Efficient Querying of Root to Leaf Xpath Queries for XML Documents
Atul D. Raut, Dr. M. Atique

Abstract: XML is self describing, extensible and most importantly it is stored in the form of text document which makes it a cross platform technology independent of hardware and software. Because of such unique features XML is now recognized as standard for data storage and exchange over the net. In spite of all these unique features XML has an inherent limitation of verbosity. Moreover size of XML database has also increased considerably. With such growing presence of XML in database technology and its inherent verbosity there is ever increasing need to design compact non redundant storage for XML which can be effectively utilized for efficient indexing and querying of XML. XML is usually modeled as an ordered labeled tree where nodes represents the elements and edges between the nodes represents the relationship i.e. either the parent child or ancestor descendant relationship. The data in XML document is stored at the leaf node. The nodes above the leaf node constitute the structure component. Hence for efficient querying of structure and contents of XML documents it is essential to index the root to leaf paths efficiently. When we say root to leaf it actually means one level above the leaf.

The proposed technique creates a structure index which is a compact summarization of all root to leaf paths of the XML document and data index which groups and stores the contents of all similar paths at one place. Based on this non redundant compact storage a novel query algorithm is proposed which can answer root to leaf xpath queries very efficiently. This approach dramatically reduces the storage requirement for XML coupled with efficient processing of xpath queries. The implementation of this technique and comparison with other techniques confirms our claim.

Index Terms--- Content Index, Inverted List, Structure Index, Xpath Queries, Non Redundant Storage.

I. INTRODUCTION

XML is widely accepted standard for storage and exchange of data over the internet. In spite of its several desirable features XML has an inherent limitation of verbosity. XML documents are verbose because of the repeated tags present in its structure. This kind of verbosity of XML leads to unusually larger size of XML document as compared to the other standard format representation of the same information. Most importantly the size of XML has also increased dramatically. Indexing techniques used for relational database cannot be used directly for XML since XML data is ordered where as the relational data is unordered. Moreover XML contains structure in addition to data. The presence of structure makes the task of indexing much more difficult as compared to relational database. The most important factor for any efficient querying system is the time required to get result. This response time can be significantly reduced with the support of efficient indexing and storing technique for XML data. XML documents can be represented with the help of an ordered tree. In this representation nodes of the tree represents attributes, elements or the text data and the edges between the nodes represents the relationship between the nodes which can be either parent child or ancestor descendant. W3C query languages Xquery and XPath specify patterns of selection predicates on multiple elements that have some specified tree structured relationship. For example the Xquery expression book [title = ‘XML’] // author [fn = ‘jane’ AND ln = ‘doe’] searches for the author element having child or sub element as fn with content jane and ln with content doe (parent-child relationship) and all author should be descendants of book element (ancestor – descendant relationship) having child element title with content ‘XML’.

Thus it is clear that any XML query has two major components the structure component and the keyword (data information) component. An XML query can be answered by finding all occurrences of such a twig pattern and then stitching (joining) together these matches. Twig pattern matching can be achieved by decomposing the twig into a set of binary structural (parent child and ancestor – descendant) relationship between pairs of nodes. The key to fast response of any XML query is efficient indexing and storing of XML data. Following are the important issues when querying an XML document.

- **Index size:**
  - The size of the index should be small so that the entire index can be kept in main memory. The structure (path) and contents are indexed separately. The structure index is used to identify the structural relationship and based on this the contents can be obtained from the content index. Some numbering schemes have been proposed to quickly examine the structural relationship.

- **Intermediate Result:**
  - Most of the earlier XML query processing algorithm made use of inverted list which extends the inverted list used in information retrieval. By using some kind of join algorithm the entries in the inverted are joined to satisfy the structural relationship. Such join algorithm produced large intermediate result. This greatly increased the query response time. Hence the query processing algorithm or indexing /storage technique should be such that ideally it should not produce any intermediate result.[1]

- **I/O Required**
  - Query response time directly depends upon the I/O required to get the data. Hence the indexing /storage technique should be such that the query processing algorithm which utilizes these techniques should perform minimum amount of I/O.

- **Versatility/Flexibility**
  - The indexing/storage technique should be such that the query processing algorithm which utilizes the indexing/
storage technique should be able to get quick response to any type of XML query including the parent – child, ancestor-descendant relationship and containing several predicates.

These considerations gives rise to development of non redundant compact storage of XML data which will ultimately help the development of efficient indexing and querying technique to query large repositories of XML database. For this reason we propose an efficient non redundant technique for storage of XML document and a novel query algorithm to answer a variety of xpath queries based on this storage. Most of the techniques implemented so far for storing and querying of XML either made use of inverted list, tree structure or relational table representation of XML. All these techniques require large amount of memory for storing of XML and complex query algorithm to query XML. Our technique makes use of a non tree or non inverted tree based representation and successfully reduces the storage requirement. It not only reduces the storage requirement but also stores the data in such a way that it can be directly utilized by the query algorithm i.e. it does not require any kind of decompression and also the complexity of query algorithm is reduced. It eliminates the complex join algorithms which were required in case of inverted list representation. This technique creates two types of indices for compact and non redundant storage of XML. The first type of index is the structure index. This structure index is a compact summarization of all root to leaf paths in an XML document and every such path is assigned a unique path id. The other type of index created is the content index which store the contents i.e. the text data. It groups and stores at one place all the data for all the paths having the same id i.e. all the similar paths. The path id from the structure index acts as a link between the structure index and the content index. The first path in the structure index gets a index id of 1. Using this indexid of 1 a file is created in the content index having a name of index1. This file index1 groups and stores all the data items present on all the paths which are similar to this path in document order. Thus all the data items for all similar paths can be found at one location and moreover the structure information is not repeated. This greatly reduces the storage requirement for XML. Using this compact representation a novel query algorithm is implemented which answers queries based on several different xpath axes and in particular the root to leaf xpath query very efficiently. The main contributions of our technique are:

1. A compact queriable storage of XML which does not require decompression at the time of querying
2. Efficient access to many xpath axes without constructing the F&B bisimilarity graph which is required in case of FIX[6]
3. A non tree or non inverted list representation of XML.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 and 4 describes our technique. Section 5 reports the experimental results and section 6 concludes.

II. RELATED WORK

Till date several techniques have been implemented for querying XML documents using different indexing techniques by different researchers. These early techniques can be broadly classified into following types.

1. By traversing the tree or its compressed representation.
2. By using IR style processing using inverted list.
3. By combination of the first two. In this context structure index plays a vital role.
4. By using a Relational Database Management System.
5. Techniques which utilizes efficient data structures like B' tree, hash table etc.

These techniques can be further classified as queriable and non queriable. A queriable compression technique is one which does not require decompression at the time of querying, while non queriable compression technique requires decompression. Among the non queriable techniques ISX, Twig structure index plays a vital role.

A queriable compression technique is one which can be directly utilized by the query algorithm i.e. it does not require any kind of decompression and also the complexity of query algorithm is reduced. It eliminates the complex join algorithms which were required in case of inverted list representation. This technique creates two types of indices for compact and non redundant storage of XML. The first type of index is the structure index. This structure index is a compact summarization of all root to leaf paths in an XML document and every such path is assigned a unique path id. The other type of index created is the content index which store the contents i.e. the text data. It groups and stores at one place all the data for all the paths having the same id i.e. all the similar paths. The path id from the structure index acts as a link between the structure index and the content index. The first path in the structure index gets a index id of 1. Using this indexid of 1 a file is created in the content index having a name of index1. This file index1 groups and stores all the data items present on all the paths which are similar to this path in document order. Thus all the data items for all similar paths can be found at one location and moreover the structure information is not repeated. This greatly reduces the storage requirement for XML. Using this compact representation a novel query algorithm is implemented which answers queries based on several different xpath axes and in particular the root to leaf xpath query very efficiently. The main contributions of our technique are:

1. A compact queriable storage of XML which does not require decompression at the time of querying
2. Efficient access to many xpath axes without constructing the F&B bisimilarity graph which is required in case of FIX[6]
3. A non tree or non inverted list representation of XML.

The rest of the paper is organized as follow. Section 2 discusses the related work. Section 3 and 4 describes our technique. Section 5 reports the experimental results and section 6 concludes.
III. NON REDUNDANT COMPACT XML STORAGE

This technique first constructs a path index. This path index is a compact summarization of the entire XML document. To facilitate Xpath queries it stores all unique paths in the XML document without considering the attributes or its values on the path and assigns it a unique id. The algorithm path index reads the xml document and finds the total number of child nodes of the root node. All the child nodes of root node are stored on stack of xml nodes in reverse order so that the nodes can be processed in document order. The first child node of root node is popped out from the stack and all its child nodes are pushed on to the stack in reverse order. In order to store the root to leaf path of the current node, the current node is traversed back to the root node, again from root node to current node remembering the path in a stack and finally from current node to the leaf node of the current path giving a complete root to leaf path of the current node.

This path is stored in the path array path ( ). An XML document may contain several repeated paths. The currently obtained path is compared with the path already found and if it is same then it is marked as repeated (“R”). Finally the entire path marked as “R” are removed and only the unique paths are left in the path array l ( ). The id of the path is equal to the index location i.e. the first path gets an id of 1, the second path gets an id of 2 and so on.

For example consider the following fragment of a sample XML document which illustrates the creation of the path index.

```xml
<Student data>
  <Student>
    <College>jdiet</College>
    <deptt>it</deptt>
    <year>iii</year>
    <name>abc</name>
  </Student>
  <student>
    <college>dbnce</college>
    <deptt>ese</deptt>
    <year>ii</year>
    <name>xyz</name>
  </student>
</studentdata>
```

The path index for the above XML document is as given below.

<table>
<thead>
<tr>
<th>Path</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>/studentdata/student/college</td>
<td>1</td>
</tr>
<tr>
<td>/studentdata/student/deptt</td>
<td>2</td>
</tr>
<tr>
<td>/studentdata/student/year</td>
<td>3</td>
</tr>
<tr>
<td>/studentdata/student/name</td>
<td>4</td>
</tr>
</tbody>
</table>

The path id provided by the structure index acts as a link between the path and all the contents on that path. All the data belonging to a particular path for example the path /studentdata/student/college is stored at one place and can be directly accessed using its id which is 1 in this case. Such grouping of data based on the path gives rise to content index.

The algorithm Content index reads the xml document and finds the total number of child nodes of the root node. All the child nodes of root node are stored on stack of xml nodes in reverse order so that the nodes can be processed in document order. The first child node of root node is popped out from the stack and all its child nodes are pushed on to the stack in reverse order. In order to get the root to leaf path of the current node, the current node is traversed back to the root node, again from root node to current node remembering the path in a stack and finally from current node to the leaf node of the current path giving a complete root to leaf path of the current node. This is somewhat similar to creation of path index. After getting the first path this path is compared with the paths in path index and when a match is found, it remembers the index id. It then writes the text data on this path to a text file whose name has a prefix which is index and suffix which is equal to the indexid of that path. For example the name of the first file in the content index will be index1; the name of the second file will be index2 and so on.

The content index for the above fragment of XML document is as shown below.

<table>
<thead>
<tr>
<th>Path id</th>
<th>Contents</th>
<th>file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>jdiet, dbnce</td>
<td>index1</td>
</tr>
<tr>
<td>2</td>
<td>it, cse</td>
<td>index2</td>
</tr>
<tr>
<td>3</td>
<td>iii, ii</td>
<td>index3</td>
</tr>
<tr>
<td>4</td>
<td>abc, xyz</td>
<td>index4</td>
</tr>
</tbody>
</table>

Thus from the above content index it is clear that the structure or path information is not repeated several times but the path information is stored only once. Moreover all the data items on particular path are stored together. An important feature of the content index is that it retains the ordering of the original XML document. For example the path /studentdata/student/college, this path appears twice in the sample XML document and may appear several times in the actual XML document. This path is stored only once and using its id (id=1) the contents are stored and can be accessed directly. The contents on this path are jdiet, dbnce and most importantly these contents are stored in document order at one place without repeating the path information. This technique works extremely well with document centric XML document and particularly if the elements of XML document does not contain attributes. It requires further enhancement for data centric XML document containing attributes.

IV. QUERYING OF XML DOCUMENTS

The algorithm Evalpath separates the input xpath expression into tokens tokens considering / and // as the delimiters (step1). Step2 stores the separators i.e. / and // in the array sep. Step 3 counts the number of tokens in the input xpath query excluding the separators / and // Step 4 repeatedly scans every path from the structure index and compares every token of the path in structure index with the respective tokens of the input xpath query. If all the tokens of the current path in the structure index matches with the
V. EXPERIMENTAL RESULTS

A. NRCX Compression Performance

We have implemented our technique on Intel Pentium IV dual core 3.0 GH processor with 2 GB of DDR Ram with VB.NET running on Windows XP platform. Some benchmark XML databases were considered for comparison with other techniques for compact storage of XML. We now briefly introduce each dataset.

1) Shakespeare: It contains records of plays written by Shakespeare. It is document centric XML dataset having a irregular structure, no attributes and contains a lot of textual data.

2) Orders: It is data centric XML data set with a regular structure

3) Lineitem: It also is data centric XML data set with a regular structure

4) Treebank: It is highly skewed data set with varying depth for different element. It is available from university of Washington XML repository.

Table I shows the results of implementation of our technique for non redundant compact storage of XML and comparison with the results of other previously implemented and tested techniques for compression. The results of other techniques have been reported from the reference cited by [16].

As seen from table I NRCX requires less amount of memory i.e. the compression ratio is better in case of NRCX. Since the size of structure and content index is small for small to medium size XML documents (1 to 100MB) the entire contents of both the index can be processed in main memory. This will significantly reduce the query response time for an xpath expression. The way the structure index is organized and stored greatly helps in finding quick response to different types of xpath queries. The simple xpath expression can be directly answered by using the structure index. The xpath queries beginning with // can be answered by just storing the paths in reverse order and performing the prefix matching which can be performed by single index look up of the structure index to get the index id. Once the id is obtained all the contents present on the path can be obtained in a single I/O operation.

B. NRCX QUERY PERFORMANCE

For evaluating the query performance the Treebank and Mondial XML data set were used. Treebank is highly skewed data set with maximum depth of 36 nodes, where as Mondial is a very flat data set. Query performance is evaluated for following root to leaf xpath queries.

Q1: //mondial/country/name
Q2: //FILE/EMPTY/S/VP/NP/NN

The results of other techniques for query Q1 is taken from the reference cited by [16], and the results of other techniques for query Q2 is taken from the reference cited by [18].

Table II. Performance Result for Q2 in seconds on TwigStack, TwigInlab, NRCX

Table III. Performance Result for Q1 in seconds on ISX, NOK, RFX, XGrind, NRCX

As seen from table II and table III our technique NRCX gives extremely good results for root to leaf xpath query. For other of types of queries NRCX has a good scope of improvement.

VI. CONCLUSION

In this paper we proposed a non redundant compact XML...
(NRX) storage for storing XML document. The experimental result shows that our technique achieves a better compression ratio. In addition to this it provides several other desirable features. The stored data is queriable i.e. it does not require decompression. It maintains the document order and the size of structure index is small even for very large XML documents. This technique is suitable for document centric XML document which do not contain attributes and requires further work for data centric XML documents having a large number of attributes.

REFERENCES


