Stack-based Pattern Matching Algorithm for XML Query Processing

Su-Cheng Haw, Chien-Sing Lee
Faculty of Information Technology
Multimedia University
63100 Cyberjaya, Malaysia
{schaw, cslee}@mmu.edu.my

ABSTRACT: With the popularity of XML as data exchange over the Web, querying XML data has become an important issue to be addressed. Since the logical structure of XML is a tree, establishing a parent-child (P-C), ancestor-descendant (A-D) or sibling relationship between nodes is essential for structural query processing. Thus, we propose using a <self–level: parent> labeling scheme to encode each element in the XML database by its positional information. Based on this labeling scheme, we further propose our TwigINLAB algorithm to optimize the query processing. Experimental results indicate that TwigINLAB can process both path queries and twig queries better than the TwigStack algorithm on an average of 27% and 14% respectively in terms of execution time using the XMARK benchmark dataset.

Categories and Subject Descriptors
D. 3.2 [Language Classification]; Exensible languages: H.2 [Database management] H.2.4 [Systems]: Query processing

General Terms
XML, Query processing, XML database, Pattern matching algorithm

Keywords: XML, pattern matching, labeling, query processing, structural query, twig query

Received 24 December 2006; Revised 21 March 2007; Accepted 27 April 2007

1. Introduction
XML semi-structured documents provide a flexible and natural way to store data within content. Correspondingly, there are two types of user queries, namely full-text queries (keyword-based search) and structural queries (complex queries specified in tree-like structure) [1]. To cope with tree-like structures in XML, several XML-specific query languages such as XPath [2] and XQuery [3] have been proposed to provide flexible query mechanisms [4].

This paper is concerned with structural queries. There are two types of structural 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. In other words, path query consists only one leaf node whilst twig query have two or more leaf nodes. 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 parent-child (P-C) or ancestor-descendant (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 notation [2], 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:__). Figures 1(a) and 1(b) show the example of path query and twig query respectively. The path query evaluates to “find all the titles of books under publications” while the twig query evaluates to “find all the book elements that have child named title and figure”.

The main focus of this paper is on twig query processing. The flow for path query processing algorithm has been reported in [5].

Consider the following sample query (as shown in Figure 1(b)):

Q1: /book/[title]/figure

To find matches for Q1, we need to “track existence of all element nodes with the path /book/title and path /book/figure”. Using the conventional top-down navigational approach [6] to process Q1, all downward paths starting from any book element should be traversed to find out whether there exists any immediate title element. Next, it traverses down all other element nodes one by one until it reaches the figure element. For the next set of matches, it needs to backtrack to its previous visited book element node and start the search again. Thus, this is certainly very exhaustive and inefficient.

Processing such queries may benefit from using the decomposition-matching-merging approach [4-8]. TWIG-XSKETCH [7], tree signature [8], MPMGJN [9], Stack-Tree [10] and TwigStack [11] are examples of query processing using the decomposition-matching-merging approaches. Both MPMGJN and Stack-Tree algorithms accept two lists of sorted
Our contribution merging phase. This resulted in higher processing time and mixed relationships (consists both A-D and P-C intermediate results as long as the nodes have edges with the matching process. In contrast, Stack-Tree algorithm is more efficient as it uses stack to maintain the ancestor or parent nodes and therefore requires only a one-time scan per input list. However, these approaches still produce large intermediate results. To address this problem, Bruno et al. propose TwigStack [11], 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. In addition, most of these approaches focus on the matching phase only and still suffer in producing large intermediate results before the merging phase.

The work presented in this paper is motivated by the following observation: although TwigStack [11] is optimal to support queries with A-D relationship, their algorithms still produce large intermediate results for queries with P-C relationships. The main problem of TwigStack is in the matching phase where it pushes all nodes into a chain of linked stacks as intermediate results as long as the nodes have edges with A-D relationships. Thus, this algorithm produces large ‘useless’ intermediate results especially for queries with P-C and mixed relationships (consists both A-D and P-C relationships). This resulted in higher processing time required to check for possible merge-able paths in the merging phase.

Our contribution. Our TwigINLAB algorithm is a generalization of the stack-based algorithm of [11]. The main difference is we decompose the twig query into a set of path queries. In addition, we focus on optimizing all three decomposition-matching-merging sub-processes. We introduce a novel robust and compact labeling scheme consisting of \(<\text{self} – \text{level} : \text{parent}>\) to allow quick determination of the types of relationships among each path edge, subsequently optimizing the matching phase based on each relationship and indices (built only once) that restrict the searching scope and finally reducing the number of inspections required in the merging phase.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents the createINLAB encoding algorithm. Section 4 gives an overview flow of theTwigINLAB algorithm. Section 5 presents the experimental setup, findings and performance results. Lastly, Section 6 concludes the paper and suggests future work.

2. Related Work
2.1 Labeling Scheme
Several labeling schemes have been proposed to facilitate faster query processing. They can be categorized into range-labeling scheme and prefix-labeling scheme. In range-labeling scheme, the label of a node is interpreted as a pair of numbers (start position, end position). When a new node is inserted, the label usually needs to be regenerated. Thus, the range-labeling scheme is also known as non-persistent labeling scheme. However, in the prefix-labeling scheme, the label of a node is single number. Under heavy update, prefix-labeling scheme may not need to be recomputed. It is therefore also known as persistent labeling scheme [12]. Some of these methods are discussed below. Nevertheless, further elaboration can be found in [1].

Most researchers [9, 11, 13] use the range labeling of \((\text{begin} : \text{end}, \text{level})\) for an element as the positional representation of XML elements and texts. A node \(\text{node} \_\_\text{node}, \text{begin}\_\text{node}, \text{end}\_\text{node}, \text{level})\) is an ancestor of \(\text{node} \_\_\text{node}, \text{begin}\_\text{node}, \text{end}\_\text{node}, \text{level})\) iff \(\text{node} \_\_\text{node}, \text{begin} < \text{node} \_\_\text{node}, \text{end} < \text{node} \_\_\text{node}, \text{end}\). Any two nodes is in P-C relationship, iff the level difference between the two nodes is 1. This labeling scheme is unable to determine the sibling relationship efficiently. For instance, to determine whether any two nodes are of sibling relationship, it needs to search the parent of a node, and then decide whether another node is a child of this parent.

Some other labeling schemes are tree location address [14], simple prefix [15], GRP [12], prime number labeling [16], ORDPATH [17], BIRD [18] and work done by Gabillon et al. [19]. 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. In GRP, each node contains the label with groupID and a group prefix label, where groupID is an integer and group prefix label (similar to simple prefix) is a binary string. In prime number labeling, 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). The concept for ORDPATH is similar to Dewey Order [20], 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. The BIRD labeling scheme is based on a structural summary similar to DataGuide [21]. 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. Gabillon et al. propose using real numbers between each interval number. Although this accommodates more number of possible updates, by naturally leaving gaps between successive values, however, if we use log n bits to represent the floating point number, then we can only handle at most n updates, in the worst case, before running out of space. As summary, although some labeling schemes [17-19] 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 the tree’s structure is skewed.

In our labeling scheme \(<\text{self} – \text{level} : \text{parent}>\), the size of the labelled node is only 12 bytes. Besides, our labeling is integer-based. Integer processing is very efficient compared to that of string or bit-vector. The details on this will be explained in Section 3.

2.2 Query Processing Using Decomposition-Matching-Merging Approach
In the first sub-process, there are typically two types of decomposition methods [7-11, 22]. First, a complex query pattern can be decomposed into a set of basic binary relationships between each pair of nodes. Second, it can be decomposed into a set of path queries. As for the second sub-process, MPMGJN [9], Stack-Tree [10] and TwigStack [11] algorithms are based on \((\text{docno}, \text{begin}: \text{end}, \text{level})\) labeling of XML elements. 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. Polyzotis et al. decompose the twig query into a set of path queries and 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 [7]. They propose both TREESKETCH and TWIG-XSKETCH. On the
other hand, Zezula et al. propose a novel technique, tree signature, to represent tree structures as ordered sequences of pre-order and post-order ranks of the nodes [8]. They use tree signatures as index structure and find qualifying patterns through integration of structurally consistent path query. Some other approaches performed using indices to aid in speeding up the query processing include XR-tree [23], Prix [24] and ViST [25].

Merging together the structural matches in the final process poses the problem of selecting a good join ordering. Wu et al. propose a cost-based join order selection of structural join [26]. Kim et al. suggest partitioning all nodes in an extent into several clusters [27]. Given two extents to be joined, they propose filtering out unnecessary clusters in both extents prior to the joining process.

As mentioned earlier, our TwigINLAB focuses on optimizing all three sub-processes; decompose the twig query into a set of path queries, using \(<self–level: parent>\) labeling scheme and index table as look-up to optimize the matching phase and finally, reducing the number of inspections required in the merging phase. Further elaboration can be found in section 4.

3. Encoding XML Data with createINLAB Algorithm

We pre-process the XML tree into a set of streams labeled with \(<self–level: parent>\) for each element occurrence. Thus, instead of checking for matches against the whole XML tree, the “qualified” streams are presented as input. In this section, we will present the createINLAB encoding algorithm (Algorithm 1), which takes a regular XML document and generates a set of encoded XML streams; and PCTable, the index table storing each element’s parent information.

Some basic operations involved in this algorithm (and in Algorithm 2 and 3) 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, elementAt(index) to retrieve

---

Algorithm 1: createINLAB encoding

1. function createINLAB  {
2.     input : an XML file X
3.     output : encoded XML assigned tag
4.     /*A stack eleStack to keep track of element sequence.
5.     A vector vExtent to store the occurrence of each element in stream
6.     A hashtable eleTable to store each distinct element in X.
7.     A hashtable PCTable to keep track of each element parent’s information
8.     A record with <self-level : parent> */
9.     int ptr = 0, level = 0, self = 0, parent = -1
10.    curRec = null
11.    while (! eof (X)) do {
12. if SAX event = a start tag <T> then {
13. if (tag has yet been stored into eleTable) {
14.                        create new instance of vector, vExtent
15.                        eleTable.put( tag, ptr++)
16.                        }
17.                        create new instance of record, curRec
18.                        curRec.self = self++
19.                        curRec.level = level++
20.                        if (eleStack.size( ) > 0)
21.                        curRec.parent = eleStack.elementAt(eleStack.size( )-1).self
22.                        else
23.                        curRec.parent = -1
24.                        int i = eleTable.get(tag).intValue
25.                        vExtent[i].addElement(curRec)
26.                        eleStack.push(curRec)
27.                        }
28. if SAX event = an end tag </T> then {
29.                        eleStack.pop( )
30.                        curRec = eleStack.peek( )
31.                        level —
32.                        }
33.                        }
34. } //end function
35. }
36. function output  {
37.     input : tag in XML file X and cursor position in data stream
38.     output : encoded XML data streams(files)
39.     create file fileData = (“myData\"+tag, with read and write mode)
40.     int self, level, parent
41.     while (as long as cursor NOT end of stream) {
42.     self = cursor.getCurSelf
43.     level = cursor.getCurLevel
44.     parent = cursor.getCurParent
45.     writeInt(fileData, self, level, parent)
46.     PCTable.put(self, parent)
47.     }
48. } //end function

---
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.

For each SAX event, if the start tag is found (lines 12-27), createINLAB function pushes the tag into eleTable if the tag has yet been stored into eleTable (lines 13-16). At the same time, an instance vector vExtent is created. Next, the label for each attribute such as self, level and parent is generated as the current record, curRec in lines 18-23. curRec is then inserted into the vExtent and pushed into eleStack (to keep track on each element sequence) as shown in lines 25-26. However, if the end tag is encounter (lines 28-32), an entry is removed from the eleStack. Thus, the entry at the topmost of eleStack is now the curRec. For each end tag, the level counter also decreases by one.

Function output() (lines 36-48) will generate the label based on the occurrences in vExtent as a set of streams group by the tag name and an index table, PCTable. Figure 2(a) shows a sample XML document and its corresponding fragment of XML streams stored in the eleTable generated during the createINLAB encoding algorithm. Likewise, figure 2(b) depicts the fragment of PCTable generated.

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

1. **P-C relationship**
   - node\(_i\) is the parent of node\(_j\) if and only if node\(_i\).self = node\(_j\).parent.

2. **Sibling and ordered relationship (predecessor and successor)**
   - node\(_i\) is the sibling of node\(_j\) if and only if node\(_i\).parent = node\(_j\).parent.
   - a. node\(_i\) is the predecessor node of node\(_j\) if and only if node\(_i\).self < node\(_j\).self.
   - b. node\(_i\) is the successor node of node\(_j\) if and only if node\(_i\).self > node\(_j\).self.

3. **A-D relationship**
   - node\(_i\) is possible as an ancestor of node\(_j\) if and only if level different, leveldiff = node\(_j\).level - node\(_i\).level \(\geq 1\). A multiple look-up via PCTable (shown in Figure 2(b)) 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\(_i\) and title <2-2:1> be node\(_j\). The leveldiff between the two nodes is 2. To determine whether these two nodes is of P-C relationship, we need to hash PCTable (as illustrated in Figure 2(b)) twice (two level up). The retrieved node parent attribute is 0 and it is equal to the self attribute of publications, which is 0 also. Thus, publications and title is of A-D relationship.
4. TwigINLAB Processing

Figure 3 illustrates the TwigINLAB processes, which consist of the analysisQueryPattern(), partitionTwig(), twigJoin(), mergeTwig() and outputSolution() functions. Below is a brief explanation on the flow of these functions.

Initially, the query pattern is analyzed using the analysisQueryPattern() function. For each query edge, if the twig is of P-C relationship, the parent and child details will be updated in the twigPC (a hashtable to store parent and child) repository as depicted in Figure 4. During this process, each node in the twig query is associated with a stream. Each stream contains the positional representations of the node appearance in the XML tree. The nodes in the stream are sorted by their self attribute, and thus, this will determine the order of the node to be processed. Associated also with each node is a stack. Stack is used to store the possible intermediate results.

For each path query, it recursively calls the twigJoin() function (depicted in Algorithm 2). Function twigJoin() is the main algorithm of TwigINLAB. It recursively calls getNext() function to get the next node, qString to be processed. At lines 6–9, partial answers from the stacks that cannot be extended to final answers are removed - in the procedure cleanParentStack() and cleanSelfADStack() - given the knowledge of the next qString to be processed. Each potential qString, 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 12). Once the node has been processed, lines 13–15 remove the node from the stack and advance to the next node.

In the getNext() function, 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 getFirstChild() 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. Lines 28–31 skip nodes that do not contribute to the results, if the two nodes is not in A-D relationship. Lines 32–33 are the important step to improve the query processing for twig query with P-C relationship. TwigStack [11] pushes all nodes into a chain of linked stacks as long as the nodes fulfill the A-D criteria. However, if a twig query is in P-C relationship only, this is a certainly insufficient criterion to filter out unnecessary nodes before the merging phase. Thus, more time is needed to merge the partial solutions, which do not contribute to the final solutions. Our algorithm blocks the unnecessary nodes to be pushed into the stack. Lastly, lines 35–36 return the next node to be processed.

Next, these matches are merged back through the mergeTwig() function (depicted in Algorithm 3). In the mergeTwig() function, 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, 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. 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 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 illustrates the merging process.

Finally, the final solutions are output through the outputSolution() function.
5.2 Evaluation of Performance

Figures 6, 7 and 8 show the execution time of TwigINLAB and TwigStack for both path and twig query. Figure 6 shows the execution time of: Q1PQ=text/keyword for path query and Q1TQ=text[\text/keyword]/bold for twig query over Standard dataset by varying the file sizes. From the result, TwigINLAB outperforms TwigStack in all the test cases by about 28% for path query and 19% for twig query.

---

Algorithm 2: 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.            } else advance(qString)
15.         }
16.     }
17. } //end function
18. 
19. function getNext(q) {
20.     input: current node in process
21.     output: node to be process
22.     if (isLeaf(q)) return q
23.     tempq = getChild(q)
24.     n = getNext(tempq) //recursive call
25.     if (n != tempq) return n
26.     while (! checkAncestor(q, n) {
27.         if (getSelf(q) > getSelf(n)) return n
28.         advance(q)
29.     }
30.     if (getPCRelation(q, n) { //hash twigPC table
31.         if (getSelf(q) != getParent(n)) advance(q)
32.     }
33.     if (getSelf(q) > getSelf(n)) return n
34.     return q
35. } //end function
36. 
37. function checkAncestor(q, n) {
38.     input: two nodes
39.     output: boolean true or false
40.     leveldiff = getLevel(n) – getLevel(q)
41.     current = getSelf(n)
42.     if (getSelf(n) != eof) {
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. } //end function
Figure 7 shows the execution time of: $Q2PQ = \text{mailbox}[/\text{date}]$ for path query and $Q2TQ= \text{mailbox}[//\text{date}]/\text{emph}$ for twig query respectively. TwigINLAB performs by about 26% better than TwigStack for path query and 16% for twig query.

Figure 8 shows the execution time of: $Q3PQ = \text{item/} \text{description/} \text{keyword}$ for path query and $Q3TQ = \text{item/} \text{description}[//\text{keyword}]/\text{bold}$ for twig query respectively. TwigINLAB performs about 26% better than TwigStack for path query and 6% for twig query.

From these figures, we draw several observations and conclusions:

- When the twig query contains only P-C edges, TwigINLAB performs around 23.5% better as compared to TwigStack (shown in Figure 6). This may be due to the INLAB labeling scheme, which is optimal to support P-C relationships.
- Although TwigINLAB still outperforms TwigStack for query with edges of A-D relationship by around 21%, the difference is less as compared to query with edges of P-C relationships. This may be 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.
- For each test case, TwigINLAB increases less drastically as compared to TwigStack. This shows that TwigINLAB is more scalable in processing large-scale datasets efficiently.

6. Conclusion

In this paper, we have presented the TwigINLAB algorithm to optimize all the sub-processes involved in the decomposition-matching-merging approaches. Experimental results show that, in terms of execution time, on average, TwigINLAB performs about 27% better for path query and about 14% better for twig query compared to the TwigStack. Also, TwigINLAB is more scalable compared to TwigStack.

The study can be further extended to compare the performance of TwigINLAB and TwigStack using larger size datasets. Hypothetically, the performance of TwigINLAB is expected to be better as it is more scalable. Besides, we will conduct experiments to test some of the basic functions of an XML database such as create, retrieve, update and delete.

References


Su-Cheng Haw received her BSc degree in 1999, and MSc (IT) degree in 2001 from University Putra Malaysia, Malaysia. She was an Analyst Programmer cum Team Leader in Infotech MSC Sdn Bhd and is currently a lecturer in Multimedia University, Malaysia. She is currently pursuing her Ph.D (Information Technology) in Multimedia University, Malaysia. Her research interests include XML database, query optimization, database tuning, data warehousing, Entity-Relationship Approach and web programming.

Dr. Chien-Sing Lee’s research interests are in data mining, agents, knowledge representation and management, ontology mapping and merging, e-commerce and e-learning.