# On Storage Policies for Semantic Web Repositories that Support Versioning

Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou

Computer Science Department, University of Crete, GREECE, and
Institute of Computer Science, FORTH-ICS, GREECE
email: {tzitzik, theohari, andreou}@ics.forth.gr

**Abstract.** This paper concerns versioning services over Semantic Web (SW) repositories. We propose a novel storage index (based on partial orders), called POI, that exploits the fact that RDF Knowledge Bases (KB) have not a unique serialization (as it happens with texts). POI can be used for storing several (version-related or not) SW KBs. We discuss the benefits and drawbacks of this approach in terms of storage space and efficiency both analytically and experimentally in comparison with the existing approaches (including the change-based approach). For the latter case we report experimental results over synthetic data sets. POI offers notable space saving as well as efficiency in various cross version operations. It is equipped with an efficient version insertion algorithm and could be also exploited in cases where the set of KBs does not fit in main memory.

## 1 Introduction and Motivation

The provision of versioning services is an important requirement of several modern applications of the Semantic Web, e.g. for digital information preservation [3, 14], and for e-learning applications [13]. Two key performance aspects of a version management system is the *storage space* and the *time* needed for creating (resp. retrieving) a new (resp. existing) version. In the Semantic Web (SW) there exist only limited support of versioning services. Most of the related works [9, 16, 10, 11] propose high level services for manipulating versions but none of these have so far focused on the performance aspect of these services (they mainly overlook the storage space perspective). In general, we could identify the following approaches:

(a) Keep stored each version as an independent triple store. This is actually the approach adopted in all previous works [16, 11, 9] on versioning for SW repositories. The obvious drawback of this approach is the excessive storage space requirements.
(b) Keep stored only the deltas between two consecutive versions [12, 2]. To construct the contents of a particular version one has to execute a (potentially long) sequence of deltas (which could be computationally expensive). However, appropriate comparison functions and change operation semantics can result in smaller in size deltas as described in [17]. The extreme case where only the change log is kept stored and marked positions (on that log) are used to indicate versions, is elaborated on [15] which also provides methods for reducing the size of a sequence of deltas.

In this paper we propose and analyze an approach (actually a storage index structure) that stands between the above two extremes. It aims at exploiting the fact that it is expected to have several versions (not necessarily consecutive) whose contents overlap. In addition, it exploits the fact that RDF graphs have not a unique serialization (as it happens with text), and this allows us to explore directions that have not been elaborated by the classical versioning systems for texts (e.g. [12, 2]). Specifically, we view an RDF KB as a *set* of triples. In a nutshell, the main contribution of this work is the introduction of a storage data structure, called `POI`, based on partial orders that exploits the expected overlap between versions' contents in order to reduce the storage space, also equipped with algorithms (including an auxiliary caching technique) for efficient version insertion and retrieval. It is important to note that the structure (and occupied space) of `POI` is independent of the version history. Knowledge of version history can be exploited just for speeding up some operations (specifically the insertion of versions that are defined by combining existing versions). It follows that the benefits from adopting `POI` are not limited to versioning. It could also be exploited for building repositories appropriate for collaborative applications, e.g. for applications that require keeping personal and shared spaces (KBs) as it is the case of modern e-learning applications (e.g. see trialogical e-learning [13]). In such cases, we need to store a set of KBs that are not historically connected and are expected to overlap.

In comparison to the change-based approaches (proposed in the context of the SW [17] or not [4, 5]) we could say that `POI` stores explicitly only the versions with the minimal (with respect to set containment) contents. All the rest versions are stored in a positively incremental way, specifically positive deltas are organizing as a partially ordered set (that is history-independent) aiming at minimizing the total storage space. It follows that `POI` occupies less space than the change-based approach in cases where there are several versions (or KBs in general) not necessarily consecutive (they could be even in parallel evolution tracks) whose contents are related by set inclusion ($\subseteq$).

Regarding version retrieval time, the cost of retrieving the contents of a version in the change-based approach is analogous to the distance from the closest stored snapshot (either first or last version, according to the delta policy adopted [5]). Having a `POI` the cost is independent of any kind of history, but it depends on the contents of the particular version, specifically on the depth of the corresponding node in the `POI` graph.

In comparison with other works on versioning (not in the SW context), [5] focuses on providing fast access to the current (or recent) versions, while we focus on minimizing the storage space, in an attempt to support applications with vast amounts of overlapping versions. [6] focuses on composite versioned objects, i.e. objects composed of other versioned objects.

In general, `POI` is an advantageous approach for archiving set-based data, especially good for inclusion-related and "oscillating"[1] data. This is verified analytically and experimentally. The remainder of this paper is organized as follows: Section 2 introduces basic notions and notations. Section 3 introduces `POI`. Section 4 elaborates on the storage requirements of `POI` by providing analytical and experimental results. Section 5 provides version insertion algorithms and reports experimental results. Sec-

[1] Suppose, for instance, a movie database describing movies, theaters that showtimes. Unlike movie descriptions, which rarely change, theaters and showtimes change daily.

tion 6 discusses other operations that can be performed efficiently with a `POI`. Finally, Section 7 summarizes and identifies issues for future research. Due to space limitations, proofs and other details are available in the extended version of this paper[2].

## 2 Framework and Notations

**Def. 1** If $\mathcal{T}$ is the set of all possible RDF triples (or quadruples if we consider graph spaces [7]), then :

- a *knowledge base* (for short KB) is a (finite) subset $S$ of $\mathcal{T}$,
- a *named knowledge base* (for short NKB) is a pair $(S, i)$ where $i$ is an identifier and $S$ is a KB,
- a *multi knowledge base* (for short MKB) is a set of NKBs each having a distinct identifier, and
- a *versioned knowledge base* (for short VKB) is a MKB plus an acyclic *subsequent* relation over the identifiers of the NKBs that participate to MKB.

Let $Id$ be the set of all possible identifiers (e.g. the set of natural numbers). If $v = (S, i)$ is a NKB (i.e. $S \subseteq \mathcal{T}$, $i \in Id$) then we will say that $i$ is the identifier of $v$, and $S$ is the content of $v$ (we shall write $id(v) = i$ and $D(i) = S$ respectively).

Let $V = \{v_1, \ldots, v_k\}$ be a MKB. We shall use $id(V)$ to denote the identifiers of $v_i$, i.e. $id(V) = \{\, id(v) \mid v \in V\}$, and $D(V)$ to denote their KBs, i.e. $D(V) = \{\, D(v) \mid v \in V\}$. We shall use $T_V$ denote the set of all distinct triples of $V$, i.e. $T_V = \cup_{v \in V} D(v)$. So $D(V)$ is a family of subsets of $T_V$.

A *subsequent* (version) relation over a MKB $V$ is any function of the form $next : id(V) \to \mathcal{P}(id(V))$, where $\mathcal{P}(\cdot)$ denotes powerset. For instance, suppose that $next(i) = \{j, k\}$. In this case we will say that $i$ is a direct previous version of $j$ and $k$, and that $j$ and $k$ are direct next versions of $i$. If $next(i) = \emptyset$, then we will call version $i$ *leaf* version of $V$. We call a version id $i$ the *root* version of $V$, if there does not exist any version id $j$, such that $i \in next(j)$. A pair $(V, next)$ is a VKB if the graph $(id(V), next)$ is acyclic.

Table 1 presents some version management services and their semantics in the form of conditions that should hold before their call and after their run. In particular, *insert* adds a new version with id $i$ and version content $S$. Additionally, *merge* differs from *insert* service in that the content of the new version (with identifier $h$) is the result of the application of a set operator, denoted by $\odot$, on the contents of two (or more) existing versions (having identifiers $i$ and $j$). In this way we could model operators like those proposed in [8].

## 3 The Partial Order Index (`POI`)

Consider a versioned knowledge base $(V, next)$. Specifically consider a $V$ comprising four versions with: $D(1) = \{a, b\}, D(2) = \{a, b, c\}, D(3) = \{a, c\}$ and $D(4) =$

[2] http://www.csd.uoc.gr/∼theohari/pub/eswc2008.pdf

| | Service | Pre-condition | Post-Condition |
|---|---|---|---|
| 1 | $insert(S, i)$ | $i \notin id(V)$ | $V' = V \cup \{(S, i)\}$ |
| 2 | $merge(i, j, h, \odot)$ | $\{i, j\} \subseteq id(V), h \notin id(V)$ | $V' = V \cup (D(i) \odot D(j), h)$, $next'(i) = next(i) \cup \{h\}$, $next'(j) = next(j) \cup \{h\}$ |

**Table 1.** Version creation services

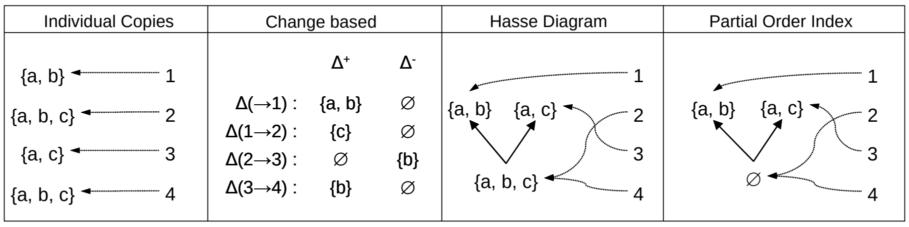

**Fig. 1.** Storage Example when the partial order index is used (right) or not (left)

$\{a, b, c\}$, where $a, b, c$ denote triples. This means that $V = \{(\{a, b\}, 1), (\{a, b, c\}, 2), (\{a, c\}, 3), (\{a, b, c\}, 4)\}$. Below we present methods for storing $V$ (we will not discuss the storage of $next$ as this is trivial and of minor importance). To aid understanding, Figure 1 sketches the storage policies that we will investigate in the sequel. One trivial approach, which is adopted by current SW versioning tools [16, 11, 9], is to store each individual NKB independently and entirely. Another approach [17, 4, 5] is to store the initial NKB and the deltas of every other version with respect to its previous version. Since versions usually contain overlapping triples, we propose the use of an index in order to reduce the number of triple copies. For instance, in the example of Figure 1 four copies of triple $a$ are stored in the trivial case (see the left part of Figure 1). However, with the use of the index we propose, only two $a$ copies are stored (see the right part of Figure 1). To describe this index, we first introduce some preliminary background material and definitions.

Given two subsets $S, S'$ of $\mathcal{T}$, we shall say that $S$ is narrower than $S'$, denoted by $S \leq S'$, if $S \supseteq S'$. So, $\emptyset$ is the top element of $\leq$, and the infinite set $\mathcal{T}$ is the bottom element. Clearly, $(\mathcal{P}(\mathcal{T}), \leq)$ is a partially ordered set (poset).

We can define the *partial order of a versioned KB* by restricting $\leq$, on the elements of $D(V)$. For brevity, we shall use the symbol $\sqsubseteq$ to denote $\leq_{|D(V)}$.

In our running example, we have $D(V) = \{\{a, b\}, \{a, b, c\}, \{a, c\}, \{a, b, c\}\}$ and the third diagram of Figure 1 shows the Hasse diagram of the partially ordered set $(D(V), \sqsubseteq)$. This diagram actually illustrates the structure of the so-called *storage graph* that we introduce below.

A *storage graph* $\Upsilon$ is any pair $\langle \Gamma, stored \rangle$ where $\Gamma = (N, R)$ is a directed acyclic graph and $stored$ is a function from the set of nodes $N$ to $\mathcal{P}(\mathcal{T})$. If $(a, b) \in R$, i.e. it is an edge, we will also write $a \rightarrow b$.

For a node $n \in N$, $stored(n)$ is actually a set of triples, so it is the storage space associated with node $n$. Table 2 shows all notations (relating to storage graphs) that will be used.

| Notation | Definition | Equiv. Notation |
|---|---|---|
| $R$ | a binary relation over the set of nodes $N$ | $\rightarrow$ |
| $R^t$ | the transitive closure of the relation $R$ | $\rightarrow^t$ |
| $R^r$ | the reflexive and transitive reduction of the relation $R$ | $\rightarrow^r$ |
| $Up(n)$ | $= \{n' \mid (n, n') \in R\} = \{n' \mid n \rightarrow n'\}$ | |
| $Down(n)$ | $= \{n' \mid (n', n) \in R\} = \{n' \mid n' \rightarrow n\}$ | |
| $Up^t(n)$ | $= \{n' \mid (n, n') \in R^t\} = \{n' \mid n \rightarrow^t n'\}$ | |
| $Down^t(n)$ | $= \{n' \mid (n', n) \in R^t\} = \{n' \mid n' \rightarrow^t n\}$ | |
| $content(n)$ | $= \cup\{\ stored(n') \mid n' \in Up^t(n)\}$ | |

**Table 2.** Notations for Storage Graphs

For each node $n$ of a storage graph we can define its *content*, denoted dy $content(n)$, by exploiting the structure of the graph and the function $stored$. Specifically we define:

$$content(n) = \cup\{\ stored(n') \mid n \rightarrow^t n'\} \tag{1}$$

so it is the union of the sets of triples that are stored in all nodes from $n$ to the top elements of $\Gamma$. We should stress that $content(n)$ is not stored, instead it is computed whenever it is necessary.

The *Partial-Order Index*, for short `POI`, is a storage graph whose structure is that of $(D(V), \sqsubseteq^r)$. Note that $\sqsubseteq^r$ denotes the reflexive and transitive reduction of $\sqsubseteq$. Consider a NKB $(D(i), i)$. Each version id $i$ is associated with a node $n_i$ whose storage space is defined as:

$$\begin{aligned} stored(n_i) &= D(i) \setminus \{\ D(j) \mid D(j) \sqsubseteq D(i)\ \} \\ &= D(i) \setminus \{\ stored(n_j) \mid n_i \rightarrow^t n_j\ \} \\ &= D(i) \setminus \{\ stored(n_j) \mid n_j \in Up^t(n_i)\ \} \end{aligned}$$

It follows easily that if $n_i \rightarrow^t n_j$ then it holds $stored(n_i) \cap stored(n_j) = \emptyset$. The fourth diagram of Figure 1 illustrates the storage graph of this policy for our running example. For each node $n_i$ the elements of $stored(n_i)$ are shown at the internal part of that node. Although we have 4 versions, $\Gamma$ contains only 3 nodes. As an example, $D(4) = \{a, b\} \cup \{a, c\} = \{a, b, c\}$.

## 4 Analyzing Storage Space Requirements

Let `IC` (from "individual copies") denote the policy where each individual version is stored independently, `POI` denotes the case where a `POI` is adopted and `CB` (from Changed Based) the case where deltas are stored. Below we compare these policies with respect to *storage space*.

If $Z$ denotes a policy, we shall use $space_t(Z)$ to denote the number of triples that are stored according to policy $Z$. It is not hard to see that

$$|T_V| \leq space_t(\texttt{POI}) \leq space_t(\texttt{IC}) = \sum_{v_i \in V} |D(v_i)|$$

$$|T_V| \leq space_t(\texttt{CB}) \leq 2 \sum_{v_i \in V} |D(v_i)|$$

Regarding the first formula, strict inequality holds, i.e. $space_t(\texttt{POI}) < space_t(\texttt{IC})$ if there is a version whose content is a subset of the content of another version. Specifically, the worst case for POI is when all nodes of the storage graph are leaves (except for the root). That case leads to space requirements equal to those of IC. On the other hand, the best case for POI, is when the content of every version is a subset of the content of every version with greater content cardinality. In that case every triple is stored only once in the storage graph. Regarding CB, in the worst case it stores $2 \times space_t(\texttt{IC})$ triples, while in the best case stores every triple only once and thus coincides with the best case of POI.

Regarding graph size, for IC and CB no index structure has to be kept as we store explicitly the entire of every version. Concerning POI, the number of nodes of the storage graph is $|D(V)|$. Notice that $|D(V)| \leq |id(V)|$, i.e. less than or equal to the number of versions[3]. The number of edges coincides with the size of the relation $\sqsubseteq^r$. This relation can have at most $\frac{N^2}{4}$ relationships. This value is obtained when $\Gamma$ is a bipartite graph, whose $\frac{N}{2}$ nodes are connected with all other $\frac{N}{2}$ nodes. More on the overall comparison of these policies are described below.

## 4.1 Experimental Evaluation

In order to measure the storage space required by each policy we created a testbed comprising from 100 to 1000 versions, each having 10,000 triples on average, where the size of each triple is 100 bytes (a typical triple size). As in real case scenarios, a new version is commonly produced by modifying an existing version, in order to generate the content of a new version, we first choose at random a parent version and then we either *add* or *delete* triples from the parent contents. The difference in triples with respect to the parent content is 10%, i.e. 1000 triples. We have an additional parameter $d$ that defines the probability to choose triple additions (so with probability $1 - d$ we subtract triples). In this respect, we create versions whose contents are either supersets or subsets of the contents of existing versions. We experimented with $d$ in the range of [0.5, 1.0] (we ignored values smaller than 0.5 as subtractions usually do not exceed additions). For additions, we assumed that the $25\%$ of the additional triples are triples which already exist in the KB (in the content of a different than the parent version), while the rest $75\%$ are brand new triples. This is motivated by the fact that in a versioning system it is more rare to re-add a triple which exists in an old version and was removed in one of the subsequent versions, than to add new triples. Clearly, as $d$ approaches 1, more new triples are created and less are deleted (so the total number of distinct triples increases). The minimum (resp. maximum) sized version contains 6000 (resp. 13000) triples, while $|T_V| = 43000$. We should also stress that, if we only create versions by adding triples, then the resulting storage graph resembles a tree. The higher the probability of deleting triples is, the higher the probability of having nodes with more than one fathers becomes. In the later case, we have increased number of edges. The more edges we have the higher the probability of having nodes with overlapping stored contents.

[3] In our running example $|D(V)| = 3$ while $|id(V)| = 4$

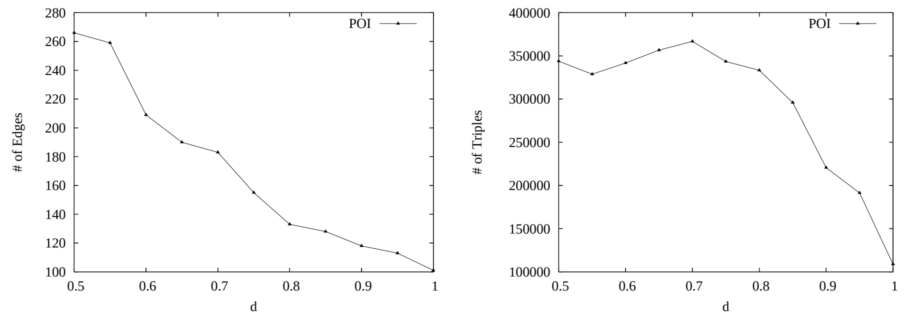

**Fig. 2.** # of graph edges (left) and triples (right) for `POI`

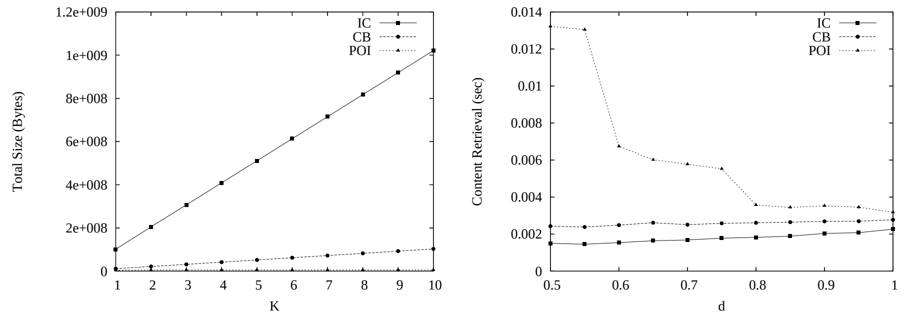

**Fig. 3.** Storage space requirements (left) and retrieval costs (right) of `IC`, `CB` and `POI`

We compared the three approaches `IC`, `CB` and `POI` using a PC with a Pentium IV 3.4GHz processor and 2 GB of main memory, over Windows XP. For the change-based approach (`CB`), we compute and store the delta $\Delta_e$ [17] between two consecutive versions. Regarding, the storage graph of `POI`, we assume node size equal to 20 bytes and edge size equal to 12 bytes. To further reduce the storage space, we used a table listing all triples each associated with a unique identifier. The contents of each version (specifically $n.stored$) is represented as a set of identifiers (rather than triples).

We first show the characteristics of `POI` and then we proceed with its comparison with the rest two policies. The left part of Figure 2 shows the number of graph edges of `POI`. As $d$ increases, the number of edges (and consequently of paths) decreases and `POI` gains advantage by its invariant to store every triple only once in a single path. The right part of Figure 2 shows the number of triples stored in `POI`. Notice that as $d$ increases, the number of stored triples tends to decrease (because duplicates decrease).

To show that `POI` saves space in cases where there are versions with equal or inclusion-related content, we used datasets that guarantee that each distinct triple set is content of $K$ versions. The left part of Figure 3 shows the total size of `IC`, `POI`, and `CB` for $d = 0.5$ and various values of $K$ (specifically $1 \leq K \leq 10$)[4]. Notice that `CB` and `POI` are much better than `IC` (9 and 18 times better respectively), and the

[4] The total number of versions is 100*K.

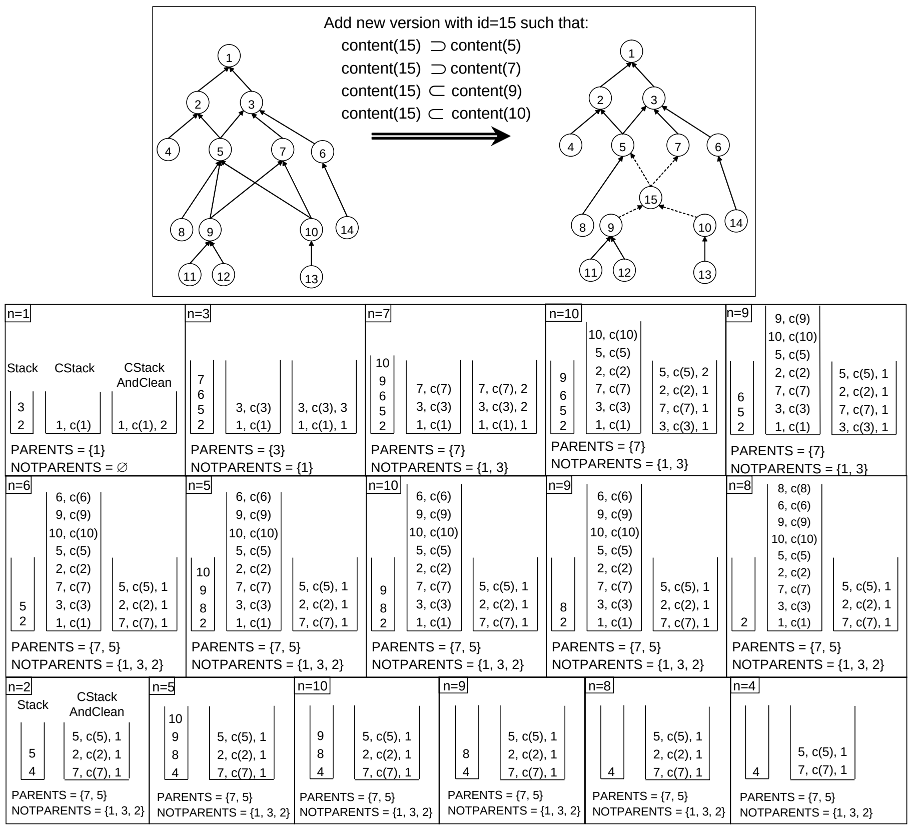

**Fig. 4.** Example of adding a new version using Algorithm *Insert POIds*

greater the value of $K$ is, the better `POI` than `CB` is. For $K = 1$ `POI` is roughly 2 times better than `CB`, while for $K = 10$, `POI` is roughly 18 times better than `CB`.

Regarding version retrieval times, we measured the time to retrieve the contents of all versions and report the average. The results are shown in the right part of Figure 3 for various values of $d$ and $K = 1$. Obviously, `IC` is the best regarding content retrieval, since no structure should be traversed, as every node is assigned to its content. In contrast, `POI` needs to traverse all the ancestors of the given node. `CB` slightly outperforms `POI`, but we have to note that the history paths in our datasets are very small. In a realistic setting `CB` would be much slower than `POI`. Once again, the decrease in number of edges as long as $d$ increases, results in shorter traversals for `POI` and as a consequence its difference with `IC` tends to decrease. In any case, `POI` content retrieval time is acceptable (i.e., max. of 0.013 sec).

## 5 Version Insertion Algorithms

The insertion algorithms exploit the structure and the semantics of the storage graph. Intuitively, we have to check whether the new version is subset or superset of one of the existing versions. To this end, we start from the root(s) of the storage graph and we descend. To be more specific, let $\mathcal{B}$ be a family of subsets of $\mathcal{T}$, let $<$ be the cover relation over these and let $<^r$ be its transitive reduction. Suppose that we want to insert a new subset $A$ (i.e. $A \notin \mathcal{B}$) and update accordingly the relation $<^r$. We define:

$$Parents(A) = \min_{<}\{\, B \mid A < B\} = \{\, B \mid A <^r B\}$$
$$Children(A) = \max_{<}\{\, B \mid B < A\} = \{\, B \mid B <^r A\}$$

To update $<^r$ we have to add the relationships $A <^r p$ for every $p \in Parents(A)$, and

**Algorithm 1** InsertInPoset($A$)

Input: a set of triples $A$
Output: updated cover relation of the poset so that to contain a node corresponding to $A$ (if there is no such node already)

```
==FIND PARENTS===============================
(1)   Stack = new STACK(); PARENTS = new Set(); NOTPARENTS = new Set();
(2)   push (Stack, $\{$root($\Gamma$)$\}$)
(3)   while not(isEmpty(Stack))
(4)       $n$ = pop(Stack)
(5)       if (n $\in$ NOTPARENTS)
(6)           push(Stack, $\{x \in Down(n) \mid x \notin PARENTS)\}$)
(7)       else if $n$.contents $= A$     // a node with contents A already exists
(8)           break
(9)       else if $n$.contents $\subset A$
(10)          PARENTS = (PARENTS $\cup\{n\}) \setminus Up^t(n)$ // all upper nodes of n are certainly
                                                              // not parents
(11)          NOTPARENTS = NOTPARENTS $\cup Up^t(n)$
(12)          push(Stack, $Down(n)$)
==FIND CHILDREN==============================
(13)  Stack = new STACK(PARENTS) // a new stack with initial contents the set PARENTS
(14)  CHILDREN = new Set()
(15)  while not(isEmpty(Stack))
(16)      $n$ = pop(Stack)
(17)      if $n$.contents $\supset A$
(18)          CHILDREN = CHILDREN $\cup\{n\}$
(19)      else
(20)          push(Stack, $Down(n)$)
==CONNECT A===============================
(21)  $nA$ = new node($A$)
(22)  For each $c \in$ CHILDREN Add($c \rightarrow nA$)     // i.e. Add( $c <^r A$)
(23)  For each $p \in$ PARENTS Add($nA \rightarrow p$)     // i.e. Add( $A <^r p$)
==ELIMINATE REDUNDANCIES===============================
(24)  for each $c \in$ CHILDREN
(25)      for each $p \in$ PARENTS
(26)          if $c \rightarrow p$ then Delete($c \rightarrow p$)     // i.e. if $c <^r p$ then Delete($c <^r p$)
```

$c <^r A$ for every $c \in Children(A)$. In addition we have to eliminate redundant relationships (that may exist between $Parent(A)$ and $Children(A)$, specifically we have to eliminate all $c <^r p$ relationships where $c \in Children(A)$ and $p \in Parents(A)$).

Let now see how we could find the children and the parents of a set $A$ and update appropriately the relation $<^r$. Returning to the problem at hand, this scenario corresponds to the case where each node $n$ of a storage graph had explicitly stored $contents(n)$. Algorithm 1 sketches the crux of the algorithm in pseudocode. It is based on a Stack and two sets called PARENTS and NOTPARENTS. The root of the storage is denoted by $root(\Gamma)$ and every storage graph has a single root corresponding to a dummy version with an empty content. Note that the relation $\rightarrow$ of the storage graph corresponds to the relation $<^r$, i.e. $a \rightarrow b \Rightarrow content(b) \subset content(a)$. An indicative example of version addition is illustrated in Figure 4. The stack contents are shown in every step.

To implement version insertion we could use Alg. 1. The only difference is that in a storage graph $n.contents$ (where $n$ is a node) is not explicitly stored (instead only $n.stored$ is stored). One naive approach would be to compute $n.contents$ by taking the union of the stored triples of its (direct and indirect) broader nodes, i.e. to use the formula (1). Each such computation would require $\mathcal{O}(d(n))$ set union operations where $d(n)$ is the depth of the node $n$ multiplied to the average number of parents. If the storage graph is a tree, then all set union operations would actually be concatenations (and thus faster). In case of DAG, we have to perform set union operations only for nodes that have more than one father. We will hereafter call this algorithm `POI`-*plain* insertion (for short `Insert POI`$_\mathbf{p}$) algorithm.

## 5.1 *Insert POI-DoubleStack (*`Insert POI`$_\mathbf{ds}$*)* Insertion Algorithm

To reduce the number of set union operations that are issued by the `Insert POI`$_\mathbf{p}$ algorithm for computing $content(n)$ for a node $n$, here we present a more time efficient algorithm which employs a second stack (actually it is a cache) for keeping stored (and thus reusing) results of operations that have already been computed. The second stack, called $CStack$ (where $'C'$ comes from contents), stores elements comprising of two components: a version id and its content (i.e. a set of triples). The extension of Alg. 1 with the second stack is Algorithm 4.

**Algorithm 2** TSContent(n)

Input: a node id
Output: the set of triples comprising the content of node $n$ (the stack CStack is updated, if necessary).

```
(1)  if (e=lookup(CStack, n))
(2)      return e.content //returns the contents of element e
(3)  else
(4)      Res = stored(n)
(5)      for each $n' \in Up(n)$
(6)          Res = Res $\cup$ TSContent(n') // this is a concatenation if $|Up(n) = 1|$
(7)      push(CStack, (n, Res)) //pushes a pair (key, content) to the stack
(8)  return Res
```

To compute the contents of a node $n$ it uses the function **TSContent** (Algorithm 2) which accesses the second stack. If the storage graph were a tree then we would be sure that all broader nodes of a node are in the stack (in both $Stack$ and $CStack$ stacks).

However the storage graph is a DAG in the general case, so this is not always true. That's why TSContent uses a lookup and if the sought element is not in $CStack$ it creates and stores it to $CStack$. We will hereafter call this algorithm *Insert POI-DoubleStack* (for short `Insert POI`$_{\mathbf{ds}}$) insertion algorithm.

**Algorithm 3** TSContentAndClean(n, PARENTS, initialN)

Input: a node id, the ids of its parents, a node id (in the root call of the routine n = initialN)
Output: the sets of triples comprising the content of node n. The stack is updated (addition, deletion) if necessary.

```
(1)  if (e = lookup(CStack, n))
(2)      Res = getElemContent(CStack, n)
(3)      if ((e.Y == 1) & (n $\notin$ PARENTS))
(4)          delElem(CStack, n)
(5)      else
(6)          editElem(CStack, (n, Res, Y), (n, Res, Y-1)) //decreases the 3rd component
                                                         //of the stack element
(7)  else
(8)      Res = stored(n)
(9)      for each $n' \in Up(n)$
(10)         Res = Res $\cup$ TSContentAndClean(n', PARENTS, initialN) //concatanation
(11)     push(CStack, (n, Res, $|Down(n)|$))
                 //pushes a triple (key, content, number of children) to the stack
(12) return Res
```

To reduce the total space needed by $CStack$, Alg. 4 actually uses a different implementation of Alg. 2 called **TSContentAndClean** (Alg. 3) that frees the contents of those versions that are not needed any more. Specifically, an element of $CStack$ should be removed if one of the following two conditions holds:
(a) its content is not a subset of $A$, so the traversal of the storage graph will not continue to its descendants and therefore its contents are not needed any more,
(b) it is not a (definite) parent of the node to be inserted and all its children are already in $CStack$. Specifically, if all its children are already in $CStack$ the version content is not needed because it is a subset of the content of every child of it.

To this end we extend the structure of each $CStack$ element with a third component, denoted by $Y$, which is actually a variable initialized to the number of children (of the corresponding node) that is decreased by one whenever lookup finds and fetches that element. When it reaches 0, the element (if not a parent) should be removed because that means that the contents of all its children are already stored in $CStack$. We will hereafter call this algorithm *Insert POI-DoubleStack(Clean)* (for short Insert `POI`$_{\mathbf{dsc}}$).

Figure 4 shows these stacks in our running example. Specifically, the left stack is the $Stack$, the center stack is the $CStack$ as employed by $TSContent$ algorithm, while the right one is the $CStack$ as employed by $TSContentAndClean$ algorithm. After the PARENTS have been computed, $CSTack$ remains the same and therefore we show only $Stack$. Notice how shorter $CStack$ is according to $TSContentAndClean$.

**Algorithm 4** Insert $\texttt{POI}_{\mathbf{dsc}}(A)$

Input: a set of triples $A$
Output: updated storage graph and CStack

```
==FIND PARENTS==============================
(1)  Stack = new STACK(); PARENTS = new Set(); NOTPARENTS = new Set()
(2)  push (Stack, $\{$root($\Gamma$)$\}$)
(3)  while not(isEmpty(Stack))
(4)      $n$ = pop(Stack)
(5)      if ($n \in$ NOTPARENTS)
(6)          push(Stack, $\{x \in Down(n) \mid x \notin$ PARENTS$)\}$)
(7)      else if $n$.contents $= A$
(8)          break
(9)      else if **TSContentAndClean**($n$, PARENTS, $n$) $\subset A$
(10)         PARENTS = (PARENTS $\cup\{n\}) \setminus Up^t(n)$
(11)         NOTPARENTS = NOTPARENTS $\cup Up^t(n)$
(12)         **delElems**(CStack, $Up^t(n)$)
(13)         push(Stack, $Down(n)$)
(14)     else
(15)         **delElem**(CStack, $n$)
==FIND CHILDREN==============================
(16) Stack = new STACK(PARENTS)
(17) CHILDREN = new Set()
(18) while not(isEmpty(Stack))
(19)     $n$ = pop(Stack)
(20)     if **TSContentAndClean**($n$, PARENTS, $n$) $\supset A$
(21)         CHILDREN = CHILDREN $\cup\{n\}$
(22)     else
(23)         push(Stack, $Down(n)$)
==CONNECT A==============================
(24) $nA$ = new node($A$)
(25) $nA$.stored = A $\setminus \cup \{$**TSContentAndClean**$(n') \mid n' \in$ Parents$\}$
(26) For each $c \in$ CHILDREN Add($c \rightarrow nA$)      // i.e. Add( $c <^r A$)
(27) For each $p \in$ PARENTS Add($nA \rightarrow p$)      // i.e. Add( $A <^r p$)
==ELIMINATE REDUNDANCIES==============================
(28) ... as in Alg. 1
==UPDATE THE STORED CONTENTS OF THE CHILDREN NODES =====
(29) for each $c \in$ CHILDREN
(30)     $c$.stored = $c$.stored - $A$
```

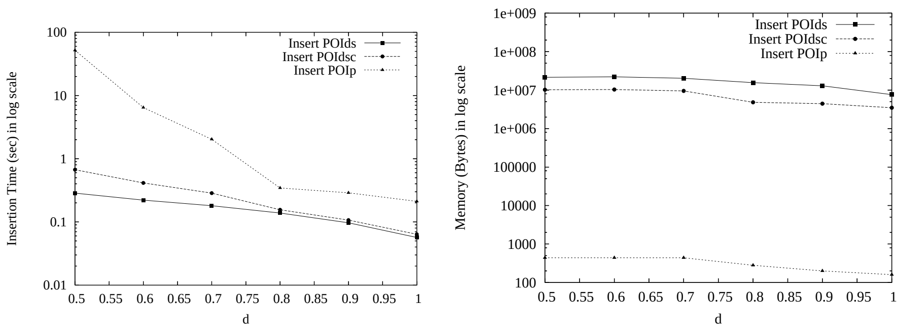

**Fig. 5.** Comparison of Insert-POI implementations

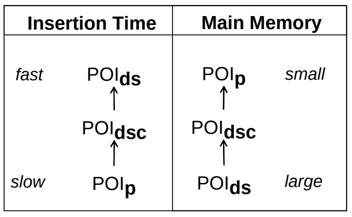

**Fig. 6.** Comparison of Version Insertion Algorithms

## 5.2 Experimental Evaluation

To compare the Insert `POI` implementations we employed the same testbed of 100 versions, of 10,000 triples on average, that has been presented in Section 4.1. The left (resp. right) part of Figure 5 illustrates the average version insertion time (resp. main memory required) in log scale. Of course, the size of the storage graph is the same irrespective of the employed insertion algorithm. As one would expect, `Insert POI`$_{\mathbf{p}}$ is the slowest but the less main memory consuming implementation of `POI`. On the other edge of the time-space tradeoff, lies `Insert POI`$_{\mathbf{ds}}$, which is the fastest but the most main memory consuming implementation. In particular, `Insert POI`$_{\mathbf{p}}$ is from 181 (for d=0.5) to 3.7 (for d=1) times slower than `Insert POI`$_{\mathbf{ds}}$, but `Insert POI`$_{\mathbf{ds}}$ needs 4 orders of magnitude more main memory. `POI`$_{\mathbf{dsc}}$ is 2-3 times less main memory consuming than `Insert POI`$_{\mathbf{ds}}$ and from 2.3 (for $d = 0.5$) to 1.1 (for $d = 1$) slower than `Insert POI`$_{\mathbf{ds}}$. Figure 6 summarizes the above results.

## 5.3 History-based Version Insertion Speedup

We have provided a general method for inserting versions and recall that the storage space requirements of `POI` are independent of the evolution history. However the knowledge of the evolution history can speed up version insertion, especially in the case we insert versions that are defined through set operations over existed versions. For instance, consider the case where we want to insert a new version $v_3 = v_1 \cup v_2$. In that case, the search for parents begins with $v_1$ and $v_2$, instead of the root, because $v_1 \subset v_3$ and $v_2 \subset v_3$. Nodes $v_1$ and $v_2$ will be the parents of $v_3$, unless there exists a descendant

$v_4$ of $v_1$ or $v_2$, such that $v_4 \subset v_3$. Additionally, the case where $v_3 = v_1 \cap v_2$ is in a sense dual to the previous one, as the children of $v_3$ will be $v_1$, $v_2$ or some nodes above them.

## 6 Other Applications & Extensions

Below we discuss a number of operations that can be performed efficiently if a `POI` is available.

Cross version operations can take advantage from the existence of a `POI`. For instance, inclusion checking can clearly benefit from a `POI`. To decide whether $D(i) \subseteq D(j)$ one could pose a reachability query on the storage graph (no need to access the contents of the versions). Moreover, by adopting a labeling scheme [1] for the storage graph $\mathit{\Gamma}$ we could decide inclusion in $\mathcal{O}(1)$.

Additionally, let $S$ be a set of triples. Suppose we want to find all versions $i$ such that $S \subseteq D(i)$ (or $D(i) \subseteq S$). Such queries would be very expensive in the `IC` or in the `CB` approach. By employing a `POI` we can use the insertion algorithm to insert $S$ to the storage graph. Let $n$ be the inserted node. The sought versions are those that point in nodes of $Nr^t(n)$ (resp. $Br^t(n)$).

We should also mention, that, if the storage graph $\langle \mathit{\Gamma}, \mathit{stored} \rangle$ does not fit in main memory, then we could keep only $\mathit{\Gamma}$ in main memory, while the function $\mathit{stored}$ could be kept in a relational storage with schema `Stored(vid,tid)` where `tid` is the triple identifier. To retrieve the contents of a version $i$, we need to compute $Br^t(i)$ using $\mathit{\Gamma}$ and then send to the db a disjunctive query (with all ids in $Br^t(i)$).

## 7 Concluding Remarks and Further Research

To the best of our knowledge, this is the first work that focuses on the storage aspect of SW repositories that support versioning. We proposed an index called `POI`, we verified the space gains of this index experimentally and we provided an efficient version insertion algorithm with acceptable main memory space requirements. From our experiments, `POI` can be 180 times more space economical compared to `IC` and 18 times compared to `CB` for parallel version tracks. Moreover, `POI` allows performing efficiently various cross-version operations.

It is worth mentioning that we have experimented also with storage graphs that have a *semi-lattice* structure, specifically with graphs that contain a node for each intersection of version contents. Such graphs guarantee that each triple is stored at most once. However, the storage gains obtained are compensated by the space required to keep the excessive number of edges of the storage graph. In future, we plan to compare `POI` with the *inverse* `POI`, i.e. with storage graphs that store explicitly the maximal elements and the internal nodes are negative deltas. We also plan to experiment with real data sets (currently we did not manage to find long version histories of SW data). Last, we could explore possible combinations of `POI` with change-based storage policies for enabling more sophisticated policies.

## References

1. R. Agrawal, A. Borgida, and H. V. Jagadish. “Efficient Management of Transitive Relationships in Large Data and Knowledge Bases”. *SIGMOD Records*, 18(2):253–262, 1989.
2. B. Berliner. CVS II: Parallelizing software development. In *Procs of the USENIX Winter 1990 Technical Conf.*, pages 341–352, Berkeley, CA, 1990. USENIX Association.
3. J. Cheney, C. Lagoze, and P. Botticelli. “Towards a Theory of Information Preservation”. In *Procs of the 5th European Conf. on Research and Advanced Technology for Digital Libraries, ECDL ’01:*, pages 340–351, London, UK, 2001. Springer-Verlag.
4. S. Chien, V. J. Tsotras, and C. Zaniolo. “Version Management of XML Documents”. In *Selected papers from the Third Intern. Workshop WebDB 2000 on The World Wide Web and Databases*, pages 184–200, London, UK, 2001. Springer-Verlag.
5. P. Dadam, V. Y. Lum, and H. D. Werner. “Integration of Time Versions into a Relational Database System”. In *VLDB ’84: Procs of the 10th Intern. Conf. on Very Large Data Bases*, pages 509–522, San Francisco, CA, USA, 1984.
6. S. Gançarski and G. Jomier. “A Framework for Programming Multiversion Databases”. *Data & Knowledge Engineering*, 36(1):29–53, 2001.
7. C. Gutierrez, C. Hurtado, and A. Mendelzon. “Foundations of Semantic Web Databases”. In *Procs of the 23th ACM Symp. on Principles of Database Systems (PODS)*, 2004.
8. Z. Kaoudi, T. Dalamagas, and T. Sellis. “RDFSculpt: Managing RDF Schemas Under Set-Like Semantics”. In *Procs of the 2nd European Semantic Web Conf. 2005 (ESWC05)*, Crete, Greece, 2005.
9. M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. “Ontology versioning and change detection on the web”. In *Procs of the 13th European Conf. on Knowledge Engineering and Knowledge Management (EKAW02)*, pages 197–212. Springer, 2002.
10. N. F. Noy, S. Kunnatur, M. Klein, and M. A. Musen. ”Tracking Changes During Ontology Evolution”. In *Procs of the 3rd Intern. Conf. on the Semantic Web (ISWC-2004)*, Japan, 2004.
11. N. F. Noy and M. A. Musen. “Ontology versioning in an ontology management framework”. *IEEE Intelligent Systems*, 19(4):6–13, 2004.
12. W. F. Tichy. RCS-a system for version control. *Software Practice & Experience*, 15(7):637–654, July 1985.
13. Y. Tzitzikas, V. Christophides, G. Flouris, D. Kotzinos, Hannu Markkanen, Dimitris Plexousakis, and N. Spyratos. “Emergent Knowledge Artifacts for Supporting Trialogical E-Learning”. *Intern. Journal of Web-based Learning and Teaching Technologies (IJWLTT)*, 2(3):16–38.
14. Y. Tzitzikas and G. Flouris. “Mind the (Intelligibily) Gap”. In *Procs of the 11th European Conf. on Research and Advanced Technology for Digital Libraries, ECDL’07*, Budapest, Hungary, September 2007. Springer-Verlag.
15. Y. Tzitzikas and D. Kotzinos. “(Semantic Web) Evolution through Change Logs: Problems and Solutions”. In *Procs of the Artificial Intelligence and Applications, AIA’2007*, Innsbruck, Austria, February 2007.
16. M. Volkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak. ”SemVersion: A Versioning System for RDF and Ontologies”. In *Procs. of the 2nd European Semantic Web Conf., ESWC’05.*, Heraklion, Crete, May 29 June 1 2005.
17. D. Zeginis, Y. Tzitzikas, and V. Christophides. “On the Foundations of Computing Deltas Between RDF Models”. In *Procs of the 6th Intern. Semantic Web Conf., ISWC/ASWC’07*, pages 637–651, Busan, Korea, November 2007.