Journal of Applied Sciences

ISSN 1812-5654

Journal of Applied Sciences 7 (19): 2736-2743, 2007
ISSN 1812-5654
© 2007 Asian Network for Scientific Information

Path Query Processing in Large-Scale XML Databases

Su-Cheng Haw and G.S.V. Radha Krishna Rao
Faculty of Information Technology, Multimedia University, 63100 Cyberjaya, Malaysia

Abstract: With the ever-increasing popularity of XML (Extensible Markup Language) as data representation and exchange on the Internet, querying XML data has become an important issue to be address. In Native XML Database (NXD), XML documents are usually modeled as trees and XML queries are typically specified in path expression. In path expression, the primitive structural relationships are Parent-Child (P-C) and Ancestor-Descendant (A-D). Thus, finding all occurrences of these relationships is crucial for XML query processing. Current methods for query processing on NXD usually employ either sequential traversing of tree-structured model or a decomposition-matching-merging processes. We adopt the later approach and propose a novel hybrid query optimization technique, INLAB comprising both indexing and labeling technologies. Furthermore, we also propose several algorithms to create INLAB encoding and analyze the path query. We implemented our technique and present performance results over several benchmarking datasets, which prove the viability of our approach.

Key words: XML databases, query processing, path query, indexing, labeling

INTRODUCTION

Due to its flexibility and efficiency in transmission of data, Extensible Markup Language (XML) has become the emerging standard of data transfer and data exchange across the Internet. As the amount of exchanged data often grows exponentially via the Web medium, this drives the requirement for storing and querying large-scale XML databases as efficient as possible (Haw and Rao, 2005).

There are two main approaches for query processing in Native XML Database (NXD). The first approach is to traverse the XML database sequentially to find the matching pattern. Every node in the XML database needs to be evaluated for possible matches. Thus, this approach certainly poses a new challenge, because it may not meet the processing requirements under heavy access requests (Li and Moon, 2001). Furthermore, it fails to support large-scale dataset efficiently. As a result, index structures have been introduced to address the problem of performance degradation due to excessive traversal. Among them are DataGuide (Goldman and Widom, 1997), T-index (Milo and Suciu, 1999), A(k)-index (Kaushik *et al.*, 2002), Index Fabric (Cooper *et al.*, 2001), APEX Index (Chung *et al.*, 2002), D(k)-index (Chen *et al.*, 2003) and M*(k)-index (He and Yang, 2004). These are general path indexes that summarize all paths in Object Exchanged Model (OEM) starting from the root to the respective node. T-index selects paths based on specific templates, while APEX, A(k)-index, D(k)-index and M*(k)-index select the most frequently used paths in queries by restricting the path length to k. The D(k)-index and M*(k)-indexes are more dynamic in the sense of they are based on the notion of the dynamic local similarity, which means different index nodes may have different local similarity to support frequently used path expressions. Unlike the other approaches, Index Fabric encodes its label path as string and stores them in a balanced Patricia trie. Recently, Zou *et al.* (2004) propose Ctree, a two-level tree index which provides a concise structure summary. Lian *et al.* (2005) propose MIS index, an index structure which index infrequent structures in the database instead. Yan and Liang (2005) propose MXI, an indexing method that support efficient path query based on embedded Data Type Definition (DTD). Kiss and Anh (2005) combine both structural and tree structure index to accelerate the query processing. Nevertheless, all these approaches suffer from large index size growth. As for INLAB indexing, in the worst case, the index size grows linearly with the XML tree. Further elaboration will be discussed in next section.

To overcome the shortcomings of structural indexing, several labeling scheme have been proposed. Labeling scheme allow quick determination of the relationships among the element nodes. Among some of the labeling schemes are tree location address (Kimber, 1993) and simple prefix (Cohen *et al.*, 2002) which utilize prefix checking for possible A-D relationship; GRP (Lu and Ling, 2004) which based on group prefix; prime number labeling (Wu *et al.*, 2004) which check for relationship based on prime number property and ORDPATH (O'Neil *et al.*, 2004) which the concept is similar to Dewey Order

Corresponding Author: Su-Cheng Haw, Faculty of Information Technology, Multimedia University, 63100 Cyberjaya, Malaysia

(Tatarinov *et al*., 2002) that encodes the Parent-Child (P-C) relationship by extending the parent label with a component for the child. Recently, Thonangi (2006) propose Sector-based labeling scheme. For example, node A is assigned with label <Ar, As> where 2^{Ar} is the radius of the sector and As is the radial-distance of the starting point of the sector with respect to the reference-axis. Although, such representation requires considerably smaller length as only the logarithm of the sector's radius is stored, the label computation is rather expensive. Zhang *et al*. (2004) propose an encoding scheme called as ES-Index, extending Dietz's labeling scheme (Dietz, 1982) based on prefix and unique gene labels to quickly determines the siblings-siblings and Ancestor-Descendant (A-D) relationship. As summary, although some labeling schemes (Wu *et al*., 2004; O'Neil *et al*., 2004; Thonangi, 2006) are able to support dynamic update, they still face the similar problem of having large labeling sizes especially if the XML tree is dense or skew structure. In our labeling scheme <self-level: parent>, the size of the labelled node is only 12 bytes. In addition, our labeling is integer based. Integer processing is very efficient compared to that of string or bit-vector.

The second approach is to preprocess the XML database into a set of data streams as the input document and executed through a series of processes involving decomposition, matching and merging. Firstly, a complex query pattern can be decomposed into a set of basic binary structural relationships between pairs of nodes. These structural relationships could be of A-D or P-C relationship. The query pattern can then be matched by matching each of the binary structural relationship against the data streams. Zhang *et al*. (2001) propose MPMGJN while Bruno *et al*. (2002) and Al-Khalifa *et al*. (2002) propose PathStack and Stack-Tree algorithm, respectively to match the binary structural relationships. The main difference between MPMGJN and both PathStack and Stack-Tree is that MPMGJN require multiple scans on input lists for the matching process. The PathStack algorithm is more efficient as it uses stack to maintain the ancestor or parent nodes and it require only one time scan per input list. The Stack-Tree algorithm uses an adaptation of a merge-sort technique and supports both path and twig query. These algorithms accepts two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matching of the binary relationships (Yao and Zhang, 2004). Yet, these approaches still suffer from producing large size of intermediate results. To address this problem, Bruno *et al.* (2002) propose TwigStack, a holistic twig join algorithm that uses a chain of linked stacks to compactly represent the intermediate results and subsequently join them to obtain the final results. However, this algorithm is only optimal for A-D relationship. Lu *et al.* (2004) extend TwigStack and propose TwigStackList, which can support both P-C and A-D relationship efficiently. Next, these matches are merged together to form the final path solution. Merging together the structural matches in the final process poses the problem of selecting a good join ordering. Wu *et al.* (2003) propose a cost-based join order selection of structural join. Kim *et al.* (2004) propose a technique to partition all nodes in an extent into several clusters. Given two extents to be joined, the proposed technique filters out unnecessary clusters in both extents before joining.

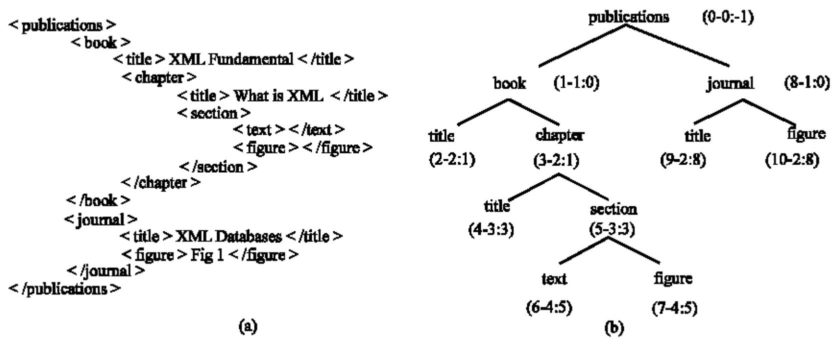
Some other XML query processing performed in a streaming fashion include XMLTK (Green *et al*., 2003), XSQ (Peng and Chawathe, 2003) and EXPedite (Chen *et al*., 2004). While XMLTK process queries using a Deterministic Finite Automaton (DFA) that are constructed lazily, XSQ process queries on streaming XML data using a hierarchical pushdown transducer (HPDT). Using the HPDT as a guide, a runtime engine responds to the incoming stream and emits the query result. In EXPedite, however, the XML database is encoded as streams with label (t, size, depth). EXPedite uses stack to store each qualified node during the matching process.

In this research we adopt the decomposition-matching-merging approach and propose a novel hybrid query processing technique, INLAB comprising both indexing and labeling technologies to evaluate the path query. This technique guaranteed that every intermediate result participate in the final results. Our contribution can be summarized as follows:

- The proposed INLAB labeling scheme can be used for determining the two main types of primitive relationships in a path query: (i) A-D and (ii) P-C efficiently.
- The proposed PathINLAB query processing algorithms process queries without traversing the whole XML database. We show the substantial performance benefits of our approach on various datasets.

Overview of INLAB: In NXD, OEM is usually used to model the data. It can be viewed as a rooted, ordered, node-labeled tree where each node represents an element or a value. For the sample XML document of Fig. 1a, its OEM representation is shown in Fig. 1b.

INLAB labeling scheme: In INLAB labeling scheme, given an XML tree T, any label consists of <self-level: parent> representation, where (i) self attribute is obtained



```
< publications >
        < book >
                < title > XML Fundamental < /title >
                < chapter >
                        < title > What is XML  < /title >
                        < section >
                                < text > < /text >
                                < figure > < /figure >
                        < /section >
                < /chapter >
        < /book >
        < journal >
                < title > XML Databases < /title >
                < figure > Fig 1 < /figure >
        < /journal >
< /publications >
```

Fig. 1: (a) A sample XML document (b) OEM representation

```
Algorithm 1: create INLAB encoding
function createINLAB  {
  input : an XML file X
  output : encoded XML assigned tag
  /*A stack eleStack to keep track of element sequence.
  A vector vExtent to store the occurrence of each element in stream
  A hashtable eleTable to store each distinct element in X.
  A hashtable PCTable to keep track of each element parent's
  information
  A record with <self-level : parent> */
  int ptr = 0, level = 0, self = 0, parent = -1
  curRec = null
  while (! eof (X)) do {
        if SAX event = a start tag <T> then {
                if (tag has yet been stored into eleTable) {
                create new instance of vector, vExtent
                eleTable.put( tag, ptr++)
           }
            create new instance of record, curRec
            curRec.self = self++
            curRec.level = level++
          if (eleStack.size() > 0)
             curRec.parent = eleStack.elementAt(eleStack.size()-
          1).self
          else
              curRec.parent = -1
            int i = eleTable.get(tag).intValue
          vExtent[i].addElement(curRec)
            eleStack.push(curRec)
        }
        if SAX event = an end tag </T> then {
                eleStack.pop()
                curRec = eleStack.peek()
                level --
      }
  }
} //end function

function output {
   input : tag in XML file X and cursor position in data stream
   output : encoded XML data streams(files)
   create file fileData = ("myData\\"+tag, with read and write
   mode)
   int self, level, parent
   while (as long as cursor NOT end of stream) {
         self = cursor.getCurSelf
         level = cursor.getCurLevel
         parent = cursor.getCurParent
         writeInt(fileData, self, level, parent)
       PCTable.put(self, parent)
  }
} //end function
```

by doing a pre-order traversal of the XML tree, T (ii) level attribute of a node is its distance from the root and (iii) parent attribute is the direct node which relates to the self node. Figure 1b depicts the INLAB labeling representation for an XML tree.

Analysis of createINLAB encoding algorithm: Algorithm 1, createINLAB() takes a regular XML document and generates a set of encoded XML data streams (files). Each element with the same tag is grouped into one data stream. Figure 2a and b show the fragment of data streams and index table, PCTable generated based on the sample XML docum ent in Fig. 1a. To parse the XML document, INLAB parser starts from the beginning of the root element, sequentially iterates over the rest of the elements to generate positional representation of each element using INLAB labeling scheme.

Structural relationships between nodes can be efficiently determined form the label as follows:

P-C relationship: $node_1$ is the parent of $node_2$ iff $node_1$.self = $node_2$.parent.

A-D relationship: $node_1$ is possible as an ancestor of $node_2$ iff level difference, leveldiff = $node_2$.level-$node_1$.level >= 1. A multiple look-up via PCTable Fig. 2b is necessary as long as the leveldiff > 1 is true to confirm the A-D relationship.

For example, let publications <0-0:-1> be $node_1$ and title <2-2:1> be $node_2$. The leveldiff between the two nodes is 2. To determine whether these two nodes is of P-C relationship, we need to hash PCTable (Fig. 2b) twice (two level up). The retrieved node parent attribute is 0 and it is equal to the self attribute of publications, which is also 0. Thus, publications and title is of A-D relationship.

Analysis of pathINLAB processing algorithm: Algorithm 2, PathINLAB() computes answers to a path query pattern. The key idea of this algorithm is to

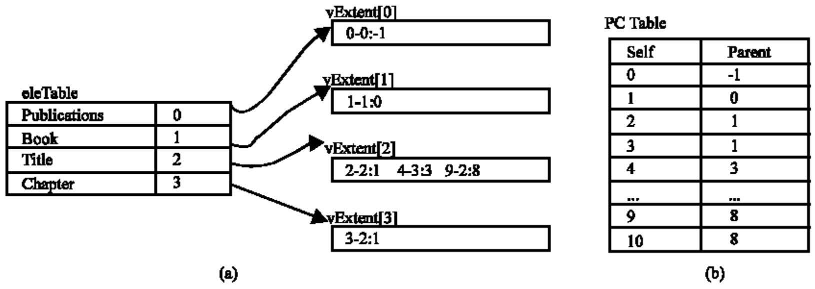


Fig. 2: (a) Fragment of generated data streams (b) fragment of index table

```
Algorithm 2 : PathINLAB Processing
1.  function PathINLAB(q) {
2.  input : INLAB encoding streams and path query
3.  output : Final solution matching to the path query
4.  while (! end(q) ) do {
5.    qSring = getNext(getRoot())
6.    if (qString != getRoot())
7.       cleanParentStack()
8.    if (qString == getRoot() || stack_size_of_ parent != empty) {
9.       cleanSelfADStack()
10.      moveToStack()
11.      if ( isLeaf(qString) )  {
12.          outputSolution()
13.          pop()
14.      }
15.      advance(qString)
16.   }
17.   else advance(qString)
18.  } //end while
19.  mergeAllPathSolution()
20. } //end function
21.
22. function end(q) {
23.   input : leaf node
24.   output : Boolean true or false
25.   if (isLeaf(q) NOT in eof(Tq))
26.        return false
27.   else return true
28. }
```

```
Algorithm 3 : get next node to be process
1. function getNext(q) {
2.   input : current node in process
3.   output : node to be process
4.   if (isLeaf(q)) return q
5.   tempq = getChild(q)
6.   n = getNext(tempq) //recursive call
7.   if (n != tempq) return n
8.   while ( ! checkAncestor(q, n) {
9.        if (getSelf(q) > getSelf(n)) return n
10.       advance (q)
11.  }
12.  if (getSelf(q) > getSelf(n)) return n
13.  return q
14. } //end function
15.
16. function checkAncestor(q, n) {
17.   input : two nodes
18.   output : boolean true or false
19.   leveldiff = getLevel(n) – getLevel(q)
20.   current = getSelf(n)
21.   if (getSelf(n) != eof) {
22.        if (leveldiff> 0) {
23.            while (leveldiff > 0) {
24.                    cursorUp = hashPCTable(current)
25.                    current = cursorUp
26.                leveldiff--
27.            }
28.        if (current = getSelf(q)) return true
29.        else return false
30.        }
31.        return false
32.   }
33.   return false
34. } //end function
```

repeatedly construct stack encoding of partial and total answers to the path query, by iterating through each element in the stream in sorted order of their self attribute starting from the root of the path query (as returned by getRoot() procedure), by the procedure getNext(). Partial answers from the stacks that cannot be extended to final answers are removed, in the procedure of cleanParentStack() and cleanSelfADStack() as in lines 6-9. Each qualified element that fulfill the matching criteria, is pushed into stack by the procedure moveToStack() for further processing. If qString is a leaf query node (checked by isLeaf() procedure), the solution should be output as in lines 11-12. Note that path solutions are output in root-leaf order so that they can be easily merged together to form final path matches (procedure mergeAllPathSolution() in line 19). Once the query node has been processed, lines 13-15 remove the query node from the stack and advance to the next query node by the advance() procedure.

Analysis of getNext algorithm: In getNext() algorithm (depicted in Algorithm 3), if q is a leaf query node (checked by procedure isLeaf()), the function directly returns to output the solution (line 4). In line 6, we recursively invoke getNext() function until it is terminated by either line 4 or 7. Path query has only one child per node, thus procedure getChild(q) returns the immediate

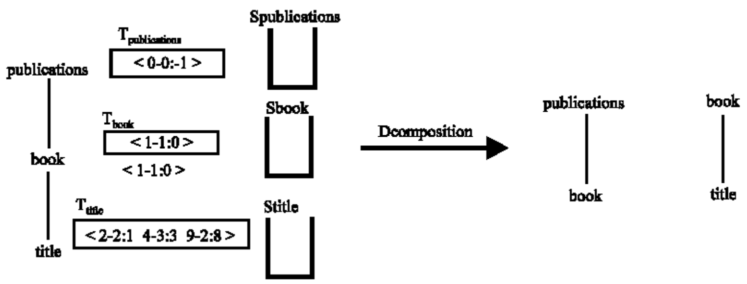


Fig. 3: Decomposition of path query publications-book-title into binary relationships

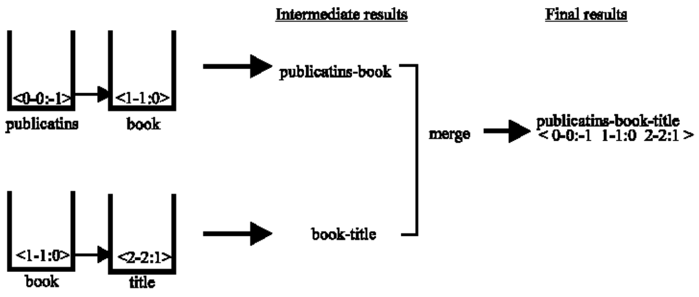


Fig. 4: Matching and merging processes on path query

children of node q. In line 7, if any returned node n is not equal to child of q, we immediately return n. Line 8 skips node that do not contribute to results, by checking whether the two nodes is of A-D relationship. During this process (line 24 of checkAncestor() function), the index table, PCTable (storing P-C relationship) is being hashed to retrieve each query node's parent for comparison. Procedures getSelf() and getLevel() return the self and level attribute of the query node in the stream. Lines 12-13 return the next node to be process.

Stack operation for PathINLAB processing: Consider a path query, Q1 = publications/book/title on the XML tree, T in Fig. 1a. Associated with each query node, q in Q1 is a data stream Tq and a stack Sq as shown in Fig. 3. Tq contains the occurrences of tag q in T. For example, associated with query node publications are $T_{publications}$ and $S_{publications}$. Firstly, Q1 undergoes the decomposition process. As a result, two sets of binary relationships are generated as shown in Fig. 3.

Initially, the binary relationship of publications-book is to be processed first. Based on the self attribute in each first occurrence in $T_{publications}$ and T_{book} query node publications is returned by the getNext() algorithm. Element <0-0:-1> is then pushed into $S_{publications}$ by procedure MoveToStack(). The next returned query node is the immediate child of publication, which is book. Element <1-1:0> is pushed into S_{book} because parent attribute of book is equal to self attribute of publications. Since book is the leaf query node, a partial solution is formed between publications-book.

Next, we process on the book-title relationship. T_{book} have one occurrence while T_{title} have three occurrences. Based on each first occurrence self attribute, query node book will be processed first. Element <1-1:0> is pushed into S_{book}. The next returned query node is title. Each occurrence will be processed in their self attribute order, that is <2-2:1>, <4-3:3> followed by <9-2:8>. Nevertheless, only element <2-2:1> is qualified being pushed into S_{title}. As title is the leaf query node, a partial solution is formed between book-title.

Finally, the two intermediate results are being merged to form the final solution in procedure mergeAllPathSolution(). Thus, there is only one final solution that fulfills the query criteria. Figure 4 shows the matching and merging processes.

EXPERIMENTAL EVALUATION

We have implemented INLAB using Java API for XML Processing (JAXP). Preliminary experimental results to compare PathINLAB with conventional top-down approach have been reported in Haw and Rao (2007).

Experimental setup: Our experimental tests are divided into four main test cases described as follow:

- T1: Comparing INLAB encoded XML file size over regular XML
- T2: Comparing PathINLAB over conventional top-down approach, PathStack and XSQ
- T3: Comparing PathINLAB processing over PathStack on a skew structured dataset
- T4: Comparing PathINLAB processing over PathStack on a flat structured dataset

For each test case, we run with certain dataset(s) as shown in Table 1.

All our experiments were performed on 1.7GHz Pentium IV processor with 512 MB SDRAM running on windows XP systems. In test case T2, we benchmarked PathINLAB with the other approaches by using the set of queries shown in Table 2 over the modified TreeBank dataset. TreeBank dataset is scaled down to approximately 3MB so that it could be supported by the conventional top-down approach. In test case T3, the same set of queries is used, but this time, we benchmarked PathINLAB over PathStack on original TreeBank dataset (84MB). We also tested the performance of PathINLAB and PathStack on a larger dataset, Protein (700MB) in test case T4 based on the set of queries shown in Table 3.

Table 1: Various test cases associated with respective dataset(s)

Test case	Dataset
T1	Protein, Original TreeBank, LineItem, Orders (UW, 2002) Sigmod (Sigmod, 2002)
T2	Modified TreeBank
T3	Original TreeBank (UW, 2002)
T4	Protein (UW, 2002)

Table 2: Queries over TreeBank dataset

Query	Path expression
Q1	S/NP
Q2	S/NP/NN
Q3	S//NP
Q4	VP/NP/PP
Q5	VP/NP/PP/IN

Table 3: Queries over Protein dataset

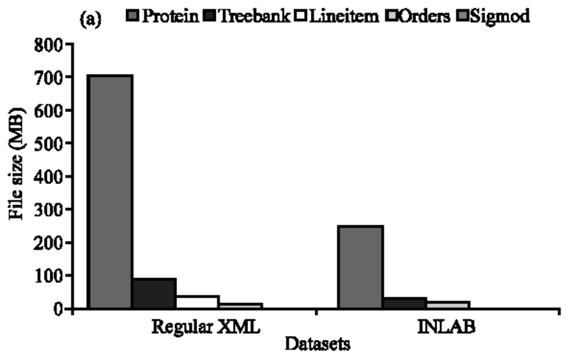
Query	Path expression
Q1	accinfo/exp-source
Q2	feature/status
Q3	ProteinEntry/function/description
Q4	organism//source
Q5	ProteinDatabase//protein/name

RESULTS AND DISCUSSION

Figure 5a shows that INLAB encoding outperforms in terms of reducing the XML file size. XML data is usually much smaller, about 15-65% than the original XML file. For a larger XML file size (as shown by Protein dataset) by using INLAB encoding, there is a major reduction in file size, about 65%. Hence, it is very suitable especially in reducing the size for a large-scale dataset.

Figure 5b shows the execution time of queries defined in Table 2 for PathINLAB, PathStack, XSQ and conventional top-down approaches. As can be observed, conventional top-down approach is much slower compared to PathINLAB, PathStack and XSQ (generally over an order magnitude). This is because conventional top-down approach is too conservative when backtracking and reads several times unnecessary nodes in the XML document when comparing for matches. Likewise, PathINLAB performs better as compared to PathStack if the path query is of P-C relationship. This is mainly due to its labeling scheme format, which enable quick determination of query nodes in P-C relationship. However, when comes to A-D relationship, PathINLAB is slower than PathStack as shown in Q3 Fig. 5b. This is because the extra time needed for multiple lookups on the index table until the ancestor level is reached. This test result indicates that PathINLAB is the fastest, followed by PathStack, XSQ and conventional top-down approach. Thus, for the rest of the test cases, we benchmarked PathINLAB with respect to PathStack only.

Figure 6a and b show the execution time of queries over original Treebank and Protein datasets, respectively. Treebank dataset is a deeply nested skew structured tree with many repeating elements while protein is a flat



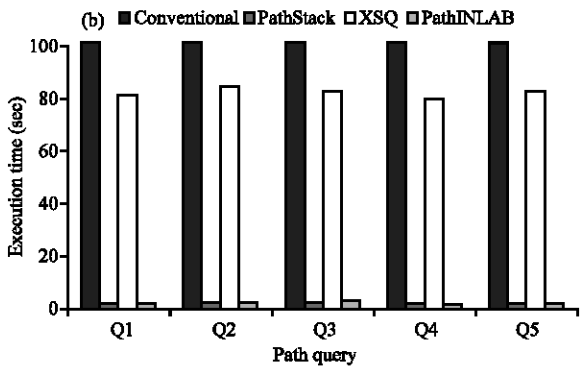
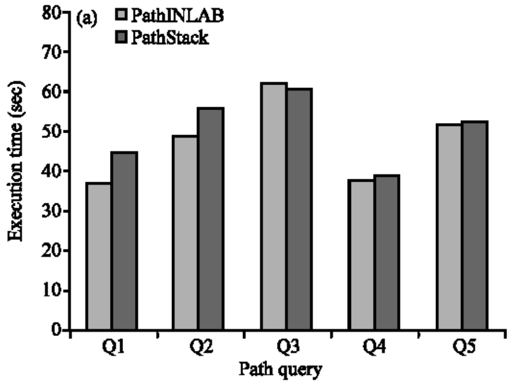


Fig. 5: Results for test cases (a) T1 and (b) T2



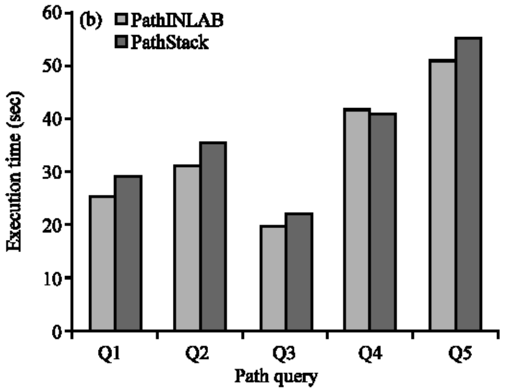


Fig. 6: Results for test cases (a) T3 and (b) T4

structured tree with many fan-outs. From the result obtained, PathINLAB performs about 10% much better than PathStack on the flat structure, which is the most common structure in most data exchange scenario. On the skew structure dataset, PathINLAB performs about 5% better as compared to PathStack.

CONCLUSIONS

In this study, we have proposed a hybrid query processing architecture, INLAB comprising both indexing and labeling technologies. Using the INLAB labeling scheme, structural relationships can be determined easily. The extensive experimental results showed that INLAB labeling scheme is efficient and yet simple. We complemented INLAB with indexing technologies to speed up the matching process among each binary structural relationship. Experimental results showed that PathINLAB outperforms the conventional top-down approach, XSQ and PathStack in most cases. Besides, PathINLAB process queries efficiently on a flat structured tree (many fan-outs), which is the most common structure found in most data exchange scenario. Moreover, PathINLAB is capable of supporting large-scale of dataset efficiently. This is crucial as currently there is a paradigm shift towards large-scale of data exchange and transfer across the Internet.

REFERENCES

Al-Khalifa, S., H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava and Y. Wu, 2002. Structural joins: A primitive for efficient XML query pattern matching. Proc. ICDE, pp: 141-152.

Bruno, N., D. Srivastava and N. Koudas, 2002. Holistic twig joins: Optimal XML pattern matching. Proc. ACM SIGMOD, pp: 310-321.

Chen, Q., A. Lim, K. Ong and J. Tang, 2003. D(k)-index: An adaptive structural summary for graph-structured data. Proc. SIGMOD, pp: 134-144.

Chen, Y., S. Padmanabhan, G.A. Mihaila and S.B. Davidson, 2004. Efficient path query processing on encoded XML. Proceeding of International Workshop on High Performance XML Processing.

Chung, C.W., J.K. Min and K. Shim, 2002. APEX: An adaptive path index for XML data. Proc. ACM SIGMOD, pp: 121-132.

Cohen, E., H. Kaplan, T. Milo, 2002. Labeling dynamic XML trees. Proc. ACM SIGMOD-SIGACT-SIGART, pp: 272-281.

Cooper, B.F., N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon, 2001. A fast index for semistructured data. In Proc. of VLDB, pp: 341-350.

Dietz, P.F., 1982. Maintaining order in a linked list. Proc. ACM Symp. On Theory of Computings, pp: 122-127.

Goldman, R. and J. Widom, 1997. Data Guides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. VLDB, pp: 436-445.

Green, T.J., G. Miklau, M. Onizuka and D. Suciu, 2003. Processing XML streams with deterministic automata. Proc. ICDT, pp: 173-189.

Haw, S.C. and G.S.V.R.K. Rao, 2005. Query optimization techniques for XML databases. Int. J. Inform. Technol., 2: 97-104.

Haw, S.C. and G.S.V.R.K. Rao, 2007. An efficient path query processing support for parent-child relationship in native XML databases. J. Digital Inform. Manage., 2: 82-87.

He, H. and J. Yang, 2004. Multiresolution indexing of XMl for frequent queries. Proc. ICDE, pp: 683-694.

Kaushik, R., D. Shenoy, P. Bohannon and E. Gudes, 2002. Exploiting local similarity to efficiently ondex paths in graph-structured data. In Proc. of ICDE, pp: 129-140.

Kim, J., S.H. Lee and H-J. Kim, 2004. Efficient structural joins with clusters extents. Inform. Proc. Lett., 91: 69-75.

Kimber, W.E., 1993. HyTime and SGML: Understanding the HyTime HYQ Query Language. Available: http://ftp2.de.freebsd.org/pub/sgml/ifi.uio.no/HyTime/HyQ-1.1.Kimber.

Kiss, A. and V.L. Anh, 2005. Combining tree structure indexes with structural indexes in query evaluation on XML data. Lecture Notes Comput. Sci., 3631: 254-267.

Li, Q. and B. Moon, 2001. Indexing and querying XML data for regular path expressions. Proc. VLDB Conference, pp: 361-370.

Lian, W., N. Mamoulist, David W.L. Cheung and S.M. Yiu, 2005. Indexing useful structural patterns for XML query processing. IEEE Trans. Knowledge Data Eng., 17: 997-1009.

Lu, J. and T.W. Ling, 2004. Labeling and querying dynamic XML trees. Lecture Notes Comput. Sci., 3007: 180-189.

Lu, J., T. Chen and T.W. Ling, 2004. Efficient processing of XML twig patterns with parent child edges: A look-ahead approach. Proc. CIKM, pp: 533-542.

Milo, T. and D. Suciu, 1999. Index structures for path expression. Proc. ICDT, pp: 277-295.

O'Neil, P., E. O'Neil, S. Pal, I. Cseri, G. Schaller and N. Westbury, 2004. ORDPATHS: Insert-friendly XML node labels. Proc. ACM SIGMOD, pp: 903-908.

Peng, F. and S.S. Chawathe, 2003. XPath queries on streaming data. Proc. ACM SIGMOD, pp: 431-442.

Sigmod, 2002. Sigmod Database, ACM. Available http://www.sigmod.org/record/xml.

Tatarinov, I., S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita and C. Zhang, 2002. Storing and querying ordered XML using a relational database system. Proc. ACM SIGMOD, pp: 204-215.

Thonangi, R., 2006. A concise labeling scheme for XML data. In Proc. of ACM SIGMOD, COMAD (In Press).

UW, 2002. University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/.

Wu, Y., J.M. Patel and H.V. Jagadish, 2003. Structural join order selection for XML query optimization. Proc. ICDE, pp: 443-454.

Wu, X., M.L. Lee and W. Hsu, 2004. A Prime Number Labeling Scheme for Dynamic Ordered XML Tree. In: Proc. of ICDE, pp: 66-78.

Yan, L. and Z. Liang, 2005. Multiple Schema Based XML Indexing. Lecture Notes Comput. Sci., 3619: 891-900.

Yao, J.T. and M. Zhang, 2004. A fast tree pattern matching algorithm for XML query. Proc. IEEE/WIC/ACM, pp: 235-241.

Zhang, C., J. Naughton, D. DeWitty, Q. Luo and G. Lohman, 2001. On supporting containment queries in relational database management systems. Proc. ACM SIGMOD, pp: 425-436.

Zhang, W., D. Liu and J. Li, 2004. An encoding scheme for indexing XML data. In: Proceeding of IEEE Internatinal Conference On e-Technology, e-Commernce and e-Service, pp: 525-528.

Zou, Q., S. Liu and W. Wesley Chu, 2004. Ctree: A compact tree for indexing XML data. Proc. WIDM, pp: 39-46.