LNV: A Storage Structure for Native XML

Mohamed E. El- Sharkawi *Neamat Abd El-Hadi El Tazi*
Information Systems Department
Faculty of Computers and Information
Cairo University, Egypt
Mel-sharkawi@acm.org Neamat_tazi@hotmail.com

Abstract

With the rapidly increasing popularity of XML as a model for data representation and exchange on the Internet, there is a lot of interest in efficient storage of data that conforms to a labeled-tree data model. In this paper, we propose a novel structure that is capable of evaluating XML queries without re-parsing the XML document. Unlike existing approaches to store XML documents, the proposed structure stores only paths of an XML document. Therefore, the proposed structure is superior as the structure size is reduced. We present algorithms for evaluation of XPath queries on documents stored in the new structure. The paper also presents algorithms that update the structure taking into consideration both value and schema updates on XML document.

Keywords
XML, Storage, XPath queries, XML Update

1 Introduction

XML data can be depicted as a tree such as the tree in Figure 1. If each node in this tree is stored as an object in an object repository, the objects have to be visited when evaluating regular path expressions. This increases the number of objects fetched when evaluating a query.

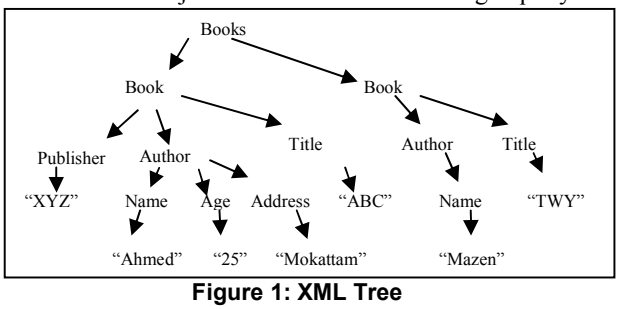

Figure 1: XML Tree

First author dedicates his work
to Yahiko Kambayshi a friend and supervisor

We depict the relationship between the nodes using a position field in our storage structure. By this way, we are not subjected to any technique of schema extraction. In our storage structure, all the information that is present in an XML document can be retrieved from the proposed storage structure.
This paper is organized as follows: Related work is presented in Section 2. Section 3 presents the architecture of our storage structure LNV. Section 4 presents the algorithms used to load the XML document to an LNV structure and construct the XML document from an LNV structure. Section 5 presents the algorithms needed to evaluate XPath queries using LNV. Section 6 presents the update algorithms required to update the structure when the XML document is updated. Performance Evaluation is presented in Section 7. Conclusion is presented in Section 8.

2 Related Work

Abiteboul examines the use of a text file [1] to store the XML document. In [2], Feng and David described the use of file as a collection of records in an object manager and evaluate alternative strategies for grouping XML elements into page-sized records. All major relational database vendors now offer some form of XML support [3,4,5]. These commercial tools are all conceptually similar to the relational DTD approach that is evaluated in [2]. The STORED [6] system utilizes data mining to extract a schema from XML data and converts them to relations. Several indexes are proposed for regular path expressions for fast retrieval. In some cases, these indexes may not cover all the possible paths because of storage requirements. These indexes include 2-Index [7] that stores the start node and the end node for an arbitrary regular expression. There are cases where the number of nodes in the 2-index is the square of the number of nodes in the data graph.
T-index [7] is similar to 2-index but the stored paths are restricted in their start node to reduce the search space. As a result there can be nodes that are not covered by the index.
Other problems include a large number of necessary joins to evaluate just a simple query. There are also

some problems generated due to complex joins between indexes. Moreover, some systems rely on approximate schema extraction to extract a schema from the XML document and use it as a relational schema. This led to loss of information.

The proposed storage structure overcomes these problems by storing the paths of the XML tree rather than the individual nodes. This makes our storages structure more compact and generates a better response time for evaluation of queries.

3. Labels-Nodes-Values (LNV) Structure

Definition 1 (*LNV Structure)*: LNV(Labels – Nodes – Values) is a storage structure for storing XML documents in which each path in the XML tree is represented by a tuple PT (L, N, V, T, P), where 'L' is the list of signatures of labels in the path ordered from the root, 'N' is the list of nodes in the path ordered from the root, 'V' is the value associated with the end of the path, 'T' is the type of leaf node (element, attribute, comment, or text); types of nodes are used according to the XPath data model [8]; and 'P' denotes the positions that represent the occurrence of each element node among its sibling. □

The type of node component 'T' is used to regenerate the result of a query as an XML tree or XML document afterwards. The position 'P' component is used to facilitate the evaluation of queries on position of elements inside a document. We also added another component DocID to our structure to be able to represent different XML documents inside the same storage structure.

The structure corresponding to the tree in Figure 1 is given in Figure 2. Figure 2a represents the labels of the XML document with its associated signatures. Figure 2b represents its LNV structure.

For example, in Figure 2b, the second tuple corresponds to the second path of the XML tree. The path labels are books, book, author and name. The associated signatures for these labels in Figure 2a are 1, 2, 3 and 4 respectively. Therefore, the path of signatures field entry in LNV structure contains the four signatures as a list [1, 2, 3, 4]. The nodes of this path are (1), (2), (5), and (6) considering the depth first order traversal of the tree. Therefore, the path of nodes entry of the LNV structure for this path contains the list [1, 2, 5, 6]. The value associated with the path is "Ahmed" as presented in the leaf node of the second path in the XML tree. Therefore, the values entry of the path contains the value "Ahmed". The type of this value is an "attribute" and it is entered as the path's type of node entry in the LNV structure.

a)Labels and its signatures

Label	Signature
Books	1
Book	2
Author	3
Name	4
Age	5
Address	6
Title	7
Publisher	8

b)LNV structure:

DocID	Path of Signatures	Path of Nodes	Values	Type of Node	Position
1	[1,2,8]	[1,2,3]	["XYZ"]	Element	[1,1,1]
1	[1,2,3,4]	[1,2,5,6]	["Ahmed"]	Attribute	[1,1,2,1]
1	[1,2,3,5]	[1,2,5,8]	[25]	Element	[1,1,2,2]
1	[1,2,3,6]	[1,2,5,10]	["Mokattam"]	Element	[1,1,2,3]
1	[1,2,7]	[1,2,12]	["ABC"]	Element	[1,1,3]
1	[1,2,3,4]	[1,14,15,16]	["Mazen"]	Attribute	[1,2,1,1]
1	[1,2,7]	[1,14,18]	["TWY"]	Element	[1,2,2]

Figure 2: LNV structure to the XML tree in Figure 1

The last field in our structure is the Position field. It contains the corresponding position of each node in the path with respect to its siblings. For example, the position of node (5) is 2 because it is the second child of its parent.

4 Loading and Constructing an XML Document using the LNV Structure

Loading the XML document to the LNV structure and constructing it afterwards is done using two functions, Load XML and Construct XML. In this section we describe the algorithms that load and then construct an XML document using LNV storage structure.

4.1 Load Function

The first function in our system is the loading function, Load (XML file) which is described in Figure 3.

This function is responsible for loading the XML document in the LNV storage structure. The input to function Load is an XML document. Function Load gives a signature to each distinct label in the XML document and put it in the signature file. Parsing the XML is done using XML Text Reader class that is present in the C# language inside the Microsoft visual studio.Net environment. This class is based on a SAX parser. We use the SAX parser because it does not load the entire XML document in main memory such as DOM parser. This enables us to use different sizes of XML documents.

Input: XML File
Output: LNV structured file, signature file

Begin

1. Read the XML document.
2. Read node by node and while reading construct each path in main memory.
3. After constructing each path, the path is stored in the LNV file.
4. Each new label is given a signature and stored in the signature file.

End

Figure 3: Load Function

We also managed to decrease the time needed to load the file in the structure by constructing the paths while reading each node from the XML document. By this way, we load the entire XML document in the LNV structure by traversing it only once. The output of the loading function is the LNV file itself and the signature file.

4.2 Construct XML Function

Before describing the Construct XML Function, we need the following definitions.

Definition 2 (*Intersection Nodes):* Given two paths in any XML tree. The intersection nodes between these two paths are the common ancestor nodes of these two paths.

For example, given the following two paths: $label_i/label_{i+1}[predicate_k]/label_n$ and $label_i/label_{i+1}[predicate_{k+1}]/label_m$. The intersection nodes are $label_i$ and $label_{i+1}$ where *i* in the first path equals to *i* in the second path. This is because $label_i$ and $label_{i+1}$ are the common ancestor of the given two paths. Another example includes the ancestor-descendant step of an XPath Query. Consider the following two paths $//label_i/label_{i+1}[predicate_k]$ and $//label_i/label_{i+1}/label_{i+2}[predicate_K]$. Unlike the first example, the intersection nodes are all the nodes from the root inclusive $label_i$ and $label_{i+1}$ not just $label_i$ and $label_{i+1}$. This is because the // step is replaced by the path from the root until the node after the step.

Definition 3 (*Query Paths):* Given an XPath query $label_i/label_{i+1}[predicate_k]/\ldots/label_n[predicate_{k+1}]$. Paths in this query are determined as follows. Every path ends at the first occurrence of a predicate, inclusive the label of the predicate. The first path is $label_i/label_{i+1}[predicate_k]$. The next path is determined by taking all nodes in the previous path and removing all previous predicates up to the first occurrence of another predicate. The next path is $label_i/label_{i+1}/\ldots/label_n[predicate_{k+1}]$.

Definition 4 (*Original Path):* The term original path denotes the signatures that must be contained in the query result path. The original path is obtained by removing all predicates from the input XPath expression query.

For example, given an XPath query $label_i/label_{i+1}[predicate_k]/label_n$. The original path is $label_i/label_{i+1}/label_n$. The original path has the same signatures of the result paths because the predicates are satisfied in the result of the query. Note that a query may have several result paths, however, it has only one original path. For example, the query books/book[title="ABC"]/author contains two paths which are books/book[title="ABC"] and books/book/author. Its original path is books/book/author. The result contains all the paths that have the signatures of the labels books, book and author but satisfy the predicate of the first query path.

The second function, Construct XML, function is described in Figure 4. Function construct XML takes any paths as an input and returns the XML that corresponds to these paths. For example, if we send the entire LNV structure to this function, the constructed XML will be equivalent to the original XML document that was stored in the LNV structure. On the other hand, if we send query result records, the result XML will be the XML document that corresponds only to these paths. Construct XML function loops on the input paths and their nodes to identify the types of the nodes. If the current node is not the last node in the path then, we just write a new element in the result XML document with the label of that node.

On the other hand, if the current node is the last node in the path then, we check for many things before writing the node. First, we check for the node type. If the type of the node is an attribute, we write an attribute to the result XML document with the label of the node as the attribute name and the value of the path as the attribute value. We then, check the next record if it contains another attribute or text value that must be written inside the current element before writing the end tag of the current element. If there is no other text or attribute node in the next path or the next path does not have the same position of the current path, we write an end tag for the current element.

If there is another attribute or a text node inside the current path, we write the attribute or the text node inside the current element and then check the next path until reaching a path that is not related to the current element. We advance the pointer that points to the current path to reach the path that is not related to the current path to begin the process of checking again. Node numbers of nodes that are written in the XML result are kept in a repository where we check on it every time we are going to write any new node so that the node is written only once in the XML result. For example if we have two paths, books/book/title and

books/book/author. The first two nodes books and book will be written once in the XML result. The step of storing the node numbers of the nodes written in the XML document helps us in writing the intersection nodes only once because they are common between the different paths of the query.

Input: paths, Signature file
Output: result file

BEGIN
-For each path in the input paths.
-For each node in the path
- Fetch its label from the signature file
-Write it as a start element if it is not the last node in the path
-If it is the last node in the path
-If the type of value is attribute then write the label with the value as an attribute in the result XML document
-If the type of value is text, write start element with the label and then write the value of the text as a string inside the XML element
-If the type of value is text after element then there was an attribute before the text of the element. Write the attribute and then write the text of the element.
- Write "NO Result" element in the XML file if there are no paths given as an input.
END

Figure 4: Construct XML Function

5 Query Processing on the LNV Storage Structure

Query processing on the LNV structure is decomposed into two separate functions. The first function is the Query Parsing Function that parses the user query to get its components. The second function uses the output of the query parser to search inside the LNV structure to get the query result. Figure 5 shows the data transfer between the different functions that are available in the algorithm.

The input to our algorithm is the XML file as shown in Figure 5. The load function loads the XML file to the LNV structure. The output of the loading function is the LNV file and the signature file. The LNV file contains all the information of the XML document and the signature file contains the different labels in the XML document and its specified signatures.

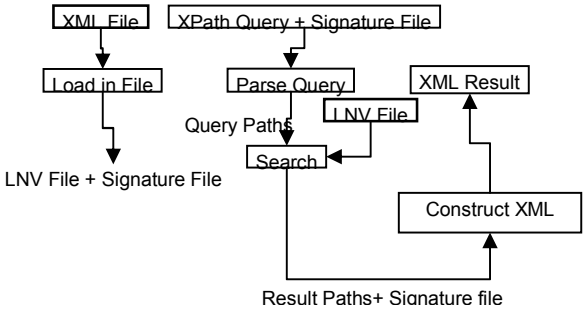

Figure 5: Data Flow between functions

To search for a query inside the LNV file, the user's XPath query and the signature file are sent as an input to the parse query function. The query parse function parses the query and determines the number of paths inside the query, the paths themselves, the intersection nodes between these paths, and the original path. The second step in evaluating the query is searching for the query paths inside the LNV structure. The query parsing function returns the query paths to the search function to begin searching inside the LNV records.

The search function takes the query paths and the LNV file as an input and outputs the result paths that are passed to the construct XML function. The construct XML function constructs the XML file that corresponds to the result of the query. In the following, we illustrate the main steps of the functions used to evaluate queries on LNV structure.

5.1 Query Parsing Function

The query parsing (XPath expression, Signature file) Function is described in Figure 6. The input to this function is an XPath expression and the signatures file. As shown in Figure 6, the function first reads the input XPath expression to be able to determine how many paths are in the query expression. The query parsing function then determines the query paths and substitutes the labels in these paths with their signatures using the input signature file.

Input: XPath expression, Signature file
Output: query paths, intersection nodes, original path
BEGIN
1. Read the input XPath expression
2. Determine how many paths in the query
3. Determine the paths themselves
4. Substitute the labels with its signatures
5. Determine the intersection nodes if more than one path exists
6. Determine the original path.

END

Figure 6: Query Parsing Function

For example, using the storage structure in Figure 2 and the XML tree in Figure 1, the path books/book/title is

substituted with the path [1, 2, 7] where 1 is the signature of label books, 2 is the signature of label book and 7 is the signature of label title. If there is more than one path in the query expression, the function has to determine the intersection nodes between these paths. For example, if the input path expression is books/book [author=”neamat”]/title. This path expression contains two paths which are books/book/author with value “neamat” and books/book/title. The intersection nodes between these two paths are the first two nodes books/book. Then the result paths should contain paths that have the same node numbers for the first two nodes of the path of nodes entry of each result path.

The function then determines the original path which denotes the paths that are needed in the result of the query. Taking the above path expression as an example, the original path is books/book/title because the query retrieves all titles of books where the author of these books is “neamat”. So the result should contain only the titles of the books not the entire two paths of the query. The output of this function is the query paths that are used afterwards to search in the LNV structure.

Example 5.1: parsing an XPath query

Consider the XML document in Figure 1 and its LNV storage structure in Figure 2. Given the Xpath query: //book[author/name=”Ahmed”]/publisher, this query retrieves the publishers of all books where the book’s author name is “Ahmed”. The query parser first determines the query paths. The query paths in this query are //book/author/name with value “Ahmed” and //book/publisher. Then the parser substitutes the query labels with their signatures that are found in Figure 2a. The parser substitutes the ancestor-descendent step ‘//’ by ‘101’. Then the signatures corresponding to the two paths are 101,2,3,4 and 101,2,8.

The parser then determines the original path of the query. The original path of the query is //book/publisher which has the same signatures of the second path. Then the parser determines the intersection nodes of the query. In this query, the two paths have ‘101,2’ signature as their intersection node. This does not mean that the query has only one intersection node because ‘101’ as mentioned before is a signature given for the ancestor-descendant step therefore it can substitute many nodes starting from the root until reaching the node after 101 in the path, node 2.

5.2 Search Function

The Search (LNV file, signature file, query paths, intersection nodes, original path) function, described in Figure 7, deals with evaluating the parsed query.

The input to this function is the LNV file, signature file, query paths, intersection nodes and the original path. The Search function searches for paths inside the LNV structure that contain the path of signatures of the query paths such that the resulting paths have the same intersection nodes.

Input: LNV file, Signature file, query paths, intersection nodes, original path
Output: result paths
BEGIN

1. For each path in the query paths
   -Search inside the path of signatures of each record in the LNV file for paths that contain the path of signatures of the query path.
   -Check if the value of the LNV record satisfies the predicate of the query path, it also should satisfy the positions if specified as predicates.
   -Due to the searching for the signatures of the query path, many records will be returned, records that do not satisfy the current query. path predicates, are neglected in the search of the next query path.
   -Check if the intersection nodes are equal in the LNV records.
2. Determine the records that satisfy the original path from the result of step one.
3. Return the result records of step two.

END

Figure 7: Search Function

The Search function checks if the value of the LNV output satisfies the predicates found in the query paths. The resulting records must all contain the original path inside their path of signatures entry.

We manage to reduce the time needed for the search by neglecting the records that do not satisfy predicates of the query paths. For example, if the query is books/book [author=”neamat”]/title. The first query path books/book/author searches all the entries of the LNV structure that satisfy the signatures of the labels books, book, and author with value equals to “neamat”. The result of the first query path is some records that contain the books where their author is “neamat”. The second query path books/book/title does not search all the records in the LNV structure but it searches only the records that have the same intersection nodes of the first query path.

Example 5.2: Searching for a query result in LNV structure

Consider the XPath query example in Example 5.1. The XPath query is: //book[author/name=”Ahmed”]/publisher. After parsing the query in Example 4.1, the parsing result contains the two query paths represented by the following signatures 101,2,3,4 with value=”Ahmed” and 101,2,8. Search

function first searches for the first path 101,2,3,4 in the path of signatures entry in the LNV structure represented in Figure 2b. The signature '101' is replaced by all signatures in the LNV record beginning from the root signature until finding the signature '2' in the LNV record. If '2' is not found then the record is not considered in the result of the query. The result of searching for the first query path is:

DocID	Path of Signatures	Path of Nodes	Values	Type of Node	Position
1	[1,2,3,4]	[1,2,5,6]	["Ahmed"]	Attribute	[1,1,2,1]
1	[1,2,3,4]	[1,14,15,16]	["Mazen"]	Attribute	[1,2,1,1]

The resulting records must satisfy the predicate associated with the path. This means that the value of the LNV record must be "Ahmed". Therefore, the result of this query path is the first record only:

DocID	Path of Signatures	Path of Nodes	Values	Type of Node	Position
1	[1,2,3,4]	[1,2,5,6]	["Ahmed"]	Attribute	[1,1,2,1]

The search function then searches for the second query path which is 101,2,8. The search is done by navigating inside the LNV structure but neglecting all records that do not have the same intersection nodes of the first query path. The intersection node in this query path is '101,2'. In the first query path result the signature '101' corresponds to the first signature in the path of signatures entry of the record. This signature is '1', the node number that corresponds to this signature in the path of nodes entry is also '1'. The second signature '2' has '2' as its node number. Therefore, the records to be searched must contain the first signature '1' and its corresponding node number must be '1' and the second signature '2' and its corresponding node number must be '2'.

The result paths of the query must contain the original query path. The original path of the query is //book/publisher which have the signatures 101,2,8. Therefore, the result of the query is:

DocID	Path of Signatures	Path of Nodes	Values	Type of Node	position
1	[1,2,8]	[1,2,3]	["XYZ"]	Element	[1,1,1]

6 Update Operations

Updates in XML, unlike relational databases, may be on values and on the document structure as well. In this section we introduce algorithms that handle the different update operations on an LNV structure. We present the basic update operations which are delete, move, insert and update [9]. We consider only these basic update operations because any sophisticated operation can be implemented through combining these operations

Updating the structure of an XML-tree *T* is done by using the following basic operations [9]:

1. Delete (*m*) that deletes the XML tree rooted in node number *m,* where *m* is not the root of *T*.
2. Insert (*n*, *k*, *T'*) that inserts the XML tree T' as the k^{th} child of *n*.
3. Move (*n*, *k*, *m*) that moves the XML tree rooted in node *m* to be the k^{th} child of *n*.

The update functions have no outputs. The output here is represented as a change in the LNV structure.

6.1 Update Value Function

Updating a value in a document is handled by the UPDATE-VALUE function which is described in Figure 8. The UPDATE-VALUE function updates the value of node number *m* to new value *v*. The input to this function is a node number that contains the value to be changed and the new value. UPDATE_VALUE function simply searches for the input node number and updates its value by the new value in the input list of parameters of the function.

Input: node number, value
Output: LNV structure is updated

Begin

1. Search for the record that contains the node number
2. Update the value of the record with the input value

End

Figure 8: Update Value Function

6.2 Delete Function

The DELETE function is represented in Figure 9. DELETE function, as mentioned above, deletes an XML subtree rooted by a specified node. It, first, determines entries in the structure that correspond to the subtree to be deleted. This is performed by searching for entries in the structure that contain the input node number, *m*, (i.e. the root of the subtree to be deleted) in their path of nodes components. Then it finds the position of the node number *m*. Note that node number *m* has the same position in all found entries in the first step.

The third step is to modify positions of the right siblings of the deleted node *m* by subtracting one from the positions of these siblings. The function then checks if the subtree to be deleted is the only child of its parent. If so, then it adds just one entry in the LNV structure that represents the path from the root until the parent of the

deleted node *m*. The last step is to delete all the entries that contain the deleted node *m*.

6.3 Insert Function

The INSERT function is described in Figure 10. This function, insert (*n*, *k*, *T'*), inserts the XML tree *T'* where *T'* is passed as a parameter to the function in the form of entries in the LNV structure. *T'* is to be inserted as the K^{th} child of node number *n*. Function INSERT finds the signature and position of the path of the node number *n*, which is the path from the root to node *n*.

The first position in the LNV entry of each record in T' is substituted by the input child number *k* in its LNV position entry. Then it updates the entries of the input tree with the signatures, nodes and positions of the path of *n*. The update is done by inserting the signatures, nodes and positions of path *n*, before the signatures, nodes and positions of the records to be inserted. It modifies the positions of the right siblings of the child *k* by adding one to all these siblings. The document identifier of T' is set to the new document identifier of the tree that we are inserting T' in. Finally, the function adds the input records of *T'* to the original tree.

Input: node number m // root of the subtree to be deleted
Output: LNV structure is updated

Begin

1. Search for the path of nodes records that contain node m.
2. Get the position of node m with respect to its siblings
3. Modify the position of its right siblings by subtracting one from each one of them
4. If the parent of node m doesn't exist in other records (not the ones to be deleted) then
   - 4.1. add a record that contains the path from the root until the parent of m
   - 4.2. delete all records resulted from step 1
5. Else
   - 5.1. delete all records resulted from step 1

End

Figure 9: Delete Function

6.4 Move Function: Delete (m), Insert (n, k, T')

The last update operation is Move, which is described in Figure 11. It accepts three input *n, k,* and *m*. It moves the XML tree rooted by node number *m* to be the K^{th} child of node number *n*. The function searches for the entries corresponding to the subtree to be moved and then stores these entries in a temporary repository. The function then deletes the signatures of the records in the temporary repository until reaching the node number *m* to be able to construct the records of the tree to be inserted after the deletion of the moved subtree. Then the function performs the same deletion for the nodes and positions of the records stored until reaching node number *m.*

Input: node number *n*, child number *k*, tree *T'*
Output: LNV structure is updated
Begin

1. Find the path of node *n*
2. Find the signatures of the path of node *n*
3. Find the node positions of the path of node *n*.
4. Update the records of *T'*
   - 4.1. put the signatures of step 2 before all entries of path of signatures
   - 4.2. put the nodes of the path in step 1 before all entries of path of nodes
   - 4.3. substitute the first position of position entries by *k*
   - 4.4. put positions in step 3 before all positions in the position entries
   - 4.5. Update document identifier to be the document identifier of the document that the sub tree will be inserted in.
5. Modify all positions of the path children in step 1 by adding one to all positions equals or greater to *k*
6. Add *T'* entries to the structure entries.

End

Figure 10: Insert Function

Input: node number *n*, child number *k*, node number *m*
Output: LNV structure is updated
Begin

1. Find the entries to be moved by searching for the node number *m*
2. Store the entries from step 1 in a temporary tree *T'*
   - 2.1. Delete the signature of the path until the signature that correspond to node m from *T'*
   - 2.2. Delete the nodes of the path until node *m* from *T'*
   - 2.3. Delete the positions of the records until node *m* from *T'*
3. Delete (*m*)
4. Insert (*n*, *k, T'*)

End

Figure 11: Move Function

The Move function then deletes the subtree to be moved from the XML tree using function Delete and finally inserts the stored entries to their new location using Insert function.

7 Performance Evaluation

In this section, we present different types of performance evaluation for our novel XML storage structure, LNV. We have implemented LNV and carried out a series of performance experiments in order to check the effectiveness of the structure. Performance is evaluated from a number of perspectives such as the LNV file size with respect to the XML document size, the time needed to load the XML document to the LNV file, the time needed to construct the whole XML document from the LNV file and the time needed to evaluate an input query

7.1 Bulk Loading Time

We illustrate the bulk loading time with respect to different types of XML trees. There are two types of XML trees, deep trees and wide ones. Deep trees are trees that are narrow and contain small number of paths but these paths have long depth. On the other hand, wide trees have a big number of paths but these paths have a short depth.

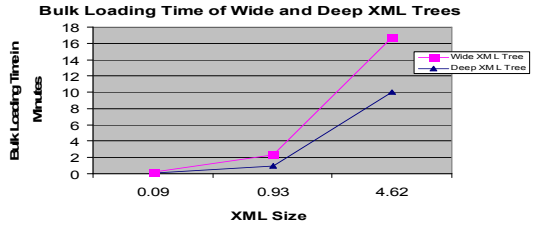

Figure 12: Bulk loading time

The time for loading a wide XML tree is much more than the time for loading a deep XML tree. This is because a wide XML tree contains more paths than a deep one and the LNV structure stores the paths of the XML document. The path is constructed in main memory while parsing the document and then the path is stored inside the LNV file. Therefore, the unit of decomposition of the XML document is the path itself. That is why the time of loading a wide XML tree is more than the time of loading a deep XML tree that has less number of paths than the wide one.

7.2 LNV size

This section presents the difference between the sizes of the stored LNV files with respect to the XML size.
Since the Wide XML tree contains more paths than the deep XML tree, the size of the LNV structure constructed from a wide XML tree is larger than the size of the LNV structure constructed from a deep XML tree.

7.3 Query Processing Time

Figures 14 and 15 show that the LNV structure for wide XML tree takes more time in evaluating both chain and tree queries. This is because of the same reason we mentioned before in the previous sections.

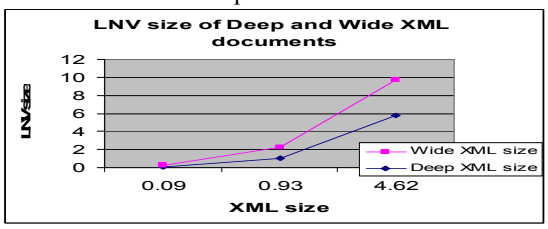

Figure 13: LNV size versus XML document size

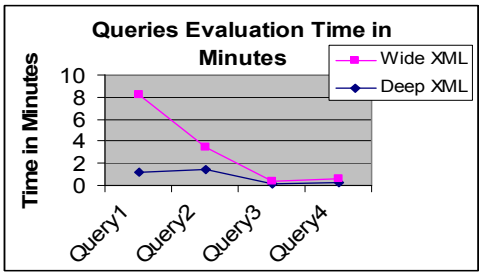

Figure 14: Query Processing time for four simple queries

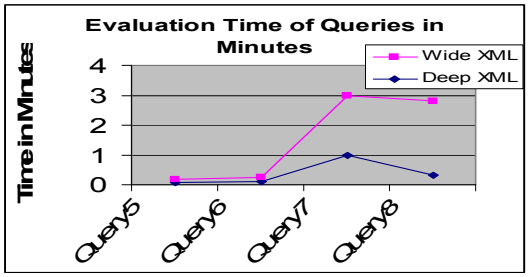

Figure 15: Query Processing time for four sophisticated queries

7.3 Construction Time

It is obvious that the construction time needed for constructing the wide XML document is much more than the time needed for constructing the deep XML document. This is because the same reason mentioned in the previous sections.

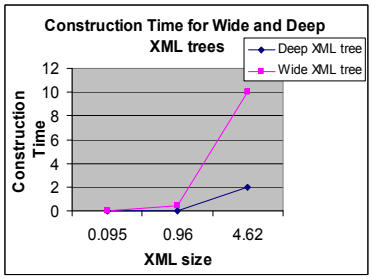

Figure 16: Construction Time for the XML documen

8 Conclusion

Motivated by applications requiring efficient evaluation of XPath queries against XML documents, we presented a novel structure to be used for the evaluation of XPath queries. Our LNV structure can be used as a storage technique because the whole document can be extracted from the structure. Our structure can be characterized as a covering index because it covers all the data inside the XML document. Evaluating a query does not require scanning the XML document because the data can be retrieved from the structure directly. As XML documents are subject to updates, we presented algorithms for updating our LNV structure to represent updates on XML documents. We considered both value and schema updates. Our structure is superior to other proposed structures[1,2]. Most of existing approaches use several structures to store an XML document. Moreover, unlike existing structures, we store paths rather than nodes or nodes and paths.

Further research can be made in continuous queries, to store the XML documents coming as a stream in our structure to decrease the storage needed and to be able to use past data as an archive. Currently, we investigate using this structure in relational databases and we are developing mechanisms to convert the XPath queries to pure SQL queries applied on the proposed structure.

References

[1] S. Abiteboul, S.Cluet "*Querying and updating the file,"* In Proceedings of the 19th International Conference on Very Large Databases, 1993.

[2] T. Feng, J. David, C. Jianjun, Z. Chun *"The Design and Performance Evaluation of Alternative XML Storage Strategies,"* ACM SIGMOD Record, Vol.31, No. 1, March 2002.

[3] IBM DB2 XML Extender. Available at: http://www4.ibm.com/software/data/db2/extenders/

[4] Oracle XML SQL Utilities. Available at: http://otn.oracle.com/tech/xml/oracle_xsu

[5] Microsoft SQL server 2000 Books Online, "*XML and Internet support*"

[6] A. Deutsch, M. F. Fernandez "*Storing and Querying XML Data using an RDBMS,*" IEEE Data Engineering Bulletin 22(3), 1999.

[7] S. Abiteboul, P. Buneman, and D. Suciu. Data on the Web: from relations to semi-structured data and XML. Morgan Kaufmann, 1999.

[8] *XQuery 1.0 and XPath 2.0 Data model*, W3C working draft. 16 August 2002. Available at: http://www.w3.org/TR/2002/WD-query-datamodel-20020816.

[9] A. Marian, S. Abitebout, G. Cobena, L. Mignet. "*Change-Centric Management of Versions in an XML Warehouse,*" Proceedings of the 27th International Conference on Very Large Databases (VLDB), Roma, Italy, 2001, 581-590.