Managing and Querying Multi-Version XML Data with Update Logging

Raymond K. Wong Nicole Lam
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{wong, s2244316}@cse.unsw.edu.au

ABSTRACT

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. This paper proposes a version management system for XML data that can manage and query changes in an effective and meaningful manner.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms

Algorithms, Documentation, Performance

Keywords

Path Expression, Versioning, XML

1. INTRODUCTION

As an increasing amount of documents are being created in XML form, authors require the ability to manage their content through version control. Content authors require flexibility and efficiency in maintaining their content, as well as the ability to query previous versions of the author's content.

Consider the situation where content authors are continuously modifying their documents and "checking-in" these documents into the database. The database system has to provide the necessary facilities for storing and querying the current version of the document, together with previous versions of the document. This presents the problem of version control.

Several solutions to this problem have been proposed by [2, 1, 4]. [3] utilise the concept of object referencing and identities to solve this problem.

In this paper, we present a Content-Based Version Management System which solves the problem of version control efficiently. Our proposed system has an efficient storage policy which stores intermediate complete versions of a document, as well as delta files of the document. By storing intermediate complete versions of documents, we are able to improve the space complexity, as well as the efficiency of the system. This is possible because the system can construct a given version from one of the intermediate complete versions rather than from the current version only, as suggested in [9]. Our concept of complete deltas also promotes efficiency as we are able to provide a mapping between backwards and forward deltas, hence reducing the space requirements for storing deltas.

Although several approaches to version control of semistructured data use the concept of object referencing, our proposed system uses complete deltas instead. This was done as we argue that deltas are more intuitive than object referencing. That is, the deltas we define in our system can be directly applied to complete versions, as edit scripts, in order to obtain the previous version.

The rest of the paper is organized as follows. The next section details works that are related to the area of version control/mangement. Section 3 provides an overview of the logical model of the proposed system. This is followed by a section on the querying facilities of the system, together with algorithms for query processing.

2. RELATED WORKS

The development of version management systems for semistructured data is closely related to the more general area of document management systems, such as [6] and [8]. The development of such commercial products illustrates that users need to be have access to a system that allows them to control and query their structured/unstructured data.

Previous solutions to the general topic of version control of documents were based on region-based diff operations, such as GNU diff. However, this approach is not suitable for Semistructured Data (e.g. XML documents) due to the rich semantics of the data. For example, if the difference between version 1 and version 2 of a document is the swap of two 20-line paragraphs, traditional region based comparison systems would state the deletions of two paragraphs and the insertions of two new paragraphs. Hence, there is a need for a more meaningful comparison of documents based on the semantics of the update operations.

More recently, [4, 9] have discussed the use of content-based version management. Identifying changes based on content result in more meaningful and intuitive results that are more efficient to query.

The Reference-Based Version Model (RBVM) proposed by [4] uses object references to manage multiversion documents. Chien et. al. focus on the storage performance of the system, while the performance of inserting a new version into the system was not

DocEng'02, November 8–9, 2002, McLean, Virginia, USA.

discussed. Our work differs from RBVM in that we use the notion of complete deltas and edit scripts, rather than object references. We believe that this approach is more intuitive as the deltas (i.e. the edit scripts) can be applied directly to a complete version.

[9] propose a system that keeps track of changes using complete deltas. We use a similar concept of completed deltas and the storing of the last version of a document. However, in contrast to [9], we also store intermediate complete versions of the document.

3. LOGICAL MODEL

The logical model of the version management system detailed in this section forms the basis of our proposal. Firstly, we look at the complete deltas used, the change detection algorithm, followed by the types of edit operations that the complete deltas contain. Next, several user operations for version insertion and retrieval are defined. A performance analysis of version insertion and retrieval can be found in [7].

3.1 Deltas

Definition: A **delta** represents the difference between two data versions. A delta can be classified as either a forward or backward delta. 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.

Example: Given Version 1 and 2 below, $e_{1,2}$ = `section/subsection!insertInto(heading/"Title")`, which is the edit operation to convert Version 1 to Version 2. The inverse of the insertInto operation corresponds to the complete delta file $e_{2,1}$ = `section/subsection/heading!delete()`.

<section>
<subsection>
</subsection>
</section>

Version 1

<section>
<subsection>
<heading>Title</heading>
</subsection>
</section>

Version 2

$e_{1,2}$ represents an edit operation in a forward delta while $e_{2,1}$ represents an edit operation in a backward delta. Note that there is a loss of information in the delete operation, $e_{2,1}$, as the operation does not specify the contents of the node that is being deleted. Hence, deltas are also referred to as 'lossy' deltas. This loss of information is significant for reconstructing a particular data version and will be elaborated on in the following sections.

3.2 Complete Deltas

One of the main components of our proposed system is the use of complete deltas for change detection. Several other change detection algorithms [1, 2, 3, 5, 12] have been proposed. However, we utilise the concept of deltas, rather than object references [4], or node annotations [1], as deltas are more intuitive. That is, the deltas we define in our system can be directly applied to complete versions, as edit scripts, in order to obtain the previous version.

We define a complete delta to be an edit script such that there exists a 1 - 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. Complete deltas allow the system to work backwards and forwards from any complete version stored. Efficiency in terms of storage space of the deltas is achieved as complete deltas only require the storage of a forward or backward complete delta and not both. This method is preferred rather than storing both the backward and forward (lossy) deltas in the system.

3.3 Change detection

The user can load a new version of data in one of two ways:

1. providing the list of edit operations (in the form of a complete delta) that were performed on the data to obtain the new/current version (using *update logging*); or

2. providing the new/current version and allowing the system process the changes that were performed on the existing version stored in the system.

The mechanisms behind constructing a complete delta, given the new/current version of data (as specified in Method 2) is defined in the xDiff algorithm, a modified version of Wang et. al.'s X-Diff [12].

```
xDiff(V_x, V_y):
1  parse V_x and V_y
2  find minimum cost matching between V_x and V_y
3  generate minimum-cost edit script
```

We find the minimum cost matching by matching each node in V_x to each node in V_y to identify the common nodes between V_x and V_y. Next the minimum-cost edit script is generated consisting of edit operations which updates nodes that are in the minimum cost matching that have different values, deletes nodes in V_x not in the minimum cost matching and inserts nodes in V_y not in the minimum cost matching.

In contrast to X-Diff, xDiff computes the delta between 2 input versions based on the ordered tree model (X-Diff detects changes using the unordered model). We find xDiff to be more meaningful especially in the context of textual documents, where the order of paragraphs and sections are important.

3.4 Edit operations

In addition to the two basic operations: Insert and Delete, we define three more main operations: 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. Furthermore, we found that including the Copy operation enabled more flexibility in representing document restructuring. We use simple path expressions, in the style of [10], to present the edit operations of our system, see Appendix A for the syntax.

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. See [7] for details on converting an edit operation to its inverse.

We consider the following operations:

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

   insertInto: Intuitively, `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. This redundancy is necessary, so as to be able to invert the delete operation to an insert. Also, `elem` is the node that was adjacent to `x` before it was moved. `isBefore` is a boolean which 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`.

   - dstpath!moveInto(srcpath, elem, boolean)
   - dstpath!moveBefore(srcpath, elem, boolean, path)
   - dstpath!moveAfter(srcpath, elem, boolean, path)

   The intuition for the three move operations is similar to those of the insert operation and will not be discussed here. However, 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`.

   - dstpath!copyInto(srcpath)
   - dstpath!copyBefore(srcpath, index)
   - dstpath!copyAfter(srcpath, index)

   The intuition for the three copy operations is similar to those of the insert operation and will not be discussed here.

3.5 Version Insertion

To insert a new version into the system, we use the xDiff algorithm. The system stores forward deltas (as opposed to backward 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 backward deltas stored instead. That is, the data model presented here can work with either forward or backward deltas as we define a function that provides a mapping between the two. However, we found that storing forward complete deltas in the system facilitates in the processing of user queries, as described in the following chapters.

A complete version is stored, V_x, depending on the number of operations required to retrieve version *x* from the most recent complete version. This is implemented by specifying an upper bound for the total number of edit operations consecutive deltas can contain. This upper bound determines if a complete version of the document is stored. As an enhancement, the value of this upper bound actually depends on the size of the document. The intuition behind this is: intermediate complete versions are stored in the system to reduce the number of operations that have to be applied to a complete version in order to reconstruct a given version, while maintaining the space efficiency of the system. We vary this value using the user defined parameter MAX_RATIO such that it depends on the size of the document too.

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

where *#ops* is the total number of operations that have to be applied to the most recent complete version stored in the system to retrieve the given version, and size(document) is the size of the complete document.

If Eq. 1 is true, then we store the complete version of the document, in addition to the delta.

```
insertNewVersion(V_x):
1  // Step 1:  Compute the difference
2  // between V_x and current version
3  delta = xDiff(V_x, current version)
4  Save complete version V_x
5  Save delta
6  // Step 2:  Process previous version
7  if Eq.  (1) is not satisfied :
8    Delete complete version (x-1)
```

3.6 Version Retrieval

To retrieve a particular version, z, we locate 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 forward 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 backwards from V_x, we have to do extra computation to work out the backward deltas as they are not explicitly stored in the system. We define a constant *FORWARD_CONSTANT* such that:

$$\frac{upperOps}{lowerOps} < FORWARD\_CONSTANT \quad (2)$$

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).

```
getVersion(ver):
 1 // Locate closest complete version stored before ver
 2 for each complete version, v, stored in the system:
 3   if v = ver:
 4     return complete version v
 5   else if v = closest complete version stored:
 6     break
 7 lowerOps = # of ops to compute ver from v
 8 upperOps = # of ops to compute ver from (v+1)
 9 if Eq.  2 is satisfied:
10   convert forward deltas [v..ver] to backward deltas
11     and apply to V_v
12 else:
13   apply forward deltas [ver..v] to V_v
```

```
convertToForward(backwardDelta):
1  for each edit operation in backwardDelta:
2    apply rules to obtain the inverse
3    store the inverse operation
4    in reverse order in forwardDelta
5  return forwardDelta
```

It is necessary to store the inverse operations in reverse order in backwardDelta. This concept can be illustrated by the following example:

Given versions V_i, V_j and V_k, where xDiff(V_i, V_j) = e_i and xDiff(V_j, V_k) = e_j (e_n is a single edit operation). To get from V_k to V_i, it is necessary to apply edit operation e_j first, then e_i to V_k. Hence, the edit operations are applied in reverse order.

4. QUERYING MULTI-VERSION DOCUMENTS

This section deals with the design of the query language used to perform queries on the data stored in the system. It also details the design of the query mechanism to process queries based on the logical model described in the preceeding section.

4.1 Query Language

We use XQuery [11] to allow the querying of multi-version documents. XQuery, which contains XPath as a subset, is a query language for XML data proposed by W3C. XQuery is a powerful query language that supports conditional statements, iterative loops, collections, set operations, user-defined functions etc. These features make XQuery a suitable choice for our query language.

The FLWR (FOR/LET) expression forms the basis of XQuery. Informally, the syntax of a FLWR expression is: FOR ... LET ... WHERE ... RETURN ... The FOR clause allows iterations over sets of elements, while the LET clause assigns a collection of elements to a variable. The WHERE clause allows conditions to be specified and the RETURN clause provides the formatting for the output to the user.

To define a function *funcName* in XQuery that takes in *parameterList* as arguments:

```
DEFINE FUNCTION funcName ( parameterList )
   RETURNS elementList {
     ...  XQuery expression ...
   }
```

The user can then invoke *funcName* in the body of an XQuery expression.

Example:

```
<catalog>
   <book>
     <title>ABC book</title>
     <year>2002</year>
     <publisher>XYZ publishing</publisher>
   </book>
   <book>
   ...
   </book>
   ...
</catalog>
```

Given the XML document above, the user can perform the following queries:

- Retrieve all book publishers which have published a book before year 2000. Here, the FOR clause iterates over each node resulting from the expansion of the path `//book` and checks that each book was published before year 2000.

  ```
  FOR $i IN //book
  WHERE $i/year < 2000
  RETURN $i/publisher
  ```

- Retrieve all the books published by each publisher, categorised by publishers.

  In particular, the user can define a function *bookPublisher* which takes the *publisher* element as an argument and returns the books associated with the *publisher* in the correct format. In the FLWR expression, the LET clause assigns the collection of nodes from the expansion of the path `//book` to the variable `$bks`. Next, the FOR clause iterates this collection of books to obtain a publisher. Note that `$p` contains a single element, while `$bks` contains a collection of elements.

  ```
  DEFINE FUNCTION bookPublisher(ELEMENT publisher $pub)
     RETURNS ELEMENT book_publisher {
       <book_publisher name=$pub>
       {
         FOR $b IN //book
         WHERE $b/publisher = $pub
         RETURN
           <book>
             <title>$b/title</title>
             <year>$b/year</year>
           </book>
       }
       </book_publisher>
     }

  LET $bks := //book
  FOR $p IN $bks/publisher
  RETURN bookPublisher($p)
  ```

We also define a query function `edit-operation()` in the style of XPath functions. As with other standard XPath functions, `edit-operation()` is executed based on the context node. That is, `edit-operation()` is performed on the most recently matched element node in the data graph. This query function returns the edit operation associated with the node (as specified by the path expression). This function provides flexibility in specifying queries in the body of a regular path expression which contains a predicate.

We define other functions for querying data versions:

HISTORY(doc, path): Returns the history of edit operations that were performed on the node specified by *path*.

DOC-HISTORY(doc, V_i, V_j): Returns a list of edit operations representing the changes between document versions *i* and *j*.

DOC-HISTORY(doc, $timestamp_i$, $timestamp_j$): Returns a list of edit operations representing the changes between time $timestamp_i$ and $timestamp_j$.

CURRENT(doc): Returns the most current version of *doc*.

VERSION-OP(doc, path, op): Returns the version of *doc* such that the operation *op* was performed on *path* or an ancestor of the node *path*.

We can define a majority of the functions listed above as user-defined functions in XQuery. In the design of the query mechanism for our proposed system, it is necessary to provide a more concrete description of the logical model of our system - in terms of the actual complete deltas that are stored in the system. There are two factors that can be included in a query that affect the method of processing the query. These are: (i) change-related or content-related queries; and (ii) queries over a range of versions or a particular version.

Firstly, we discuss our approach to query processing, together with a detailed description of the indexes maintained by the system for query processing. Finally, we explore the different types of queries that can be performed on the system.

4.2 Approach

A naive approach to processing a user query based in the changes or content of a given document version, v, would involve reconstructing version v by applying a sequence of complete deltas to the closest complete version of the document stored in the system. This is very inefficient because if a user specifies a range of versions to query, a range of complete versions of the document will have to be reconstructed, while only a few versions may match the user's query and are returned to the user.

In our approach to query processing, the system scans the deltas stored in the system and returns the corresponding version(s) that matches the query without having to reconstruct an intermediate versions. This is possible through an index structure, which stores all the deltas in a single structure indexed on the tags that were involved in an update operation. With this approach, the system only retrieves the complete version of a document that satisfies the query.

4.3 Index Structure

The basic intuition behind our index structures is the indexing of tags that are involved in an edit operation. We present the two index structures maintained by the system to facilitate query processing: Tag Index and Path Index.

For illustration purposes, we provide an example of a sample state of our version management system.

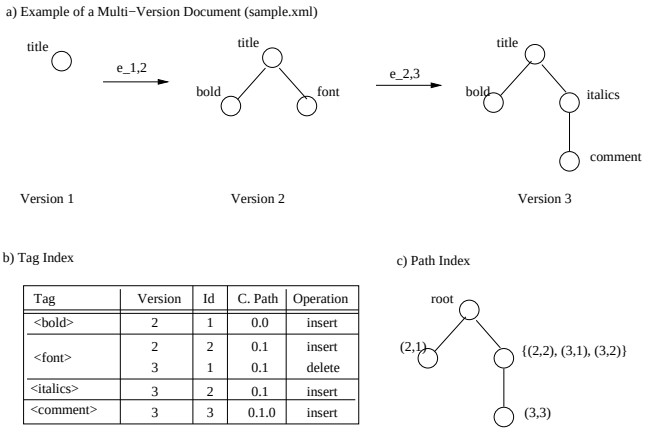
Example 1: The Figure 2 shows 3 versions of a document. $e_{1,2}$ represents the edit operation that was performed on Version 1 to obtain Version 2, while $e_{2,3}$ represents the edit operation to obtain Version 3 from Version 2.

where $e_{1,2}$ = title!insertInto(bold); title/bold!insertAfter(font) and $e_{2,3}$ = title/font!delete(); title/bold!insertAfter(italics); title/italics!insertInto(comment);

4.3.1 Tag Index

The system maintains a hash table, Tag Index, which maps a tag to a list of edit operations. With this index structure, the query processor is able to access the required edit operation associated with a given tag in near constant time. This is important as a majority of user specified queries are based on a specific tag (and can hence be easily obtained from the hash table).

Each row in the Tag Index represents a tag that was involved in an edit operation in the given document's history. Associated with each tag is a list of nodes that represent the particular nodes that



Tag	Version	Id	C. Path	Operation
<bold>	2	1	0.0	insert
<font>	2	2	0.1	insert
	3	1	0.1	delete
<italics>	3	2	0.1	insert
<comment>	3	3	0.1.0	insert

Figure 1: Sample State of Version Management System

participated in the edit operation. More specifically, each node in the list of nodes contains the following variables:

Node ID: Node identifier. This identifier uniquely identifies a particular node within a given version. Hence, the Node ID, together with the Version (i.e. (Version number, Node ID)) uniquely identifies each node in the Tag Index. The Node ID also implicitly orders the edit operations that are performed on a document version.

Version: The document version that corresponds to the edit operation executed.

Canonical path: The canonical path of a node is a unique representation of a node in terms of its order with respect to its siblings.

Definition: Given an XML document, represented as a data graph, the **Canonical Path Representation**, *C*, of a node consists of a sequence of numbers separated by '.' of the form: $C = x_1.x_2.x_3.....x_k$ where $x_i \in NAT$.

The Canonical Path is defined recursively as follows:

1. $x_1 = 0$, as each XML document has exactly 1 root node; and
2. the *i*th number in the sequence corresponds to the x_i-th child of the node represented by $x_1.x_2.x_3.....x_{i-1}$ at level *i* of the data graph.

Note that *C* corresponds to the XPath expression $*[x_1]/*[x_2]/*[x_3]/.../*[x_k]$.

Example 2: The XPath representation for the nodes for '<title>', '<bold>' and '<italics>' in Example 1 are *[0], *[0]/*[0] and *[0]/*[1] respectively. Hence, the Canonical paths are 0, 0.0 and 0.1. The canonical path represents the final position of the node after the operation was executed on the document.

Operation: The operation that was performed on the node. Possible operations include: insert, delete, updateOld, updateNew, moveSource, moveDest, copySource and copyDest. Note that it is not neccessary to include *op*Before/After as an operation type because the canonical path of the node is the final location of the node after the operation has been executed. We also keep track of the old tag value of an update operation via the operation *updateOld*. In addition, we include the

*op*Source and *op*Dest for *move* and *copy* so as to maintain information on the previous location of the node.

It is important to note that the Canonical path representation of a node containing either a *delete* or *moveSource* operation is the original location of the node before the operation was executed. Hence, for a node with Version: *x* and Operation: *delete* or *moveSource*, its Canonical path identifies the node in Version (x-1). Note also that the node list associated with each tag is ordered based on the Version number followed by the Node ID. This enables efficient evaluation of user queries.

In our Tag Index, we store the *Canonical path* of a node. This path value is derived depending on the method that the data version is loaded into the system. If the user loads the complete modified data version into the system, we user xDiff to detect the modifications that were performed on the data. In this case, the xDiff algorithm returns the *Canonical path* representations of the nodes that were involved in edit operations. On the other hand, if the user loads a list of edit operations into the system to represent the modifications to the data version, the edit operations may contain complex path expressions which cannot be immediately converted into their corresponding *Canonical path*. Hence, the system has to perform the provided list of edit operations onto the existing data version to determine the specific node in question and traverse the data graph to obtain the *Canonical path* of the node. This is best illustrated with an example.

Example 3: Given the data version, *V*:

```
<a>
   <b>
     <c></c>
     <c></c>
   </b>
</a>
```

and the edit operation performed on this version to obtain a new, modified version:

```
e_1 = a//c!insertInto(d)
```

The user loads the new version of the data by entering the edit operation, e_1, into the system. When the system performs e_1 onto the exisiting data version, *V*, it identifies `a/b/c[0]` and `a/b/c[1]` as the nodes that are affected by the edit operation. Hence, the *Canonical path* of the inserted node corresponding to `a//c!insertInto(d)` are `0.0.0.0` and `0.0.1.0`.

By applying the edit operations onto the existing document versions, we are able to obtain the *Canonical path*s of all edited nodes. This is applicable to all Regular Path Expressions.

4.3.2 Path Index

In addition to the Tag Index, we also present an index structure, Path Index, to maintain the relationships between the paths of the edit operations that are performed on the data version. This enables the efficient processing of complex queries on the changes made to the data.

The Path Index is a tree consisting of path index nodes. Each index node contains a list of (Version, Node ID) pairs (uniquely identifying each node in the Tag Index). The parent-child relationship between the path index nodes illustrates the parent-child relationship between the nodes within the path index node (i.e. A node in the parent path index node is actually the parent of a node in the child path index node). Each path index node can be identified by the *Canonical Path* of the nodes that it represents. Hence, each node (in the node list of the Tag Index) also holds a pointer to the path index node of the Path Index, corresponding to the path index node that it is in.

4.4 Change-Related Queries

4.4.1 Queries spanning a single version:

This is the most basic type of query that can be performed by the user on the system. These queries involve the changes that were performed on a single document version. That is, these queries involve querying on one or more edit operations executed on a single document version.

Example 4: From the running example, the user can issue a query to locate the version of document where the '<font>' tag was deleted and the '<italics>' tag was inserted into the document. The system would return Version 3. This is executed by looking-up the index structure for the '<font>' and '<italics>' tag. The node list associated with '<font>' contains the node (3,1) which was involved in a delete operation, while the node list for '<italics>' contains the node (3,2) for an insert operation. Hence, as both matched nodes are associated with Version 3, the system returns Version 3 of the document to the user.

Query:

```
FOR $i IN /
WHERE edit-operation($i//font) == DELETE AND
   edit-operation($i//italics) == INSERT
RETURN version($i);
```

4.4.2 Queries spanning multiple versions:

For queries spanning multiple versions, it is necessary to combine the query over several deltas in order to keep track of the modifications that the user may have performed on different document versions.

Example 5: From the running example, the system returns Version 3, when the user issues a query to locate the version of document that has both the '<bold>' and '<italics>' tag inserted into the document. To execute this query, the query processor performs a lookup in the Tag Index for the '<bold>' and '<italics>' tag, obtaining their corresponding node lists. By identifying the nodes which were involved in the required (insert) operation (i.e. (2,1) for '<bold>' and (3,2) for '<italics>'), the processor next scans both node lists to ensure that either tag was not deleted or updated within Versions 2 and 3. Finally, the processor returns Version 3 to the user, in response to the query.

Query:

```
FOR $i IN /
WHERE edit-operation($i//bold) == INSERT AND
   edit-operation($i//italics) == INSERT
RETURN version($i);
```

4.4.3 Content-Related Queries

Content-related queries have to be handled differently from change- related queries as the particular content of a document may not have been involved in an update operation. An example of a content-related query: *Locate the document version which contains that phrase "hello world"*.

We augment the Tag Index structure in order to be able to handle content-related queries. We combine the Inverted File Index together with the Tag Index to obtain the Augmented Tag Index. Here, each row contains either a tag or a word located in the data version. If the row contains a word,

Example 6: From Example 1, suppose the user performed the operation `title/bold!insertAfter(font/"hello world")` instead of `title/bold!insertAfter(font)` in $e_{1,2}$. The Augmented Tag Index would look like Figure 2. For "hello", the Order = 1 indicating that "hello" is the first word in document version 2. The node information associated with "hello" and "world" are the same as for <font> (2,2). This was done to

Tag	Version	Id	Order	C. Path	Operation
...	...	...	...	...	...
hello	2	2	1	0.1	insert
world	2	2	2	0.1	insert

Figure 2: Augmented Tag Index (from Example 1)

keep track of the operations performed on the words in the document.

By including the content (in terms of the words) of the data version in the Tag Index, the query processor is able to perform change-related queries on both the tags as well as the content of the document.

4.5 Algorithms

4.5.1 Insertion

We provide the algorithm for inserting an edit path (or Canonical path of the node) into the Path Index. Here, *P* is the Path Index currently being constructed. As a user performs an edit operation on the data version, the edit operation is inserted into P using `INSERT()`. `INSERT-PATH` locates the appropriate location that the *path* should be located and inserts *pid* into the Path Index. It determines the correct location by performing a `PREFIX` comparison on the current index node and the argument *path.* If an index node to represent path does not exist, an new index node is created using `CREATE-NODE`.

```
// pid = (ver, id)
// 'path' is the Canonical path representation of the node
// that was affected by a user operation.
INSERT(pid, path):
1  P.root ← INSERT-PATH(pid, path, P.root());

INSERT-PATH(pid, path, node) :
 1 if node = {} then
 2    return CREATE-NODE(pid, path);
 3 if path == node.path then
 4    node.append(pid);
 5    return node;
 6 else if PREFIX(path, node.path) then
 7    n ← CREATE-NODE(pid, path);
 8    update children of node.path
 9    return n;
10 else if PREFIX(node.path, path) then
11    c ← child node of 'node' that is a prefix of path
12    if (c == null) then
13       n ← CREATE-NODE(pid,path);
14       node.children.append(n);
15    else
16       INSERT-PATH(pid, path, c);
17    return node;
18 else
19    return CREATE-NODE(pid,path);

CREATE-NODE(pid, path) :
1  n ← INDEX-NODE();
2  n.path ← path;
3  n.nodes ← pid;
4  n.children ← [];
5  return n;
```

4.5.2 Change-Related Queries

When the user specifies a change-related query, the query processor executes `PROCESS-QUERY` with the regular path expression and edit operation the user is interested in as arguments. We define a function `NODE-NAME` which extracts the tag name of the node specified by the argument path. This enables the processor to perform a lookup on the Tag Index to identify the relevant edit operations (this identifying process is executed using `FILTER`). Note that `PROCESS-QUERY` can handle the wildcard (*) operator as the end of a path too, as `TAG-INDEX` would return all operations in the Tag Index. If there are nodes that match the 'tag' and the 'op' required, `PROCESS-QUERY` then checks that the matching node has a Canonical path equivalent to the input *path*, using `IS-VALID`. The function also traverses the Path Index tree upwards, in an attempt to locate other data versions that had the required operations performed on the ancestor of the current matching nodes.

```
// returns a set, R, of version numbers that
// match the 'path', 'op' inputs
PROCESS-QUERY(path, op):
 1 tag ← NODE-NAME(path);
 2 C ← TAG-INDEX(tag);
 3 if C ← {} then
 4    // path does not exist
 5    return {};
 6 N ← FILTER(C, op);
 7 R ← {}
 8 if N != {} then
 9    for each node ∈ N do
10       if IS-VALID(node.path, path, node.ver, node.id) then
11          R ← R ∪ node.ver;
12 for each c ∈ C do
13    i ← *(c.ptr);
14    R ← R ∪ PROCESS-QUERY-UP(c.path, path, op, i.parent, R);
15 return R;

PROCESS-QUERY-UP(cpath, path, op, i, V) :
 1 // traverse the tree upwards to see if
 2 // the ancestors were involved in op
 3 R ← {};
 4 N ← FILTER(i, op);
 5 if N != {} then
 6    for each node ∈ N do
 7       if node.ver ∈ R then
 8          // version already matched
 9          continue;
10       if IS-VALID(cpath, path, node.ver, node.id) then
11          R ← R ∪ node.ver;
12    if i = P.root() then
13       return R;
14    return R ∪ PROCESS-QUERY-UP(path,op, i.parent, R);
15 else
16    if i = P.root() then
17       return R;
18    return PROCESS-QUERY-UP(path, op, i.parent, R);
```

In `IS-VALID` we include a parameter *l* to keep track of the current token in *cpath* being matched to *path.* We define a *token* to be either a tag, wildcard operator (*), child operator (/) or descendant operator (//). This enables the query processor to handle queries with regular path expressions.

```
IS-VALID(cpath, path, ver, id) :
1  return IS-VALID(cpath, path, ver, id, 0, 0, P.root(), 1);

// (lowerVer, lowerId) = lower bound for matching nodes
// (ver, id) = upper bound for matching nodes
IS-VALID(cpath, path, ver, id, lowerVer, lowerId, start, l) :
 1 if start = null ∧ path.length() = l then
 2    // matched path with cpath
 3    return (lowerVer, lowerId);
 4 // scan the Path Index
 5 valid ← null;
 6 C ← start.children();
 7 for each c ∈ C do
 8    if PREFIX(c.path, cpath) then
```

```
 9      for each i ∈ c do
10        if !  (lowerVer, lowerId) < i) then
11        // previously matched tokens are not valid here
12        continue;
13      res ← (i.ver, i.id) < (ver, id);
14      if path[l] equiv i.tagc ∧ res then
15        for each j ∈ c do
16          res ← (j.ver, j.id) < (ver, id);
17          if j > i ∧ res ∧ j.tag = i.tag then
18            if j.op = "delete" || j.op = "updateOld" ||
19            j.op = "moveSrc" then
20              break;
21        if j != c.last then
22          // i not valid anymore
23          continue;
24        else
25          value ← IS-VALID(cpath, path,
26            ver, id, i.ver, i.id, c, l+1));
27            valid ← min(valid, value);
28  return valid;
```

The above algorithm only handles regular path expressions without predicates. To handle predicates, the code fragment below has to be inserted into IS-VALID, after Line 12. The code fragment below checks if the condition is true before allowing the algorithm to continue processing the next token in the *path*. It executes IS-VALID-QUERY to check for path validity. The function PREDICATE-PATH extracts the predicate path at position *l* and returns the corresponding path and operation.

```
13        if CHANGE-PREDICATE(path, l) then
14          (p, op) ← PREDICATE-PATH(path, l);
15          if !  IS-VALID-QUERY(cpath, p, op, ver, id,
16            i.ver,i.id, c, l+1) then
17            // predicate not satisfied
18            continue;
```

The algorithms for IS-VALID-QUERY is similar to that for PROCESS-QUERY and thus will not be presented here.

4.6 Index Representation

In practice, we store both indexes: Tag Index and Path Index as XML documents in the database. The DTD for the Tag Index and Path Index is:

```
<!DOCTYPE tagIndex[
   <!ELEMENT tagIndex (tagRow)*>
   <!ELEMENT tagRow (node)+>
   <!ELEMENT node (path, ver, id)>
   <!ELEMENT path (#PCDATA)>
   <!ELEMENT ver (#PCDATA)>
   <!ELEMENT id (#PCDATA)>
   <!ATTLIST tagRow
     tag CDATA #REQUIRED>
]>
<!DOCTYPE pathIndex[
   <!ELEMENT pathIndex (indexNode)*>
   <!ELEMENT indexNode (insert|delete)*, (indexNode)*>
   <!ELEMENT insert (ver, id, tag)>
   <!ELEMENT delete (ver, id, tag)>
   <!ELEMENT ver (#PCDATA)>
   <!ELEMENT id (#PCDATA)>
   <!ELEMENT tag (#PCDATA)>
]>
```

The advantages of storing the indexes as XML documents include:

1. Ease in performing lookups on the indexes. This is possible by applying complex XPath expressions on the XML document indexes. This enables the query language to be more expressive as all the information contained in the indexes can be queried efficiently and easily;
2. Intuitive mapping between the syntax and semantics of the query language; and
3. Ease in updating the indexes as concurrency/ access control mechanisms are provided by the database management system. Therefore, the design of our query mechanism can focus on the querying of changes rather than the index locking mechanisms.

Hence, the user has two avenues for querying changes in documents, either directly, via the actual XML copies of the indexes, or indirectly, via the functions defined in Section 2.1.

Therefore, when the user performs an operation on a document version, the query mechanism issues update statements to the Tag Index and Path Index XML documents in order to maintain the indexes.

5. CONCLUSION

In this paper, we have presented an efficient version management system for multi-version documents. By storing intermediate versions of the document as complete deltas, we are able to efficiently query these data versions without having to reconstruct each version. We utilise XQuery to facilitate the querying of changes and/or content in the system.

6. 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. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, May 1997.

[3] 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.

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

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

[6] CVS. Concurrent versions system. *http://www.cvshome.org*.

[7] N. Lam and R. Wong. Managing and querying changes for xml data. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics*, July 2002.

[8] Hummingbird Ltd. Hummingbird's document management and content management solutions. *http://www.hummingbird.com/products/dkm/*.

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

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

[11] W3C Recommendation. Xquery 1.0: An xml query language. *http://www.w3.org/TR/xquery*, April 2002.

[12] 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.