Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 7 (20): 2934-2946, 2007
ISSN 1812-5654

Structural Query Optimization in Native XML Databases: A Hybrid Approach

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

Abstract: As XML (eXtensible Mark-up Language) is gaining its popularity in data exchange over the Web, querying XML data has become an important issue to be addressed. In native XML databases (NXD), XML documents are usually modeled as trees and XML queries are typically specified in path expression. The primitive structural relationships are Parent-Child (P-C), Ancestor-Descendant (A-D), sibling and ordered query. Thus, a suitable and compact labeling scheme is crucial to identify these relationships and henceforth to process the query efficiently. We propose a novel labeling scheme consisting of <self-level:parent> to support all these relationships efficiently. Besides, we adopt the decomposition-matching-merging approach for structural query processing and propose a hybrid query optimization technique, TwigINLAB to process and optimize the twig query evaluation. Experimental results indicate that TwigINLAB can process all types of XML queries 15% better than the TwigStack algorithm in terms of execution time in most test cases.

Key words: XML, query processing, query optimization, twig query, hybrid approach

INTRODUCTION

eXtensible Mark-up Language (XML) is primarily used as a data exchange format. However, the amount of data exchanged and transmitted often grows exponentially via the Web medium. As such, web applications such as search engines, e-commerce and e-learning portals require advanced tools for managing data. Furthermore, user demands are no longer restricted to retrieval of full-text queries but also requests for more specific data (structural queries). This drives the requirement for efficient and reliable storage and query of large-scale XML data (Haw and Rao, 2005).

This study is concerned with structural queries. There are two main approaches for structural query processing in native XML databases (NXD). The first approach is to traverse the XML database sequentially to find the matching pattern. This approach certainly poses a new challenge, because it may not meet the processing requirements under heavy access requests (Li and Moon, 2001). As a result, some researchers had complemented it with index-based techniques in order to reduce the portion to be scanned during query evaluation. In this context, many indexing techniques have been introduced to optimize the query processing speed.

The second approach is executed through a series of processes involving decomposition, matching and merging (Yao and Zhang, 2004). Firstly, a complex query pattern is decomposed into a set of basic binary structural relationship between pairs of nodes. These relationships can be either of ancestor-descendant (A-D) or parent-child (P-C) relationship. The query pattern can then be matched by matching each of the binary structural relationships against the NXD. Next, these matches are merged together to form the path solution. Another similar approach is to decompose the twig query into a set of path queries, followed by a join operation to reconstruct the matched twig pattern. However, these approaches still suffer from having large intermediate results, which may not participate in the final results. This certainly imposes serious scalability and efficiency issues.

In this study, we adopt the decomposition-matching-merging approach to decompose the twig query into a set of path queries and propose a novel hybrid query optimization technique, INLAB (combination of INdexing and LABeling techniques) which consists of createINLAB encoding and TwigINLAB algorithms. The index structures of INLAB allow us to efficiently find all elements that belong to the same parent or ancestor. The proposed labeling scheme quickly determines the A-D, P-C, sibling and ordered query relationship between elements in the NXD. Moreover, our approach can reduce the number of inspections performed on irrelevant results during the merging process and hence improves on the efficiency.

Our contribution can be summarized as follows:-

- The proposed createINLAB labeling scheme can be used for determining the four main types of relationships: (i) A-D (ii) P-C (iii) sibling and (iv) order-based queries efficiently.

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

- A query optimization algorithm, TwigINLAB to process twig queries without traversing the whole XML tree.

XML query: Much research has been done on XML query languages. Examples of prior work on XML query languages are XML-QL, Lorel, Quilt, XPath and XQuery (Abiteboul *et al*., 1997; Anonymous, 2005) and XML query algebras such as XOM, XAL, YATL and Lore algebra (Frasincar *et al*., 2002; Zhang and Dong, 1999; Abraham *et al*., 2004). XML algebra typically provides a solid ground to define the semantics of a query language, to analyze its power of expression and to perform query optimizations. The Lore algebra (McHugh and Widom, 1999) uses cost-based optimization to efficiently evaluate path expressions based on an idiosyncratic set of logical and physical operators. Christophides *et al.* (2000) propose YATL, which introduces a set of idiosyncratic operators based on hybrid technologies of relational and object-oriented databases. XOM (Zhang and Dong, 1999) is a complete and closed algebra composed of six object operators, but do not provide any optimization support. El-Sayed *et al.* (2003) study the challenges related to handling order in the XML context and propose a key encoding for XML nodes to support node identity and node order. Recently, Abraham *et al.* (2004) propose to extend the Niagara algebra (Galanis *et al*., 2002; Viglas *et al*., 2002) by proposing an unnest-and-select operator to simplify the logical expression of query, improves processing time and to reduce the storage requirement. On the other hand, XML query languages typically provide the means to extract and manipulate data from XML documents. Although most of the query languages differ in detailed grammars and representation, they share a common feature, that is, queries usually make use of path expression for query evaluation (Yao and Zhang, 2004). Hence, we will discuss two types of queries, namely path query and twig query.

Path query defines query on one single element at a time while twig query defines query on two or more elements. Thus, they are also known as Simple Path Expression and Branching Path Expression, respectively. In both cases, query nodes may be elements, attributes or texts. However, query edges for path query are either P-C or A-D relationships, whereas query edges for twig query pattern may be either P-C, A-D or sibling (preceding and following) relationships. In XPath (Anonymous, 2004) notation, P-C relationship is denoted by "/" while A-D relationship is denoted by "//". There are two types of sibling relationships, which also determine the ordering of the relationship; namely preceding-sibling (denoted by preceding-sibling::*) and following-sibling (denoted by following-sibling::*). Figure 1 shows the queries and their corresponding path expression specified in XPath notation. For present study, our focus is on twig query.

Indexing and labeling: As mentioned earlier, some researchers have complemented structural queries with index-based techniques. 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*., 2002a), Index Fabric (Cooper *et al*., 2001), APEX Index (Chung *et al*., 2002), D(k)-index (Chen *et al*., 2003), M(k)-index (He and Yang, 2004), F&B index (Kaushik *et al*., 2002b) and Structural Map (Zheng *et al*., 2002). DataGuide provides general path indices that summarize all paths in the tree starting from the root to the respective node. T-index selects paths based on specific templates, while APEX, A(k)-index and D(k)-index select the most frequently used paths in queries by restricting the path length to k. M(k)-index improves D(k)-index by avoiding over-refinement of irrelevant index by targeting the data nodes which are relevant to frequent queries only. Zheng *et al.* (2002) propose two types of structural maps namely Full Structural Map and Partial Structural Map. Full Structural Map indexes all of the path segments between any two nodes in the guide while Partial Structural Map indexes only partial paths that are frequently accessed. The F&B index (Kaushik *et al*., 2002b) uses the Forward and Backward bi-similarity technique to answer all branching

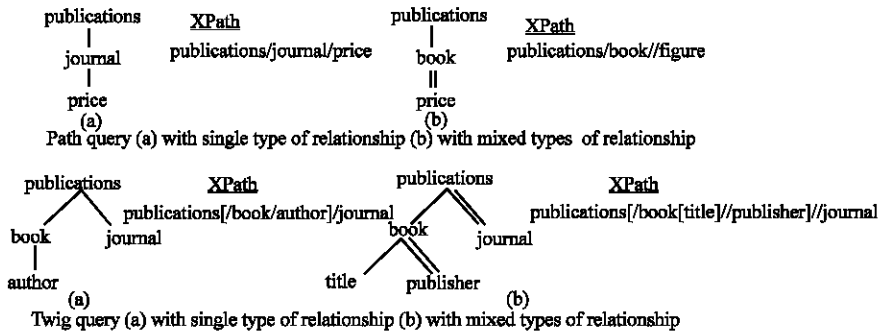

Fig. 1: Various types of queries

path expressions. However, the size of the F&B index is usually as large as the original document. Unlike the other approaches, Index Fabric encodes its label path as string and stores them in a balanced Patricia trie. However, all these approaches still suffer from large index size growth.

As such, several labeling schemes have been proposed to overcome the shortcomings of structural indexing and to annotate the hierarchical relationship present in the XML tree. Among some of the labeling schemes are tree traversal order (Dietz, 1982), tree location address (Kimber, 1993), extended preorder traversal (Li and Moon, 2001), simple prefix (Cohen *et al*., 2002), ORDPATH (O'Neil *et al*., 2004), prime number labeling (Wu *et al*., 2004) and BIRD (Weigel *et al*., 2005). In tree traversal order, each node is labeled with a pair of unique integer consisting of preorder and postorder traversal sequences. Thus, given any two nodes A and B, A is an ancestor of B if and only if A occurs before B in the preorder traversal of XML tree and after B in the postorder traversal. Li and Moon (2001) propose to extend Dietz's labeling scheme and to integrate with an indexing mechanism to enable efficient search by value and structure. Their labeling scheme is based on a pair of numbers <order, size>, where order is generated during the preorder traversal and size is assigned as an arbitrary integer larger than the total number of descendants for each node to accommodate future insertion gracefully. In tree location address and simple prefix, each ancestor node is a prefix of its descendant. A node id (nid) is the concatenation of the nids through the path from the root to the respective node. The concept for ORDPATH is similar to Dewey Order (Tatarinov *et al*., 2002), which encodes the P-C relationship by extending the parent label with a component for the child. However, ORDPATH reserves the even numbering for further node insertion. On the other hand, the BIRD labeling scheme is based on a structural summary similar to DataGuide (Goldman and Widom, 1997). BIRD labeling is compatible with document order where the nodes visited later in a pre-order traversal of the document tree have larger BIRD numbers. In addition, each node has an integer weight, which determines whether the reconstruction process is necessary. In prime number labeling, using the top-down approach, each non-leaf node will be given a unique prime number. The label of each node is the product of its parent nodes' label (parent-label) and its own assigned number (self-label).

Twig query processing

Navigational approach based on algebraic expression: Algebraic expressions render the possible algebraic optimization for faster evaluation in a Query Optimizer (Jagadish *et al*., 2001; Abraham *et al*., 2004; Zhang, 2006). Recently, there emerged two types of navigational approaches to speed up query processing, namely, query-driven and data-driven. In the query-driven approach such as Natix (Brantner *et al*., 2005), each location step in the path expression is translated into a transition of a set of nodes. Conversely, in the data-driven approach such as XNav (Josifovski *et al*., 2005), the query is translated into an automaton and the data tree is traversed according to the current state of the automaton.

Basically, the Natix query-processing engine translates a path query into a native XML algebraic expression. Next, each location step is translated into an Unnest-Map operator, which is then translated into a physical operator to directly retrieve the navigation in the Natix storage system. Kanne *et al*. (2005) optimize the Natix I/O performance by considering multiple Unnest-Map operators as a whole and schedule I/O accordingly.

XNav is a navigational processing technique based on finite state automata. This algorithm requires only one pass of the input XML data, possibly skipping some tree nodes. Therefore, it is analogous to the sequential scan operator in relational database systems. The difference is that XNav has a more complex data access pattern.

There are strengths in the algebraic approach, i.e., it facilitates iterator-based applications, support full-text and structural queries and proven performance in relational database systems (Brantner *et al*., 2005). However, in this study, we choose to do optimization using the decomposition-match-merge approach, because our main concern is on structural queries only as well as data streaming processing under constrained devices with limited computing and disk storage space ability. Future work may involve comparisons between the algebraic and the decomposition-match-merge approaches.

Decomposition-matching-merging approach: Twig query pattern matching typically involves decomposition-matching-merging processes with (1) decomposition of query pattern into a set of binary relationships between each pair of nodes or into a set of path queries (2) matching of each binary component of the query pattern against the NXD and (3) merging-joining matched pairs to obtain the final result.

Our INLAB approach focuses on optimizing all three sub-processes; introducing novel compact labeling scheme, optimizing the matching phase and reducing the number of inspection required in the merging phase.

In the first sub-process, most researchers build inverted indices (very popularly used in information retrieval systems) on the XML document. Text words are

indexed in T-index while elements are indexed in E-index, which maps elements to inverted lists (Zhang *et al*., 2001; Bruno *et al*., 2002; Lu *et al*., 2004). These techniques use the labeling of (docno, begin: end, level) for an element and (docno, wordno, level) for a text word as the positional representation of XML elements and texts. However, we use <self-level: parent> as the positional representation instead.

Most literature focuses on the second sub-process: matching binary structural relationships. Zhang *et al.* (2001) and Al-Khalifa *et al.* (2002) propose MPMGJN (multi-predicate Merge Join) and Stack-Tree algorithm, respectively to match the binary structural relationship. These algorithms accept two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matching of the binary relationships. The difference between MPMGJN and Stack-Tree is that MPMGJN requires multiple scans on input lists for the matching process. In contrast, Stack-Tree algorithms are more efficient as they use stack to maintain the ancestor or parent nodes and therefore require only one time scan per input list. However, these approaches still produce large intermediate results. To address this problem, Bruno *et al.* (2002) propose TwigStack, a holistic twig join algorithm which 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 relationships. Lu *et al.* (2004) extend TwigStack and propose TwigStackList, which can support both P-C and A-D relationships efficiently. Jiang *et al.* (2003) extended the holistic approach and propose pipelining joining multiple inverted lists at one time so that no intermediate results are generated. Besides, they also extended the holistic approach to process twig queries with an OR predicate (Jiang *et al*., 2004). Recently, Jiao *et al.* (2005) and Yu *et al.* (2006) propose a holistic path join algorithm for path and twig query, respectively using NOT predicates.

Another similar approach is to decompose the twig query into a set of path queries instead. Polyzotis *et al.* (2004) propose methods to reduce the number of intermediate results by introducing a filtration step based on some notion of synopses to facilitate query-approximate answers. They propose both TREESKETCH and TWIG-XSKETCH. Amer-Yahia *et al.* (2002) propose to preprocess the query patterns before the matching phase is executed. Since the efficiency of tree pattern matching depends on the size of the pattern, it is essential to identify and eliminate redundant nodes in the pattern before the matching phase takes place. On the other hand, Zezula *et al.* (2004) propose a novel technique, tree signature, to represent tree structures as ordered sequences of pre-order and post-order ranks of the nodes. They use tree signatures as index structure and find qualifying patterns through integration of structurally consistent path query.

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) suggest partitioning all nodes in an extent into several clusters. Given two extents to be joined, Kim *et al.* (2004) propose filtering out unnecessary clusters in both extents prior to the joining process.

OVERVIEW OF INLAB

CreateINLAB labeling scheme: In our createINLAB labeling scheme, given an XML tree T, any label consists of <self-level:parent> representation, where (i) self is obtained by doing a pre-order traversal of the tree nodes (ii) level of a node is its distance from the root and (iii) parent is the direct node which relates to the self node.

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

P-C relationship: $Node_1$ is the parent of $node_2$ if and only if $node_1.self = node_2.parent$.

Sibling relationship: $Node_1$ is the sibling of $node_2$ if and only if $node_1.parent = node_2.parent$.

Ordered query relationship (predecessor and successor):

- $Node_1$ is the predecessor node of $node_2$ if and only if $node_1.self < node_2.self$.
- $Node_1$ is the successor node of $node_2$ if and only if $node_1.self > node_2.self$.

A-D relationship: $Node_1$ is possible as an ancestor of $node_2$ if and only if level difference, $leveldiff = node_2.level - node_1.level >= 1$. A multiple look-up via PCTable shown in Fig. 2b is necessary as long as the $leveldiff > 1$ is true to confirm the A-D relationship. Further explanation is shown in Algorithm 3.

Figure 2a and b show the fragment of streams and index table generated based on the sample XML document during the createINLAB encoding process.

Overview of TwigINLAB processing: Figure 3 shows the overall processes involved in TwigINLAB processing. Initially, the query pattern is being analyzed in

```
<publications>
    <book ISBN = "0-7895-6514-5">
        <title category = "Multimedia"> Discovering … </title>
        <author>Gary B. Shelly </author>
        <author>Thomas J. Cashman </author>
        <author>Misty E. Vermaat </author>
        <publicationInfo>
            <place>Boston</place>
            <publisher>Thomson Learning</publisher>
            <dateIssued>2002</dateIssued>
            <dateRevised>2003</dateRevised>
        </publicationInfo>
            <price>60.99</price>
    </book>
    <journal number = "J1" year="2003">
        <title category = "XML"> XML Query …</title>
        <author>S.C. Haw </author>
        <price>20.00</price>
    </journal>
</publications>
```

publications	0-0:-1
book	1-1:0
title	2-2:1 13-2:12
author	3-2: 1 4-2: 1 5-2: 1 14-2: 12

(a)

Self	Parent
0	-1
1	0
2	1
...	...
15	12

(b)

Fig. 2: (a) Fragment of streams generated (b) fragment of index table generated based on a sample XML document

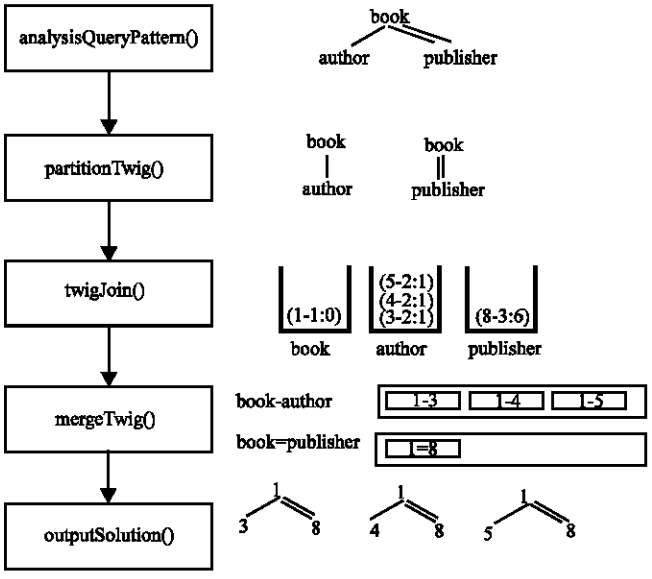

Fig. 3: Overall flow of TwigINLAB

analysisQueryPattern() function. For each query edge, if the twig is of P-C relationship, the parent and child(s) details will be updated in the twigPC repository. Next, the partitionTwig() function takes place. During this function, the twig pattern is decomposed into two or more path queries. Each possible match is pushed into the stack in the twigJoin() function. Next, these matches are merged back through the mergeTwig() function. Finally, the final solutions are output through the outputSolution() function.

Some basic operations involved in these functions include operation over stack, operation over hashtable and operation over vector. Operations over a stack are empty() to examine if the stack contains no entry, pop() to remove an entry, push() to add an entry, peek() to peek on the entry at the topmost, element At (index) to retrieve the entry at position specified and size() to return the total entry in a particular stack. Operations over a hashtable includes get(key) to retrieve each value which belong to the key and put(key, value) to add an entry into the hashtable. Operation over a vector is addElement(entry) to add an entry. Function isRoot(q) and isLeaf(q) each examines whether a query node is a root or a leaf node, respectively.

The analysisQueryPattern() function: The analysisQueryPattern() function is presented in Algorithm 1. For each SAX event, if the start tag is found (lines 8-24), this function pushes the tag into tStack. At every point during computation, each entry in tStack (from bottom to top) is guaranteed to lie on a root to leaf sequence. This function also identifies each relationship between two adjacent nodes and stores it into the twigPC table as in line 18. Every entry in the twigPC contains unique parent query node information as control in line 11. However, if there are more query nodes, which belong to the same parent, they will be added as a collection of child's (line 13). If the query edge is of P-C relationship as specified by character 1, this information will be updated in the twigPC table by the function setPCRelation() as in line 32. On the other hand, for each SAX event where an end tag is found, the top entry in tStack is removed. The output of this function using the query specified in Fig. 3 is shown in Fig. 4.

The partitionTwig() function: The partitionTwig() function is depicted in Algorithm 2. This function partitions the twig query pattern into a set of path queries. Starting from the root of twig query pattern, for each start tag event, it pushes the tag into twigStack as in lines 9-10. When it reaches an end tag event (lines 11-33), it checks whether the currentNode at the top of twigStack is a leaf node. If it is a leaf, the query node will be added one by one to the vpathList according to the Last In First Out (LIFO) sequence of a stack operation. Finally, if the currentNode is the root, it will firstly be added to the vpathList and next the vpathList sequence will be output in reverse order by the function reverse() (lines 44-50). Thus, the final output of this function is a set of path queries in root-to-leaf order in pq hashtable. However, if the current tag is a non-leaf node, the top entry at the top of twigStack will be removed. If the SAX event is end of

```
Algorithm 1: Analysis Query Pattern
1.    Function analysisQueryPattern  {
2.       input : twig query pattern
3.       output : twigPC table
4.       /* A hashtable twigPC to store parent and child(s)
5.          A stack tStack to store each query node sequence
6.          A vector vTemp to keep track of the current query
7.          node child(s) information*/
8.          if SAX event = a start tag <T> then {
9.                if (! tStack.empty()) {
10.                   parent = tStack.peek()
11.                   if (twigPC contains parent) {
12.                         vTemp = twigPC.get(parent)
13.                          vTemp.addElement(tag)
14.                   }
15.                   else {
16.                         create new instance of vTemp
17.                         vTemp. addElement (tag)
18.                         twigPC.put (parent, vTemp)
19.                   }
20.               }
21.               else
22.                   ROOT = tag
23.               tStack.push(tag)
24.         }
25.         if SAX event = an end tag </T> then {
26.               tStack.pop()
27.          }
28.          if SAX event = character then {
29.                if (character == "1") {
30.                      child = tStack.peek()
31.                      parent = tStack.elementAt(tStack.size()-2)
32.                      setPCRelation(parent, child)
33.               }
34.          }
35.   } //end function
```

twigPC

Parent	Child(s)
book	[author, 1] [publisher, 0]

Fig. 4: Fragment of twigPC generated during the analysisQueryPattern() function

document, for each entry in pq hashtable, it recursively calls the initialize() and twigJoin() functions (lines 34-41).

The next process involves finding matches for the set of path queries. This process is similar to pathINLAB() as presented in work (Haw and Rao, 2007). The difference is partitionTwig() processes a set of path queries recursively. For clarity, this function is rewritten again and depicted in Algorithm 3.

The twigJoin() function: Let q denote the query node in the path query and e denote the positional representation of occurrence in the encoded data streams, Tq. Function getRoot() is self-explaining; returning the root of path

```
Algorithm 2 : partition twig query into path queries
1.  Function partitionTwig(twigPC) {
2.    input : twigPC hashtable and twig query pattern
3.    output: a set of path queries
4.    /* A stack twigStack to keep track of twig node sequence
5.       A vector vpathList to store query nodes in leaf-to-root order
6.       A vector vpathQuery to store query nodes in
7.       root-to-leaft order
8.       A hashtable pq to keep each distinct path query */
9.     if (SAX event == a start tag <T>)
10.       twigStack.push(tag)
11.    if (SAX event == an end tag </T>) {
12.       index = twigStack.size() –1
13.       currentNode = twigStack.peek()
14.       if (isLeaf(currentNode)) {
15.            create new instance of vector, vpathList
16.            create new instance of vector, vpathQuery
17.            numOfLeaf++
18.            while (index > 0) {
19.              vpathList.addElement(currentNode)
20.              index—
21.              currentNode = twigStack.elementAt(index)
22.              if (isRoot(currentNode) {
23.                   vpathList.addElement(currentNode)
24.                   vpathQuery = reverse(vpathList)
25.                    if (vpathQuery has yet been stored into pq)
26.                            pq.put(vpathQuery, 0)
27.             }
28.           }
29.           twigStack.pop()
30.      }
31.      else
32.           twigStack.pop()
33.    }
34.    if (SAX event == an end document) {
35.       while (each entry in pq) {
36.           s = nextElement()
37.           initialize(s)
38.           twigINLAB(s)
39.       }
40.       mergeTwig(getPathList, numOfLeaf)
41.    }
42.  } //end function
43.
44.  Function reverse (pathlist) {
45.    input : pathlist in leaf-to-root order
46.    output: path query in root-to-leaf order
47.    for (int i = 0; i < pathlist.size(); i++)
48.       reverse.addElement(pathlist.elementAt(pathlist.size()-i-1));
49.    return reverse
50.  }  //end function
```

query. Associated with each q in a path query is Tq, which contains occurrences, eq. Each eq in the stream is sorted by their self attributes. We use Cq to point to the current eq in Tq. Initially, Cq points to the first occurrence. Function eof(Cq) tests whether Cq is at the end of Tq. To proceed to the next occurrence, we use the advance(Cq) function. In addition, we also associate a stack Sq for each query node. Each entry in the stack consists of eq that matches to the query node.

In the twigJoin() algorithm (Algorithm 3a), it repeatedly calls the getNext() function to get the next q to process as in line 5. At lines 6-9, partial answers from the

```
Algorithm 3a : TwigJoin processing
1.  Function twigJoin(pathquery) {
2.       input : INLAB encoding streams and partitioned twig pattern
3.       output: final solutions matches to the twig pattern
4.       while (! end()) {  //if cursor not end of Tleaf
5.         qString = 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.                  formPathListStack()
13.                  pop()
14.             }
15.             advance()
16.        }
17.        else advance(qString)
18.     }
19.  }//end function
20.
```

stacks that cannot be extended to final answers are removed-in the procedure cleanParentStack() and cleanSelfADStack() -given the knowledge of the next eq to be processed. Procedure cleanParentStack() is invoked if the returned node to be processed is other than the root (a root does not have any parent). During this function call, if the parent stack is not empty, the entry at the topmost in the parent stack is compared with the current node and will be removed if it does not participate in the partial solution. However, if the returned node to be processed is the root or its parent stack is not empty, procedure cleanSelfADStack() is invoked. During this function call, if the stack of the current node is not empty, the topmost entry will be compared to the current node and will be removed if it does not participate in the partial solution. Each potential eq, which may fulfill the matching criteria, is pushed into the stack by the procedure moveToStack() for further processing. If qString is a leaf node, the solution should be output as in lines 11-12. Note that path solutions should be output in root-leaf order so that they can be easily merged together to form final path matches (line 19). Once the eq has been processed, lines 13-15 remove the eq from the stack and advance to the next eq.

In the getNext() function (Algorithm 3b), if q is a leaf query node (checked by procedure isLeaf()), the function directly returns to output the solution (line 24). In line 26, we recursively invoke the getNext() function until it is terminated by either line 24 or 27. Path query has only one child per node, thus the procedure getChild(q) returns the immediate children of node q. In line 27, if any returned node n is not equal to child of q, we immediately return n. Line 28 skips nodes that do not contribute to results, by

```
Algorithm 3b : getNext
21. function getNext(q) {
22.       input : current node in process
23.       output : node to be process
24.       if (isLeaf(q)) return q
25.       tempq = getChild(q)
26.       n = getNext(tempq) //recursive call
27.       if (n!=tempq) return n
28.       while ( ! checkAncestor(q, n) {
29.             if (getSelf(q) > getSelf(n)) return n
30.             advance (q)
31.       }
32.       if (getSelf(q) > getSelf(n)) return n
33.       return q
34.    } //end function
35.
36. function checkAncestor(q, n) {
37.       input : two nodes
38.       output : boolean true or false
39.       leveldiff = getLevel(n) – getLevel(q)
40.       current = getSelf(n)
41.       if (getSelf(n) != eof) {
42.             if (leveldiff> 0) {
43.                   while (leveldiff > 0) {
44.                         cursorUp = hashPCTable(current)
45.                         current = cursorUp
46.                         leveldiff--
47.                   }
48.                  if (current = getSelf(q)) return true
49.                 else return false
50.             }
51.             return false
52.    }
53.    return false
54. } //end function
```

checking whether the two nodes are of A-D relationship. During this process (line 44 of the checkAncestor() function), the index table, PCTable (storing P-C relationship) is hashed to retrieve each query node's parent for comparison. Procedures getSelf() and getLevel() return the self and level attributes of the current eq. Lines 32-33 return the next eq to be processed.

The mergeTwig() function: In the mergeTwig() function (Algorithm 4), all partial solutions from the twigJoin() function are merged together to generate the final solutions. This function begins by comparing each entry in the partial solutions of two path queries at a time. All the occurrences in the partial solutions are in sorted order of their self-attributes. If each entry first node is equal (line 12), or if the query edge is of P-C relationship and the second query node is of sibling and predecessor relationship, the partial solution will be added to the final solutions as in lines 13-18. For query edge with A-D relationship, if the second query node is a predecessor, it will be added as a final solution. In both cases, the inner loop begins the iteration from the current j position. Hence, this function skips the unnecessary iteration of

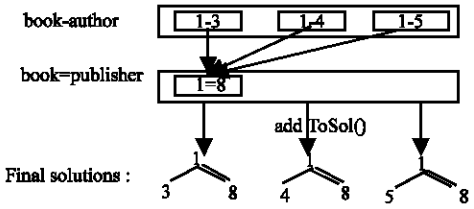

Fig. 5: The merging process scenario

non-feasible partial solutions. However, if the first node in the second path query is greater than node1, the next inner loop will begin from position j-1 (for cases where, j > 0). Figure 5 shows the merging process.

Complexity analysis of twiginlab: Here, we discuss the correctness of the TwigINLAB algorithm and analyze its complexity.

Definition 1: Let Q be the twig query. For each node q in Q, we say that the cursor, Cq initially points to the first occurrence of element q, eq_0, which is also known as head element in the stream Tq.

Definition 2: After splitting Q into several path queries, pq_i, each q (except the leaf node) in pq_i has only one child, tempq and has the same root.

Definition 3: We say that a node q has the descendant extension if the child, tempq has an element occurrence e_{tempni}, which is a descendant of eq_i.

Lemma 1: Suppose that for an arbitrary node q in the path query, we have getNext(q) = q'. Then, the following properties hold:

- q' has the descendant extension
- either (a) q = q' or (b) parent(q) does not have the descendant extension because of q'.

Proof: For each path query, solution is generated by function twigJoin() which recursively calls the getNext() function to get the next q to be processed. In the getNext() function, if q is a leaf node, it verifies all properties, so we return it in line 24. Otherwise, we get n=getNext (tempq), where tempq is the direct child of q in line 25. If we have that n ≠ tempq, n verifies properties (1) and 2(b) with respect to tempq, so we return n in line 27. Next, we check if q and n have any A-D relationship. If they are not in A-D relationship and if q self attribute is larger than n self attribute, we return n, else we advance

```
Algorithm 4: match and merge path queries into twig query
1.      Function mergeTwig (pathList, numOfLeaf) {
2.        input hashtable which contains a list of path query
3.        output: final solutions
4.         int counter = 0
5.         while (counter < (numOfLeaf-1)) {
6.      sizePath1 = pathList [counter].size()
7.      sizePath2 = pathList [counter+1].size()
8.      int start = 0
9.      for (int i = 0; i < sizePath1; i++) {
10.       node1 = pathList [counter].Node[i].first
11.       for (int j = start; j <sizePath2; j++)  {
12.         if (pathList [counter+1].Node[j].first == node1) {
13.           if (isPCRelationship()) {
14.             if((isSibling(pathList [counter].Node[i].second,  pathList [counter+1].Node[j].second)
15.              && (isPredecessor(pathList [counter].Node[i].second, pathList [counter+1].Node[j].second))
16.                start = j
17.                addToSol()
18.           }
19.          else if (isADRelationship()) {
20.            if ((isPredecessor(pathListStack[counter].Node[i].second, pathListStack[counter+1].Node[j].second)
21.             && (!isAD(pathListStack[counter].Node[i].second,  pathListStack[counter+1].Node[j].second)))
22.                start = j
23.                addToSol()
24.          }
25.          else if (pathList [counter+1].Node[j].first > node1 && j > 0) {
26.                  start = j-1
27.                  break
28.          }
29.         else if (pathList [counter+1].Node[j].first > node1  && j==0)
30.                  break
31.       }
32.    }
33.    counter++
34.    }
35.  } //end function
```

q to the next occurrence, eq_i. If q and n are of A-D relationship and the q self attribute is smaller than n self attribute, it satisfies properties (1) and 2(a), thus we return it in line 33.

Lemma 2: Suppose getNext(q) = q' returns a query node q' where, q' ≠ q in line 27 of algorithm 3b. If the stack is empty, then the head element does not participate in any partial solution.

Proof: If node q' is other than root, procedure cleanParentStack() will be invoked as in line 7 algorithm 3a. However, this function does nothing because the parent stack is empty. Thus, it directly proceeds to line 17 to advance to the next eq. Hence, the head element does not participate in any partial solution.

Lemma 3: All node occurrences, eq_i are processed based on the order of their self attributes. So, the cursor, Cq always moves forward. Consequently, each eq_i is accessed once only.

Proof: In Algorithm 3b, getNext(q) always returns the node with the smallest self attribute as shown in line 29 (if the two nodes are not in A-D relationship, but the node n self attribute is less than node q) and line 32 (if the two nodes are in A-D relationship but the node n self attribute is less than node q self attribute).

Lemma 4: Suppose that, after the partitionTwig() function, there are m number of path queries, pq_1, pq_2, …, pq_m. Function mergeTwig() merges two paths at a time to form the final solution and skip the 'non-potential' path.

Proof: Algorithm mergeTwig() check for potential merge-able path recursively two paths at a time as in line 5. Each partial solution of the first path is compared to each partial solution in the next path. If the first node in each path is the same (based on definition 2), the paths are merge-able and added to the solution as shown in lines 13-24, else the function skips the unnecessary iteration of non-feasible partial solutions as shown in lines 25-30.

Theorem 1: Given a twig query, Q and an XML document, D, algorithm TwigINLAB correctly returns all answers for Q on D.

Proof: Based on Lemma 2, we know that when getNext() returns a query node q' in line 27 of getNext(), if the stack Sparent(q') is empty, then the head element, eq'_0 does not participate in any partial solution. Thus, any element in the ancestor of q' that use eq' in the descendant extension is returned by getNext() before eq'. By using Lemma 1 and 2, for each node q in the query, the element involved in the root-to-leaf path solution is in Stack Sq. If getNext(q) is a leaf node, we output all path solutions that use eq. Finally, using Lemma 4, the path solutions are merged into a final solution.

Theorem 2: The complexity of our TwigINLAB algorithm is O (|Q| |I|), where, |I| and |Q| is the size of the XML stream and query, respectively.

EXPERIMENTAL EVALUATION

Experimental setup: We have implemented INLAB using Java API for XML Processing (JAXP). We run the experiment on all types of queries as:

Q(A) : Path query with single type relationship
Q(B) : Path query with mixed types relationship
Q(C) : Twig query pattern with single type relationship
Q(D) : Twig query pattern with mixed types relationship

Our experimental tests are divided into six main test cases as:

T1 : Measuring the execution time of Q(A), Q(B), Q(C) and Q(D) on TwigINLAB
T2 : Comparing the execution time of Q(C) on TwigINLAB and TwigStack
T3 : Comparing the execution time of Q(D) on TwigINLAB and TwigStack
T4 : Measuring the execution time of Q(A) with increasing path length on TwigINLAB and TwigStack
T5 : Measuring the execution time of Q(C) with increasing number of branches on TwigINLAB and TwigStack

All our experiments were performed on 1.7 GHz Pentium IV processor with 512 MB SDRAM on Windows XP. In all test cases, we use the Treebank dataset (84 MB) obtained from the University of Washington XML repository (Anonymous, 2002). The treebank dataset is a skew structured tree. Table 1 shows the experimental setup environment.

Table 1: Experimental setup environment

Test case	Query	Path expression
T1	Q1	S/NP/NN
	Q2	S/NP//NN
	Q3	PP[/IN]/NP
	Q4	PP[/IN]//NP
T2	Q1	PP[/IN]/NP
	Q2	S[/NP/_None_]/VP/VB
	Q3	S[//JJ]//NP
	Q4	S[//JJ]//IN
T3	Q1	PP//NP[/DT]/NN
	Q2	PP[//VP/VBN]/NP
	Q3	S[/NP]/VP[//PP//JJ]/VBZ
	Q4	S[[/NP]/MD]/VP//JJ
T4	Q1	FILE/EMPTY/S
	Q2	FILE/EMPTY/S/VP
	Q3	FILE/EMPTY/S/VP/NP
	Q4	FILE/EMPTY/S/VP/NP/NN
T5	Q1	S/NP
	Q2	S[/NP]/MD
	Q3	S[[/NP]/MD]/VP
	Q4	S[[[/NP]/MD]/VP]/PP

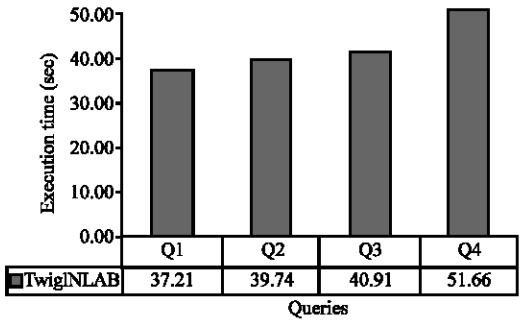

Fig. 6: Results for T1

Performance results: Figure 6 shows the execution time of all types of queries on TwigINLAB. From the results obtained, queries with single type of relationship perform better as compared with queries with mixed types of relationship. Besides, queries with edges of A-D relationship are slower due to the extra time needed to determine whether the two nodes is in A-D relationship by multiple lookups on the index table until the ancestor level is reached.

Figure 7 and 8 show the execution time of several queries with single type and mixed types of relationship over TwigINLAB and TwigStack, respectively. From Fig. 7 and 8, we draw several observations and conclusions:-

- When the twig query contains only a single type of relationship, TwigINLAB performs about 15% on average better as compared to TwigStack.
- TwigINLAB performs about 3% on average better as compared to TwigStack on mixed types of relationship.

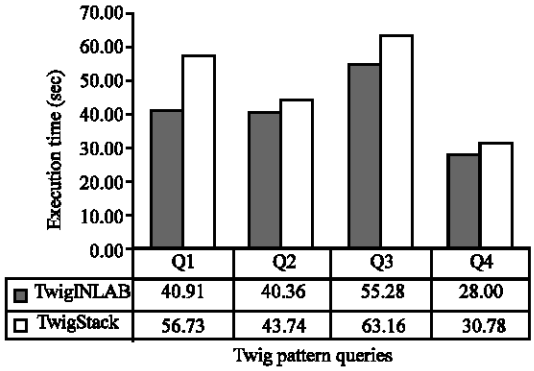

Fig. 7: Results for T2

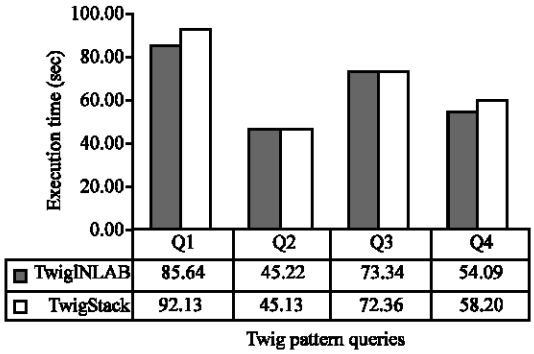

Fig. 8: Results for T3

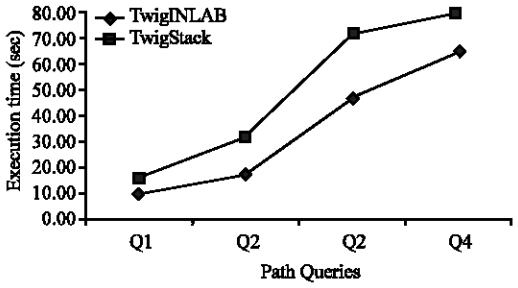

Fig. 9: Results for T4

- When the twig query contains only P-C edges, TwigINLAB performs around 18% better as compared to TwigStack (shown in Q1 and Q2 in Fig. 7). This may be due to the createINLAB labeling scheme which is optimal to support P-C relationships.
- When edges below branching nodes contain only P-C relationships, TwigINLAB performs better about 7% as compared to TwigStack (shown in Q1 and Q4 in Fig. 8).

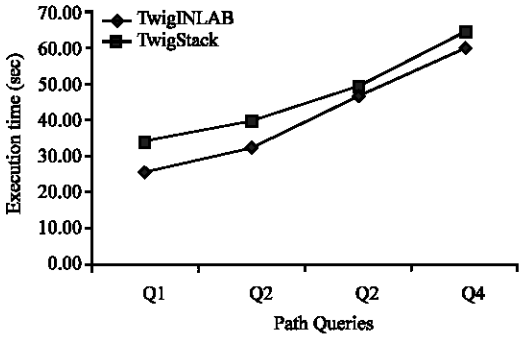

Fig. 10: Results for T5

- When edges below branching nodes contain mixed relationships (as shown in Q2 and Q3 Fig. 8), the performance of TwigINLAB is comparable to TwigStack.

In Fig. 9, by using TwigINLAB processing, the execution time increases linearly (only slightly) with the increment of path length. This result shows that TwigINLAB is more scalable as compared to TwigStack in terms of execution time. As such, TwigINLAB can support query on large-scale datasets efficiently. In Fig. 10, by using TwigINLAB processing, the execution time increases slightly with the increment of number of branches. Thus, TwigINLAB performs better in terms of scalability as compared to TwigStack.

CONCLUSIONS

In this study, we have presented a hybrid query optimization technique, INLAB, comprising the createINLAB to create INLAB encoding and the TwigINLAB algorithm to optimize the twig query pattern matching process. Most of the labeling schemes are affected by the fan-out of the tree. Compared to existing labeling schemes, the size for createINLAB labeling is fixed-length labels resulting in better storage utilization. In addition, using the createINLAB labeling scheme, structural relationships can be determined easily. Moreover, the proposed labeling scheme is integer based. Integer processing is very efficient compared to that of string or bit-vector. Using the createINLAB indexing, elements that belong to the same parent or ancestor can be determined easily by multiple lookups to the index table. Another advantage to createINLAB indexing is that the index size grows linearly with the XML tree.

Experimental results show that, in terms of execution time, on average, TwigINLAB performs about 15% better compared to TwigStack on a skew structured tree such as

the Treebank dataset. Besides, TwigINLAB outperforms TwigStack if (i) query has a single type of relationship only and (ii) the edges below branching nodes contain only P-C relationships. For edges with A-D relationships below the branching nodes, the performance of TwigINLAB is comparable to TwigStack. In addition, TwigINLAB is more scalable compared to TwigStack in terms of execution time. As such, TwigINLAB supports large-scale query of datasets efficiently.

The study can be further extended to compare the performance of TwigINLAB and TwigStack using a flat-structured tree. Hypothetically, the performance of TwigINLAB is expected to be better as it requires less number of lookups on the index table as compared to a skew-structured tree.

REFERENCES

Abiteboul, S., D. Quass, J. McHugh, J. Widom and J. Wiener, 1997. The lorel query language for semistructed data. J. Digital Libraries, 1: 68-88.

Abraham, J., N.S. Chaudhari and E.C. Prakash, 2004. XML query algebra operators and strategies for their implementation. Proc. TENCON, IEEE, pp: 286-289.

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.

Amer-Yahia, S., S. Cho, L.V.S. Lakshmanan and D. Srivastava, 2002. Tree pattern query minimization. VLDB J., 11: 315-331.

Anonymous, 2002. http://www.cs.washington.edu/research/xmldatasets/.

Anonymous, 2004. XPath, XML path language. http://www.w3.org/TR/xpath.

Anonymous, 2005. XML Query (XQuery). http://www.w3.org/XML/XQuery.

Brantner, M., S. Helmer, C.C. Kanne and G. Moerkotte, 2005. Full-fledged algebraic XPath processing in natix. Proc. ICDE, pp: 705-716.

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.

Christophides, V., S. Cluet and J. Simeon, 2000. On wrapping query languages and efficient XML integration. Proc. ACM. SIGMOD., pp: 141-152.

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 and 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. Proc. VLDB, pp: 341-350.

Dietz, P.F., 1982. Maintaining order in a linked list. Proceeding of ACM Symposium on Theory of Computing, pp: 122-127.

El-Sayed, M., K. Dimitrova and E.A. Rundensteiner, 2003. Efficiently supporting order in XML query processing. Proc. ACM SIGMOD, pp: 147-154.

Frasincar, F., G. Houben and C. Pau, 2002. XAL: An Algebra for XML Query Optimization. Proceeding of 13th Australasian Database Conference, pp: 49-56.

Galanis, L., E. Viglas, D.J. DeWitt, J.F. Naughton and D. Maier, 2002. Following the paths of XML Data: An algebraic framework for XML query evaluation. Technical Report, University of Wisconsin.

Goldman, R. and J. Widom, 1997. Data guides: Enabling query formulation and optimization in semistructured Databases. Proc. VLDB, pp: 436-445.

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. Digit. Inform. Manage., 2: 82-87.

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

Jagadish, H., L. Lakshmanan, D. Srivastava and K. Thompson, 2001. Tax: A tree algebra for XML. Proceeding of Database Programming Language, pp: 149-164.

Jiang, H., W. Wang, H. Lu and J.X. Yu, 2003. Holistic twig joins on indexed XML documents. Proc. VLDB, pp: 273-284.

Jiang, H., H. Lu and W. Wang, 2004. Efficient processing of twig queries with OR-predicates. In Proc. of ACM SIGMOD, pp: 59-70.

Jiao, E., T.W. Ling, C.Y. Chan and S. Yu, 2005. PathStack]: A holistic path join algorithm for path query with not-predicates on XML data. Proc. DASFAA, pp: 113-124.

Josifovski, V., M. Fontoura and A. Barta, 2005. Querying XML Streams. VLDB J., 14: 197-210.

Kanne, C.C., M. Brantner and G. Moerkotte, 2005. Cost sensitive reordering of navigational primitives. Proceeding ACM SIGMOD, pp: 742-753.

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

Kaushik, R., P. Bohannon, J.F. Naughton and H.F. Korth, 2002b. Covering indexes for branching path queries. Proc. ACM SIGMOD, pp: 133-144.

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

Kimber, W.E., 1993. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation.

Li, Q. and B. Moon, 2001. Indexing and Querying XML Data for Regular Path Expressions. Proc. VLDB, pp: 361-370.

Lu, J., T. Chen and T.W. Ling, 2004. Efficient processing of XML Twig Patterns with Parent Child Edges: A look-ahead approach. In Proc. CIKM, pp: 533-542.

McHugh, J. and J. Widom, 1999. Query Optimization for XML. Proc. VLDB, pp: 315-326.

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.

Polyzotis, N., M. Garofalakis and Y. Ioannidis, 2004. Approximate XML query answers. Proc. SIGMOD, pp: 263-274.

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. In: Proc. ACM SIGMOD, pp: 204-215.

Viglas, S.D., L. Galanis, D.J. DeWitt, D. Maier and J.F. Naughton, 2002. Putting XML query algebras into context. Technical Report, University of Wisconsin.

Weigel, F., K.U. Schulz and H. Meuss, 2005. The BIRD numbering scheme for XML and tree databases-deciding and reconstructing tree relations using efficient arithmetic operations. Lecture Notes Computer Science, 3671: 49-67.

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

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

Yao, J.T. and M. Zhang, 2004. A Fast Tree Pattern Matching Algorithm for XML Query. In: Proc. IEEE/WIC/ACM, pp: 235-241.

Yu, T., T.W. Ling and J. Lu, 2006. TwigStackList ¬: A Holistic twig join algorithm for twig query with not-predicates on XML data. In: Proc. DASFAA, pp: 249-263.

Zezula, P., F. Mandreoli and R. Martoglia, 2004. Tree signatures and unordered XML pattern matching. Proc. SOFSEM, pp: 122-139.

Zhang, D. and Y. Dong, 1999. A Data Model and Algebra for the Web. Proceeding of Workshop on Database and Expert System Application, IEEE, pp: 711-714.

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, N., 2006. Query processing and optimization in native XML databases. Technical Report, University of Waterloo.

Zheng, S., A. Zhou, J.X. Yu, L. Zhang and H. Tao, 2002. Structural map: A new index for efficient XML path expression processing. Proc. WAIM, pp: 25-36.