American Journal of Applied Sciences 5 (9): 1212-1218, 2008
ISSN 1546-9239
© 2008 Science Publications

TwigX-Guide: Twig Query Pattern Matching for XML Trees

Su-Cheng Haw and Chien-Sing Lee
Multimedia University, Faculty of Information Technology, 63100 Cyberjaya, Malaysia

Abstract: The growing importance of XML and the lack of efficient solutions for managing and querying XML data have led to the development of hybrid systems. We present a hybrid system, TwigX-Guide; an extension of the well-known DataGuide index and region encoding labeling to support twig query processing. With TwigX-Guide, a complex query can be decomposed into a set of path queries, which are evaluated individually by retrieving the path or node matches from the DataGuide index table and subsequently joining the results using the holistic twig join algorithm TwigStack. TwigX-Guide improves the performance of TwigStack for queries with parent-child relationships and mixed relationships by reducing the number of joins needed to evaluate a query. Experimental results indicate that TwigX-Guide can process path and twig queries on an average 38% better than the TwigStack algorithm, 29% better than TwigINLAB and 11% better than TwigStackList in terms of execution time.

Key words: XML, query pattern matching, region encoding, DataGuide, hybrid system

INTRODUCTION

With the rapid emergence of XML as an enabler for data exchange and data transfer over the Web, querying XML data has become a major concern. Since XML is a semi-structured data, there are typically two types of user queries; namely full-text queries (keyword-based search) and structural queries (complex queries specified in tree-like structure). A keyword search is similar to content retrieval in information retrieval technology. However, to support structural query, we need an effective way to match (i) the query node tag and (ii) the query edge which is either Parent-Child (P-C) or Ancestor-Descendant (A-D) against the XML database.

There are two main approaches to processing such queries, namely: 1) Traversing the XML database sequentially to find the matching pattern and 2) Query processing using the decomposition-matching-merging approach. Using the first approach, many researchers have complemented it with indexes to address the degradation problem due to excessive traversal. These indexes reduce the search scope by creating and traversing the path summary of the XML tree instead. DataGuide[1] indexes each distinct raw data path to facilitate the evaluation of simple path expression. T-index[2] selects paths based on specific templates, while APEX[3], A(k)-index[4] and D(k)-index[5] select the most frequently-used paths in queries by restricting the path length to k. Although path indexing greatly speeds up the evaluation of path queries with P-C edges, it needs expensive join operations for processing queries with multiple branches and queries with A-D edges.

MPMGJN[6], Stack-Tree[7], TwigStack[8] and TwigStackList[9] algorithms are examples of the second approach. These algorithms are based on region encoding <*start, end, level*> labeling of the XML tree. MPMGJN and Stack-Tree algorithms decompose query pattern into basic binary relationships between pairs of nodes and structurally join pairs of nodes from the two lists to produce the matching results. TwigStack evaluates twig query as a whole without decompositing it into binary relationships. Hence, memory is not used unwisely to store irrelevant nodes. Nevertheless, TwigStack is sub-optimal for queries with A-D edges only. Lu *et al.*[9] extend TwigStack by proposing TwigStackList, which can support both P-C and A-D edges efficiently. Their technique is to look-ahead by reading some elements in input data streams and cache 'potential' elements to the main memory.

The work presented in this paper is motivated by the following observations:

- Although TwigStack[8] is optimal in supporting queries with A-D edges, the algorithm is still inefficient for queries with P-C and mixed edges. This limitation was due to its less selective criteria,

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

which pushes all nodes as intermediate results as long as it has the matching tag and is in the region range (node *A* is an ancestor of node *B* iff *A.start* <*B.start* && *A.end*> *B.end*). Thus, this algorithm produces large 'useless' intermediate results, leading to higher processing cost to check for possible merge-able paths in the merging phase. In addition, this algorithm requires a total of (*N*-1) joins, where *N* is the number of query nodes in an input query. Since join is expensive in query processing, a method to reduce it is crucial.

- Although DataGuide[1] is effective in summarizing all path information, it is unable to support twig queries and queries with A-D edges because it does not preserve the hierarchical relationship among individual nodes.

The contribution of this paper is as follows:

- We propose the TwigX-Guide system architecture, which extends the existing DataGuide and TwigStack to accelerate twig query processing.
- We propose two new algorithms: The CutMatchMergePath algorithm to process path query and the CutMatchMergeTwig algorithm to process twig query. We show that using these two algorithms require fewer joins and produce smaller intermediate results than other approaches, which are based solely on region encoding.
- We implement the TwigX-Guide system and experimentally show that TwigX-Guide outperforms TwigStack by about 38%, TwigINLAB[10] by about 29% and TwigStackList[9] by about 11% in terms of execution time on Nasa[11] dataset.

MATERIALS AND METHODS

The key idea: A query consists of a sequence of alternate edges and tags. As mentioned earlier, the edges may be either P-C or A-D relationships. To process A-D edges efficiently, region encoding[8, 9] was commonly used. Each node in the data tree is labeled with <*start, end, level*> where start and end attributes can be generated by doing a pre-order traversal of the tree and sequentially assigning a number at each visit and level is the distance of a node from the root of the tree. Using this labeling scheme, *node1* is the ancestor of *node2* iff *node1.start* <*node2.start* and *node1.end*> *node2.end*. The key idea of this labeling is to speed up query with A-D edges. However, since this labeling does not contain hierarchical path information, it is insufficient to provide quick determination of query with P-C edges.

Table 1: Summary on characteristics of TwigStack, DataGuide and TwigX-Guide

TwigStack	DataGuide	TwigX-Guide
No path information	Path index	Path index
Node labeled based on region encoding	Node label based on nodeID	Node labeled based on region encoding
Support twig query	No support for twig query	Support twig query
Support query with A-D edges	No support for query with A-D edges	Support query with A-D edges
Less efficient to support query with P-C edges	Support query with P-C edges efficiently	Support query with P-Cedges efficiently

On the other hand, DataGuide provides general path indexes that summarize all paths in the data tree starting from the root. Each label path in DataGuide is unique. DataGuide is effective to answer query with P-C edges by matching the query path against the label path directly. Nevertheless, it is unable to answer twig query and query with A-D edges since it does not preserve the hierarchical relationships among individual nodes. Combining the beautiful features of region encoding in TwigStack and the path summary of DataGuide, we propose to "pre-match" the query nodes along the root-to-leaf paths in DataGuide (instead of evaluating each node individually) and structurally join the results with TwigStack algorithm. As a result, the number of joins required is reduced. Eventually, this will speed up query evaluation.

Table 1 summarizes the pros and cons of TwigStack, DataGuide and our proposed system, TwigX-Guide.

Figure 1 shows our TwigX-Guide system architecture, which consists of the Labeling and Indexing Generator in the offline processing and the Query Engine in the online processing.

Offline processing-label and index generator: In its traditional way, each DataGuide node is in the form of (*nodeID, data path*). We observe that the *nodeID* does not provide any information besides annotating a particular node. Thus, instead of using *nodeID*, we annotate each node by their region encoding label as introduced earlier. On the other hand, we remove the level attribute in the region encoding because it is no longer needed to check for P-C relationship. All P-C relationships can be determined directly from the DataGuide.

During the offline processing, we pre-process the XML tree into a set of streams labeled with <*start-end*> for each element and path occurrence. Thus, instead of checking for matches against the whole XML tree, only the "qualified" streams are presented as input.

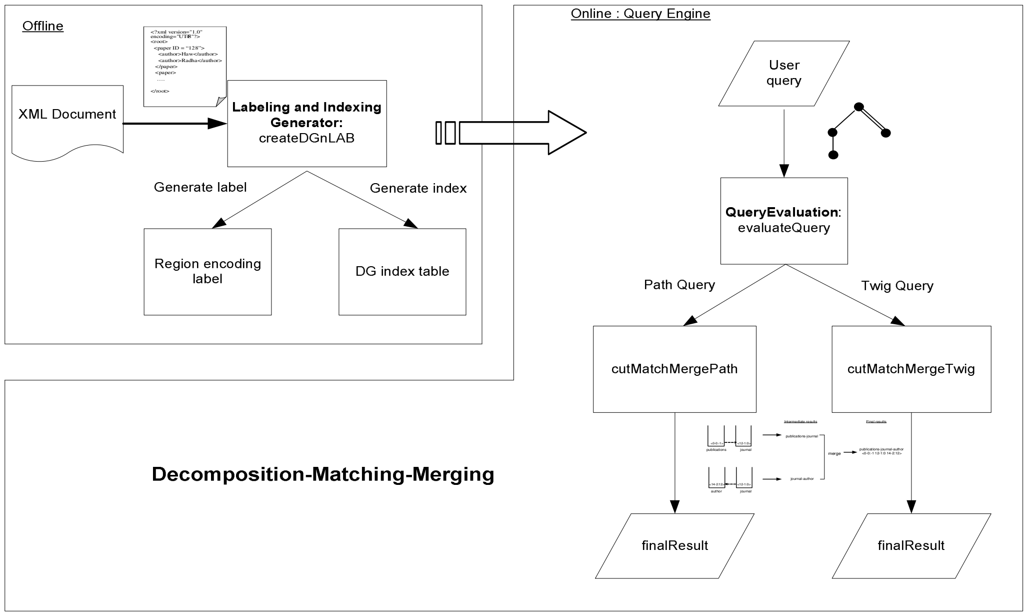

Fig. 1: TwigX-Guide system architecture

Figure 2a shows an example of an XML tree for the domain "Publications" while Figure 2b depicts the fragment of XML streams stored in the DG hashtable during the offline processing.

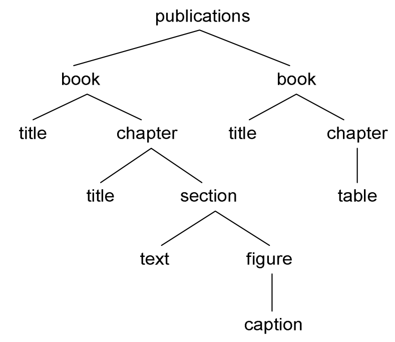

Fig. 2a: Example of Publications XML

DG	
//publications	<0-25>
//book	<1-16> <17-24>
//title	<2-3><5–6><18-19>
//book/title	<2-3><18-19>
//chapter	<4-15><20-23>
//book/chapter	<4-15><20-23>
…	…
//caption	<11-12>
//figure/caption	<11-12>
//section/figure/caption	<11-12>
//chapter/section/figure/caption	<11-12>
//book/chapter/section/figure/caption	<11-12>
…	…

Fig. 2b: Fragment of the DG index table

Online processing-query evaluation: The input query is analyzed in the function evaluateQuery (depicted in Algorithm 1) to determine the type of query for processing. If it is a path query, the algorithm CutMatchMergePath is invoked. Else if the input query is a twig query, the algorithm CutMatchMergeTwig will be executed.

Algorithm 1: to evaluate query based on type of query

```
1.  function processQuery  {
2.     input : a path or twig query Q
3.     output : final solutions matches the input query
4.      Pre-order traversal (Q) {
5.         for each node q' in Q
6.                 count number of children
7.      }
8.      if (each node except the leaf node have only one child)
9.                 finalSolution = CutMatchMergePath (Q)
10.  else if (any node has more than one child)
11.                finalSolution = CutMatchMergeTwig (Q)
12.  } //end function
```

CutMatchMergePath algorithm: The algorithm (depicted in Algorithm 2) decomposes the input path query into one or more segments if there are any A-D edges such as p//q into p and //q (lines 6-13). The segment that contains A-D edges is put into temp_result (temporary results). However, the remainder segmented path query with P-C edges is formed in the function partitionTwig. Lines 16-22 are the matching and merging processes. If the path query only contains P-C edges (lines 16-17), the final results can be obtained directly from the DG index table. Conversely, if the path query only contains A-D edges (lines 18-19), the final results can be obtained by joining the temporary results with the TwigStack algorithm. However, if the path query contains both P-C and A-D edges (lines 20-21), the function connectPQ (lines 25-32) will be invoked to match-merge the final results.

Algorithm 2: Cut node and its subtree whenever there is A-D relationship for a path query

```
1.function CutMatchMergePath  {
2.   input : a path query, P and data guide table, DG
3.   output : final solution matches the input path query
4.   /*Vector temp_result to store each suffix path expression
5.   Vector vPathQuery to store path query after partition  */
6.   Pre-order traversal (P) {
7.      if (current node reached by a // edge)   {
8.         let P' = the subtree rooted at the current // edge
9.         Cut P' from P
10.        temp_result.add (cutDescendantPathQuery (P'))
11.     }
12. }
13. vPathQuery = partitionTwig(P)
14. t_array[] size = temp_result.size()
15. p_arrayp[] size = vPathQuery.size()
16. if ((temp_result.size()==0) && (vPathQuery.size()==1))
17.    finalSolution = hashDG to get path label
18. else if ((temp_result.size()>0) && (vPathQuery.size()=0))
19.    finalSolution = TwigStack(temp_result)
20. else if ((temp_result.size()>0)&&(vPathQuery.size()==1))
21.     finalSolution = connectPQ(temp_result, vPathQuery)
22. return finalSolution
23. }// end function
24.
25. function connectPQ(descendantAxis, vPathQuery) {
26.   input: descendant axis path/node and path query
27.   output: connected path query
28.   for each ti in each descendantAxis
29.      for each pi in vPathQuery
30.         if ( ti.start > pi.start && ti.end < pi.end)
31.                    addToSol(ti, pi)
32.  }
```

Example 1: Let Q1: book/chapter//figure/caption. Since there is an A-D edge, the path query is split into two paths /book/chapter and //figure/caption. As a result, only one join is needed to merge the paths. From the DG index table, only two nodes, <4-15> and <20-23> fulfill the first path and one node <11-12> fulfill the second path. Next, the function connectPQ is invoked to match and merge the final results. Node <11-12> is in the range of <4-15>. As such, a solution is formed. However, node <11-12> is out of the range of <20-23>. Therefore, there is no solution for this case. As a summary, there is only one final answer.

CutMatchMergeTwig algorithm: The CutMatchMergeTwig algorithm is presented in Algorithm 3. This algorithm takes an input query, do a pre-order traversal and cut any nodes and its subtree that are reached by the descendant axis (lines 6-11). Besides, it decomposes any branch axis of the form p[[[/q1]/q2]/…]/r into p/q1, p/q2, …, p/r as in lines 12-19. The remainder segment of path query with only P-C edges is formed in the function partitionTwig. In contrast to the CutMatchMergePath algorithm, this algorithm needs to determine the TopBranchNode in order to merge the path queries as in lines 22-23. Finally, all these results are joined holistically using the TwigStack algorithm.

Algorithm 3: Cut node and its subtree whenever there is a A-D relationship or branch node

```
1. function CutMatchMergeTwig {
2.   input : a twig query, Q and data guide table, DG
3.   output : final solution matches the input query
4.   /* Vector temp_result to store each suffix path expression
5.   Vector vPathQuery to store path query after partition */
6.    Pre-order traversal (Q) {
7.      if (current node reached by a // edge)   {
8.         let Q' = the subtree rooted at the current // edge
9.         Cut Q' from Q
10.        temp_result.add (cutDescendantAndBranch (Q'))
11.     }
12.     if (current node has more than one child) {
13.        branchnode = current node tag
14.        let  Q' = the child under the branchnode
15.        Cut Q' from Q
16.        for each Q' {
17.           let Q'= //Q'
18.           temp_result.add(cutDescendantAndBranch(Q'))
19.        }
20.     }
21.  }
22.  vPathQuery = partitionTwig(Q)
23.  TopBranchNode = branchnode nearest to the root of the
     twig
24.  finalSolution = twigStack(temp_result, vPathQuery,
     TopBranchNode)
25. return finalSolution
26. }// end function
```

Example 2: Let Q2: book/chapter[/title]//figure/caption. Since Q2 contains branching nodes and an A-D edge, this input query is decomposed into two paths; /book/chapter/title and //figure/caption. Note that the title node is not segmented because it is a leaf node. Only nodes <5-6> and <11-12> fulfill the two paths respectively. The TopBranchNode has two nodes, <4-15> and <20-23>. The final result is then obtained by holistically joining these nodes together as illustrated in Figure 3.

RESULTS AND DISCUSSION

We have implemented TwigX-Guide using Java API for XML Processing (JAXP). Experiments have been carried out on the Nasa dataset obtained from the University of Washington repository[11]. All our experiments are performed on 1.7GHz Pentium IV processor with 512 MB SDRAM running on Windows XP using the queries listed in Table 2. Fig. 4 shows the query execution time of TwigX-Guide with respect to TwigStack[8], TwigINLAB[10] and TwigStackList[9].

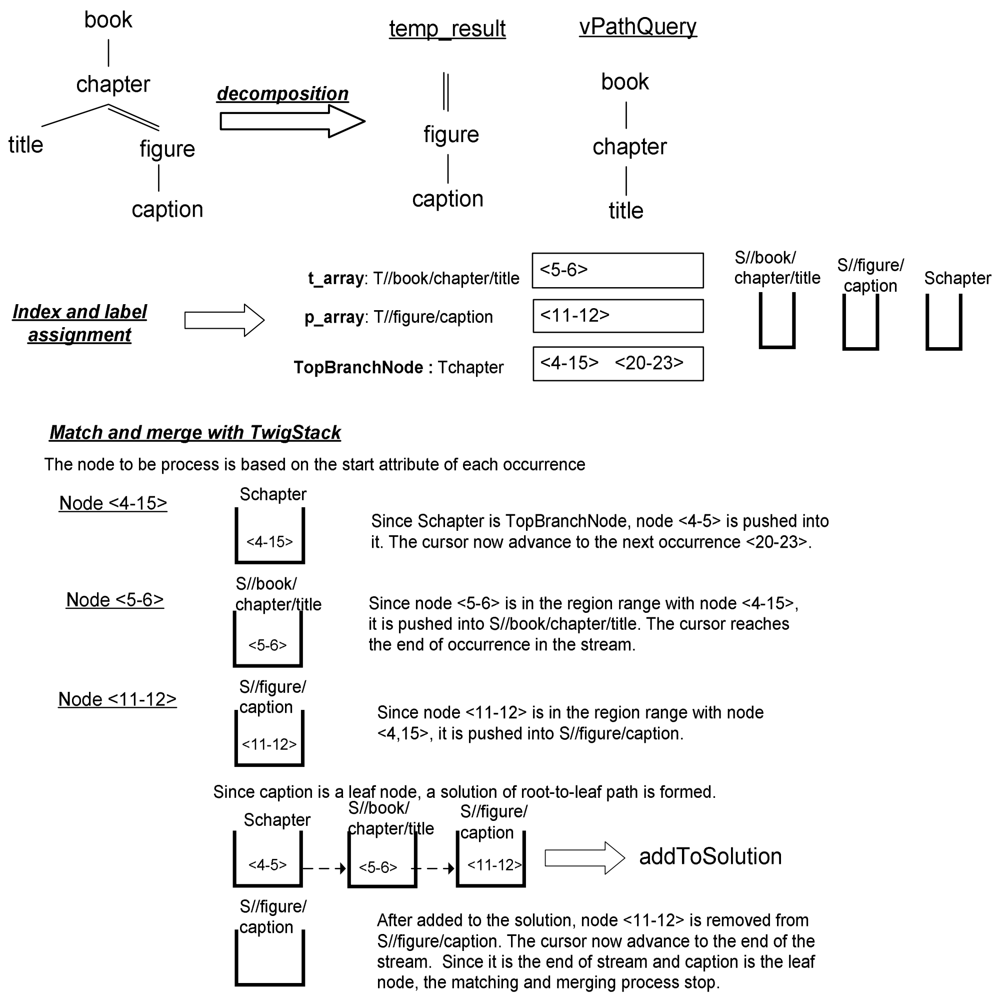
book
chapter
title
figure
caption
decomposition
temp_result
figure
caption
vPathQuery
book
chapter
title
Index and label assignment
t_array: T//book/chapter/title
<5-6>
p_array: T//figure/caption
<11-12>
TopBranchNode : Tchapter
<4-15> <20-23>
S//book/chapter/title
S//figure/caption
Schapter
Match and merge with TwigStack
The node to be process is based on the start attribute of each occurrence
Node <4-15>
Schapter
<4-15>
Since Schapter is TopBranchNode, node <4-5> is pushed into it. The cursor now advance to the next occurrence <20-23>.
Node <5-6>
S//book/chapter/title
<5-6>
Since node <5-6> is in the region range with node <4-15>, it is pushed into S//book/chapter/title. The cursor reaches the end of occurrence in the stream.
Node <11-12>
S//figure/caption
<11-12>
Since node <11-12> is in the region range with node <4,15>, it is pushed into S//figure/caption.
Since caption is a leaf node, a solution of root-to-leaf path is formed.
Schapter
<4-5>
S//book/chapter/title
<5-6>
S//figure/caption
<11-12>
addToSolution
S//figure/caption
After added to the solution, node <11-12> is removed from S//figure/caption. The cursor now advance to the end of the stream. Since it is the end of stream and caption is the leaf node, the matching and merging process stop.

Fig. 3: TwigX-Guide query processing steps

Table 2: Query set

Query Edges	Query	Query notation
P-C only	Q1	dataset[/title]/referenes/source/other/author/lastname
	Q2	field[/name[/definition/footnote/para]]/units
A-D only	Q3	datasets//fererence//year
	Q4	history[//creator]//day
Mixed edges	Q5	reference/source[//title]//reference
	Q6	datasets[//reference/source/other/title]//descriptions/description/para

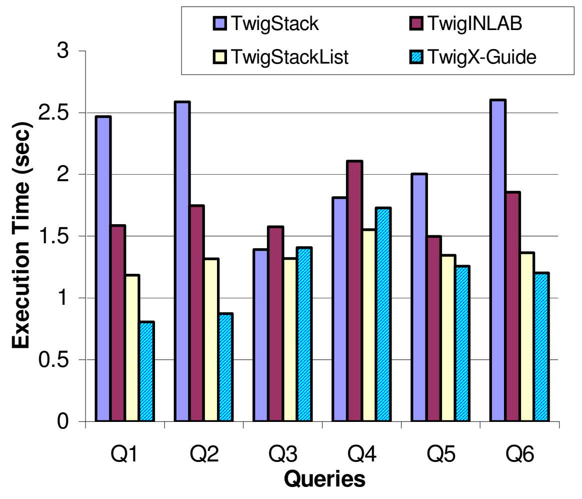

Fig. 4: Performance evaluation results

From Fig. 4, we draw several observations and conclusions:-

- When the query contains only P-C edges as shown in Q1 and Q2, TwigX-Guide performs significantly faster; about 66% faster as compared to TwigStack, 49% faster as compared to TwigINLAB and about 33% faster as compared to TwigStackList. This is because all the other approaches require six joins with a lot of disk access, while our approach requires only one join. However, TwigStackList is much faster as compared to TwigStack and TwigINLAB because it caches 'potential' elements to the main memory for faster processing.
- When the query contains A-D edges only as shown in Q3 and Q4, the performance of all the approaches are comparable. This is because all approaches are based solely on region encoding labeling to retrieve "qualified" streams for further processing. Besides, all the approaches require the same number of joins.
- For query with mixed edges (containing both P-C and A-D edges) as shown in Q5 and Q6, TwigX-Guide is 45% faster than TwigStack, 25% faster than TwigINLAB and 9% faster than TwigStackList approaches. This is because, the number of joins required in TwigX-Guide is far less. For example, for Q6, our approach requires only two joins while the other approaches need seven joins.

CONCLUSION

We have presented TwigX-Guide for efficient processing on complex queries. The system extends DataGuide and region encoding to accelerate query processing. With this extension, DataGuide is able to process twig queries and queries with A-D relationships efficiently. Region encoding, on the other hand, benefits DataGuide in terms of utilizing the path index to annotate the hierarchical relationships among individual nodes. With TwigX-Guide, a complex query can be decomposed into a set of path queries, which are evaluated individually by retrieving the path or node matches from the DataGuide index table and subsequently joining the results using the holistic twig join algorithm TwigStack. TwigX-Guide improves the performance of TwigStack for queries with P-C edges and mixed edges by reducing the number of joins needed to evaluate a query.

REFERENCES

1. Goldman, R. and J. Widom, 1997. Data Guides: Enabling Query Formulation and Optimization in Semistructured Databases, Proceedings of VLDB, pp: 436-445.
2. Milo, T. and D. Suciu, 1999. Index structures for path expression, Proceedings of ICDT, pp: 277-295.
3. Chung, C.W., J.K. Min and K. Shim, 2002. APEX: An Adaptive Path Index for XML data, Proceedings of ACM SIGMOD, pp: 121-132.
4. Kaushik, R., D. Shenoy, P. Bohannon and E. Gudes, 2002. Exploiting Local Similarity to Efficiently Index Paths in Graph-Structured Data, Proceedings of ICDE, pp: 129-140.
5. Chen, Q., A. Lim, K. Ong and J. Tang, 2003. D (k)-index: An adaptive structural summary for graph-structured data, Proceedings of SIGMOD, pp: 134-144.

6. Zhang, C., J. Naughton, D. DeWitty, Q. Luo and G. Lohman, 2001. On Supporting Containment Queries in Relational Database Management Systems, Proceedings of ACM SIGMOD, pp: 425-436.
7. 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, Proceedings of ICDE, pp: 141-152.
8. Bruno, N., D. Srivastava and N. Koudas, 2002. Holistic Twig Joins: Optimal XML Pattern Matching, Proceedings of ACM SIGMOD, pp: 310-321.
9. Lu, J., T. Chen and T.W. Ling, 2004. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. Proceedings of CIKM, pp: 533-542.
10. Haw, S.C. and C.S. Lee, 2007. Stack-based Pattern Matching Algorithm for XML Query Processing, Journal of Digital Information, 2: 82-87.
11. University of Washington XML Repository. Available
<http://www.cs.washington.edu/research/xmldatasets/>