Adaptive Change Management for Semi-structured Data

Raymond K. Wong   Nicole Lam
School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia
`wong@cse.unsw.edu.au`

Technical Report
UNSW-CSE-TR-0314
June 2003

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

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.

Traditional change detection systems which computed the differences between two documents (such as GNU diff) were based on a line-by-line comparison. This was appropriate for flat textual files but cannot be extended to handle semistructured data, such as XML documents.

Change detection algorithms have been proposed by [4] and [8]. 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 [2].

Marian et. al. [5] developed a change-centric method for version management, which is similar to our approach. In [5], the system stores the last version of a document and the sequence of forward completed deltas. In contrast to the approach by [5], we also store intermediate complete versions of the document. A disadvantage of [5] 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 [10]. 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 [9]. 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.

Finally we discuss the query mechanisms to facilitate meaningful queries to our proposed version management system, The mechanisms behind querying previous versions of data (such as a document) are greatly dependant on the logical and physical model of the version management system in question. That is, it is impossible to explore query mechanisms of a version management system without considering the context in which previous versions of data are stored in the system.

Our intended Version Management System allows for content-based change detection, which enables content-based queries in relation to changes in documents. Our system provides performance efficiency by constructing a given version from one of the intermediate complete versions rather than from the current version only, as suggested in [5]. Furthermore, we provide a mapping between backwards and forward deltas, hence reducing the space requirements for the system. The prototype of our proposed Version Management System has been integrated with a native XML database system called SODA3 that is available at [7].

The rest of the paper is organised as follows. Section 2 describes the design of our proposed system. Section 3 details the basic edit operations and version insertion and retrieval. This is followed by Section 4 containing some of the pseudo code of our system to illustrate the algorithms used. Section 5 presents the adaptive parameters that are used to tune the performance of the proposed system and Section 6 estimates its adaptivity. We then provide the performance results from experiments at Section 7 and Section 8 summarizes the query mechanisms that supports querying of changes. Finally Section 9 concludes the paper.

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 [5]. Different from [5], 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.

3 Edit Operations

3.1 Complete Deltas

This paper utilises the concept of deltas, rather than object references [3], or node annotations [1], as deltas are more intuitive. Firstly, we define a forward (backward) **delta** to be an edit script that can be applied to a complete version of a document to obtain the next (previous) complete version of a document. Such deltas are also termed 'lossy' as, given two versions V_i and V_j such that *f* is the forward delta which can be applied to V_i to obtain V_j, it is not possible to obtain V_i from V_j and *f*. Similarly for a backward delta. Hence, such deltas contain operations that are not invertable. Next, we define a **complete delta** to be an edit script such that there exists a 1 to 1 mapping between the forward and backward deltas between two versions. Although a lot of redundancy is introduced to the edit operations in a complete delta, this method is preferred as we argue that complete deltas enable our system more flexibility and efficiency.

3.2 Edit operations

In addition to the two basic operations: Insert and Delete, there are three more main operations supported by SODA3 [7]: 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.

For simplicity and clarity, we use simple path expressions, in the style of [6], to present the edit operations of our system, refer to Appendix A for the syntax. In our actual prototype implementation, we use more complicated, extended XQL expressions as described in [9].

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.

We consider the following operations:

1. **Insert:** We define three sub-operations under the Insert operation: **insertInto, insertBefore, insertAfter**.

   insertInto: `path!insertInto(`
   `insertedpath)` inserts `insertedpath` as a subtree of `path`, given that `path` has no children.

   insertBefore: `path!insertBefore(`
   `insertedpath, i)` assumes that there exists a subtree, t, rooted at `path`, such that t has nodes other than the root. This operation inserts `insertedpath` before subtree t, such that it has a final element index `i`.

   insertAfter: `path!insertAfter(`
   `insertedpath, i)` assumes that there exists a subtree, t, rooted at `path`, such that t has nodes other than the root. This operation inserts `insertedpath` after subtree t such that it has a final element index `i`.

2. **Delete:** This operation is the inverse of the Insert operations. That is, `path!delete(x, elem, isBefore)` removes the subtree `x` (which has nodes other than the root node) rooted at `path`. The subtree `x` is the raw XML that represents the subtree being deleted. Also, `elem` is the node that was adjacent to `x` before it was moved. `isBefore` identifies if `elem` is the left/right neighbour or neither.

   We also define `path!delete(elem, isBefore)` which assumes that `path` describes a leaf node and thus removes the leaf node.

3. **Update:** Intuitively, `path!update(newvalue, oldvalue)` updates the `oldvalue` of `path` to the `newvalue` specified. It is necessary to keep track of the `oldvalue` in order to invert the operation.

4. **Move:** We define three sub-operations under the Move operation: `moveInto, moveBefore, moveAfter`.

   (a) `dstpath!moveInto(srcpath, elem, boolean)`

   (b) `dstpath!moveBefore(srcpath, elem, boolean, path)`

   (c) `dstpath!moveAfter(srcpath, elem, boolean, path)`

   We also include extra parameters for these operations to aid in the mapping between backward and forward deltas. For example, in `path!moveAfter(srcpath, origNeighbour, isBefore, newpath)`, `newpath` is the final path of `srcpath` in its new location. The parameters for the other move operations are similarly defined.

5. **Copy:** We define three sub-operations under the Copy operation: `copyInto, copyBefore, copyAfter`.

   (a) `dstpath!copyInto(srcpath)`

   (b) `dstpath!copyBefore(srcpath, index)`

   (c) `dstpath!copyAfter(srcpath, index)`

We next define the 1 to 1 function that maps an operation to its inverse:

1. **Insert:** The _ in the mappings indicates that the value of that field is irrelevant (as it should contain redundant information unnecessary in the mapping).

   (a) `path!insertInto(name/restpath)`
   $\mapsto$ `path/name[0]!delete()`

   (b) `e1/n1[num1]!insertBefore(n2/`
   `restpath, index)` $\mapsto$ `e1/n2[index]!delete(_,_)`

   (c) `e1/n1[num1]!insertAfter(n2/`
   `restpath, index)` $\mapsto$ `e1/n2[index]!delete(_,_)`

2. **Delete:** To aid in the conversion from a delete operation to insert, we augment the insert operations (defined above), such that each insert operation can also take as an argument the raw XML representation of the subtree that is to be inserted (rather than a single path). To invert the delete operation on attributes, we include as a parameter to the delete operation the attribute's value.

   (a) `e1/end!delete(s, n, isBefore)`
   $\mapsto$

   ```
   if (isBefore == NULL)
      e1!insertInto(s)
   else if (isBefore)
      e1/n!insertBefore(s, _)
   else
      e1/n!insertAfter(s, _)
   ```

   (b) `e1/n1[num1]!delete(n, isBefore)`
   $\mapsto$

```
if (isBefore == NULL)
   e1!insertInto(n1)
else if (isBefore)
   e1/n!insertBefore(n1, _)
else
   e1/n!insertAfter(n1, _)
```

(c) `e1/@name!delete(text)`
$\mapsto$ `e1!insertInto(@name/text)`

3. **Update:** `path!update(oldvalue, newvalue)`
$\mapsto$ `path!update(newvalue, oldvalue)`

4. **Move:**

(a) `e1/n1[i1]!moveInto(e2/n2[i2], n, isBefore)`
$\mapsto$

```
if (isBefore == NULL)
   e2!moveInto(e1/n1[i1]/n2[i2],
     _, _)
else if (isBefore)
   e2/n!moveBefore(e1/n1[i1]/
     n2[i2], _, _, _)
else
   e2/n!moveAfter(e1/n1[i1]/
     n2[i2], _, _, _)
```

(b) `e1/n1[i1]!moveBefore(e2/n2[i2], n, isBefore, newpath)`
$\mapsto$

```
if (isBefore == NULL)
   e2!moveInto(e1/n2[i2], _, _)
else if (isBefore)
   e2/n!moveBefore(newpath,
     _, _, _)
else
   e2/n!moveAfter(newpath,
     _, _, _)
```

(c) `e1/n1[i1]!moveAfter(e2/n2[i2], n, isBefore, newpath)`
$\mapsto$

```
if (isBefore == NULL)
   e2!moveInto(e1/n2[i2], _, _)
else if (isBefore)
   e2/n!moveBefore(newpath,
     _, _, _)
else
   e2/n!moveAfter(newpath,
     _, _, _)
```

5. **Copy:**

(a) `path!copyInto(elem/name[index])`
$\mapsto$ `path/name[0]!delete()`

(b) `e1/n1[num1]!copyBefore(e2/`
`n2[num2], index)` $\mapsto$ `e1/n2[index]!delete(_, _)`

(c) `e1/n1[num1]!copyAfter(e2/`
`n2[num2], index)` $\mapsto$ `e1/n2[index]!delete(_, _)`

3.3 Insert a new version

The system stores backwards deltas (as opposed to forward deltas). This was a design decision, as the type of delta stored by the system is arbitrary. The algorithm presented below will work equally well with forward deltas stored instead, as we define a function that provides a mapping between the two. Our implementation shows that the mapping can be done in reasonably low cost, as shown in Figure 3 later.

Either a backwards delta or a complete version is stored, depending on the number of operations required to convert between the given versions . This is implemented by specifying an upper bound which depends on the size of the document. We provide a user defined parameter MAX_RATIO such that, given:

$$\frac{\#ops}{size(document)} \geq MAX\_RATIO \tag{1}$$

where *#ops* is the number of operations in the delta, and size(document) is the size of the complete document.

If Eq. (1) is true, then we store the complete version of the document, rather than the delta.

3.4 Retrieve a version

To retrieve a particular version, z, we define a method getVersion which finds the 2 complete versions (V_x and V_y) that bound version z. That is, there consist of only deltas between V_x and V_y, such that one of the deltas belong to z. The method then constructs version z by applying the forward/backward deltas, depending on the number of operations each involve. The construction of a version using backward deltas is favoured in the algorithm, as less computation is required to construct the version required - we just need to apply the stored delta step-wise on version V_y. On the other hand, to work forwards from V_x, we have to do extra computation to work out the forward deltas as they are not explicitly stored in the system. We define a constant *FORWARD_CONSTANT* such that:

$$\frac{upperOps}{lowerOps} < FORWARD\_CONSTANT \tag{2}$$

where $upperOps$ ($lowerOps$) is the number of operations required to get from the V_y (V_x) complete version to the version z.

If Eq. (2) is true, it is more efficient to use V_y to construct version z (using backward deltas).

4 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 ← $\phi$
```

```
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^1 in
    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 (using the rules defined in Section 3).

5 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.

5.1 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:

$$cost_l = T_l \tag{3}$$

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:

$$T_l = l * \frac{i+d}{2} \tag{4}$$

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:

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

$$c < i \tag{6}$$

$$up < d + i \tag{7}$$

This assumption is valid because, for example, if Eq. 5 was not true, we could replace the move operation to a delete and insert operation to improve the time complexity. Similarly for Eq. 6 and 7.

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

$$cost_u = u + T_u \tag{8}$$

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:

$$T_u = u * \frac{i+d}{2} \tag{9}$$

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

$$\frac{cost_u}{cost_l} > 1 \quad or \quad \frac{u(\frac{i+d}{2})}{l + l(\frac{i+d}{2})} > 1 \tag{10}$$

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.

5.2 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*(Δ_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:

$$\frac{min(C_f, C_b)}{\mid V_x \mid} > K \;\; where \;\; K \in INT, K \geq 1 \tag{11}$$

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,

$$C_f = \Sigma_{i=m}^{x-1} Ops(\Delta_i) * E \tag{12}$$

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.

$$C_b = \Sigma_{i=x}^{k} Ops(\Delta'_i) + \Sigma_{i=x}^{k-1} (Ops(\Delta'_i) * E) \tag{13}$$

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.

6 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.

6.1 Nonlinear Regression

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

$$Op(\Delta'_y) = (A * Op(\Delta_x) + B \mid V_x \mid)^C \;\; where \;\; A, B, C \geq 1 \tag{14}$$

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.

6.1.1 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.

6.2 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.

$$T = \frac{\Sigma_{i=1}^{x-1} Pr_i * Op(\Delta_i)}{x-1} \tag{15}$$

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,

$$l < u * (\frac{i+d}{2+i+d}) \tag{16}$$

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.

7 Experimental Results

Extensive experiments were performed for collecting the data presented in this section. XML documents (upto 512,000 nodes per document) were randomly generated with depth upto 20. The experiment was carried out on a dual pentium PC with two Pentium 850MHz processors, 512MB memory, and a 10,000RPM SCSI harddisk. Each experiment was repeated 20 times with different randomly generated documents (and the average runtime is then taken). All optional database indexes were disabled throughout the experiment. The corresponding edit operations (with valid path expressions) were randomly generated for updating the documents in the experiments.

In the case when versoning mechanism is enabled, the runtime presented (the Y-axis) in each figure represents the total time to perform 50 *check_in*'s, with each *check_in* consists of upto 50 edit operations. Depends on the path expressions being randomly generated, some of these operations may involve modifying multiple document nodes.

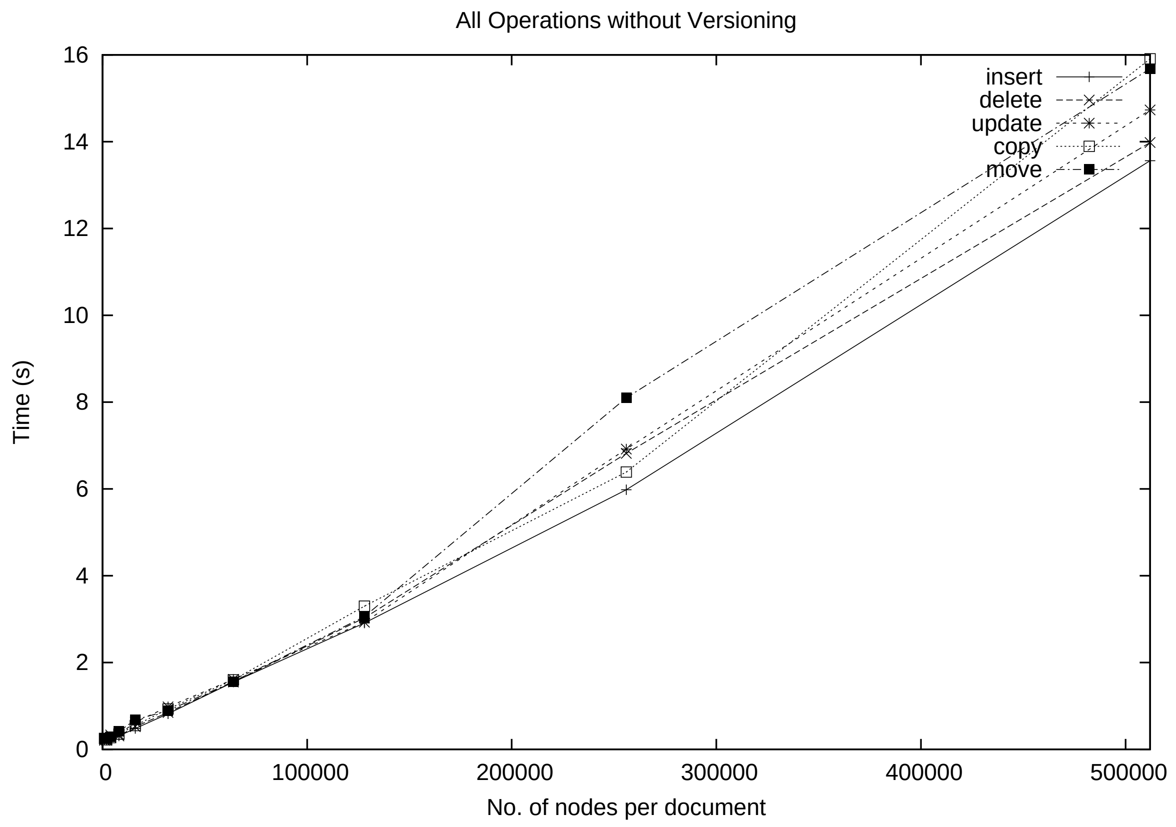

Figure 1: Costs of edit operations in the original SODA3 system

Otherwise, when versioning mechanism is disabled or unavailable (such as Figure 1), upto 2500 edit operations are simply generated and used. For example, in Figure 1, the time taken for processing 250 random edit operations on a document with 512,000 nodes is about 14 seconds.

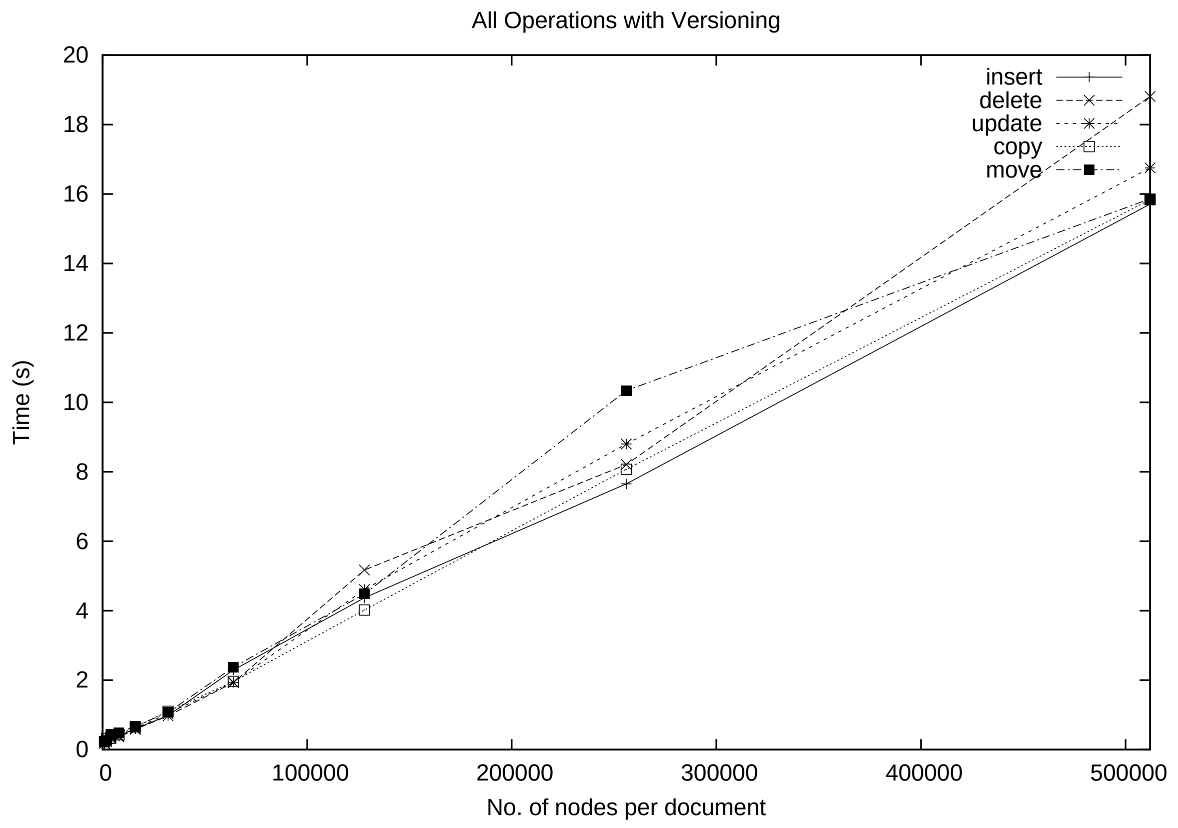

Figure 2: Costs of edit operations in SODA3 after the proposed versioning mechanism was implemented

Figures 1 and 2 show the performance of each of these edit operations in the original SODA3 system and those in the system extended with the proposed versioning mechanism, respectively. The version index proposed previously [10] has been implemented. New version can be created by typical *check_in* and *check_out* operations. These two figures show that the costs of processing each type of the edit operations (e.g., move, delete, copy) are very close. They also show that our implementation is efficient so that the overhead of having the versioning mechanism added is reasonably small.

Figure 3 shows that the computation from forward deltas to backward deltas can be done efficiently. In this experiment, the Y-axis represents the time taken for randomly retrieving 10 versions according to the settings mentioned above.

Finally, Figures 4 and 5 show the effects of varying the value of MAX_RATIO. In Figure 4, edit operations become significantly more expensive (compared to the system without versioning mechanism built-in) when the MAX_RATIO is low. This is due to the fact that every *check_in* creates a complete version that needs to be stored by the system. However, as shown in Figure 5, low MAX_RATIO will increase the retrieval performance (as fewer deltas are involved in the re-construction of a document version).

8 Querying Changes

To facilitate the processing of queries on changes, we need to be able to query the edit operations contained by each delta file. In some circumstances, we need to combine the results from the queries on the data with those from the queries on the edit operations. Due to space limitations, this section

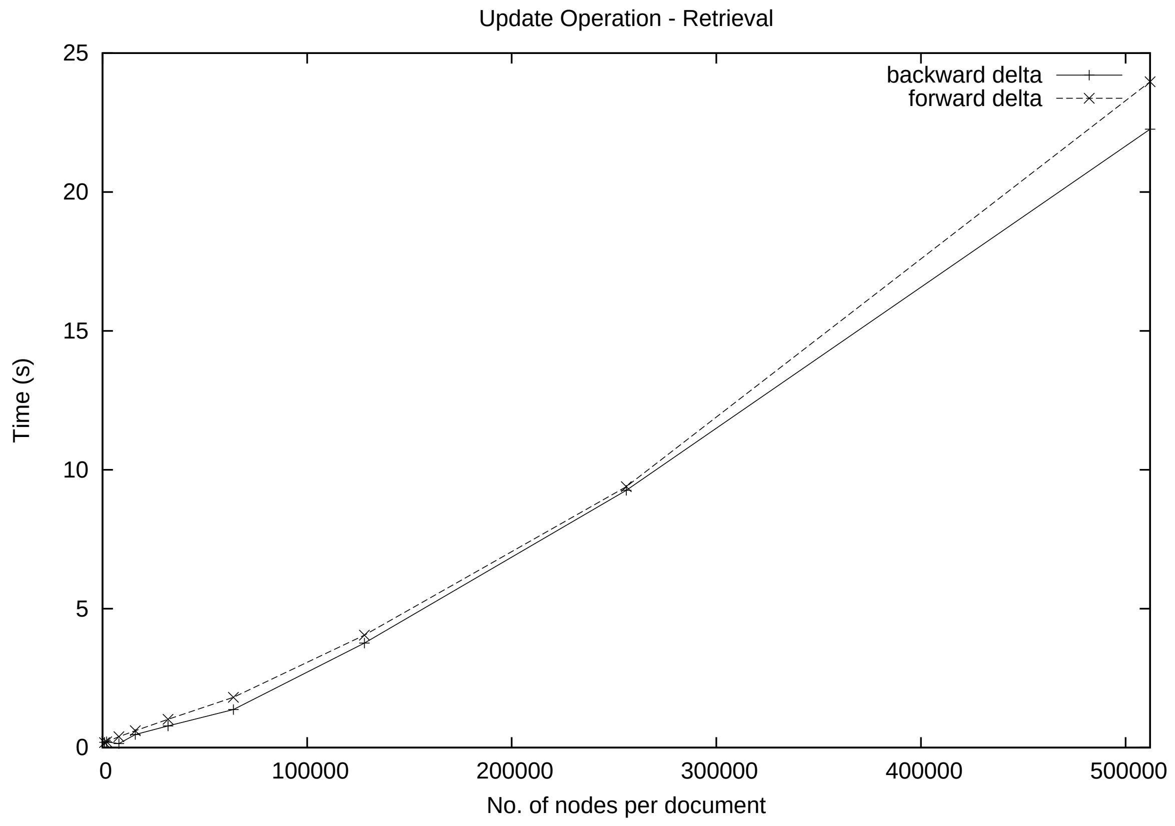

Figure 3: Cost of retrievals via Forward deltas vs that via Backward deltas

illustrates the basic idea through the following examples:

- *Find the latest version of the document just before the product with brand "cse" was increased its price by 10%.*
  ```
  v1.1:  .../product[brand=''cse'']/*/
  price!update(110, 100)
  ```
- *Find the lastest version of the document just before the product with brand "cse" was deleted from the document.*
  ```
  v1.3:  .../product[brand=''cse'']
  !delete()
  ```
- *Find the version with maximum number of changes being done to it.* We can measure this by counting the number of edit operations contained in each delta file, or alternatively, we can count the total number of XML elements being involved in all the edit operations of each delta file.

After the correct delta file is identified, the computation of the full document version takes place in either backward or forward manner (depending on which way is more efficient).

8.1 Major Query Operators

8.1.1 diff(V_i, V_j)

This operator takes as arguments two versions of a document (with i $\leq$ j) and computes the difference between the two. In the case where the two versions are actually consecutive versions (i.e. i = j),

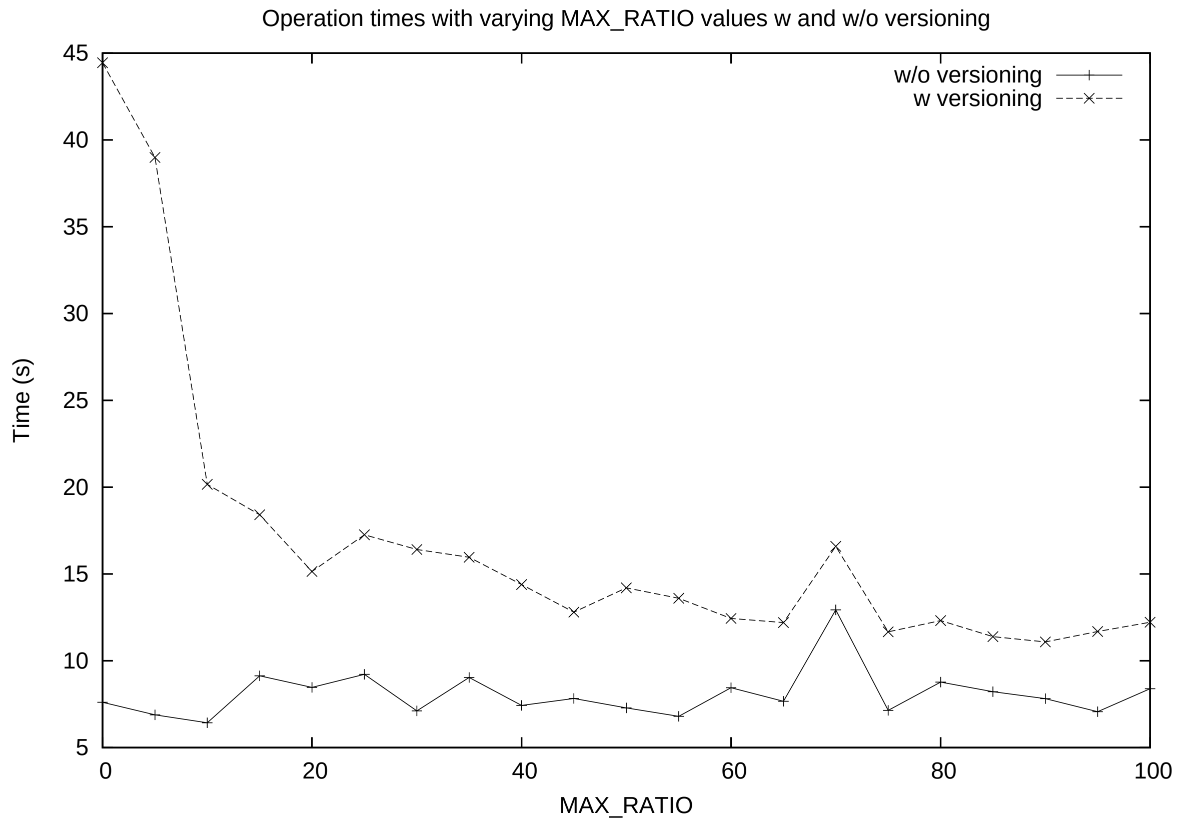

Figure 4: The cost of edit operations by varying MAX_RATIO

it is trivial to compute their differences. That is, either the differences are already stored explicitly in the form of a complete delta, or the differences are implicit (i.e. the two consecutive versions are stored as complete versions in the system and we have to apply xDiff to determine the differences).

On the other hand, if i $<$ j we have to combine several deltas and complete versions of documents together to obtain the differences between the two versions. In this case, we have to perform a merge on the differences contained in the complete deltas, while eliminating an redundant operations. To illustrate this, suppose we are merging two complete deltas Δ_{i+1} and Δ_{i+2} to obtain the differences between V_i and V_{i+3}. Let Δ_{i+1} represent the difference where `a/b/c` is inserted into the document, and Δ_{i+2} represent the difference where `a/b/c` is deleted from the document. Hence, the merged delta of Δ_{i+1} and Δ_{i+2} should not contain either operation as both are irrelevant to the differences between V_i and V_{i+3}.

8.1.2 searchPaths(*paths*)

searchPaths(*paths*) returns the most recent version of a document such that all *paths* are valid. By executing `searchPaths({'/product/name', '/item/name'})`, we can process a query to determine the most recent version of a document where '/product/name' and '/item/name' are both valid paths.

8.1.3 versionOperation(op, path)

Returns the version of document where the operation *op* was performed on *path* or an ancestor of *path*. For example, a query to retrieve the version of a docu-

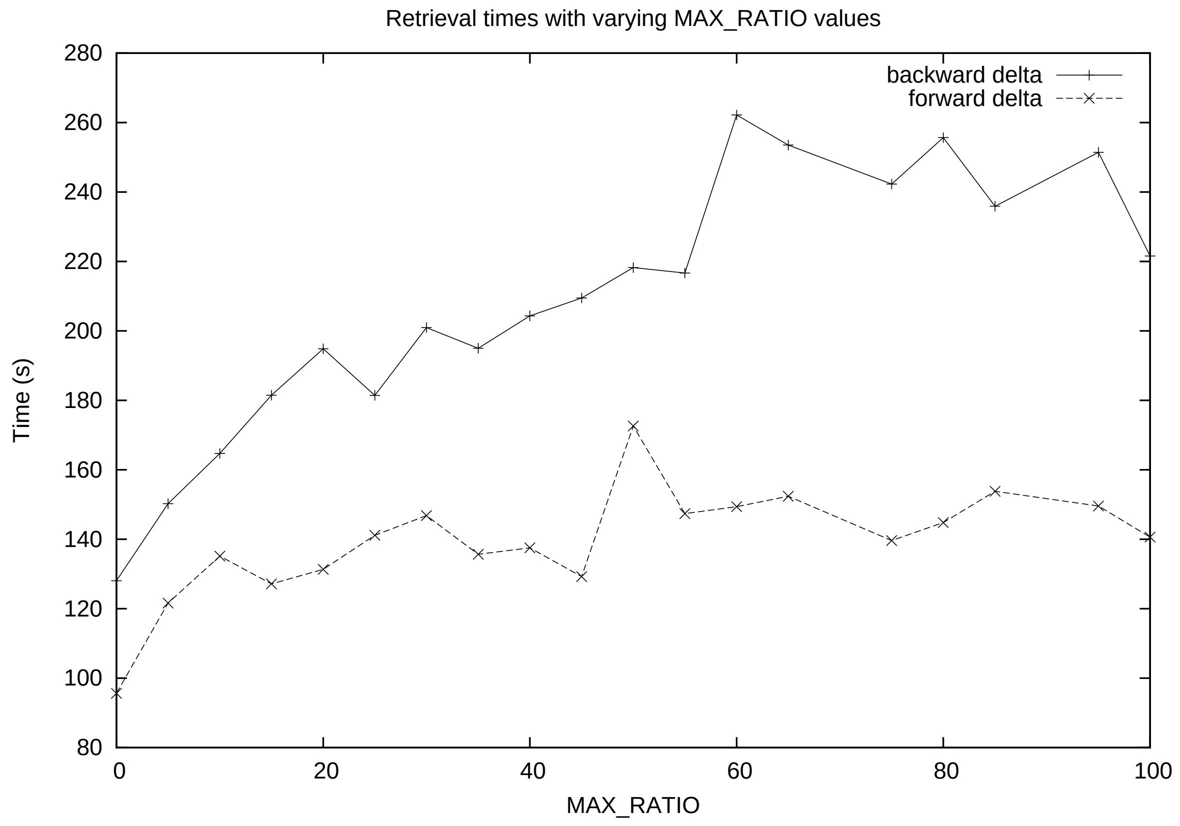

Figure 5: The cost of version retrieval by varying MAX_RATIO

ment where '/product/name' was affected by an 'insert' operation, can be processed using `versionOperation('insert','/product/name')`.

8.1.4 Other operations

We define below other operations that are necessary to efficiently process queries on changes.

completeVersion(V_i) : Obtain the complete version of a document. This may involve simply returning the complete version - if it is explicitly stored in the system, or reconstructing the version using a complete version and its complete deltas.

time(V_i) : This returns the *timestamp* associated with the document version, V_i.

history($time_i$, $time_j$) : The *history* operator returns all complete versions of a document that were loaded into the database within the specified time range. That is, it returns all complete versions, V_k, such that $time_i \leq$ time(V_k) $< time_i$.

8.2 Query Processor

We propose a model for processing queries based on the system model (using forward complete deltas) in Section 2. Note that a model using backward complete deltas can be similarly defined. To process a query based on the edit operations performed on the system, we maintain an Operation Index to keep track of all relevant operations that were performed on the document. The Operation Index is created incrementally once a query has been issued. The system starts processing the versions from the version, V_i, as indicated by the query and works backwards from V_i, adding the edit operations in each forward complete delta to the Operation Index. The system processes each complete delta sequentially and checks if the delta if the delta processed satisfies the query issued. This is done by traversing the tree structure of the Operation Index.

For example, given:

```
delta_1 = a/b/c!delete()
delta_2 = a/b!delete()
```

the system is able to answer the query: identify the version where a/b, which has a child c, was involved in a delete operation. The system will return Version 2. Note that by scanning the complete deltas independantly, the query processor would not have been able to answer this query. This is because the edit operation in delta_2 does not specify that the a/b previously had a child 'c'.

Structure of Operation Index is summarized as follows (details can be found from [10]):

1. Tree structure
2. Each node represents an element in the document which is part of the target and/or sub path of an edit operation
3. Each node contains additional information to indicate the history of operations that were performed on that node, together with the version in which the operation was executed on the document

When constructing the index, it is necessary to process the relative-location-based operations (e.g. opBefore, opAfter where op is a Move or Copy operation) differently from the other operations. This different handling is not necessary for insertBefore and insertAfter as new data is inserted into the database and the data is regarded as new elements. On the other hand, the relative location based operations mentioned above require special handling so as to keep track of the edit operations being performed on the same element, despite it's location. To handle these operations, we simply apply the inverse of the operation to the Operation Index, hence maintaining the edit history of each element.

In order to determine if the Operation Index created matches the user's query, we apply the query directly onto the Operation Index. The query takes into account the history of operations contained in each node of the Index and matches the query path to the nodes in the Index. If the processor locates a node that matches the query, the version corresponding to the matching is returned.

The above method is applicable for handling Simple Path Expressions (see Appendix A for details). To handle complete deltas which have XPath expressions as a target and/or sub path of an edit operation, we expand the XPath expression with respect to the DTD or schema provided and insert these expanded paths into the Operation Index.

8.3 Algorithm

This subsection presents the algorithm that corresponds to the query processor above.

```
   // returns the most recent version of data
   // (up to 'version') that satisfies the query
processQuery(query, version):
 1 paths = {query}
 2 for each $v \in$ [version..$V_1$] :
 3   // process editOp in reverse order
 4   for each editOp $\in$ $delta_v$ do:
 5     if overlap(editOp, paths):
        if editOp = opAfter or editOp = opBefore:
          e = invert editOp
          T = apply(e, T.root(), 0, paths)
        else :
          T = apply(editOp,T.root(), 0, paths)
  ans = T(query)
  if ans ≠ { } :
    return ans.version
```

```
apply(editOp, node, level, paths):
if node == { }:
  return createNode(editOp, level);
if node.element == editOp.elementAt(level):
  if node.childs() == { } :
    node.addHistory(editOp.operation,
      editOp.version)
  else:
    for each n ∈ node.childs() :
      child = apply(editOp, n, level+1, paths)
      if child ≠ n :
        node.insertChild(child)
  update paths to include sub path of
    move/copy operations
  return node
else:
  n = createNode(editOp, level)
  return n
```

```
createNode(editOp, level):
for each l ∈ [level..editOp.pathLength]:
  n = NEW-NODE();
  prev.insertChild(child)
  prev = n;
n.history = {(editOp.operation, editOp.version)}
```

9 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. Adaptive selection between forward and backward deltas based on the justifications from the experimental performance data were presented. Finally we summarized the query operators and the query processor for facilitating meaningful queries on data changes.

References

[1] 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.

[2] 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.

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

Appendix A: Syntax of Simple Path Expressions

```
path          ::=  elem/end
end           ::=  @name | @name/"text" |
                   name[num]/"text"
elem          ::=  elem/elem | name[num]
num           ::=  [0-9]+
name          ::=  [a-zA-Z0-9:._-]+
text          ::=  [^<>&"']*
srcpath       ::=  path
dstpath       ::=  path
newpath       ::=  path
newvalue      ::=  "text"
oldvalue      ::=  "text"
insertedpath  ::=  elem2/end2
elem2         ::=  elem2/elem2 | name
end2          ::=  name | @name/"text" |
                   name/"text"
```