Indexing and Querying Semistructured Data Views of Relational Database

B.M. Monjurul Alom, Frans Henskens and Michael Hannaford

School of Electrical Engineering. & Computer Science, University of Newcastle, AUSTRALIA

Summary
The most promising and dominant data format for data processing and representing on the Internet is the Semistructured data form termed XML. XML data has no fixed schema; it evolved and is self describing which results in management difficulties compared to, for example relational data. XML queries differ from relational queries in that the former are expressed as path expressions. The efficient handling of structural relationships has become a key factor in XML query processing. It is therefore a major challenge for the database community to design query processing techniques and storage methods that can manage semistructured data efficiently. The main contribution of this paper is querying semistructured data using bitmap to represent path-value relationship and compress the bitmap to save space. The presented bitmap indexing and querying scheme termed BIQS data that stores the element path, token of the word, attribute and document number in a dynamically created matrix structure. We use word, attribute and path dictionaries for the construction of a Bitmap structure. This paper describes an algorithm to query semistructured data in a more time efficient way than is provided by other relational and semistructured query processing techniques. The presented BIQS structure provides storage and query performance improvement due to the compression of semistructured data.

Key words:
Structural Join, XQuery, XPath, Bitmap, TwigStack, MySQL.

1. Introduction

Query processing is an essential part of any type of databases as well as Semistructured (XML) databases [1]. Semistructured data does have some structure, but this structure is not as rigid, regular, or complete as the structure required by traditional database management systems [20]. The use of XML as a semistructured data format is becoming more prevalent, specially when performing tasks such as the simple integration of data from multiple sources [21]. The growth of XML repositories on the Web has led to much research on storing and indexing for efficient querying of XML data.

One option for managing semistructured, as well as XML, data is to build a specialized data manager that contains a XML data repository at its core [22]. It is difficult to achieve high query performance using XML data repositories, since queries are answered by traversing many individual element-to-element links requiring multiple index lookups [23] . In the case of XML data, queries are even more complex because they may contain regular path expressions [24]. Thus additional flexibility is needed in order to traverse data whose structure is irregular or partially unknown to the user. Another option for managing semistructured data is to store and query it with a relational database [22]. In the database community many researchers argue that the relational (and object-relational) model, due to its maturity and widespread usage, is still the best option [25].

XML query processing is much more complicated than traditional query processing methods because of the structure of XML [1]. A path expression specifies patterns of selection predicates on multiple elements related by a tree structure named Query Tree Pattern (QTP). Consequently, In order to process an XML query, all occurrences of its related QTP should be distinguished in the XML document. This is an expensive task when huge XML documents are attended.

The well known query processing method termed as Structural Join is described in [2]. In Structural Join, query is decomposed into some binary join operations. Thus, a huge volume of intermediate results are produced in this method. The Holistic twig join approach [3] do not decompose the query into its binary Parent-Child (P-C) or Ancestor-Descendant (A-D) relationships but they need to process all of the query nodes in the document. The query processing method termed TJFast [12] which only process elements which belong to the leaves of QTP instead of processing all the nodes in the XML document. But this method use a structure named Finite State Transducer (FST) for decoding the code of nodes into the same name of the path traversed from the root for each node, so FST waste a lot of time.

The contribution of this paper is querying semistructured data using bitmap to represent path-value relationship and compress the bitmap to save space. The presented BIQS supports the Structural Join query, Path query, and Tree structure query which are processed by joining path results. The BIQS technique also supports the

Manuscript received June 5, 2010
Manuscript revised June 20, 2010

type of query where only a portion of the path name is mentioned in the query. The paper presents the comparison of query execution time of BIQS to the other XML query processing time (Structural Join and TwigStack) and relational query (Oracle, MySQL) processing time.

Experimental results show that the proposed technique queries the semistructured data in a time efficient way than is provided by some of the other existing XML and relational query processing techniques. The paper presents the "time and space" complexity of the proposed BIQS algorithm. The paper also addresses the issue of relational (Semistructured data) querying using compressed bitmap structure, the word, path, and attribute dictionaries.

The bitmap structure provides the facility to store huge information of words and paths into each cell of a single block to compress the data. This compression leads the data retrieval can be performed efficiently with low latency. To understand the functionality of the proposed technique, the algorithm shows the storing of sixteen words and paths information into each memory cell of a single block by a decimal value for the data compression. But the compression is possible for the presented structure upto the information of thirty two words and paths into each memory cell of a single block. No lose of any XML information is always maintained for the proposed techniques.

The remainder of this paper is organized as follows: Related work in section 2, a framework of the proposed method is described in section 3. The algorithm for Bitmap Structure is presented in section 4. Searching and querying documents is described in 5. Section 6 experimental results. The paper concludes with a discussion and final remarks in section 7.

2. Related Work

Many query processing techniques such as Holistic Twig Join methods have been proposed in [6, 8, 13, 18] to process a twig query efficiently; however, they still suffer from large number of redundant function calls. A new approach termed TwigStack+ is proposed in [19] to solve this problem, which based on holistic twig join algorithm that greatly improve query processing performance. The TwigStack+ technique is used to reduce the query processing cost, simply because it can check whether other elements can be processed together with current one. The proposed technique also used to check the usefulness of an element from both forward and backward directions. Different XML query processing techniques are also elaborated in [4, 7, 9, 11, 15].

TSGeneric+[6] made improvements on TwigStack by using XR-Tree to skip some useless elements which have Solution Extensions but cannot participate in any path solution. TwigStackList [8] handles the sub-optimal problem by attaching an element list to each query node to cache some elements, TJFast [12] improved the query processing performance by scanning elements of leaf nodes in the query to reduce the I/O cost. Although the existing methods [6] can guarantee the optimality of CPU time and I/O when only AD edges involved in the twig pattern, they all suffer from large number of redundant function (getNext(root) calls.

A query processing and update processing method termed EXEL (Efficient XML Encoding and labeling) is presented in [10].

SIGOPT (schema information graph) to optimize XML query processing is described in [17]. The presented technique explores the opportunities for schema information to affect the query evaluation performance. Multi-level operator combination in XML query processing is described in [16] which elaborates the importance to consider the operations at each level. Specifically, the technique considers the influence of projections and set operations on pattern-based selections and containment joins.

Database management systems support indexing to provide better query performance. Indexing provides a flexible, uniform and efficient mechanism to access data [22]. There are some path indexes like Strong DataGuide[26], Fabric Index, ToXin[27], APEX[28], Index1[24] , A(k) Index, and Fix[29] which are indexing the path of document's nodes to facilitate access to nodes required in XML query processing methods. These path indexes are other kinds of query processing methods which are against the Structural Join[2], Twig Join[3] and TJFast[12] methods.

Most of the indexing schemes can only be applied to some limited query processing stages or limited class of queries. To overcome these limitations an indexing scheme called ToXin [27] has been developed. ToXin fully exploits the overall path structure of the database in all query processing stages, consisting of Path index and value index. A three dimensional bitmap indexing scheme named Bitcube [30] considers a more complex frequency table that represents a set of documents together with both a set of element paths and a set of words for each path. A new system for indexing and storing XML data based on a numbering scheme for elements is proposed in [11].

Query capabilities are provided through Structural Join and Twig queries, which are the core components of standard XML query language, e.g. XPath[31] and XQuery[32]. Techniques also exist for querying XML data such as Lorel[21], XML-QL[33] , XQL[34], UnQL[35], XML-GL[34], XSL[34], Quilt[25]; however these query languages are complicated to use and have some limitations. A comprehensive effort has been done on storing and querying XML data using relational databases are described in [4, 7, 9, 15, 36-40] [23, 36-48] also a comprehensive effort has been done on XML data compression are presented in [22].

3. Framework of the Proposed Technique

3.1 Overall Architecture of the System

To understand the functionality of the proposed technique the overall architecture is presented in Fig. 1. Data processing engine is used to generate a word dictionary, a path dictionary, and an attribute dictionary, which together become the basis of a bitmap matrix to store XML document information. The path elements are calculated from root to nested sub element among all the XML documents. The attribute dictionary records all the attributes (not distinct attributes) including the content of each attribute and the corresponding document number. The word dictionary records a token number for each distinct word. The path dictionary stores all the distinct path elements including their path number.

Multi block compressed structure stores all the raw information in compressed form. Token and Path (TP) structure are used to represent the token and path number. Secondary indexing is used for searching the token and path number from the Token and Path structure to reduce the search time. The compressed structure with dictionary and TP (Token Path structure) are maintained on main memory. Input query through query manager is applied to the compressed structure to obtain the output query.

The developed structure is not always same, if the set of documents are considered but in different order. Different order of the documents provides the differentiation of the structure that does not mean the structure loses some XML information. The structure always maintains the exact information of the original XML database whether the set of documents are considered in different order or same order. For any order of the documents, the data are stored in a multi block compressed structure, which leads the searching efficiency. Also for the use of dynamic matrix structure the efficiency of the updating is not degraded.

3.2 Construction of Bitmap Structure

BIQS dynamically creates a two dimensional matrix structure which represents the existence of all the words and element paths in the corresponding documents. The first row of the matrix structure records all the token numbers for the corresponding words and the associated path numbers for the words. All the tokens exist (are bounded) within their corresponding path numbers in the first row of the BIQS structure.

. We use a negative (-) sign before all path numbers to differentiate them from token. The first column of the matrix stores the document number. The entries of the matrix use a bit value (1/0) to represent the existence or not of the word and element path within the document number. To represent a new path from an XML document, this approach initially generates a new column in the matrix structure. The first row (entry) of the column stores the path number (from path dictionary) and a value 1 is inserted to the next row of the created column. The value 1 denotes a path’s existence in the document. The tokens (from word dictionary) of all the words within the selected path number are stored similarly by creating new columns in the matrix structure.

A value 1 is inserted into the next entry of each of the created columns for the token. Each row of the matrix structure records all the information of each distinct XML document. The system similarly completes the matrix creation for all XML documents. BIQS does not create new columns within an existing path, for the same word even for different documents. The technique always creates new columns for the same word but different path number, whatever the document number. We consider the XML documents given in Fig. 2, Fig. 3, Fig. 4, and Fig. 5 for use in demonstrating our proposed bitmap construction.

3.3 Construction of Dictionary and BIQS with Example

The word dictionary, path dictionary and attribute dictionary (comprising Tables I, II and III) have been created from XML documents shown in Fig. 2, Fig. 3, Fig. 4 and Fig. 5. The attribute dictionary, given in Table III, shows that an attribute named *key* has four different values in different documents such as 2 and 4. In path dictionary, *nasa.datasets.dataset.title* and *dblp.msthesis.title* represent two different path number.

The system creates a new column in the matrix structure (given in Table IV) to record the path name “*nasa.datasets.dataset*” from document$_1$ (given in Fig. 2), and the path number (-1) is assigned to the first row of the created column, and a value 1 is assigned to the next row of the created column to indicate existence of the document$_1$. There are no words within this path number except some attributes. Therefore no token is updated within this path number. Similarly, for the path number 2, a new column is created in the structure. For all the words within this path number, a new separate column is created and the value 1 is assigned to the next row of the created column indicating their existence to the corresponding document. Thus the token 1 for the word *ProperMotion* is recorded within the path number 2. The value 1 is assigned to the next row of the created column indicate their existence to the corresponding document. In Table IV, 1 is the token within path number 2. The created matrix structure after extracting all the words and paths from document$_1$ to document$_4$, is given in Table V.

3.4 Methodology to Compress the Bitmap Structure

The system separates the BIQS structure into two structures to compress the XML data. The first row is one structure named *Token and Path structure* used to represent the token and path number. This row is indexed serially starting from 0. Later, these indexes are used for searching the token and path number from the Token and Path structure. Another structure named *Compressed (BIQS) Bitmap Index Structure* consists of all other remaining rows of the matrix (except the first row).

In this structure each row is divided into blocks. In each block, the information of 16 (words and paths) bit cells is compressed. As each row represents the information of each XML document, there may be a different number of blocks for each document and each block consists of different values for different documents. The compression is also possible using 32 bit cells. The token and path structure is presented in Table VI. The compressed bitmap structure is presented in Table IX. The value of each 16 bit cells is recorded in decimal form. If there is not enough of the data to form a block with 16 bit cells, we fill these bits with zero.

The compressed BIQS structure is given in Table VIII; the first column of the structure represents the document number and the other three columns represent the blocks. The value of each block is generated from the BIQS structure given in Table V. The values of the Block$_0$ are 65472, 57, 0 and 39. These values represent the compressed information of data for different XML documents. This compression does not lose any information. We use the compressed BIQS structure for searching the data. Realistically, we are not converting the binary values (from Table V) into decimal values (into Table VIII) rather than we store the information for 16 words and paths into a single cell of a block.

The system separates the BIQS structure into two structures to compress the XML data. The first row is one structure named *Token and Path structure* used to represent the token and path number. This row is indexed serially starting from 0. Later, these indexes are used for searching the token and path number from the Token and Path structure. Another structure named *Compressed (BIQS) Bitmap Index Structure* consists of all other remaining rows of the matrix (except the first row).

In this structure each row is divided into blocks. In each block, the information of 16 (words and paths) bit cells is compressed. As each row represents the information of each XML document, there may be a different number of blocks for each document and each block consists of different values for different documents. The compression is also possible using 32 bit cells. The token and path structure is presented in Table VI. The compressed bitmap structure is presented in Table IX. The value of each 16 bit cells is recorded in decimal form. If there is not enough of the data to form a block with 16 bit cells, we fill these bits with zero.

The compressed BIQS structure is given in Table VIII; the first column of the structure represents the document number and the other three columns represent the blocks. The value of each block is generated from the BIQS structure given in Table V. The values of the Block$_0$ are 65472, 57, 0 and 39. These values represent the compressed information of data for different XML documents. This compression does not lose any information. We use the compressed BIQS structure for searching the data. Realistically, we are not converting the binary values (from Table V) into decimal values (into Table VIII) rather than we store the information for 16 words and paths into a single cell of a block.

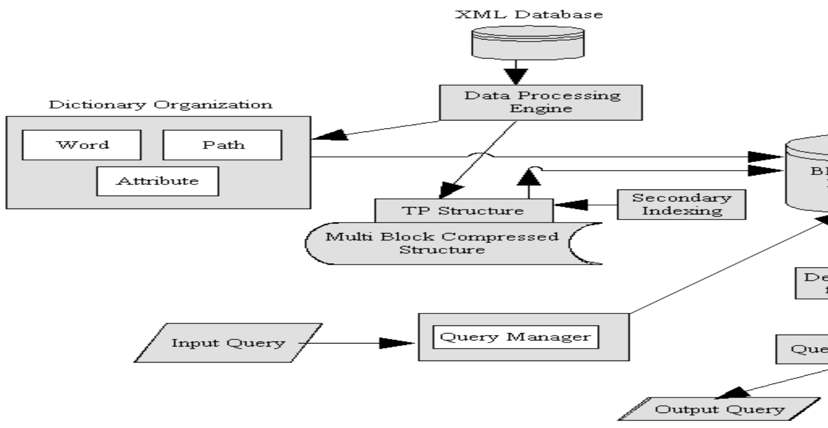


Fig. 1 Architecture of the Query Processing Methodology.

```
<nasa>
<datasets>
<dataset subject="astronomy" xmlns:xlink="http://www.w3.org">
        <title> ProperMotions </title>
         <altname type="ADC">I/1005 </altname>
         <altname type="CDS">I/5 </altname>
         <author>
               <firstname>Jack</firstname>
               <lastname>Spencer</lastname>
          </author>
   </dataset>
 </datasets>
 </nasa>
```

Fig. 2: XML document-1

```
<dblp>
    <msthesis   key="ms/Brown92">
        <author>Brown </author.
        <title> DB System </title>
        <year>1992</year>
       <school>Madison</school>
    </msthesis>
    <msthesis key="ms/Yurek97">
         <author>Yurek</author>
         <title>DataWarehouse</title>
         <year>1997</year>
         <school>california</school>
     </msthesis>
 </dblp>
```

Fig. 3: XML document-2

```
<Yahoo>
<listing>
      <seller_info>
             <seller_name>Katich</seller_name>
             <seller_rating>new</seller_rating>
       </seller_info>
       <item_info>
              <memory>128MB RAM</memory>
              <HD>40GB</HD>
              <cpu>Pentium-III</cpu>
       <item_info>
 </listing>
 <listing>
        <item_info>
              <memory>256MB RAM</memory>
              <HD>80GB</HD>
              <cpu>Pentium-IV</cpu>
       <item_info>
</listing>
</Yahoo>
```

Fig. 4 :XML Document-3

```
<dblp>
    <msthesis   key="ms/Korth94">
        <author>Korth</author.
        <title> DataMining </title>
        <year>1994</year>
       <school>MIT</school>
    </msthesis>
    <msthesis key="ms/Martin98">
         <author>Martin</author>
         <title>DSP</title>
         <year>1998</year>
         <school>Texas</school>
     </msthesis>
 </dblp>
```

Fig. 5: XML document-4

Table 1. Word dictionary.

Word	Token
ProperMotions	1
I/005	2
I/5	3
Jack	4
Spencer	5
Brown	6
DB	7
System	8
1992	9
Madison	10
Yurek	11
DataWarehouse	12
1997	13
california	14
Katich	15
new	16
128MB	17
40GB	18
Pentium-III	19
256MB	20
80GB	21
Pentium-IV	22
Korth	23
Datamining	24
1994	25
MIT	26
Martin	27
DSP	28
1998	29
Texas	30

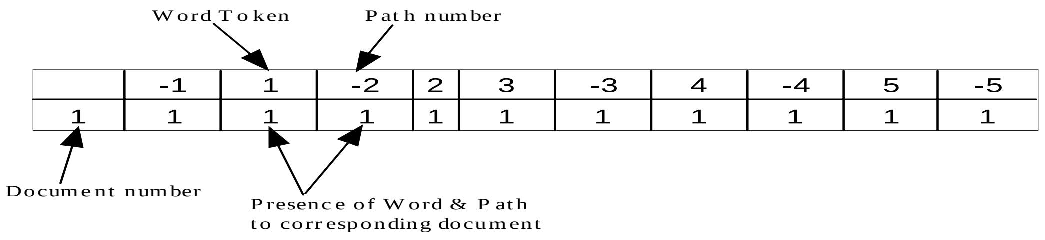
Table 2. Path dictionary.

Path Number	Path
1	nasa.datasets.dataset
2	nasa.datasets.dataset.title
3	nasa.datasets.dataset.altname
4	nasa.datasets.dataset.author.firstname
5	nasa.datasets.dataset.author.lastname
6	dblp.msthesis
7	dblp.msthesis.author
8	dblp.msthesis.title
9	dblp.msthesis.year
10	dblp.msthesis.school
11	Yahoo.listing.seller_info.seller_name
12	Yahoo.listing.seller_info.seller_rating
13	Yahoo.listing.item_info.memory
14	Yahoo.listing.item_info.HD
15	Yahoo.listing.item_info.cpu

Table 3. Attribute dictionary.

Attribute Name	Content	Doc Number	Path Number
subject	astronomy	1	1
xmllns:xlink	"http://www.w3.org"	1	1
type	ADC	1	3
type	CDS	1	3
key	ms/brown92	2	6
key	ms/yurek97	2	6
key	ms/korth94	4	6
key	ms/Martin98	4	6

Table 4: The Structure after completing Document-1



	-1	1	-2	2	3	-3	4	-4	5	-5
1	1	1	1	1	1	1	1	1	1	1

Table 5: BIQS Structure.

	-1	1	-2	2	3	-3	4	-4	5	-5	-6	6	11	23	27	-7	7	8	12	24	28	-8	9	13	25	29	-9
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	1	1	1	0	0	1	1	1

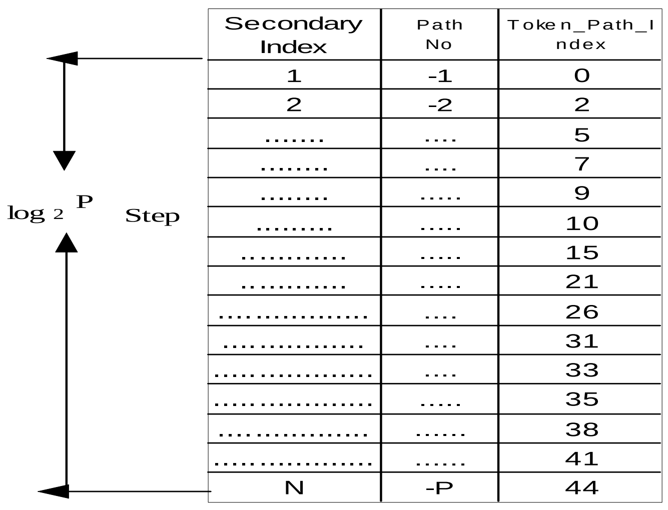
	10	14	26	30	-10	15	-11	16	-12	17	20	-13	18	21	-14	19	22	-15
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
4	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6: Token and Path Structure.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
	-1	1	-2	2	3	-3	4	-4	5	-5	-6	6	11	23	27	-7	7	8	12	24	28	-8	9	13	25	29	-9

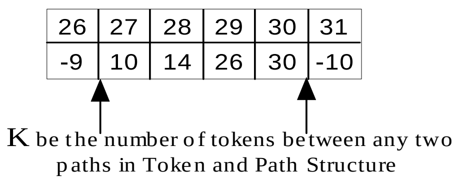
27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
10	14	26	30	-10	15	-11	16	-12	17	20	-13	18	21	-14	19	22	-15

Table 7: Path Searching from Token and Path Structure.



Secondary Index	Path No	Token_Path_Index
1	-1	0
2	-2	2
....	....	5
....	....	7
....	.....	9
....	.....	10
....	.....	15
....	.....	21
....	....	26
....	....	31
....	....	33
....	.....	35
....	......	38
....	......	41
N	-P	44

Table 8. Token searching between two paths.


K be the number of tokens between any two paths in Token and Path Structure

26	27	28	29	30	31
-9	10	14	26	30	-10

Table 9. Compressed BIQS Structure.

Document Number	Block-0	Block-1	Block-2
1	65472	0	0
2	57	59193	0
3	0	0	65528
4	39	7399	0

.

4. Algorithm of the Technique

We have developed an algorithm shown in Fig. 6. To understand the algorithm the following data structures are necessary:
DPath: Distinct Path; *DPNumber:* Distinct Path Number; *PDic[][]:*Path Dictionary; *DocNumber:* Document Number; *DWord:* Distinct Word; *WDic[][]:* Word Dictionary; *ADic[][]:* Attribute Dictionary; *Att:* Attribute; *NPath:* New Path; *BMS:* BIQS Matrix Structure; *NC*: New Column; *PIndex*: Path Index; *NPIndex:* New Path Index; *FR:* First Row; *TNumber:* Token Number; *WNPIndex:* Word New Path Index; *CBS:* Compressed Block Structure; *CIndex:* Compressed Index; *TNDC:* Total No of Dynamic created Column in BMS; *WPIndex:* Word Path Index; *BIndex:* Block Index; *BinDec():* Binary to Decimal; *SIndex:* Secondary Index; *CPnumber:* Current Path Number; *IPCPath:* Immediate Previous Path of Current Path; *BN:* Block Number; *IToken:* Index of the Token; *OPos:* Offset Position; *BN:* Block Number; *TP:* Token Path Structure; *WPInfoVal:* Word Path Info Value.
Dictionary_Construction (): This function is used to create the word, path and attribute dictionaries.
Dynamic_Matrix_Structure(): This function constructs the matrix structure including all tokens, paths, and attributes with their associated documents.
Searching_Structure(): This function describes the compression of the XML information, division into blocks, and storage of XML data into the compressed bitmap structure.
Index: Is used to store the index number of the searching token from Token and Path structures.
//Block_no: Each row has multiple block_no; each block consists of a16 bit information cell.
//Offset position: Determines the position for the existence of word or path or attribute in the //document.

```
Algorithm BIQS()
Begin
      Dictionary_Construction();
      Dynamic_Matrix_Structure();
      Searching_Structure();
End.
Dictionary_Construction()
 Begin
Calculate All the DPath from root to
nested sub-element;
 For each Dpath do
Begin
  PDic[PIndex][1]=DPNumber;
  PDic[PIndex++][2]=DPath;
End
For each DWord do
Begin
  WDic[WIndex][1]=DWord;
  WDic[WIndex++][2]=WTNumber;
End
For each Att do
Begin
        ADic[AIndex][1]=AttName;
      ADic[AIndex][2]=AttContent;
      ADic[AIndex][3]=DocNumber;
      ADic[AIndex++][4]=PNumber;
 End
     End. // Word_Path_Attribute


Dynamic_Matrix_Structure()
 Begin
For each NPath do
Begin
Create a NC in BMS;
//store the path number in the first
row of the created column;
//Store the negative sign before the
path number;
//Insert 1 to the next row of the
created path number;
        BMS[NPIndex][FR++]=-PNumber;
        BMS[NPIndex][FR]=1;
       End
For each DWord do
Begin
//Create a new column in matrix
structure within the path number;
//Insert the token number onto the
first entry of created column;
//Insert 1 to corresponding doc..number
(next row of created column) for
created token;
 Create a NC in BMS within PNumber;
 BMS[WNPIndex][FR++]=TNumber;
 BMS[WNPIndex][FR]=1;
End

//Separate the first row of (BIQS)
Bitmap Index structure as a TP
Structure;
//For all other rows form BMS
   TPS=First Row of the BMS;
For CIndex=i+1 to DocNumber do
   Begin
      For WPIndex=1 to TNDC do
//Convert the values of 16 bit memory
cell into a decimal form;
Begin
      CBS[BIndex][Col++]=BinDec(BMS[WPI
      ndex][1]to BMS[WPIndex][16]);
End
```

```
 BIndex++;
End
End

Searching_Structure()
  Begin
   //Path number are sorted in ascending
   order //through the Token and Path
   structure
   //Use a secondary index on these path
   numbers
   //Apply Binary searching to find the
   path number from these (index)
   structures;
   //Apply Binary Searching within this
   (current) path number to immediate
   previous path //number to find the
   Index of the token from token and
   path structure;

   Search the TNumber of input query
   (word) from Wdic;
   Apply BS to find PNumber from SIndex;
   Apply BS to find IToken within
   CPnumber to IPCPath from TP;
    BN=IToken/ 16;
    OPos=IToken % 16;

//For each Block from Compressed Bitmap
Index Structure do
//If a 1 is found in the offset
position, the searching word is found
//so return the row number which is the
document number;

    For each Block of CBS do
  Begin
     WPInfoVal=DecToBin();
   If (WPInfoVal[OPos])==1){Return RNo;}
     Else {Return 0;}
          End
       End.
```

Fig. 6 Algorithm for the Proposed Technique.

5 Searching and querying the Documents

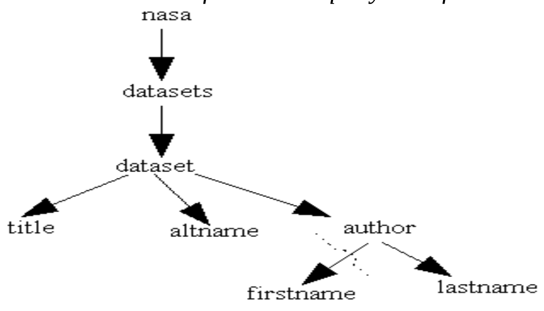
The BIQS technique supports different types of querying and searching, applied to the compressed structure of the data. Users search the word dictionary to find the token for the corresponding word; after obtaining the token, the system finds (using Binary Search technique) the index position of the token from the Token and Path structure, and the path number in which the token is bounded. Because information for 16 memory cells is compressed in each block, the Block_no is calculated, dividing the index number by 16. The Offset_position is also calculated as the index number modulo 16. From the compressed data structure (Table IX), the corresponding values of each block are converted into binary forms to check the word's existence in the document. The approach checks the existence of value 1 in the corresponding offset position in each of the block values. The presence of the value 1 in the corresponding offset position indicates the existence of the word in the corresponding document number. The system can search for a single word or for multiple words.

To search for an element-path (and path contents), the system initially searches the path number within the path dictionary and then searches all the token numbers within this path number from the Token Path structure. Whilst searching the path number from Token Path Structure, always we use Binary search technique. To get the Index of the Token, from Token and Path Structure, we also apply Binary search within this (obtained) path number and immediate previous path number. This is because all the word's token for a specific path are recoded, from the previous path number to the obtained (current) path number. As the system uses the (-) negative sign before the path number, it is easy to find the range of searching within the path numbers. After obtaining the token number, the corresponding words are searched for in the word dictionary.

If searching for an attribute (from an XML document), the technique can search directly from the attribute dictionary. In the attribute dictionary, every attribute has its content, name and corresponding document number recorded. The overall (element's content) word searching structure is shown in Fig. 7.

The functionality of the searching scheme is demonstrated in the following examples:

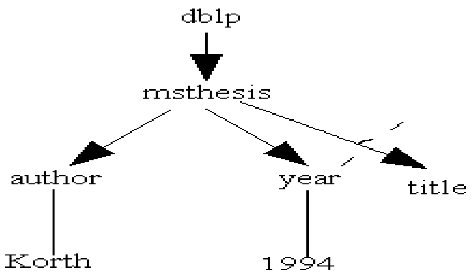
Query # 1*: Find all the author's firstname from all documents.*
The above query is represented in XPath is as follows:
/nasa/datasets/dataset/author/firstname
The tree structure of the above query is as follows:



According to BIQS the query is represented as follows:
Select nasa.datasets.dataset.author.firstname
From documents;

The system supports these types of query. It is required to find all the contents of this path name from all XML documents. The technique finds the path (nasa.datasets.datset.author.firstname) number from the path dictionary, and it is 4. The token stored within this path number (from Token and path structure, Table VI), is 4. The system searches for the words from word dictionary to the corresponding token 4, which is Jack and see it is correct from the XML documents shown in Fig. 2-5.

Query #2*: Find the title of the msthesis under dblp which have sub-element (author) containing "Korth" and sub-element (year) with a value is 1994.*
The above query is represented in XPath is as follows: /dblp/msthesis[contains(./author,"Korth")and year=1994]/title
The query tree structure of the above query is as follows:



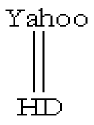
According to BIQS the query is represented as follows:
Select title
From documents
Where (dblp.msthesis.author= "Korth"and dblp.msthesis.year=1994)

The system supports these types of query. The search technique searches for the token of the words Korth, 1994 and finds it to be 23 and 25 from the word dictionary. The path number of "dblp.msthesis.author" and "dblp.msthesis.year" are 7 and 9. We see token 23 is within path number 7 and 25 is within path number 9 (from the Token and Path structure). As both conditions are true, the technique finds the index of the token 23 is 13 and 24 for token 25. So Block no=13/16=0 and 24/16=1, the system searches the values of Block-no 0 from (BIQS) Compressed Bitmap index structure. The offset position= 13%16=13. The offset position 13 is checked to have a value 1, from all the values of Block-0**.** We see only in the 4th value 39 (0000000000100111) (which correspond to 4th row) has 1 in the offset position 13 (starting from left to right, 0 to 15) 1. Hence this represents the existence of the word in document-4. But this token is not present in 3rd value that is in document-3 or in document-2 or in document-1.

Similarly the offset position (24%16=8) 8 is checked through the values of Block-no 1. We see only in the 4th value 7399 (0001110011100111) (which correspond to 4th row) has 1 in the offset position 8 (starting from left to right, 0 to 15) 1. Hence this represents the existence of the word in document-4. Also this token is not present in 3rd value that is in document-3 or in document-2 or in document-1 in all other values of Block-no 1.

As the query works on satisfying both of the conditions, so both of the tokens are found only in document-4 (path named "dblp.msthesis.author" and "dblp.msthesis.year") within path numbers 7 and 9. The paths title, author, and year are the siblings of "dblp.msthesis". The path number of the title is 8 which contains the token 24 for both of the given satisfied conditions. So the query lists the output "DataMining" for the title, of the corresponding token 24.

Query # 3*: Find all the information of Hard Disk (HD) from all the documents (given in Fig.2-5).*
This query (which is an example of structural join (i.e. // ancestor-descendant relationship) does not matter where HD are?
The above query is represented in XPath is as follows: /Yahoo//HD
The tree structure of the above query is as follows:

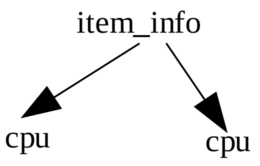


According to BIQS *(supports these types of query where a portion of the path name)* the query is represented as follows:
Select HD
From documents;
The BIQS technique, finds the path number (which matches the paths from path dictionary either full path name or portion of path name), and it is 14. All the tokens are stored within this path number (from Token and path structure, Table VI), and are 18 and 21. The system searches for the words from word dictionary to the corresponding tokens (18 and 21) and these are 40GB and 80GB and see it is correct from the XML documents shown in Fig. 2-5.

Query # 4*: Select all the attributes with the name key from documents.*
Select @key
From documents;
In this query the existence of key as an attribute in XML documents is investigated. The system uses @ sign before any name of an attribute to distinguish from path name. In the case of attribute searching, the BIQS technique searches for the attribute directly in attribute dictionary. It is found four instance of key exists in the attribute dictionary. The contents of the attribute are ms/brown92, ms/yurek97, ms/korth94, and ms/Martin98. The first two attributes exist in document 2 and the last two in document

4 (from the attribute dictionary). These attributes represent the information of the "ms/thesis" which is also associated with a path number (6) within path dictionary.

Query # 5: Find all the cpu information of item_info.
This query is an example of in which the path does not start from root.
The above query is written in XPath query language as follows:
/item_info/cpu
The tree structure of the above query is as follows:



According to BIQS the query is as follows:
Select item_info.cpu
From documents
In this query the "item_info.cpu" is part of path name "Yahoo.listing.item_info.cpu". The technique finds the full path name and matches the portion of the path name given by the query. After satisfying this, the technique finds the path number from the path dictionary, and it is 15. The token stored within this path number (from Token and path structure, Table VI), is 19 and 22. The system searches for the words from word dictionary to the corresponding token 19 and 22, which is Pentium-III and Pentium-V see it is correct from the XML documents shown in Fig. 2-5.

6 Experimental Results

To We used Oracle 9i (Enterprise Edition Release 9.2.0.8.0) and Stylus Studio 2009 XML Enterprise Suite Release 2, to evaluate the different query results in the case of a centralized system. We also used the C++ language using BorlandC compiler (32 bit that supports up to 4 GB RAM) to implement our proposed technique. We used an Intel Processor with 2.13 GHz, 1.99 GB of RAM under the Windows XP professional operating system. To support the Oracle 9i database we used the Linux operating system. We used the (well structured) XML datasets (Bib.xml, Yahoo.xml, Protein_Sequence.xml, Dblp.xml) in [49] to run the comparisons of the XQuery language and our proposed BIQS technique using file sizes of .004MB, .024MB, 5.78 MB, 11MB, 130MB, and 683MB. We have presented the other types of the query used for the experiments of BIQS in Table 10. We took all the average of these measured times for different queries.

To compare with other relational databases like MySQL, we used our own (custom) generated database named Personal-info comprising different file sizes (5.78 MB, 11 MB, 34.14, 53.03 MB, 104.46 MB, 130 MB, and 683 MB) and consisting of millions of tuples in the database relations. We measured the time (in sec) with respect to each query operation using different numbers of predicates by using *Java Eclipse*. The Java Eclipse is connected with the MySQL database for the execution of different (MySQL DB) query operations. The query execution time (in sec) using MySQL is presented in Fig. 7. It is clear from Fig. 7 that the measured time for the query operation increases due to increasing the number of Predicates and file sizes.

The measured time for different XQuery operations is shown in Fig. 8. To run the query operation we use different types of predicates in path expressions. Fig. 8 clearly shows that the XQuery evaluation time for AND operations is always larger than for the OR operations. Also the query time increases due to increasing the number of conditions or predicates. It is concluded from Fig.7 and Fig.8 that it takes more time for semistructured XQuery operations than Relational query operation.

We also used an Oracle database to evaluate the query performance for our generated database named "Personal Information", with different file sizes. The query time for both AND & OR operations as measured with respect to different numbers of predicates in the WHERE clause. The execution times are presented in Fig. 9 and Fig. 10. We ran the query with respect to all true conditions, all false conditions, and a combination of the conditions. We took all the average of these measured times. It is concluded from Fig.7, Fig.8, Fig. 9 and Fig. 10 that it takes more time for semistructured XQuery operations than Relation query operations (MySQL) but less time than Oracle SQL operation.

The measured query processing time for different query operations using BIQS is shown in Fig. 11 and Fig. 12. The query execution time (Fig. 11 and Fig. 12) is better than the XQuery execution time (shown in Fig. 8.) for the same database size. Comparing Fig 8, Fig. 9, Fig. 10, Fig. 11 and Fig 12, it is clear that BIQS is time efficient. The query processing time presented in Fig. 11, Fig. 12, and Fig. 13 for BIQS also includes the pre-processing time (such as dictionary construction time).

The comparison analysis for XQuery execution time, execution time in Oracle, MySQL, and BIQS are presented in Fig. 13. It can be seen that BIQS execution times were slightly slower than MySQL, but better than those of XQuery, and also that BIQS outperforms the highly regarded Oracle execution times across the range of predicates tested. The results show that the improvement achieved by BIQS increases with the number of predicates.

In order to evaluate the construction and manipulation of bitmap indexing [30], we used their execution time to compare with our presented BIQS execution time. A set of experiments was also performed to compare the query execution times of BIQS and Bitcube [30] as shown in Fig. 14. These experiments used 500, 1000, 1500, 2000 and 2500 element paths (ePaths) per document and a variety of words per element path. For all numbers of ePaths per document, and all numbers of documents, BIQS outperformed.

The comparison analysis for query execution time of BIQS using different file sizes with other XML query processing techniques is presented in Fig. 14. The comparison analysis in Fig.15 is presented based on the query Q-7 (Structural Join) presented in Table XI on TreeBank Data set. A tabular form is also presented in Table X. In Table X, the first row represents the different file sizes and the other rows represent the query execution time (in Sec) for structural join operation of different query processing techniques corresponding to the file sizes. We have compared the running time of BIQS with the query execution time of other query processing techniques presented in [19]. The graph presented in Fig. 15, shows that the BIQS has better time efficiency than that of some other (such as TSGeneric+, TwigStack) techniques. The preprocessing time of BIQS is also presented in Fig. 16. The compressed and uncompressed comparison is presented in Fig. 17.

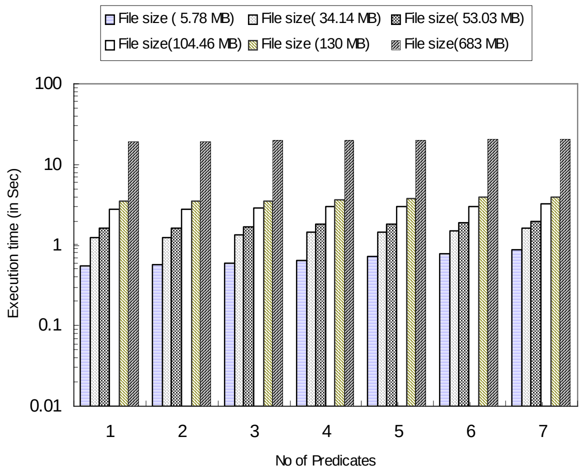


Fig. 7: The Query Execution Time (AND condition) using MySQL

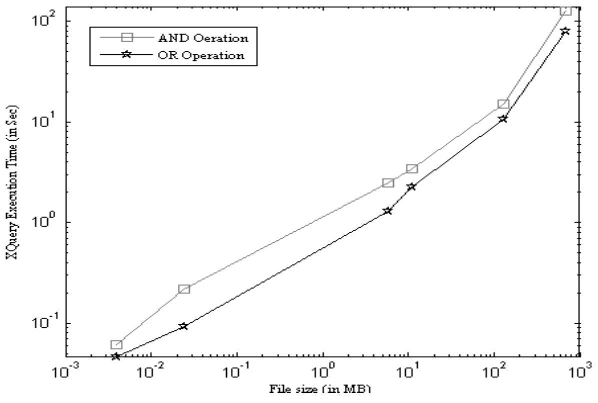


Fig. 8: The XQuery Execution Time for different file sizes.

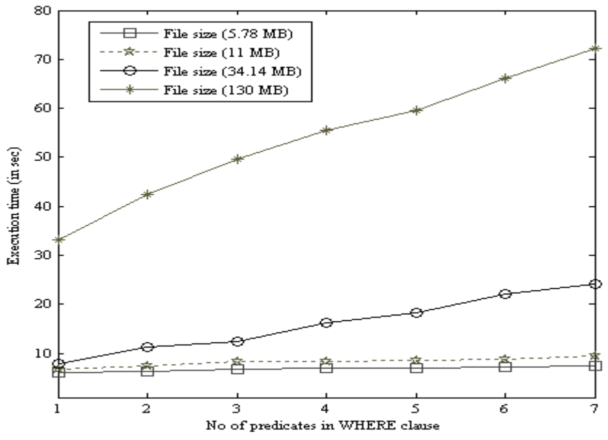


Fig. 9: The Query Execution time using OR operation in Oracle DB.

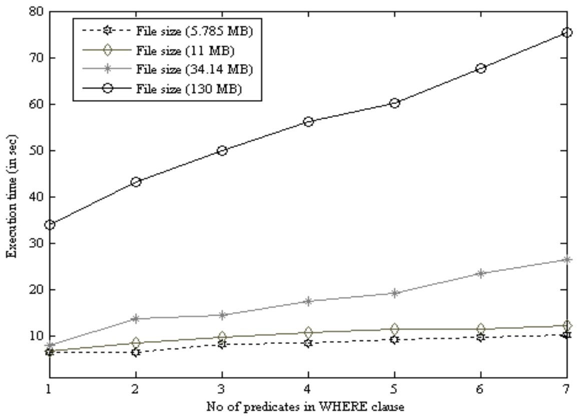


Fig. 10: The Query Execution time using AND operation in Oracle DB.

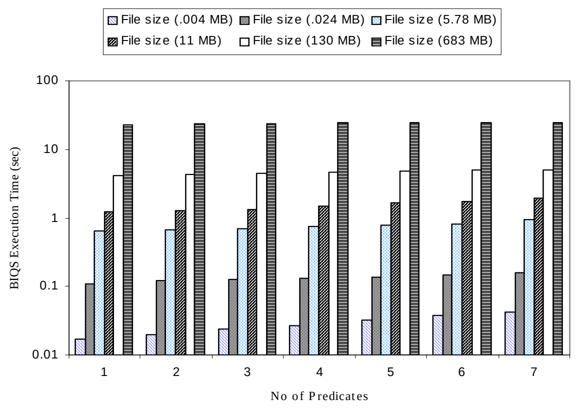


Fig. 11: Query Execution Time by BIQS Technique (wrt AND condition).

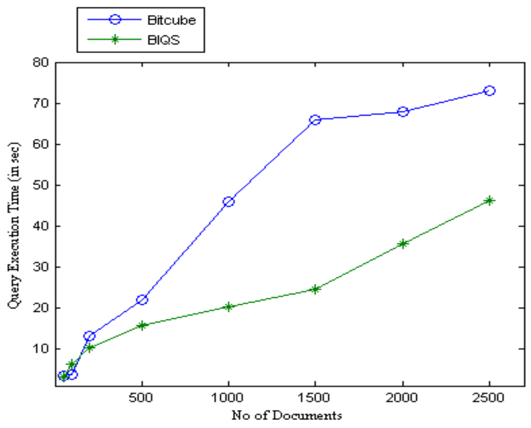


Fig. 14: Execution time wrt word/ePath/doc.

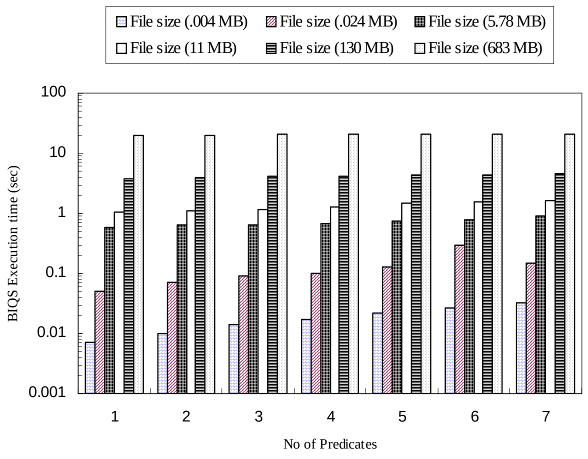


Fig. 12: The Query Execution Time Using BIQS (wrt OR condition).

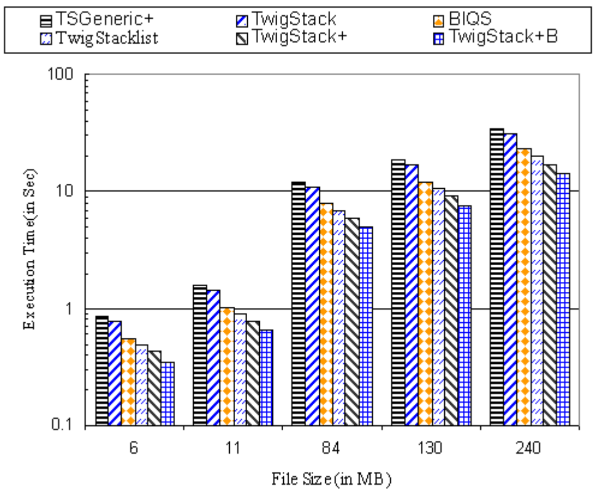


Fig. 15: Comparison of Query Execution Time over Different Data size using Q-7 (Table 10).

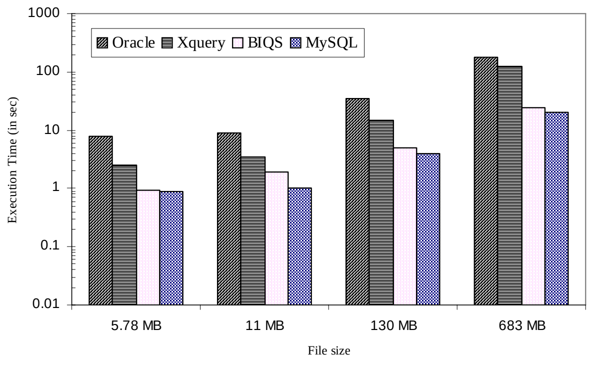


Fig. 13: Comparison of Query Execution Time.

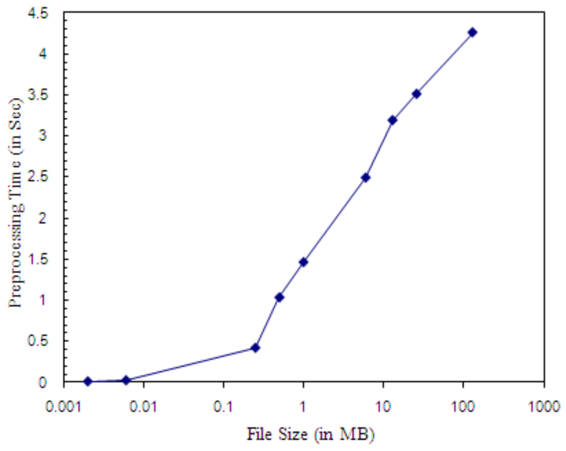


Fig. 16: Preprocessing Query Execution Time for BIQS.

Table 10. Query used in our Experiments.

Query	Dataset	XPath Expression
Q-1	DBLP	/dblp/msthesis/year
Q-2	Yahoo	/Yahoo/listing/Seller_info/Seller_rating
Q-3	Nasa	/Nasa/Datasets/dataset/author/lastname
Q-4	Yahoo	Yahoo/cpu
Q-5	DBLP	/dblp/msthesis/[author= “ korth ” and year=1994]/title
Q-6	Protein Databse	/ProteinDatabase/ProtenEntry/Reference/Referenceinfo/ authors/author
Q-7	Tree Bank	//SS[//JJ]/NP

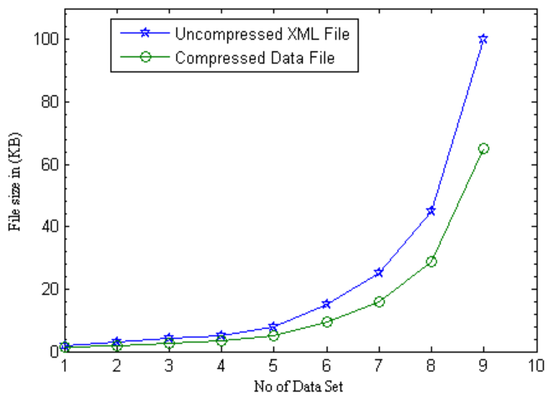


Fig. 17: Uncompressed to Compressed XML files using BIQS structure.

7 Conclusions and Future Work

XML is a convenient, semistructured data format for information exchange and some data processing tasks. Other activities, particularly searching and sorting of data, are better supported if the data is represented in a more structured form, such as that used by relational databases.

This paper describes the BIQS technique to query semistructured data in a more time efficient way than is provided by some of the other relational and semistructured query processing techniques. The presented BIQS supports the Structural Join query, Path query, and Tree structure query. The paper presents the comparison of query execution time of BIQS to the other XML query processing time (Structural Join and TwigStack) and relational query (Oracle, MySQL) processing time. Experimental results show the proposed technique queries the semistructured data in a time efficient way than is provided by some of the other existing XML and relational query processing techniques.

The presented BIQS provides storage and query performance improvement due to the compression of semistructured data. Our experiments show that XML data compression is almost 35-38% comparing to the uncompressed data. The execution time of BIQS also demonstrates better time efficiency when compared to the highly regarded Bitcube, Oracle, and XQuery.

Issues such as Aggregate function, Database updating, more complex Twig query, aggregate function, deleting, dynamic data updating will be the future research work.

References

[1] V. Garakani, M. Harizi, and M. Harizi, "Effective Guidence-Based XML Query Processing," in *International Conference on High Performance Computing and Communications*, Dalian, China 2008, pp. 605-612.

[2] Al-Khalifa, J. S, K. H.V, P. N, S. J.M, and W. Y, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," in *International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002, pp. 141-152.

[3] N. Bruno, N. Koudas, and D. Srivasta, "Holistic Twig Joins:Optimal XML Pattern Matching," in *International Conference on Management of Data (SIGMOD)*, Madison, Wisconsin, 2002, pp. 310-321.

[4] A. David, G. David, N. Ashish, C. Knight, and B. Peter, "Semistructured Data Management in the Enterprise: A Nimble, High-Throughput, and Scalable Approach," in *The 9th International Conference on Database Engineering & Application Symposium (IDEAS)*, 2005.

[5] G. Gottlob, C. Koch, and R. Pichler, "The Complexity of XPath Query Evaluation," in *PODS* San Diego, CA, 2003.

[6] J. Haifeng, W. Wei, and L. Hongjun, "Holistic Twig Joins on Indexed XML Documents," in *International Conference on Very Large Databases (VLDB)*, Berlin, Germany, 2003, pp. 273 - 284

[7] S. Hartmann and S. Link, "XML Query Optimization:Specify your Selectivity," in *The 18th International Workshop on Database and Expert Systems Applications (DEXA)* IEEE Computer Society, 2007.

[8] L. Jiaheng, C. Ting, and W. L. Tok, "Efficient Processing of XML Twig Patterns with Parent, Child Edges: A Look-ahed Approach," in *International Conference on Information and Knowledge Management*, Washington Dc, 2004, pp. 673-682

[9] V. Josifovskil, M. Fontoura, and A. Barta, "Querying XML Streams," *The journal on Very Large Databases (VLDB)* vol. 14, pp. 197-210, 2005.

[10] M. Jun-Ki, L. Jihyun, and C. Chin-Wan, "An Efficient XML Encoding and Labeling method for Query Processing and Updating on Dynamic XML Data," *The Journal of Systems and Software,* vol. 82:2009, pp. 503-515, 2008.

[11] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," in *The 27th International Conference on Very Large Databases(VLDB)* Roma, Italy, 2001.

[12] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen, "From Region Encoding to extend dewey: On efficient processing of XML twig pattern matching," in *International Conference on Very Large Databases*, Trondheim, Norway, 2005, pp. 193-204.

[13] B. Nicolas, K. Nick, and S. Divesh, "Holistic Twig Joins: Optimal XML Pattern Matching," in *International Conference on Management of Data (ACM SIGMOD)*, Wisconsin, USA, 2002, pp. 310-321.

[14] P. Ramanan, "Covering Indexes for XML Querries: Bisimulation -Simulation=Negation," in *The 29th International Conference on Very Large Databases(VLDB)* Berlin, Germany, 2003.

[15] J. Shanmugasadaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk, " Querying XML Views of Relational Data," in *The 27th International Conference on Very Large Databases (VLDB)* Roma, Italy, 2001.

[16] A.-K. Shurg and J. H.V, "Multi-level Operator Combination in XML Query Processing," in *CIKM*, Virginia, USA 2002, pp. 134-141.

[17] P. Stelios, P. Jignesh, and J. H.V, "SIGOPT:Using Schema to Optimize XML Query Processing," in *International Conference on Data Engineering (ICDE)*, Istanbul, Turkey, 2007, pp. 1456-1460.

[18] C. Ting, L. Jiaheng, and W. L. Tok, "On Boosting Holism in XML Twig Pattern Matching Structural Indexing Techniques," in *International Conference on Management of Data (ACM SIGMOD)*, Maryland, USA 2005, pp. 455-466

[19] J. Zhou, M. Xie, and X. Meng, "TwigStack$^+$:Holistic Twig Join Pruning Using Extended Solution Extension," *Wuhan University Journal of Natural Sciences (WUJNS)*, vol. 8:2B, pp. 603-609, 2007.

[20] S. Abiteboul, "Querying Semistructured Data," in *The International Conference on Database Theory (ICDT)* Delphi, Greece., 1997.

[21] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal on Digital Libraries*, vol. 1(1), pp. 68-88, 1997.

[22] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," in *The 27th International Conference on Very Large Databases (VLDB)* Roma, Italy, 2001.

[23] J. McHugh and J. Widom, "Query Optimization for XML," in *VLDB* Edinburgh, Scotland, 1999.

[24] T. Milo and D. Suciu, "Index Structures for Path Expressions," in *ICDT* Jarujalem, Israel, 1999.

[25] A. A. d. Sousa, J. L. Perira, and J. A. Carvalho, "Querying XML Databases," in *The 12th International Conference of the Chilean Computer Science Society (SCCC)* IEEE, 2002.

[26] R. Goldman and J. Widom, "DataGuides:Enabling Query Formulation and Optimization in Semistructured Databases," in *International Conference on Very Large Databases (VLDB)*, Athens, Greece, 1997, pp. 436-445.

[27] F. Rizzolo and A. Mendelzon, "Indexing XML Data with ToXin," in *Research Report* Department of Computer Science, University of Toronto, Canada, 2001.

[28] C.-W. Chung, J.-K. Min, and K. Shim, "APEX: An Adaptive Path Index for XML Data," in *ACM SIGMOD* Madison, Wisconsin, USA, 2002.

[29] N. Zhang, M. T. Ozsu, I. F. llyas, and A. Aboulnaga, "FIX:Feature-based Indexing Technique for XML Documents," in *The 32nd International Conferences on Very Large databases(VLDB)* Seoul, Korea, 2006.

[30] J. P. Yoon, V. Raghavan, and V. Chakilam, "BitCube: A Three-Dimensional Bitmap Indexing for XML Documents," *Journal of Intelligent Information Systems*, vol. 17, pp. 241-254, 2001.

[31] M. Benedikt, W. Fan, and F. Geerts, "XPath Satisfiability in the Presence of DTDs," in *PODS* Baltimore, Maryland, 2005.

[32] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan, "XQuery Implementation in Relational Database System," in *The 31st International Conference on Very Large Databases* Trondheim, Norway, 2005.

[33] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "XML-QL: A Query Language for XML," http://www.w3.org/TR/NOTE-xml-ql.

[34] A. Bonifati and S. Ceri, "Comparative Analysis of Five XML Query Language," *SIGMOD*, vol. 29:1, pp. 68-79, 2000.

[35] P. Buneman, M. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *The VLDB Journal*, vol. 9, pp. 76-110, 2000.

[36] A. Balmin and Y. Papakonstantinou, "A Storing and Querying XML Data using Denormalized Relational Databases," *The journal on Very Large Databases (VLDB)*, vol. 14, pp. 30-49, 2005.

[37] Y. Chen, S. Davidson, C. Hara, and Y. Zheng, "RRXS: Redundancy reducing XML storage in relations," in *The 29th International Conference on Very Large Databases (VLDB)* Berlin, Germany, 2003.

[38] S.-Y. Chien, Z. Vagena, and D. Zhang, "Efficient Structural Joins on Indexed XML Documents," in *The 28th International Conference on Very Large Databases (VLDB)* Hong Kong, China, 2002.

[39] F. Du, S. Amer, and J. Freire, "ShreX: Managing XML Documents in Relational Databases " in *The 30th International Conference on Very Large Databases (VLDB)* Toronto, Canada: , 2004.

[40] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, and M. Morschel, "ROX: Relational Over XML," in *The 30th International Conference on Very Large Databases* Toronto, Canada, 2004.

[41] B. M. Alom, F. A. Henskens, and M. R. Hannaford, "Storing Semistructured Data Into Relational Database Using Reference Relationship Scheme," in

International Conference on Software & Data Technologies (ICSOFT) Porto, Portugal 2008.

[42] A. A. A. Aziz and H. Okasha, "Mapping XML DTDs to Relational Schemas," in *IEEE*, 2005.

[43] F. Bry and S. Schaffert, "The XML Query Language Xcerpt: Design Principles, Examples, and Semantics," in *LNCS 2593* Berlin Heidelberg: Springer-Verlag, 2003.

[44] A. Deutsch, M. Fernandez, and D. Suciu, "Storing Semistructured Data with STORED," in *International Conference on Management of Data (SIGMOD)* Pennsylvania, USA., 1999.

[45] A. Deutsch, M. F. Fernandez, and D. Suciu, "Storing Semistructured Data in Relations," in *ICDT*, 1999.

[46] D. Florescu and D. Kossman, "Storing and Querying XML Data using an RDMBS," *The IEEE Data Engineering Bulletin,* vol. 22(3), pp. 27-34., 1999.

[47] P. J. Harding, Q. Li, and B. Moon, " XISS/R: XML Indexing and Storage System Using RDBMS," in *The 29th International Conference on Very Large Databases (VLDB)* Berlin, Germany, 2003.

[48] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov, "Indexing XML data Stored in a Relational Database," in *The 30th International Conference on Very Large Databases* Toronto, Canada., 2004.

[49] http://www.cs.washington.edu/research/xmldatasets/.

[50] J. Shanmugasadaram, R. Krishnamurthy, I. Tatarinov, E. Shekita, E. Vigias, J. Kiernan, and J. Naughton, " A General Technique for Querying XML Documents using a Relational Database System," *The journal (SIGMOD),* vol. 30(3), pp. 20-26, 2001.

Authors Biography

B.M. Monjurul ALom who born in Bagherpara, Jessore, Bangladesh, is a research (PhD) student in the School of Electrical Engineering and Computer Science, The University of Newcastle, Australia. Mr Alom has completed his MSc engineering degree from Bangladesh University of Engineering and Technology, Dhaka. His research interest is Distributed (Structured and Semistructured) Database Management. Mr. Alom was an assistant professor in CSE dept from 2004 to 2007 and a lecturer from 2000 to 2004 in Dhaka University of Engineering and Technology, Gazipur, Bangladesh.

Dr. Frans Henskens is an Associate Professor in the School of Electrical Engineering and Computer Science, Newcastle University Australia. He is also Head, Discipline of Computer Science & Software Engineering, Deputy Head, School of Electrical Engineering & Computer Science, and Assistant Dean IT in Faculty of Engineering & Built Environment. His research interests include engineering of flexible software systems, bioinformatics, operating systems and computer forensics, distributed and grid computing, resilience and availability in database systems.

Dr. Michael Hannaford is Assistant Dean (Postgraduate Studies) of FEBE, and a Senior Lecturer in the School of Electrical Engineering and Computer Science at the University of Newcastle. His research interests are in the areas of Distributed Computing, and Programming Language Design and Implementation.