An Adaptive Document Version Management Scheme

Boualem Benatallah[1], Mehregan Mahdavi[1], Phuong Nguyen[1], Quan Z. Sheng[1], Lionel Port[1], and Bill McIver[2]

[1] School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
[2] School of Information Science and Policy
University at Albany, State University of New York
Albany, New York 12222, USA
{boualem,mehrm,ntp,qsheng,lionelp}@cse.unsw.edu.au, mciver@albany.edu

Abstract. This paper addresses the design and implementation of an adaptive document version management scheme. Existing schemes typically assume: (i) *a priori* expectations for how versions will be manipulated and (ii) *fixed* priorities between storage space usage and average access time. They are not appropriate for all possible applications. We introduce the concept of document version pertinence levels in order to select the best scheme for given requirements (e.g., access patterns, trade–offs between access time and storage space). Pertinence levels can be considered as heuristics to dynamically select the appropriate scheme to improve the effectiveness of version management. We present a testbed for evaluating XML version management schemes.

1 Introduction

In several applications (e.g., digital government applications) there is a need for processing an increasing amount of data usually formatted using mark-up languages such as HTML and XML. In these applications, the issue of document content evolution is very important. For example, during a legislative mark-up process (i.e, the process of creating bills that will be proposed as laws), an initial document (i.e, the draft bill) is presented [1]. Legislators then begin a process of suggesting changes to the document, which ultimately voted out or are reconciled in a final document (i.e, the final bill).

1.1 Background

The work described in this paper focuses on the issue of managing multiple versions in document management systems such as file systems, HTML, and XML repositories. A variety of version management schemes has been developed for various data models including object, XML, HTML, text document models [2, 3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]. Indeed, some these schemes can be used to

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 46–62, 2003.

manage versions in different document representation models. However, existing schemes typically assume: (i) *a priori* expectations for how versions will be manipulated and (ii) *fixed* priorities between storage space usage and average access time. Consider the following schemes as examples [18,17,2]:

- **Scheme 1 - Store last version and backward deltas:** In this scheme, only the most recent version is stored. In order to be able to generate other versions, backward deltas (i.e, changes from a version to the previous one) are stored. The generation of recent versions is rather efficient (e.g., access to the last version is immediate). However, this scheme is not appropriate for applications which require access to historical information (i.e, temporal queries).
- **Scheme 2 - Store all versions:** In its simplest form, this scheme stores all versions. Thus, this solution is appropriate for applications which require access to historical information. The main drawback of this scheme is storage overhead. In addition, extra processing is needed to generate deltas which makes this solution inappropriate for change queries.

Clearly, a fixed scheme is unlikely to be appropriate for requirements (e.g., access to multiple versions may not necessarily follow predictable patterns, priorities between storage space usage and average access time may vary) of all possible applications. For example, once a legislative process has been completed, historical search may be performed across different versions of bills, by academics and citizens alike. Access to bills and versions thereof may be highly unpredictable. Some bills may have greater social significance or be more controversial than others. In addition, researchers may engage in different types of analyses across versions. The co–existence of all these situations in this type of document management system calls for an adaptive versioning scheme.

1.2 Contributions

In this paper, we propose an adaptive document version management scheme, which supports different version management schemes. Our work aims at providing a unified framework that can be used to combine several document versioning schemes. More precisely, the salient features of our approach are:

- A version is *stored* only if its existence is justified on performance, information–loss avoiding, or trade–offs between access time and storage space grounds, for example. Otherwise, it is generated on–demand. We refer to the latter as a *calculated version*. For example, a version may be kept as a stored version when it is intensively requested. However, when it is rarely requested, it may be kept as a calculated version. Furthermore, our approach features the deletion of old versions that are no longer useful.
- To decide if a version is to be stored, calculated, or deleted, we introduce the concept of document version pertinence levels. A version may be *pertinent*, *relevant*, or *obsolete*. Intuitively, the pertinence level of a version is a means to characterise the importance of the availability of a version in the history of a document with regard to application needs [8]. A version is:

  - pertinent if its availability is deemed highly important (e.g., intensively requested). Maintaining such a version as a stored version may, e.g., help increasing access performance.
  - relevant if its availability is not deemed highly important, but its existence is still necessary (e.g., rarely requested). Maintaining such a version as a calculated version may not have, e.g., any major negative impact on the overall access performance.
  - obsolete if its availability is deemed useless (e.g., no longer needed by applications, not expected to be requested in the future). The deletion of an obsolete version may, e.g., help improving performance and reducing the volume of a document history.

  This classification is based on a number of criteria, including past user access patterns, storage space and processing time trade–offs.
- We propose a technique that continuously adjusts the history of a document to match the intensively requested versions. We show that transformations of a document history over time, result in offering better alternatives of its content. Our approach helps in reducing the volume of a document history by enabling deletion of obsolete versions and replacement of stored versions by calculated versions when their status changes from pertinent to relevant. We give experimental results which demonstrate the effectiveness of the proposed technique.

The remainder of this paper is organised as follows. Section 2 presents the adaptive document versioning scheme. Section 3 discusses the refreshing of a document history. Implementation and evaluation issues are discussed in Sect. 4. Section 5 gives some conclusions.

2 Adaptive Document Versioning Scheme

The adaptability of our approach lies in the following features. Firstly, the history of a document is continuously adjusted to provide an effective versioning scheme based on criteria such as estimation of future usage of versions. Secondly, our approach allows explicit tuning of: (i) the used criteria and (ii) how such criteria contribute to the selection of a given versioning scheme.

We propose the concept of *pertinence agent* to facilitate the adaptability of a versioning scheme. The role of this agent is to continually gather and evaluate information about versions and recommend changes to a document history. A pertinence agent is an extensible object that can be attached to a document[1]. It contains operational knowledge such as number of versions, number of times a version is requested, size of versions, and change control policies related to the document history (e.g., a pertinence agent can be programmed to periodically refresh versions). It also provides operations for refreshing a document history (e.g., deleting versions, generating new versions).

[1] Note that a pertinence agent can be attached to several documents (e.g., documents of an XML repository).

A pertinence agent is extensible in the sense that it is possible to change its operational knowledge, e.g., tuning certain parameters in order to provide applications with ability to dynamically adjust the version management scheme to their precise needs. The remainder of this section is organized as follows. Section 2.1 presents the document version model used in our approach. Section 2.2 discusses the usage of the pertinence agent to determine version pertinence levels.

2.1 Basic Concepts and Definitions

In this section, we present the version model used in our approach. In this model, a document is associated to the set of its versions. This set can be regarded as the physical representation of the document itself. At the system level, a document is represented by a pair `(d,vers)`, where `d` is the document identifier (e.g., name) and `vers` is the set of its versions. A version is identified by a pair `(d,num)`, where `d` is the identifier of the document and `num` is the number that the system associates to the version, at its creation time. We use the notation $v_i(d)$ to denote the ith version (i.e, version number i) of the document d. We designate the latest version of a document as the *current version*. A *historical version* of a document is any of its versions which is not the current version.

As we already mentioned before, our approach differentiates between stored and calculated versions. A calculated version is generated on–demand. To generate a calculated version $v_i(d)$, a sequence of change operations is applied to stored version $v_j(d)$, called the *generation root* of $v_i(d)$. In Section 4, we will discuss how to determine a generation root. A version is represented by a tuple:

$$(v_i(d), i, status, p_i(d))$$

where *status* can be either "`stored`" or "`calculated`". $p_i(d)$ is a pointer by which the content of the version can be obtained. In particular, if *status* is "`stored`" then $p_i(d)$ refers to the document associated to $v_i(d)$. Otherwise, $p_i(d)$ refers to a sequence of change operations which allows to generate the version $v_i(d)$ from its generation root.

In order to generate a version from another one, we maintain changes between versions. We use the term *delta* to refer to the sequence of changes between two versions of a document. It should be noted that our versioning model does not target a specific model such as XML or HTML. Instead, it uses concepts that are common in any document model.

Below, we introduce some notations that will be used in the remainder of the paper. We denote by:

- $A(t, d)$ the set of version numbers i such that $v_i(d)$ is accessible at time t,

$$A(t, d) = \{i : v_i(d) \text{ is accessible at time } t\}$$

- $D(t, d, i, j)$ the distance between $v_i(d)$ and $v_j(d)$ at time t,

$$D(t, d, i, j) = \begin{cases} \#\{l : l \in A_t(d), i < l \leq j\} \text{ if } i < j \\ \#\{l : l \in A_t(d), i > l \geq j\} \text{ if } i \geq j \end{cases}$$

- $L(t, d, i)$ the distance between $v_i(d)$ and its generation root at time t.
- $Z(t, d)$ the average of the distance between a calculated version and its generation root at time t (taken over all existing calculated versions).
- $N(t, d, i)$ the number of accesses to $v_i(d)$ at time t since the last refreshing.
- $S_c(t, d)$ the average size of calculated versions of document d at time t. If none of the versions is calculated at time t, a default value is assigned to $S_c(t, d)$. Since maintaining a calculated version requires 2 deltas, the size of a calculated version is approximated using the size of deltas associated with it.
- $S(d, i)$ the size of $v_i(d)$.
- $S_s(t, d)$ the average size of stored versions of document d at time t.

2.2 Pertinence Criteria

As mentioned before, the pertinence level of a version is determined based on a number of criteria including user access patterns, storage space and processing time trade–offs. Based on these criteria, the pertinence agent uses a scoring function to determine the pertinence levels. $Score(t, d, i)$ is a number between 0 and 1 that represents the score given by the pertinence agent to $v_i(d)$ at time t. Higher value of $Score(t, d, i)$ means that $v_i(d)$ is more likely to be pertinent. The pertinence agent sorts versions according to their scores. A version is classified obsolete if its score is less than certain threshold τ. The top p % ($0 \leq p \leq 100$) of non–obsolete versions are classified pertinent. The remaining versions are classified relevant. p is a pre–defined parameter (e.g., set by an administrator). For some applications, it may not be practical to delete any existing version. To avoid deleting versions, τ should be set to a negative number (e.g., -1). Calculation of the scores is based on the following parameters: *storage space saving* and *average access time saving*.

Storage Space Saving. Since a version $v_i(d)$ can be maintained as either stored or calculated version, the storage space saving is measured as the difference between the storage space requirement of these two possibilities, i.e, $S(d, i) - S_c(t, d)$. Let $\delta_S(t, d, i)$ be the difference between $S(d, i)$ and $S_c(t, d)$ (i.e, $\delta_S(t, d, i) = S(d, i) - S_c(t, d)$). The scoring function associated with storage space saving parameter is defined as follows:

$$Score_S(t, d, i) = \begin{cases} 1 & : \text{ if } \delta_S(t, d, i) < 0 \\ k_S/(k_S + \delta_S(t, d, i)) & : \text{ otherwise} \end{cases}$$

$Score_S(t, d, i)$ represents the score given to the storage space saving of maintaining $v_i(d)$ as a stored version at time t. If it takes less space to maintain $v_i(d)$ as a stored version than as a calculated version, then $Score_S(t, d, i)$ is 1, otherwise $Score_S(t, d, i)$ takes a value between 0 and 1. The higher $\delta_S(t, d, i)$ is, the lower is $Score_S(t, d, i)$, and the more likely is $v_i(d)$ going to be kept as a calculated version. k_S is a positive constant that is used to scale the measurement unit of the storage space (e.g., if space is measured in K-bytes, k_S can be 1, but if space is measured in bytes, then k_S should be 1000).

Total Average Access Time Saving. The scoring function associated with this parameter takes into account both *version popularity* and *version extraction time*.

Version Popularity. User interest in a version $v_i(d)$ is reflected by the number of accesses to it since the last refreshing time. Higher number of accesses to $v_i(d)$ implies higher popularity of this version.

If $v_i(d)$ is an old version (i.e., created before last refreshing), its popularity may be measured by $N_t(d,i)$, i.e., the number of accesses to it since last refreshing time. On the other hand, since a new version is created after last refreshing time (i.e., it is created after we started recording number of accesses to versions), the number of accesses to a new version $v_i(d)$ is "scaled up" to $\nu N(t,d,i)/(D(t,d,i,n)+1)$ where n is the version number of the current version. Here, ν is called the *youth factor* which is used to account for the possible low number of accesses to younger versions (e.g., there may be only few accesses to recently created versions). Choosing an appropriate value of the youth factor depends on the number of new versions. More specifically, the value of ν should be greater than or equal to the number of new versions.

For example, assume that: (i) there are 20 new versions at each refreshing time, (ii) the average number of accesses to the 10 most heavily accessed old versions is 10000, and (iii) $\frac{1}{20}\sum_{v_i(d)}$ is new $\frac{N(t,d,i)}{D(t,d,i)+1} = 200$.

Let's assume that the new versions should be as popular as the 10 most heavily accessed old versions (i.e., the average number of accesses to new versions is equals to the average number of accesses to the 10 most heavily accessed old versions), then ν can be set to $10000/200 = 50$.

The number of accesses to a version $v_i(d)$ since the last refreshing time is computed using the following function:

$$M(t,d,i) = \begin{cases} N(t,d,i) & : \text{ if } v_i(d) \text{ is an old version} \\ \nu N(t,d,i)/(D(t,d,i,n)+1) & : \text{ otherwise} \end{cases}$$

A straightforward way to compute the popularity of a version $v_i(d)$ is to use the function $M(t,d,i)$. A potential limitation of this approach is that versions are considered in isolation (i.e., the popularity of versions is computed independently of each other). However, if we consider the history as a whole, it may be beneficial, e.g., to maintain a $v_k(d)$ as a calculated if the version $v_{k-1}(d)$ is maintained as a stored version. affect the decision of whether or not to keep $v_k(d)$ as a stored version. Thus, it is important to consider correlation between versions when computing the popularity metric. For instance, if $v_{k-1}(d)$ is the generation root of $v_k(d)$, then accessing $v_k(d)$ requires accessing $v_{k-1}(d)$ first. The correlation between these two versions can be reflected in their popularity. Clearly, the effect of correlation between versions decreases as distance between them increases. Based on the above discussion, the popularity $P(t,d,i)$ of $v_i(d)$, at time t, can be approximated using the following function:

$$P(t,d,i) = M(t,d,i) + \sum_{j \in A(t,d), j \neq i} \frac{\mu M(t,d,j)}{(D(t,d,i,j)+1)^2}$$

We measure the contribution of correlation between $v_i(d)$ and $v_j(d)$ to the popularity of $v_i(d)$ at time t using:

$$\frac{\mu M(t,d,j)}{(D(t,d,i,j)+1)^2}$$

The predetermined constant μ is called the *correlation factor* ($\mu \geq 0$). Here the expression $\mu/(D(t,d,i,j)+1)^2$ represents the effect of distance between versions on their correlation. Normally, storing the XML document of $v_i(d)$ has more effect on the pertinence level of $v_j(d)$ if $v_j(d)$ is closer to $v_i(d)$ (i.e., $D(t,d,i,j)$ is small). If $\mu = 0$ there is no effect of correlation between versions on their popularity. The higher μ is, the higher is the effect of correlation between versions on their popularity.

Version Extraction Time. Suppose that $v_i(d)$ is to be kept as a stored version, then the processing time for extracting it is proportional to its size:

$$T(d,i) = \lambda_1 S(d,i)$$

Here λ_1 is a constant called *direct access factor*, which accounts for costs such as CPU processing time and disk access. On the other hand, if $v_i(d)$ is to be maintained as a calculated version, then its XML document needs to be generated on–demand. Recall from Section 2.1 that this is done by applying the change operations specified in the deltas to the generation root of $v_i(d)$. Therefore, the processing time to extract $v_i(d)$ at time t is approximated using the following function:

$$T_c(t,d) = \lambda_2 Z(t,d) S_c(t,d) + \lambda_1 S_s(t,d)$$

where $\lambda_2 Z(t,d) S_c(t,d)$ is the approximated time for obtaining the deltas and applying the change operations on them, and $\lambda_1 S_s(t,d)$ is the approximation of access time to the root of generation.

Here λ_2 is a constant called *indirect access factor*, which accounts for costs such as CPU processing time and disk access as well as the time required for applying change operations. λ_1 and λ_2 should not be seen as tuning parameters. Their values depend only on the characteristics of the computing environment (e.g., CPU speed, memory size).

As pointed out before, the scoring function associated with the total average access time saving parameter incorporates both version popularity and extraction time. Let $\delta_T(t,d,i)$ be the difference between $T_c(t,d)$ and $T(d,i)$ (i.e, $\delta_T(t,d,i) = T_c(t,d) - T(d,i)$). The scoring function associated with the total average access time saving is defined as follows:

$$Score_T(t,d,i) = \begin{cases} 0 & : \text{ if } \delta_T(t,d,i) < 0 \\ P(t,d,i)\delta_T(t,d,i)/(k_T + P(t,d,i)\delta_T(t,d,i)) & : \text{ otherwise} \end{cases}$$

$Score_T(t,d,i)$ returns a score measuring the contribution of the average access time parameter in the decision of whether to keep $v_i(d)$ as a stored version. The

higher the value of $Score_T(t,d,i)$ is, the more likely $v_i(d)$ is to be kept as a stored version. If it takes less time to access $v_i(d)$ as a calculated version than as a stored version, then $Score_T(t,d,i)$ is 0, otherwise $Score_T(t,d,i)$ takes a value between 0 and 1. The higher $P(t,d,i)\delta_T(t,d,i)$ is, the higher is $Score_T(t,d,i)$. k_T is a positive constant that is used to scale the measurement unit of access time.

Combined Weighted Score. The decision function $Score(t,d,i)$ combines the scores of storage space saving and total average access time saving. It can be tuned to capture trade–offs between space and access time. This is done by assigning weights to the importance of storage space saving and average access time saving. The combined score of a version $v_i(d)$ is computed as follows:

$$Score(t,d,i) = \lambda Score_S(t,d,i) + (1-\lambda)Score_T(t,d,i)$$

where $\lambda \in [0,1]$ is a the weight assigned to the importance of storage space saving and $1-\lambda$ is the weight assigned to the importance of average access time saving.

3 Document History Refreshing

In this section, we present the algorithm that adapts the history of a document. During the refreshing: (i) a previously pertinent version may become relevant or obsolete, and (ii) a previously relevant version may become obsolete or pertinent.

In the adaptation process, the system needs to determine which of the versions need to be:

- Deleted, i.e, obsolete versions.
- Converted from stored versions to calculated versions, i.e, versions whose status changes from pertinent to relevant.
- Converted from calculated versions to stored versions, i.e, versions whose status changes from relevant to pertinent.

In the remainder of this section, we will consider the adaptation of a sequence of versions $v_k(d), \ldots, v_n(d)$. First, we introduce the basic operations used to convert or delete versions. We then present the history refreshing algorithm.

3.1 History Refreshing Operations

Some refreshing operations involve composing deltas. Before formally introducing these operations, we briefly discuss the composition of deltas[2]. The advantage of composing deltas is that: given $\Delta_{i,j}(d)$ and $\Delta_{j,l}(d)$, it is possible to generate $\Delta_{i,l}(d)$ without accessing any version of d.

[2] Note that the focus of this paper is not on composing deltas. We mention this aspect here for completeness

We will use the following notations. We denote by $\otimes$ the composition of deltas (i.e., $\Delta_{i,l}(d) = \Delta_{i,j}(d) \otimes \Delta_{j,l}(d)$), c_c the average time for processing the composition, c_g the average time for generating a version $v_j(d)$ from $v_i(d)$ and $\Delta_{i,j}(d)$, and c_d the average time for obtaining the changes between two versions (i.e., getting $\Delta_{i,j}(d)$ from $v_i(d)$ and $v_j(d)$).

Recall that $D(t, d, i, j)$ represents the distance between $v_i(d)$ and $v_j(d)$. If some versions between $v_i(d)$ and $v_j(d)$ have been deleted, then $D(t, d, i, j) < |j - i|$. For clarity, in the following discussion we assume that numbers associated to available versions (i.e, identifiers of versions) are always consecutive natural numbers. This can be achieved by a temporary renumbering of versions $v_k(d), \ldots, v_n(d)$.

Conversion from Calculated to Stored Version. Suppose that, the status of $v_i(d)$ changes from relevant to pertinent. In this situation, $v_i(d)$ needs to be converted to a stored version. As we already mentioned in Section 2.1, the generation of a stored version requires the identification of its generation root. The generation root of $v_i(d)$ is its nearest stored version $v_j(d)$. In case $v_i(d)$ has two nearest stored versions, the youngest one is selected.

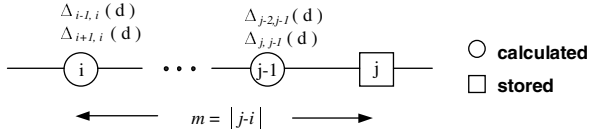


Fig. 1. Generating $v_i(d)$

$v_i(d)$ is generated by applying $\Delta_{j,i}(d)$ to $v_j(d)$. We use composition of deltas to get $\Delta_{j,i}(d)$:

$$\begin{aligned} \Delta_{j,j-2}(d) &= \Delta_{j,j-1}(d) \otimes \Delta_{j-1,j-2}(d) \\ \Delta_{j,j-3}(d) &= \Delta_{j,j-2}(d) \otimes \Delta_{j-2,j-3}(d) \\ &\cdots \\ \Delta_{j,i}(d) &= \Delta_{j,i+1}(d) \otimes \Delta_{i+1,i}(d) \end{aligned}$$

It should be noted that the deltas $\Delta_{j,l}(d)$ $(i \leq l < j - 1)$ are discarded after $v_i(d)$ has been generated. Let $m = L(t, d, i) - 1$, then the time needed to generate the stored version is $mc_c + c_g$.

Conversion from Stored to Calculated Version. Suppose that the status of $v_i(d)$ changes from pertinent to relevant. In this situation, $v_i(d)$ needs to be converted to a calculated version. Before the conversion, $p_i(d)$ refers to the XML document of $v_i(d)$. After the conversion, $p_i(d)$ should refer to $\Delta_{i-1,i}(d)$ and

$\Delta_{i+1,i}(d)$. Thus, it is necessary to generate $\Delta_{i-1,i}(d)$ and $\Delta_{i+1,i}(d)$. In order to do this, we need to get $v_{i-1}(d)$ and $v_{i+1}(d)$).

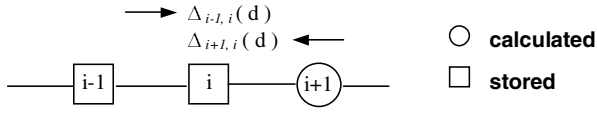


Fig. 2. Converting $v_i(d)$ into a calculated version

Suppose that among the versions $v_{i-1}(d)$ and $v_{i+1}(d)$, s versions are stored where $s \in \{0, 1, 2\}$. The average processing time for generating $v_{i-1}(d)$ and $v_{i+1}(d)$ is $(2-s)c_g$. The average processing time of the conversion is $(2-s)c_g+2c_d$.

Deletion of Versions. A version can be deleted when it becomes obsolete. This section discusses the deletion of a number of consecutive versions $v_i(d), \ldots, v_j(d)$ $(k \leq i \leq j \leq n)$.

If $v_{i-1}(d)$ and $v_{j+1}(d)$ are stored versions, then the deletion is simply done by removing $v_i(d), \ldots, v_j(d)$. However, if either $v_{i-1}(d)$ or $v_{j+1}(d)$ is a calculated version, then the situation becomes more complex. The following cases need to be considered:

Case 1. $v_{i-1}(d)$ is a stored version, $v_{j+1}(d)$ is a calculated version (Fig. 3)

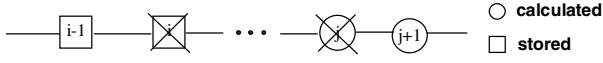


Fig. 3. Deleting $v_i(d), \ldots, v_j(d)$ (case 1)

Before the deletion, $p_{j+1}(d)$ refers to $\Delta_{j,j+1}(d)$ and $\Delta_{j+2,j+1}(d)$. After the deletion, $p_{j+1}(d)$ should refer to $\Delta_{i-1,j+1}(d)$ and $\Delta_{j+2,j+1}(d)$. Thus $\Delta_{i-1,j+1}(d)$ needs to be generated. There are two possible alternative strategies.

A first strategy is to generate $v_{j+1}(d)$ and $\Delta_{i-1,j+1}(d)$. The generation of $v_{j+1}(d)$ is done as discussed above. The generation processing time is $mc_c + c_g$, where $m = L(t, d, j+1) - 1$. The overall deletion average processing time is $mc_c + c_g + c_d$. Here we have $L(t, d, j+1) \leq 2 + D(t, d, i, j)$ and thus $m \leq D(t, d, i, j) + 1$.

A second strategy is to generate $\Delta_{i-1,j+1}(d)$ from other deltas, i.e.,

$$\Delta_{i-1,i+1}(d) = \Delta_{i-1,i}(d) \otimes \Delta_{i,i+1}(d)$$
$$\ldots$$
$$\Delta_{i-1,j+1}(d) = \Delta_{i-1,j}(d) \otimes \Delta_{j,j+1}(d)$$

The average processing time of the compositions in this strategy is $(D(t,d,i,j)+1)c_c$. However, if not all of the deltas $\Delta_{i-1,i}(d),\ldots,\Delta_{j-1,j}(d)$ are available (i.e., some of $v_i(d),\ldots,v_j(d)$ are stored versions), then the generation of these deltas requires additional processing.

In the algorithm which is presented in Section 4.2, there is a preprocessing phase where we estimate and compare the costs of the two strategies, and then select the most efficient one.

The case where $v_{i-1}(d)$ is a calculated version and $v_{j+1}(d)$ is a stored version, is handled similarly.

Case 2. $v_{i-1}(d)$ and $v_{j+1}(d)$ are both calculated versions (Fig. 4).

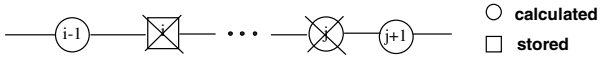


Fig. 4. Deleting $v_i(d),\ldots,v_j(d)$ (case 2)

Before the deletion, $p_{j+1}(d)$ refers to $\Delta_{j,j+1}(d)$ and $\Delta_{j+2,j+1}(d)$, while $p_{i-1}(d)$ refers to $\Delta_{i-2,i-1}(d)$ and $\Delta_{i,i-1}(d)$. After the deletion, $p_{j+1}(d)$ should refer to $\Delta_{i-1,j+1}(d)$ and $\Delta_{j+2,j+1}(d)$, while $p_{i-1}(d)$ should refer to $\Delta_{i-2,i-1}(d)$ and $\Delta_{j+1,i-1}(d)$. Thus $\Delta_{i-1,j+1}(d)$ and $\Delta_{j+1,i-1}(d)$ need to be generated.

Similar to case 1, this case can be handled using two different strategies. A first strategy is to generate $v_{i-1}(d)$, $v_{j+1}(d)$, $\Delta_{i-1,j+1}(d)$, and $\Delta_{j+1,i-1}(d)$. The average processing time of generating $v_{i-1}(d)$ and $v_{j+1}(d)$ is $(m_1+m_2)c_c+2c_g$ (where $m_1=Z_t(d,i-1)-1, m_2=Z_t(d,j+1)-1$). The overall processing time is $(m_1+m_2)c_c+2c_g+2c_d$.

A second strategy is to compose $\Delta_{i-1,j+1}(d)$ using $\Delta_{i-1,i}(d),\ldots,\Delta_{j,j+1}(d)$ and $\Delta_{j+1,i-1}(d)$ using $\Delta_{j+1,j}(d)$,
$\ldots,\Delta_{i,i-1}(d)$. In this strategy, the overall processing time is $2(D_t(d,i,j)+1)c_c$. However, similarly to the second strategy in case 1, if $\Delta_{i-1,i}(d),\ldots,\Delta_{j-1,j}(d)$ and $\Delta_{j+1,j}(d),\ldots,\Delta_{i+1,i}(d)$ are not available, the generation of these deltas requires additional processing. The most efficient strategy is selected at the preprocessing phase of the refreshing algorithm.

3.2 Refreshing Technique

This section describes the history refreshing algorithm. In this algorithm, operations for converting calculated versions to stored versions are scheduled in order to avoid repetition in generating deltas. For example, suppose that $v_i(d)$ and $v_l(d)$ are calculated versions which need to be converted to stored versions, and which have the same root of generation $v_j(d)$. Suppose also that: (i) $v_l(d)$ is located between $v_i(d)$ and $v_j(d)$ (i.e., either $i<l<j$ or $i>l>j$) and (ii)

$D(t,d,l,j) < D(t,d,l,i)$ (i.e, $v_j(d)$ is closer to $v_l(d)$ than $v_i(d)$). In this situation, $v_l(d)$ is converted before $v_i(d)$ in order to avoid repetition in composing deltas.

The detailed description of the algorithm is given in [18]. We summarise here the different phases of the algorithm, namely, *preprocessing*, *generation of versions*, and *history update*.

- **Preprocessing**: This phase consists of three main steps. In the first step, pertinence levels of versions are determined. In the second step, for each deletion, a strategy is selected based on the cost analysis discussed in 2. In the third step, the versions which need to be generated are identified. A sorted linked list, called L, is created. This list contains pairs (i,j), where $v_i(d)$ is a version which needs to be generated and $v_j(d)$ is the generation root of $v_i(d)$. L is sorted by $L(t,d,i)$ (i.e., $D(t,d,i,j)$).
- **Generation of versions:** In this phase, versions identified in the previous phase are generated. In order to optimise the processing cost, the version $v_i(d)$ which has the smallest value of $L(t,d,i)$ is generated first. In other words, the algorithm avoids repetition in generating the deltas. Thus, the algorithm minimises the overall generation processing time as well as the storage space.
- **History update:** In this last phase, pertinent versions which were relevant (before the refreshing) are stored, obsolete versions are deleted, and relevant versions which were pertinent (before the refreshing) are converted to calculated versions.

It can be seen that following this algorithm, all the documents which needed to be generated are identified in the first phase and generated in the second phase. After that, they are updated in the last phase. The processing time for the generation of documents is minimized by always generating the documents with smallest distance to its generation root first. This is because the processing time for generating the document of a calculated version $v_i(d)$ is $mc_c + c_g$, where $m = L(t,d,i) - 1$ and $L(t,d,i)$ is the distance from $v_i(d)$ to its root of generation.

4 Experiments and Analysis

In order to evaluate the performance of different XML version management schemes, we built a simulation testbed. In this section, we first present the architecture and implementation of the testbed. Then, we present the experimental results for evaluating the performance of different schemes.

4.1 Testbed Architecture

Figure 5 shows the generic architecture of the testbed. This testbed consists of the following components: *Simulator*, *VersionGenerator*, *RequestGenerator*, and *StrategySimulator*. The `Simulator` is used to initialise the experimental environment using configuration information (e.g., p, λ, μ, ν). The `VersionGenerator`

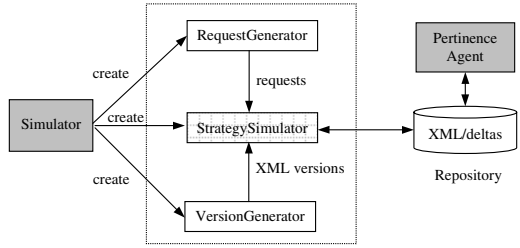


Fig. 5. Architecture of the testbed

acts as an XML document change simulator. Document versions are generated by randomly applying `deleteNode`, `updateText` and `insertElement` operations on each node of the XML document[3]. The `RequestGenerator` periodically generates requests (i.e., retrieving specific XML versions) based on a specific user's access pattern. Currently, the testbed supports three access patterns: (a) *recent* access in which younger versions (i.e., most recent versions) are intensively requested; (b) *early* access in which older versions are intensively requested; and (b) *uniform* access in which user's access does not depend on the age of versions. The `StrategySimulator` is used to either insert a new version or retrieve an existing version to/from the repository according to a versioning scheme. Currently, three versioning schemes are supported: *Store All Versions* (SAV for short), *Store Last Version And Backward Deltas* (SLVBD for short) and the *Adaptive Scheme* (AS for short). AS is the scheme proposed in this paper.

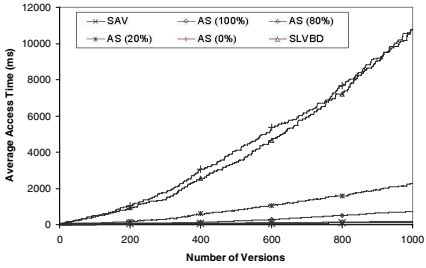
All the components of the testbed have been implemented in Java 2. In the current implementation, a PostgreSQL 7.1 database server is used as XML repository. For each scheme, versions are stored in separate tables as `text` fields in PostgreSQL tables. Each table has specific attributes to maintain version-specific meta–data. For example, in the adaptive scheme, these attributes include status, previous access frequency, etc. Document-specific meta–data is stored in another table. For example, in the adaptive scheme, these attributes include youth factor, correlation factor, etc. The storage of deltas is dealt with in the same way as stored versions because deltas are maintained as XML documents.

4.2 Experiments and Discussions

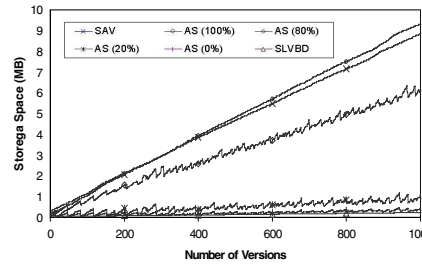
In this section, we first compare different strategies, namely SAV, SLVBD and AS. Then we show the effect of varying the refreshing interval on the performance of AS. A PC with Pentium III processor and Windows NT 4.0 operating system has been used to run the testbed.

Storage Space vs. Access Time. In the first experiment, we study the effect of changing the value of p on the average access time and storage space usage

[3] Taxonomy of XML change operations can be, e.g., found in [17,19,18]



(a) Average access time



(b) Storage space usage

Fig. 6. Tuning AS using p

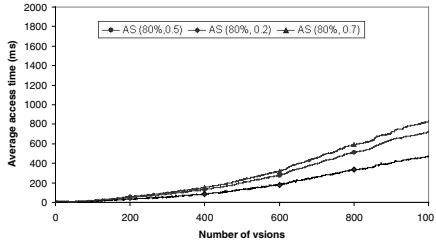
of AS. We use SAV (time optimal) and SLVBD (space optimal) as baselines for comparison. We use the following experimental settings:

- An average size of 80 K-bytes for versions
- An average size of 2 K-bytes for deltas
- A uniform access pattern on the history of versions
- λ is set to 0.5. ν is set to 50 (the average number of accesses to new versions is equal to the average number of accesses to the 10 most heavily accessed old versions, as in the example of section 3.4.2). μ is set to 1.

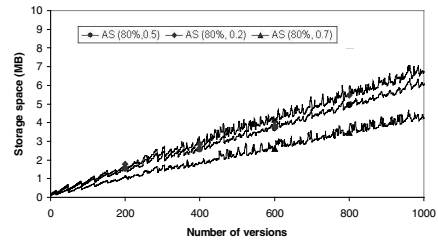
Figure 6(a) shows the average access time for the three schemes. We can see that SAV has the smallest average access time which remains almost constant (i.e., around 100 ms) even when the number of available versions increases. When p is equal to 100, the average access time for AS is close to those of SAV. The difference is due to the additional computation costs in AS (e.g., updating the meta–data). The average access time in SLVBD increases dramatically as the number of versions increases. When p is equal to 0, the access time results for AS are closer to those of SLVBD. When maintaining 20% of versions as stored versions (i.e., $p = 20$), the average access time decreases by a factor of 4.7. By maintaining more versions as stored versions, e.g., p equal to 80, the average access time is reduced.

Similarly, Fig. 6(b) shows the storage space usage. We can see that SLVBD has the least storage space usage. By choosing p equal to 0, the storage space usage of AS is slightly higher than SLVBD. This is due to the fact that more meta–data is stored in AS. It can also be seen that SAV consumes more space than the other schemes. When p is equal to 100, storage space usage results of AS are closer to those of SAV.

In the second experiment, we study the effect of changing the value of λ. We use the same settings as in the previous experiment except that p is set to 50 and λ varies. Figure 7(a) shows the resulting average access times. It confirms that small values of λ result in less access time. Similarly, Fig. 7(b) confirms that large values of λ result in less storage space usage.



(a) Average access time



(b) Storage space usage

Fig. 7. Fine–tuning AS using λ

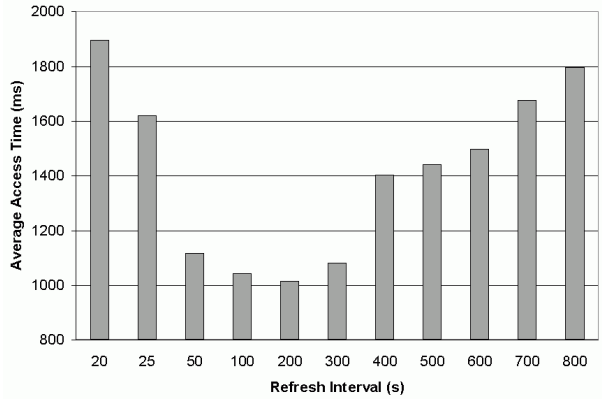


Fig. 8. Varying refreshing interval

Effect of Varying Refreshing Interval. This experiment studies the effect of changing the value of the refreshing interval on performance of AS. We use the same settings as the previous experiments except that $p = 20$ and $\lambda = 0.5$. The results are shown in Fig. 8. We can see that the length of the refreshing interval can significantly affect the average access time. When the interval value is too small, there is a performance degradation because the system does not have sufficient time to gather realistic information about usage patterns (e.g., number of accesses). On the other hand, when the interval value is too large, there is a performance penalty because changes in access patterns are not dealt with immediately. For example, an old version may remain calculated for a long time even if it should have been transformed into a stored version because of changes in its access patterns.

5 Conclusions

In summary, we have proposed an adaptive technique for continuously adjusting and improving the effectiveness of a versioning scheme taking into account both

performance and storage space requirements. We also conducted simulation experiments to gauge the behaviour of few schemes. The comparison results show clear indications of the potential of the adaptive scheme. They show the viability of the adaptive scheme in depicting the behaviour of an effective versioning scheme for given requirements (e.g., access patterns, trade–offs between access time and storage space).

References

1. Bach, S.: The Committee Markup Process in the House of Representatives. `http://www.house.gov/rules/crs_reports.htm` (1999)
2. Chien, S.Y., Tsotras, V.J., Zaniolo, C.: Efficient Management of Multiversion Documents by Object Referencing. In: Proc. of 27th Int. Conf. on Very Large Data Bases (VLDB), Rome, Italy (2001)
3. Chien, S.Y., Tsotras, V.J., Zaniolo, C.: XML Document Versioning. SIGMOD Record **30** (2001) 46–53
4. Sommerville, I., Rodden, T., Rayson, P., Kirby, A., Dix, A.: Supporting information evolution on the WWW. World Wide Web **1** (1998) 45–54
5. Douglis, F., Ball, T.: Tracking and Viewing Changes on the Web. In: Proc. of the 1996 USENEX Technical Conference, Berkekey, CA (1996) 165–176
6. Vitali, F., Durand, D.G.: Using Versioning to Support Collaboration on the WWW. World Wide Web Journal (1995)
7. Benatallah, B.: A Unified Framework for Supporting Dynamic Schema Evolution in Object Databases. In: 18th Int. Conf. on Conceptual Modeling - ER'99, Paris, France, Springer-Verlag (LNCS series) (1999)
8. Benatallah, B., Tari, Z.: Dealing with Version Pertinence to Design an Efficient Object Database Schema Evolution Mechanism. In: The IEEE Int. Database Engineering and Applications Symposium – IDEAS, Cardiff, Wales, (UK) (1998)
9. Ra, Y., Rundensteiner, E.: A Transparent Object-Oriented Schema Change Approach Using View Evolution. Technical Report CSE-TR-211-94, Dept. of EECS, Univ. of Michigan (1994)
10. Monk, S., Sommerville, I.: Schema Evolution in OODBs Using Class Versioning. SIGMOD RECORD **22** (1993)
11. Ferrandina, F., Meyer, T., Zicari, R.: Schema and Database Evolution in the O2 system. In: Proc. of 21th Int. Conf. on Very Large Data Bases (VLDB), Zurich (1995)
12. Tichy, W.F.: RCS - A System for Version Control. Software Practice and Exprience **15** (1985) 637–654
13. Rochkind, M.J.: The Source Code Control System. IEEE Transactions on Software Engineering **1** (1975) 255–265
14. Chien, S.Y., Tsotras, V.J., Zaniolo, C.: Copy-Based versus Edit-Based Version Management Schemes for Structured Documents. In: RIDE-DM'2001, Heidelberg, Germany (2001)
15. Chawathe, S.S., Abiteboul, S., Widom, J.: Representing and Querying Changes in Semistructured Data. In: Proc. of Int. Conf. on Data Engineering (ICDE). (1998) 4–13
16. Chien, S.Y., Tsotras, V.J., Zaniolo, C.: Version Management of XML Documents. In: WebDB (Informal Proceedings). (2000) 75–80

17. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.: Change-Centric Management of Versions in an XML Warehouse. In: Proc. of 27th Int. Conf. on Very Large Data Bases (VLDB). (2001) 581–590
18. Port, L.: Managing Changes in XML Documents and DTDs. Honour's thesis, Computer Science and Engineering, The University of New South Wales, Australia (2001)
19. Tatarinov, I., Ives, Z.G., Halvey, A.Y.: Updating XML. In: SIGMOD Conference. (2001)