constructing the latest version of the ontology the cost is $\mathcal{O}(n)$. Clearly, if O_i is already constructed and $j > i$, then the cost for constructing O_j is $\mathcal{O}(j - i)$. It is worth noticing here that the ability to compute reductions could be exploited for reducing the ontology construction costs (as well as the storage space requirements of the change logs), according to application-oriented requirements. For instance, if the desired policy is to keep only one version per week, then instead of keeping stored the entire sequence of change operations for a given week, we can reduce them and keep stored only their reduction.

Now suppose that we want an efficient method to decide whether a particular triple t belongs to version O_i (independently of whether or not reduction has been computed). Below we present a data structure which can be constructed once and then used for answering quickly this kind of questions. Specifically, this data structure keeps for each triple t two lists of identifiers, denoted by $added(t)$ and $deleted(t)$: the first is the set of identifiers of the operations that add t, while the second is the set of identifiers that delete t. The algorithm *MakeVO* takes as input a sequence us and constructs this data structure.

Alg. MakeVO

Input: a sequence of change operations us
Output: a VO data structure

(1) $T := \emptyset$
(2) for $i = 1$ to n do
(3) if $u_i = Add(t)$ then
(4) if $t \notin T$ then
(5) $T := T \cup \{t\}$;
(6) $added(t) := \emptyset$; $deleted(t) := \emptyset$
(7) $added(t) := added(t) \cup \{i\}$
(8) else // this means that $u_i = Del(t)$
(9) $deleted(t) := deleted(t) \cup \{i\}$
(10) return

After the execution of this algorithm T will contain all triples that have been created during the lifetime of the ontology. Notice that for every triple $t \in T$, the sets $added(t)$ and $deleted(t)$ are disjoint. The storage space of this data structure is $\mathcal{O}(n)$ where n the size of us. In practice, it will be less than the space occupied by us, because each triple is stored once. Notice that the same algorithm can be also used for incrementally updating the data structure when new change operations are issued and logged.

Now suppose that we want to decide whether a particular triple t is an element of a version with $id = X$. Let
$a = max\{\ i \in added(t) \mid i \leq X\}$, and
$b = max\{\ i \in deleted(t) \mid i \leq X\}$.
It is evident that t belongs to version X, iff a is defined and it holds $a > b$. For example, suppose that $added(t) = \{3, 9, 18\}$ and $deleted(t) = \{5, 14\}$. For $X = 11$ we get $a = 9$ and $b = 5$, so t belongs to version 11. For $X = 8$ we get $a = 3$ and $b = 5$, so t does not belong to version 8. It follows that the cost to decide whether a triple t belongs to a version is $\mathcal{O}(f)$ where f is the number of times t has been created and deleted. As $added(t)$ and $deleted(t)$ are disjoint, both lists can be stored in one single list. In this way (and with binary search) the cost reduces to $\mathcal{O}(log(f))$. At last note that from this data structure we can reconstruct the original sequence of update operations.

Summarizing, we can say that the proposed approach

is realistic in terms of storage space requirements and computational cost.

6 Concluding Remarks

This paper described a Semantic Web that is based on a *change*-based version model, instead of the current *state*-based version model. According to this view every change operation should be logged as the resulting change logs can be a source of very valuable information that could be exploited in several different ways now and in the future. Realizing this approach requires devising efficient methods and tools for managing such change logs. The latter requires specializing and elaborating on the problems stated and formalized in Section 4, which revolve around the notions of *equivalence* and *reduction*. For these problems we provided efficient algorithms whose time complexity is linear. From the analysis presented in this paper we can draw the conclusion that a *change*-based Semantic Web would not suffer from any performance drawback. There are several issues for further research including algorithms for managing change operations of different granularity, and semantics-based optimizations. Furthermore, as ontologies commonly import and reuse other, remotely stored, ontologies, it is worth devising distributed change log management algorithms (e.g. along the lines of [2]).

Acknowledgement: This work was partially supported by the EU project CASPAR (FP6-2005-IST-033572).

References

[1] Heraclitus (535 - 475 bc).

[2] Mikhail J. Atallah, Michael T. Goodrich, and S. Rao Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *J. ACM*, 41(6):1049–1088, 1994.

[3] Tim Beners-Lee and Dan Connoly. ”Delta: An Ontology for the Distribution of Differences Between RDF Graphs”, 2004. http://www.w3.org/DesignIssues/Diff (version: 2004-05-01).

[4] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. *Proceedings of the 14th international conference on World Wide Web*, pages 613–622, 2005.

[5] S. Chawathe and H. Garcia-Molina. ”Meaningful Change Detection in Structured Data”. In *Procs. of SIGMOD'97*, 1997.

[6] Russell Cloran and Barry Irwin. ”Transmitting RDF graph deltas for a Cheaper Semantic Web”. In *Procs. of SATNAC'2005*, South Africa, September 2005.

[7] G. Cobena, S. Abiteboul, and A. Marian. ”Detecting Changes in XML Documents”. In *Procs. of IEEE Intern. Conf. on Data Engineering, ICDE*, San Joce, CA, 2002.

[8] Mukesh Dalal. “Investigations Into a Theory of Knolwedge Base Revision”. In *Conference on Artificial Intelligence, AAAI-88*, pages 475–479, St. Paul, Minesota, August 1988.

[9] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 261–273, 1992.

[10] Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. “On Applying the AGM Theory to DLs and OWL”. In *International Semantic Web Conference*, pages 216–231, 2005.

[11] P. Gärdenfors. “Belief Revision: An Introduction”. In P. Gärdenfors, editor, *Belief Revision*, pages 1–20. Cambridge University Press, 1992.

[12] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. “The Meaning of Erasing in RDF under the Katsuno-Mendelzon Approach”. In *Proceedings WebDB-2006*, Chicago, Illinois, 2006.

[13] Jeff Heflin, James Hendler, and Sean Luke. “Coping with Changing Ontologies in a Distributed Environment”. In *Proceedings of AAAI-99 Workshop on Ontology Management*, 1999.

[14] H. Katsuno and A. O. Mendelzon. “On the Difference between Updating a Knowledge Base and Revising it”. In *Proceedings KR-91*, pages 380–395, 1991.

[15] M. Klein and D. Fensel. ” Ontology versioning for the Semantic Web”, 2001. International Semantic Web Working Symposium (SWWS), USA.

[16] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. In *Procs of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, pages 197–212. Springer, 2002.

[17] Kyriakos Komvoteas. “XML Diff and Patch Tool”. Master's thesis, Computer Science Department, Heriot-Watt University in Edinburgh, Scotland, 2003.

[18] Pieter De Leenheer. ”Revising and Managing Multiple Ontology Versions in a Possible Worlds Setting”. In *Procs of On The Move to Meaningful Internet Systems Ph.D. Symposium (OTM 2004)*, pages 798–818, Agia Napa, Cyprus, 2004.

[19] Yaozhong Liang, Harith Alani, and Nigel Shadbolt. ”Ontology Change Management in Protege”. In *Procs of the 1st AKT Doctoral Symposium*, Milton Keynes, 2005.

[20] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. ”RUL: A Declarative Update Language for RDF”. In *Procs. 4th Intern. Conf. on the Semantic Web (ISWC-2005)*, Galway, Ireland, November 2005.

[21] E. Miller, R. Swick, and D. Brickley (editors). RDF and RDF Schema, W3C, 2003. http://www.w3.org/RDF.

[22] Michel Klein Natalya F. Noy, Sandhya Kunnatur and Mark A. Musen. ”Tracking Changes During Ontology Evolution”. In *Third International Conference on the Semantic Web (ISWC-2004)*, Hisroshima, Japan, 2004.

[23] N. F. Noy and M. A. Musen. ”PromptDiff: A Fixed-point Algorithm for Comparing Ontology Versions”. In *In 18th Nat Conf on Artificial Intelligence (AAAI-2002)*, pages 744–750, Edmonton, Alberta, 2002.

[24] N. F. Noy and M. A. Musen. “Ontology versioning in an ontology management framework”. *IEEE Intelligent Systems*, 19(4):6–13, 2004.

[25] D. Ognyanov and A. Kiryakov. ”Tracking Changes in RDF(S) Repositories”. In *Procs. of the 13th Intern. Conf. on Knowledge Engineering and Management, EKAW'02*, Spain, 2002.

[26] D. E. Oliver, Y. Shahar, E. H. Shortliffe, and M. A. Musen. “Representation of Change in Controlled Medical Terminologies”. *AI in Medicine*, 15:53–76, 1999.

[27] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW*, volume 2473, pages 285–300. Springer, 2002.

[28] Max Volkel, Wolf Winkler, York Sure, Sebastian Ryszard Kruk, and Marcin Synak. ”SemVersion: A Versioning System for RDF and Ontologies”. Heraklion, Crete, May 29 June 1 2005. Procs. of the 2nd European Semantic Web Conference, ESWC'05.

[29] M. Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

[30] K. Zhang, J.T.L Wang, and D. Shasha. ”On the Editing Distance Between Undirected Acyclic Graphs and Related Problems”. In *Procs. of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.