Efficient Querying of Structure and Contents for Xml Documents

Atul D. Raut
I. T. Deptt. J.D.I.E.T. Yavatmal

M. Atique
P.G. Deptt of C. S.
S.G.B.A.U. Amravati

ABSTRACT
XML is recognized as a standard for data storage and exchange for web applications. This is because it has certain unique features like it is self describing, extensible and it is stored in the form of text document. In spite of all these unique features XML has an inherent limitation of verbosity. Because of the strong presence of XML in database technology and its inherent verbosity there is ever increasing need to design compact storage for XML which can be effectively utilized for efficient indexing and querying of XML. The proposed technique creates a structure index which is a compact summarization of the XML document and data index which groups and stores the contents of all similar paths at one place. Based on this compact storage a novel query algorithm is proposed which can answer 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.

General Terms
Indexing, Querying, Xpath expression.

Keywords
Compact storage, Structure index, Content index.

1. INTRODUCTION
XML is widely accepted standard for storage and exchange of data over the internet. In spite of its several advantages XML has a very serious drawback 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. 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. 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.[1][2] Hence the query processing algorithm or indexing /storage technique should be such that ideally it should not produce any intermediate result.

- **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 predicat**es.**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. [3][4] 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 [5][6]. 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. is 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 0. Using this indexid of 0 a file is created in the content index having a name of index0. This file index0 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:

- A compact queriable storage of XML which does not require decompression at the time of querying
- Efficient access to many xpath axes without constructing the F&B bisimilarity graph which is required in case of FIX[7]
- A non tree or non inverted list representation of XML.

The rest of the paper is organized as fallows. Section2 discusses the related work. Section 3 and 4 describes our technique. Section 5 reports the experimental results and section 6 concludes

2. 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.

- By traversing the tree or its compressed representation.
- By using IR style processing using inverted list.
- By combination of the first two. In this context structure index plays a vital role.
- By using a Relational Database Management System.
- 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 includes XMill[8], XMLPPM[9] etc. XMill achieves a good compression ratio and does not require a DTD. But the main drawback of XMill is that it requires decompression before querying. Hence the query response time is more in case of XMill. XMLPPM achieves higher compression than XMill but requires longer compression time. Inverted list techniques and techniques using relational tables to store XML data are somewhat nearer to queriable techniques as it does not require decompression before querying. The main drawback of inverted list techniques and techniques using tables is that it requires complex join operation for querying. [10][11]. When the XML data is stored in the form of relational tables a sql query is to be written which is then translated to xpath query.

Xgrind, Xpress, XQuec, Tbitmap, PCIDindex, RFX, ISX ,Twig-Inlab[19],etc are some of the queriable XML compression techniques. Xgrind has lower compression as compared to Xmill. Both Xgrind and Xpress requires two scans over the document and hence requires longer compression time. Moreover both of these techniques does not support set based queries such as the join queries. [12][13]XQuec's compression factor is slightly inferior to that of Xgrind and comparable to that of Xpress but XQuec has grater query capabilities than Xgrind.[14] The Tbitmap technique represents XML data in the form of a tree and assigns a bitmap to nodes of the tree. If the number of nodes increases the tbitmap requires more space and processing time.[15] PCIDindex uses dewey id for nodes. Again the dewey id of node becomes very long if the number of nodes in the tree increases considerably. This results in requirement of more memory space.[16] ISX is compact XML storage engine to store XML in a more concise structure, but it is a schema aware storage technique.[18] Different indexing and querying technique can be compared on the basis of various factors such as intermediate results size, index size, response time, I/O required, flexibility etc .

3. 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 by considering the attributes and its values on the path and assigns it a unique id.

Following is the algorithm for constructing the path index

Algorithm PathIndex

1. xmd ← Read the XML document
2. xmn ← Root node of xmd
3. count ← Number of nodes of xmn
4. top ← 0

// Store the children of root node on stack

5. For i = count -1 to 0

   5.1 stack(top) = xmn.childnodes.index(i)

   5.2 top ← top + 1

End For

```
6.  While count > 0
6.1 xmn ← stack.pop()
      // Process first child node of root node
   6.2 temp  ← xmn
   6.3 count1 ← temp.childnodes.count
//push all the child nodes of temp on stack
         6.4  For i = count -1 to 0
                  6.4.1  stack(top) = temp.childnodes.index(i)
   6.4.2  top ← top + 1
End For
   // traverse the current node back up to the root node
6.5 While temp.parent < > # document
      6.5.1 tstack(top1)=temp.parent
      6.5.2 temp=temp/parent
    End While
   //From root node traverse up to the current node remembering the path
6.6.  temp2← tstack.pop()
6.7   t← "/"+ temp2.localName
6.8   While tstack.count() > 0
        6.8.1  temp2← tstack.pop()
        6.8.2   t← t+ temp2.localName
      End While
  // From current node traverse up to the leaf node of current node
6.9   While xmn.HasChilNodes = True
        6.9.1 xmn← xmn.FirstChild
        6.9.2  t← t+ "/"+ xmn.localName
      End While
  //  Store the newly found path in a path array

6.10 path(u)← t
6.11 nx←u
  // Mark the repeated path as "R"
6.12 u← u+ 1
6.13   While nx > 0
      6.13.1  If path(u-1).equals(path(nx-1))
                 Path(u-1)← "R"
                 Exit While
               End If
       6.13.2 nx← nx- 1
    End While
End While
// Steps to eliminate repeated paths

7.0  For v← 0 to u-1
       7.1 If path(v1) NotEqualsTo "R"
      // copy the unrepeated path to another array
          L(loc) = path(v1)
          loc =loc + 1
        End if
    End For
 End While
End PathIndex
```

The algorithm path index reads the xml document and finds the total number of child nodes of the root node.(steps 1 to 3). 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. (step5) The first child node of root node is poped out from the stack and all its child nodes are pushed on to the stack in reverse order.(step6.4) In order to store the root to leaf path of the current node , the current node is traversed back to the root node (step 6.5), again from root node to current node remembering the path in t(step 6.6.to 6.8) and finally from current node to the leaf node of the current path giving a complete root to leaf path of the current node. (step 6.9)

This path is stored in the path array ptah(). 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") .(step (6.10 to 6.13)). Finally all the path marked as "R" are removed and only the unique paths are left in the path array l().(Step 7). The id of the path is equal to the index location i.e. the first path gets an id of 0 ,the second path gets an id of 1and so on.

Consider the following fragment of a sample XML document which illustrates the creation of the path index.

```
<studentdata>
    <student @university=amt>
        <college>jdiet</college>
        <deptt> it </deptt>
        <year> iii </year>
       <name>abc</name>
   </student>
<student @university=nag >
        <college>dbnce</college>
        <deptt> cse </deptt>
        <year> ii </year>
       <name>xyz</name>
   </student>
</studentdata>
```

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

Path	Id
/studentdata/student@university=amt/college	0
/0/deptt	1
/0/year	2
/0/name	3
/studentdata/student@university=nag/college	4
/4/deptt	5
/4/year	6
/4/name	7

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@university=amt/college is stored at one place and can be directly accessed using its id which is 0 in this case. The size of the path index is reduced by removing the repeated component in a path. For example for the second path the path components till deptt is same as that of the first path till college component. Hence the repeated component from the second path is removed and only a 0 is placed to get the required component during query processing. Such grouping of data based on the path gives rise to content index. Following is the algorithm for constructing the content index.

Algorithm ContentIndex

1. xmd ← Read the XML document
2. xmn ← Root node of xmd
3. count ← Number of nodes of xmn
4. top ← 0

// Store the children of root node on stack

5. For i = count -1 to 0
   a. stack(top) = xmn.childnodes.index(i)
   b. top ← top + 1

End For

6. While count > 0

6.1 xmn ← stack.pop()

// Process first child node of root node

6.2 temp ← xmn

6.3 count1 ← temp.childnodes.count

//push all the child nodes of temp on stack

6.4 For i = count -1 to 0

6.4.1 stack (top) = temp.childnodes.index(i)

6.4.2 top ← top + 1

End For

// traverse the current node back up to the root node

6.5 While temp.parent < > # document

6.5.1 tstack(top1)=temp.parent

6.5.2 temp=temp/parent

End While

//From root node traverse up to the current node remembering the path

6.6. temp2← tstack.pop()

6.7 t← "/"+ temp2.localName

6.8 While tstack.count() > 0

6.8.1 temp2← tstack.pop()

6.8.2 t← t+ temp2.localName

End While

// From current node traverse up to the leaf node of current node

6.9 While xmn.HasChilNodes = True

6.9.1 xmn← xmn.FirstChild

6.9.2 t← t+ "/"+ xmn.localName

End While

// load the path index in an array of strings path() from the path index file

// created by the above algorithm

6.10 path()←load path index from path index file

// For the newly found path compare it with the paths in path index

//if match is found get the index id and store the data of leaf node in document whose name has index as the prefix and index id as the suffix

6.11 nx←path.length

6.12 For q← 0 to nx-1

6.13 If path(q) = t

6.13.1 indexid← q

// write the contents to content index file

6.13.2 write ("index"+indexid , t.innertext)

6.13.3 Exit For

End If

End For

End While

End ContentIndex

The algorithm Contentindex reads the xml document and finds the total number of child nodes of the root node.(steps 1 to 3). 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. (step5) The first child node of root node is poped out from the stack and all its child nodes are pushed on to the stack in reverse order.(step6.4) In order to get the root to leaf path of the current node , the current node is traversed back to the root node (step 6.5), again from root node to current node remembering the path in t(step 6.6.to 6.8) and

finally from current node to the leaf node of the current path giving a complete root to leaf path of the current node. (step 6.9)

After getting the first path this path is compared with the paths in pathindex 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(step 6.13). 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.

Path id	Contents	file name
0	jdiet	index0
1	it	index1
2	iiii	index2
3	abc	index3
4	dbnce	index4
5	cse	index5
6	ii	index6
7	xyz	index7

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@university=amt/college, this path appears once 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=0) the contents are stored and can be accessed directly. The contents on this path are jdiet, and most importantly these contents are stored in document order at one place without repeating the path information.

4. QUERYING OF XML DOCUMENTS

In this section we present the algorithm Evalpath which effectively utilizes the structure and content index for finding quick response to XML queries.

Algorithm Evalpath

1. Load the structure index file
2. query = Read the xpath expression to evaluate
3. qtokens = tokens of the input query which are separated by /or //
4. sep = all separators of the query / or //
5. nt = No. of tokens in query excluding / or //
6. indexid=-1
7. Repeat for i= 0 to n-1 paths in the structure index

7.1 j=0, lt=0

7.2 If indexid != -1then /* indexid already found

7.2a Exit for

7.3 strindex = read the i^{th} path

7.4 sitokens = tokens of the i^{th} path

7.5 While lt < nt

7.5.1If sep(j)='/'&& qtokens(j)= sitokens(j) then

a. j = j + 1

b. lt = lt + 1

End if

7.5.2 If sep(j) = '//'

a. indexed = obtain index of qtokens(j) in strindex

b. j = j + 1, lt = lt + 1

End if

7.5.3 If indexid != -1 then

indexid= i

Else

Exit while

End if

End While

Next i

8. If indexid > 0 then

Load the data file with name equal to indexid.

Else

Print : No such path in document

9. Exit

The algorithm Evalpath separates the input xpath expression into tokens qtokens considering / and // as the delimeters (step3). Step 4 stores the separators i.e. / and // in the array sep. Step 5 counts the number of tokens in the input xpath query excluding the separators / and //. Step 7 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 respective tokens of the

input xpath query then indexid is assigned the value of I and the inner while loop (step 7.5) and also the outer for loop (step 7.2) terminates. If any token of the current path in the structure index does not match with the token of input xpath expression (step 7.5.3 else part) then the inner while loop terminates and the same process is applied to the next path in the structure index. If all the paths in the structure index do not match with the xpath expression then the outer for loop completes execution (step 7) and index id retains the value of -1. Once the indexid is obtained all the data for this path can be obtained in minimum number of I/O since the data is present in a single file. Thus the compact storage of XML not only reduces the storage size but also provides efficient querying of XML documents.

5. EXPERIMENTAL RESULTS

5.1 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 [17].

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 fully specified i.e. root to leaf 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.

5.2 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

Table I. Storage Size Required of RFX, ISX, XMill, Xgrind, NRCX

Bench Mark Database	Source Data (MB)	RFX (MB)	ISX (MB)	XMILL (MB)	XGRIND (MB)	NRCX (MB)
Orders	5.1	3	3	0.5	1.3	1.821
Shakespeare	7.5	5.1	5.3	0.9	2.1	3.7
Lineitem	30.8	15.8	21	3.7	8.6	7.021
Treebank	84	NA	NA	NA	NA	45

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

Bench Mark Database	Source Data (MB)	TwigStack (seconds)	TwigInlab (seconds)	NRCX (seconds)
Treebank	84	80	63	03

Table III Performance result for Q1 in seconds on ISX,NOK,RFX,NRCX

	1MB	16MB	64MB	128MB
ISX	0.001	0.021	0.13	0.85
NOK	0.005	0.015	0.76	1.25
RFX	0.001	0.013	0.087	0.21
NRCX	0.001	0.013	0.081	0.29

The results of other techniques for query Q1 is taken from the reference cited by [17], and the results of other techniques for query Q2 is taken from the reference cited by [19].

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.

6 . CONCLUSION

In this paper we proposed a non redundant compact XML (NRCX) 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 and requires further work for data centric XML documents having a large number of attributes

7 REFERENCES

[1]Dayanand P, Dr.Rajashree Shettar. "Survey on Information Retrieval in Semi Structured Data," *International Journal of Computer Applications*, vol 32 ,no 8, pp 1-5, Oct 2011.

[2] S. Al. Khalifa, H. V. Jagdish, N Koudas, J. M. Patel, D Srivastava and Y Wu. "*Structural Joins: A Primitive for Efficient XML Query Pattern Matching"*, Proc. of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, pp. 141-152, February 26-March 1, 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava. *Holistic Twig Joins: Optimal XML Pattern Matching*, Proc. of 21st ACM SIGMOD Int"l Conference on Management of Data (SIGMOD"02), pp. 310–321, 2002.

[4]Ibrahim Dweib, Ayman Awadi and Joan Lu. "MAXDOR: Mapping XML Document into Relational Database," *The Open Information System Journal*, vol. 3, pp. 108-122, June 2009

[5]Peter Bunaman , Martin Grohe, Christioph Koch. *Path Queries on Compressed XML*, Proc. of the 29 thVLDB conference, Berlin Germany ,2003.

[6]Raghav Kaushik, Rajasekar Krishnamurthy, Jeffery F. Naughton, Raghu Ramkrishnan. *On the Integration of structure Index and Inverted Lis*t, Proc. of the 204 ACM SIGMOD international conference on Management of data, Paris, France, pp.779-790, June 13-18 2004.

[7] Ning Zhang , M. Tamer. *et.al.* "FIX: Feature-based Indexing Technique for XML Documents" in Pro. 32 nd VLDB conference, Seoul, Korea,2006.

[8] H. Liefke and D. Suciu, "XMill: an efficient compressor for XMLdata," in *ACM SIGMOD international conference on management of data pages*, 2000, pp. 153-24.

[9] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models," in *Proceedings of the IEEE Data Compression Conference*, 2000, pp. 163-172

[10]Zhuyan Chan, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundram, Divesh Srivastava, " Index Structures for Matching XML Twigs using Relational Query Processor,"in *Proceeding of 21 st International Conference on Data Engineering Workshop ICDEW* ,Tokyo-Japan, pp 1273-1283,5-8 April 2005.

[11]Igor Totarinov, Stratis D Vigals, Kevin Beyer, Jayavel Shanmugasundram, Eugene Shekita, Chun Zhang, " Storing and Querying Ordered XML using a Relational Database System," in *Proceeding of ACM SIGMOD Int'l Conference on anagement of Data*, Madison Wisconsin USA, pp. 204-215, June 3-6 *2002*.

[12] P. Tolani and J. Haritsa, "XGRIND: A query-friendly XML compressor,"in *18th International Conference on Data Engineering (ICDE)* IEEE Computer Society, 2002, pp. 225-234.

[13] J. Min, M. Park and C. Chung, "XPRESS: A queriable compressionfor XML data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California,2003.

[14] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese, "XQueC: Pushing queries to compressed XML data," in *Proceedings of the 29th International Conference on Very LargeData Bases (VLDB'03)*, 2003.

[15]Yin Fu Huang and Shin-Hang Wang, " An efficient XML Processing based on combining T bitmap and Index Techniques," in *Proceeding of IEEE Symposium on Computers and Communication ISCC 2008*, Marrakech, Morocco, pp 858-863, July 6-9 2008.

[16]Li Ying, MaJun Sun Yun, "Applying Dewey Encoding to Construct XML Index for Path and Keyword Query," in *Proceeding of First International Workshop on Database Technology and Application 09*, Wuhan, Hubie,China, pp553-556, 25-26 April 2009.

[17]Radha Senthilkumar, Priyaa Varshinee and A. Kannan. "Designing and Querying a Compact Redundancy Free XML Storage," *The Open Information System Journal*, vol. 3, pp. 98-107, June 2009.

[18] R. Wong, F. Lam and W. Shui, "Querying and maintaining a compact XML storage," in *16thinternational conference on World Wide Web*, Banff, Alberta, Canada, 2007.

[19]Su-Cheng Haw and Chien-Sing Lee, "Structural Query Optimization in Native XML Databases : A Hybrid Approach," *Journal of Applied Sciences*, vol20, pp 2934-2946,2007