Efficient Re-construction of Document Versions Based on Adaptive Forward and Backward Change Deltas

Raymond K. Wong   Nicole Lam

School of Computer Science & Engineering, University of New South Wales, Sydney 2052, Australia, wong@cse.unsw.edu.au

Abstract. This paper presents an efficient content-based version management system for managing XML documents. Our proposed system uses complete deltas for the logical representation of document versions. This logical representation is coupled with an efficient storage policy for version retrieval and insertion. Our storage policy includes the conditional storage of complete document versions (depending on the proportion of the document that was changed). Based on the performance measure from experiments, adaptive scheme based on non-linear regression is proposed. Furthermore, we define a mapping between forwards and backwards deltas in order to improve the performance of the system, in terms of both space and time.

1 Introduction

With the increasing popularity of storing content on the WWW and intranet in XML form, there arises the need for the control and management of this data. As this data is constantly evolving, users want to be able to query previous versions, query changes in documents, as well as to retrieve a particular document version efficiently. A possible solution to the version management of data would be to store each complete version of data in the system. Although this would maintain a history of the data stored on the system so far, the performance of such a system would be poor. This leads to the use of change detection mechanisms to identify the differences between data versions. The storage of these differences may provide an increased performance of the system, especially in relation to its space requirements.

Change detection algorithms have been proposed by [CAM02] and [WDC01]. In each case, the algorithm utilises the concept of persistent identifiers and node signatures in order to find matchings between nodes of the 2 input documents. We adopt a similar approach. An alternative solution to the change detection problem is via object referencing as suggested by [CTZ01a]. Marian et. al. [MACM01] developed a change-centric method for version management, which is similar to our approach. In [MACM01], the system stores the last version of a document and the sequence of forward completed deltas. In contrast to the approach by [MACM01], we also store intermediate complete versions of the document. A disadvantage of [MACM01] is: if we have already stored 100 versions of a document, retrieving the 3rd version would involve applying 97 deltas to the curent version - a very inefficient process. On the other hand, by storing intermediate versions, our system is likely to result in a more efficient retrieval of the 3rd version (for example, by working forward from the initial version).

In this paper, we present an adaptive selection scheme between forward and backward deltas for an efficient content-based version management system that have been previously proposed in [WL02]. The system is primarily designed for managing and querying changes on XML documents based on update logging. The proposed system uses complete deltas for the logical representation of document versions. Our storage policy includes the conditional storage of complete document versions (depending on the proportion of the document that was changed). Furthermore, we define a mapping between forwards and backwards deltas in order to improve the performance of the system, in terms of both space and time. We also adapt a set of basic edit operations, which provide the necessary semantics to describe the changes between documents, from our previous work regarding the extensions of XQL with a useful set of update operations [WON01]. Although these operations are based on XQL, since they are designed and implemented based on regular path expressions, they can easily be extended as other query languages such as XPath or XQuery. The prototype of our proposed Version Management System has been integrated with a native XML database system called SODA3 that is available at [SODA3].

2 System Model

This section defines a system model for version management. The logical model of the system consists of the representation of intermediate versions similar to the notion of Complete Deltas in the style of [MACM01]. Different from [MACM01], we here define an efficient storage policy for the document versions to reduce the storage requirements of the system. The system also maintains the time at which the document was loaded into the system in order to perform time related queries on this data.

2.1 Complete Deltas

Our proposed system uses the concept of Complete Deltas to store the different versions of a document in the database. That is, instead of storing the complete versions of all documents in the system, we chose to represent only the differences between versions to conserve storage space.

The Complete Deltas used here are representations of the differences between versions. They are termed 'Complete' as it is possible, given two versions, V_i and V_j and their Complete Delta $\Delta_{i,j}$ to reconstruct either document version. That is, given V_i and $\Delta_{i,j}$, we can reconstruct V_j; and given V_j and $\Delta_{i,j}$, we can reconstruct V_i.

2.2 Storage Policy

We define an efficient storage policy for our proposed system. Suppose there are many differences between two versions of a document, it may be more efficient to store the complete version of the more recent document, rather than storing the large complete delta. This is the intuition behind the storage policy defined.

Depending on the relative size of a complete delta, as compared to the complete document version, we either store the complete delta or the complete version. This reduces the storage requirements of the system significantly. However, due to this unconventional storage policy, there arises the need to define new query mechanisms in order to efficiently query these document versions.

2.3 Representing Time

We associate with each version of data (i.e. a complete delta or a complete version) a time value, also called a *timestamp*. This *timstamp* represents the time that the version was entered into the system. This facilitates the processing of time related queries, detailed in the next section.

2.4 Edit operations

In addition to the two basic operations: Insert and Delete, there are three more main operations supported by SODA3 [SODA3]: Update, Move and Copy. Although the Insert and Delete operations are sufficient to describe the differences between two versions, we find that the three additional operations provide a more meaningful and intuitive approach to the description of differences. Moreover, for the insert, delete, move and copy operations, it is necessary to include the element's final index as this facilitates the inversion of the operations. The operations also contain some redundant information (for example the *oldvalue* in Update operation) so as to aid in the mapping between forward and backward deltas. The detailed semantics of these operations [WL02] and the version index can be found at [LW03].

3 Main Algorithms

In this section, we consider the major parts of the system and present their key algorithms.

global:
    currentVersion ← 1

    // flag to indicate if the complete version of
    // (*currentVersion* - 1) should be stored
    storeComplete ← false

    // stores the delta between

```
// (currentVersion - 2) and (currentVersion - 1)
prevDelta ← null

// list of version numbers that have
// their complete versions stored
fullVersion ← []

// hash table or stucture to store the number of
// operations associated with each delta stored
numOp ← φ


insertNewVersion(File version):
delta ← version;
operations ← countOperations(delta);
write 'version' out to disk ;
if !storeComplete ∧ !prevDelta :
  write prevDelta out to disk;
  delete complete version of 'currentVersion';
  prevDelta ← null;
if operations > MAX_RATIO * size(version) :
  fullVersion.append(currentVersion++);
  storeComplete ← true;
else :
  numOp{currentVersion++} ← operations;
  prevDelta ← delta;
  storeComplete ← false;

getVersion(int ver):
i ← complete version closest and > ver
prev ← complete version closest and < ver
uBound ← fullVersion[i];
lBound ← fullVersion[prev];
for j ← lBound to ver do :
  lowerOps ← lowerOps + numOp{j};
for k = uBound to ver do :
  if upperOps > (lowerOps * FORWARD_CONSTANT) :
    break;
  upperOps ← upperOps + numOp{k};
if upperOps < (lowerOps * FORWARD_CONSTANT) :
  cVersion ← complete file, 'uBound';
  constructBackwards(uBound, ver, cVersion);
else :
  cVersion ← complete file, 'lBound';
  contructForwards(lBound, ver);

constructBackwards(int upper, int version, File f):
if upper != version :
  delta ← retrieve delta file, 'version';
  applyDelta(delta,f);
```

```
  constructBackwards(upper-1, version, f);
return;

constructForwards(int lower, int version, File f):
if lower != version :
  applyForwardDelta(lower, f);
  constructForwards(lower+1, version, f);
return;

applyForwardDelta(int version, File fileSoFar):
backwardDelta ← retrieve delta file, 'version';
forwardDelta ← convertToForward(backwardDelta);
applyDelta(forwardDelta, fileSoFar);
return;

applyDelta(File deltaFile, File fileSoFar):
for each e ∈ fileSoFar do:
  apply e to fileSoFar;

convertToForward(File backwardDeltaFile):
File forwardDeltaFile;
for each e ∈ backwardDeltaFile do :
  apply rules in Section 2: Edit operations
  to obtain the inverse of e;
  store the inverse operation e^1in
  reverse order in forwardDeltaFile ;
return forwardDeltaFile;
```

To insert a new version of a document into the system, we firstly process the new version - by storing it in its entirety into the system. Next, we process the previous version of the document using Eq. (1). This determines whether the backward delta of the previous version or the complete version of the document is stored.

For the retrieval of a given version, we have to iterate through the complete versions of the document stored in the system, in order to identify the version that can be used to most efficiently reconstruct the required version. This can be achieved by applying the backward deltas directly to a complete version of the document, or inverting the backward deltas and then applying the operations to the complete version.

We define a function `applyDelta` which applies the edit operations to its argument file. `convertToForward` is a function that converts each operation to its inverse.

4 Adaptive parameters

This section proposes an alternative to the adaptive parameters FORWARD_CONSTANT and MAX_RATIO in an attempt to automate the process of version management, without having the user specify the value of the respective parameters.

4.1 FORWARD_CONSTANT

We first consider the parameter FORWARD_CONSTANT. We propose another equation which takes into account the extra computation cost associated with inverting a forward complete delta to a backward complete delta.

Suppose the total number of operations that have to be performed on a complete document version stored in the system (using forward deltas) is l, while the total number of operations using backward deltas is u. The cost ($cost_l$) of using forward deltas is:

$$cost_l = T_l \tag{1}$$

where T_l represents the cost of performing all l operations on the complete version. We estimate the cost of T_l using the formula:

$$T_l = l * \frac{i+d}{2} \tag{2}$$

where i represents the cost of applying an insert operation to a document and d represents the cost of applying a delete operation to a document. Here, we assume that the time complexity for move (m), update (up) and copy (c) are such that:

$$m < d + i \tag{3}$$

$$c < i \tag{4}$$

$$up < d + i \tag{5}$$

This assumption is valid because, for example, if Eq. 3 was not true, we could replace the move operation to a delete and insert operation to improve the time complexity. Similarly for Eq. 4 and 5.

The cost ($cost_u$) of using backward deltas is:

$$cost_u = u + T_u \tag{6}$$

where T_u represents the cost of performing all u operations on the complete version. We estimate the cost of T_u using the formula:

$$T_u = u * \frac{i+d}{2} \tag{7}$$

Note that in contrast to $cost_l$, $cost_u$ includes an additional u to the cost of retrieval. This is because the forward complete deltas that are stored explicitly in the system have to be converted to their inverse (i.e. backward complete deltas). It takes constant time to invert each operation in a complete delta, hence to invert u operations, it costs u.

Hence, the final cost is

$$\frac{cost_u}{cost_l} > 1 \quad or \quad \frac{u(\frac{i+d}{2})}{l + l(\frac{i+d}{2})} > 1 \tag{8}$$

The intuition behind the above equation is: if the cost of using backward deltas is higher than the cost of using forward deltas, it is more efficient to use a forward delta to retrieve the document version.

4.2 MAX_RATIO

By analysing the adaptive parameter MAX_RATIO, we find that the main issue involves the efficient retrieval of the version being inserted. The factors that affect whether a complete delta or the complete version, V_x of a document is stored in the system include:

1. cost of executing the edit operations, $E = \frac{i+d}{2}$;
2. the size of the complete delta, $\mid \Delta_x \mid$;
3. the size of the current version being inserted, $\mid V_x \mid$;
4. the size of the previous complete version stored in the system, $\mid V_{x-1} \mid$;
5. number of operations in each complete delta stored in the system, $Ops(\Delta_i)$; and
6. total number of operations since the last complete version.

Hence, by analysing the costs associated with retrieving version, V_x, in the long run, we are able to identify a meaningful relationship between the factors listed above and whether a complete delta or document version of V_x is stored in the system.

More precisely, we divide the problem into two sections: (i) the cost of retrieving the current version using forward deltas (C_f); and (ii) the cost of retrieving the current version using backward deltas (C_b).

Hence, the cost is:

$$\frac{min(C_f, C_b)}{\mid V_x \mid} > K \quad where \quad K \in INT, K \geq 1 \tag{9}$$

This inequality represents the relationship between the cost of version retrieval and the size of the current complete document version. It indicates that if the cost of version retrieval using complete deltas is large relative to the size of the actual document, the system should store the complete version of the document rather than the complete delta. Here, K is a system-defined constant.

Cost to retrieve a version with forward deltas (C_f): The cost associated with retrieving a version using forward deltas (which are stored explicitly in the system) is mainly attributed to the total number of operations since the last complete version. Hence,

$$C_f = \Sigma_{i=m}^{x-1} Ops(\Delta_i) * E \tag{10}$$

In the above equation, m represents the previous closest complete version stored in the system.

Cost to retrieve a version with backward deltas (C_b): Given that we are currently performing the version insertion of V_x, it would be impossible to determine accurately the cost associated with retrieving V_x using backward deltas. This is because it would involve having some knowledge of the document versions that are yet to be stored in the system. Hence, the best approach to determining a cost value would be to approximate the costs associated with retrieval by predicting the the number of edit operations contained in future complete deltas (Op(Δ'_i)) to be stored in the system.
We use nonlinear regression to predict Op(Δ'_i) on the basis that V_x is not stored as a complete version. Hence we are able to identify whether a subsequent document version is stored as a complete version (using the threshold, T, presented in next section). We consider all subsequent document versions up to the version specified by the cost equation, such that using backward complete deltas to retrieve the current version is more efficient than using forward complete deltas.
Also, as backward deltas are not stored explicitly in the system, we have to consider the extra computational cost associated with converting a forward complete delta to a backward complete delta.

$$C_b = \Sigma_{i=x}^{k} Ops(\Delta'_i) + \Sigma_{i=x}^{k-1}(Ops(\Delta'_i) * E) \tag{11}$$

In the equation above, k represents the predicted version number which is the closest complete version of the document stored in the system, such that x $\leq$ k. That is, $Ops(\Delta'_i) < T, \ \forall i \in \{x..(k-1)\}$ and $Ops(\Delta'_k) \geq T$. Hence, C_b represents the predicted cost associated with retrieving V_x using backward deltas in an efficient manner.

5 Adaptivity

The adaptivity of the system is defined by a nonlinear regression model detailed in this section. This model enables the system to operate autonomously, without any user input specifying the value of MAX_RATIO. In addition, this model is adaptive in terms of being able to modify its storage plan based on the version history of the documents that are currently stored in the system. This results in a highly efficient version management system, especially for the retrieval of a given document version.

By comparing the estimated number of operations in the next delta: Op(Δ'_x) (using nonlinear regression) with a probability threshold: T, we can predict if the complete version of x will be stored in the system. These concepts will be presented in this section.

5.1 Nonlinear Regression

We use nonlinear regression to estimate the number of operations in each subsequent delta to be entered into the system.

$$Op(\Delta'_y) = (A * Op(\Delta_x) + B \mid V_x \mid)^C \quad where \quad A, B, C \geq 1 \tag{12}$$

This equation forms the model for the nonlinear regression. We observe that the number of operations in a subsequent delta is largely dependant on the number of operations in each previous delta stored in the system, together with the size of each complete version of the document stored in the system. The equation above contains variables A, B and C that vary independantly. In particular, C represents the *order* of the equation. The value of C is adjusted accordingly, based on the percentage error on a given prediction.

Error Each prediction of Op(Δ'_y) is verified when version y of the document is loaded into the system by the user. Hence, it is possible to verify the accuracy of the regression, especially with regard to the *order* variable C . Initially, we set the value of C to be 1. However, as the user loads more versions of the document into the system, the value of C adapts accordingly. This enables a more accurate equation for regression, and hence provides a more accurate prediction on the number of operations contained in the subsequent deltas and improves the efficiency of the system.

More specifically, the system allows a minimal error rate before adjusting the value of C in order to improve performance. For example, if C = 2 and regression has predicted the wrong size of subsequent deltas for the last 4 out of 5 document versions loaded into the system, we increase the value of C to 3 in an attempt to obtain a more accurate estimate for the size of a delta. Once the error rate converges to a specific range, we increment the value of C by smaller amounts, as this range is the most appropriate for regression.

5.2 Threshold

We define a threshold, T, that specifies the maximum probability to store a complete delta in the system (rather than a complete version) based on the number of edit operations in the deltas currently stored in the system.

$$T = \frac{\Sigma_{i=1}^{x-1} Pr_i * Op(\Delta_i)}{x-1} \tag{13}$$

Pr_i indicates the probability that the number of operations for a delta stored in the system is equal to $Op(\Delta_i)$. It is based on the version history of the document stored in the system.

From the equations in last section,

$$l < u * (\frac{i+d}{2+i+d}) \tag{14}$$

We use this equation to limit the number of deltas the system attempts to predict the size of, using nonlinear regression. Therefore, while the above equation is true, the system estimates the number of operations in the next delta that is to be stored in the system. This process continues until the estimated number of edit operations in a subsequent delta exceeds the threshold, T, resulting in a complete version of the document being stored in the system.

6 Conclusion

In this paper, we have addressed the problem of content-based version management of XML data. We presented a system which had an efficient logical representation and storage policy for managing changes of such data, which involved the storage of intermediate complete versions, together with complete deltas. Automatic conversion between the forward and backward deltas was also defined, which can be used to derive the complete deltas without storing both types of deltas. Finally adaptive selection between forward and backward deltas based on the justifications from the experimental performance data was presented.

References

[CAM02] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE (San Jose)*, 2002.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the International Conference on Data Engineering*, February 1998.

[CTZ01a] S-Y. Chien, V. Tsotras, and C. Zaniolo. Copy-based versus edit-based version management schemes for structured documents. In *RIDE-DM*, pages 95–102, 2001.

[CTZ01b] S-Y. Chien, V.J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *Proceedings of VLDB*, September 2001.

[LW03] N. Lam and R.K. Wong. A fast index for xml document version management. In *Proceedings of the Asia Pacific Web Conference (APWEB)*, September 2003.

[MACM01] A. Marian, S. Abiteboul, G. Cobna, and L. Mignet. Change-centric management of versions in an xml warehouse. In *Proceedings of VLDB*, September 2001.

[W3C99] W3C Recommendation. Xml path language (xpath) version 1.0. *http://www.w3.org/TR/xpath*, November 1999.

[SODA3] Soda Technologies. Soda3 xml database management system version 3.0. *URL: http://www.sodatech.com.*

[WDC01] Y. Wang, D. J. DeWitt, and J-Y. Cai. X-diff: An effective change detection algorithm for xml documents. Technical report, University of Wisconsin, 2001.

[WL02] R.K. Wong and N. Lam. Managing and querying multi-version xml data with update logging. In *Proceedings of the ACM International Symposium on on Document Engineering (DocEng)*, November 2002.

[WON01] R.K. Wong. The extended xql for querying and updating large xml databases. In *Proceedings of the ACM International Symposium on on Document Engineering (DocEng)*, November 2001.