JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 25, 603-617 (2009)

Short Paper

TwigX-Guide: An Efficient Twig Pattern Matching System Extending DataGuide Indexing and Region Encoding Labeling

SU-CHENG HAW AND CHIEN-SING LEE
Department of Electrical and Computer Engineering
Faculty of Information Technology
Multimedia University
63100 Cyberjaya, Malaysia
E-mail: {schaw; cslee}@mmu.edu.my

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. In this paper, 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 twig queries on an average 38% better than the TwigStack algorithm, 30% better than TwigINLAB, 10% better than TwigStackList and about 5% better than TwigStackXB in terms of execution time.

Keywords: XML query, indexing, labeling, DataGuide, region encoding, query optimization

1. INTRODUCTION

XML documents consist of nested elements enclosed by user-defined tags, which indicate the meaning of the content contained. Thus, to query XML typically means querying over data that conforms to a labeled tree data model. There are two types of user queries, namely full-text queries (keyword-based search) and structural queries (complex queries specified in tree-like structure). Much work has been done by W3C on full-text search and has been published in the working draft [1]. Hence, the main concern in this paper is on structural queries. There are two types of structural queries, namely path query (query on one single element at a time) and twig query (query on two or more elements). In other words, path query consists of only one leaf node while twig query involves two or more leaf nodes. Query edges for both queries are either parent-child (P-C) -denoted by "/" or ancestor-descendant (A-D) relationships-denoted by "//" [2]. There are two main approaches to processing such queries, namely: (1) Traversing the XML

Received May 31, 2007; revised December 3; accepted April 24, 2008.
Communicated by Jonathan Lee.

database sequentially to find the matching pattern and (2) Query processing using the decomposition-matching-merging approach.

Using the first approach, if the query contains edges with A-D relationships, multiple forward and backward traversals are needed to find the matches. Thus, this is certainly very exhaustive and inefficient. As a result, many researchers have complemented it with indices [3-7] to speed up the query processing. Among one of the well-known indexing methods is DataGuide [3].

Processing such queries may benefit from using the decomposition-matching-merging approach (the second approach) [8-12]. Firstly, a complex query can be decomposed into a set of basic binary relationship between pairs of nodes, which are then matched against the XML document. Next, these matches are merged together to form the path solution. However, these approaches still suffer in producing large intermediate results because the join results of individual binary relationships may not actually participate in the final results. To reduce the number of intermediate results, Bruno *et al.* proposed TwigStack [12], which processes the twig query holistically without decompositing it into binary relationships.

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

1. Although TwigStack [12] 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, 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.
2. Although DataGuide [3] 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:

1. We propose the TwigX-Guide system architecture, which extends the existing DataGuide and TwigStack to accelerate twig query processing.
2. We propose three new algorithms: (1) The *createDGnLAB* algorithm to create path indexing and region encoding of an XML document; (2) The *CutMatchMergePath* algorithm to process path query and (3) The *CutMatchMergeTwig* algorithm to process twig query.
3. We analyze and observe that in a typical XML document, the number of distinct path labels is far less than the number of possible path labels. For example:
   - In the Standard dataset [13], there are about 9.24×10^6 nodes, but the number of distinct paths is only 66.
   - In the Mondial dataset [14], there are about 80,560 nodes, but only 60 distinct paths.
4. We implement the TwigX-Guide system and experimentally show that TwigX- Guide outperforms TwigStack [12] by about 38%, TwigINLAB [15] by about 30%, TwigStackList [16] by about 10% and TwigStackXB [12] by about 5%.

The rest of the paper is structured as follows. Section 2 presents the background of section 2.1: region encoding and TwigStack and section 2.2: path summary and DataGuide. Section 3 is the core of the paper and gives an overview of our TwigX-Guide system architecture and all its components. Moreover, we propose three main algorithms in this section. Section 4 presents the experimental setup and performance results. Section 5 reviews related work. Lastly, section 6 concludes the paper.

2. BACKGROUND

2.1 Region Encoding and TwigStack Join Algorithm

Fig. 1 shows the XML tree encoded with <*start*, *end*, *level*> label. Using this labeling scheme, structural relationships between nodes can be determined as follows: *node*1 is the ancestor of *node*2 iff *node*1.*start* < *node*2.*start* and *node*1.*end* > *node*2.*end*. However, *node*1 is the parent of *node*2 iff *node*1.*start* < *node*2.*start* and *node*1.*end* > *node*2.*end* and *node*1.*level* = *node*2.*level*-1.

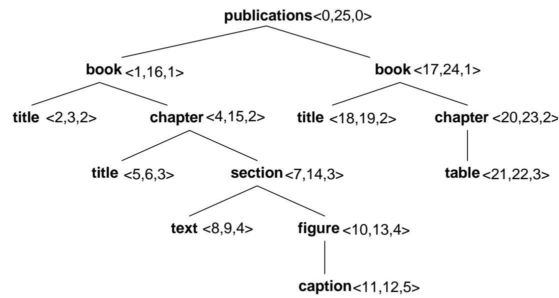


Fig. 1. XML document labeled with region encoding.

Example 1: Let Q1: *book//section//figure*. To evaluate this query, we first retrieve all the nodes reachable by *book*, *section* and *figure* tags. This results in three lists of streams. We then test each occurrence in the streams for possible A-D relationships based on the rule defined above and structurally join them via TwigStack join algorithms. Consider *book* <1, 16, 1> as $node_1$ and *section* <7, 14, 3> as $node_2$. Looking at their *start* and *end* attributes, they have an A-D relationship iff *book.start* < *section.start* < *book.end*, that is, in this case, 1 < 7 < 16 is true. Similarly this evaluation applies for the rest of the occurrences in the streams.

However, this labeling does not contain hierarchical path information. Thus, it is insufficient to determine query with P-C relationships as illustrated in the Example 2.

Example 2: Let Q2: *book*[//*title*]/*table*/*caption*/*text*. To evaluate this query, we need to retrieve all nodes reachable by the *book*, *title*, *table*, *caption* and *text* tags. If we evaluate this query using TwigStack, it will push nodes <1, 16, 1> and <17, 24, 1> into the stack and output all root-to-leaf path solutions of *book*//*title*: [1-2, 1-5, 17-18]. Notice that in

this example, there is actually no match at all because there is no *book*/*table*/*caption*/*text* path. Furthermore, this query needs a total of four joins. Since joins are very expensive in query processing, we propose to reduce the number of joins by "pre-matching" the node along the root-to-leaf path instead of matching it individually.

2.2 Path Summary and DataGuide

The DataGuide [3] provides general path indices that summarize all paths in the data tree (source) that start from the root. An important characteristic of the DataGuide (strong DataGuide) is that each label path from the source appears exactly once in the index tree. Each node in a DataGuide has an extent for the corresponding nodes in the source. Nevertheless, the path summary, does not preserve the hierarchical relationships among individual nodes. Therefore, the path summary is unable to answer twig queries.

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 the 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.

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 onregion 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-C edges efficiently.

3. TWIGX-GUIDE SYSTEM ARCHITECTURE

Fig. 2 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.

3.1 Offline Processing: Labeling and Indexing Generator

In its traditional way, each DataGuide node is in the form of (*nodeID*, *data path*). Nevertheless, we annotate each node by their region encoding label as introduced earlier in section 2.1. On the other hand, we remove the *level* attribute in the region encoding because it is no longer needed to check for P-C relationships. All P-C relationships can be determined directly from the DataGuide.

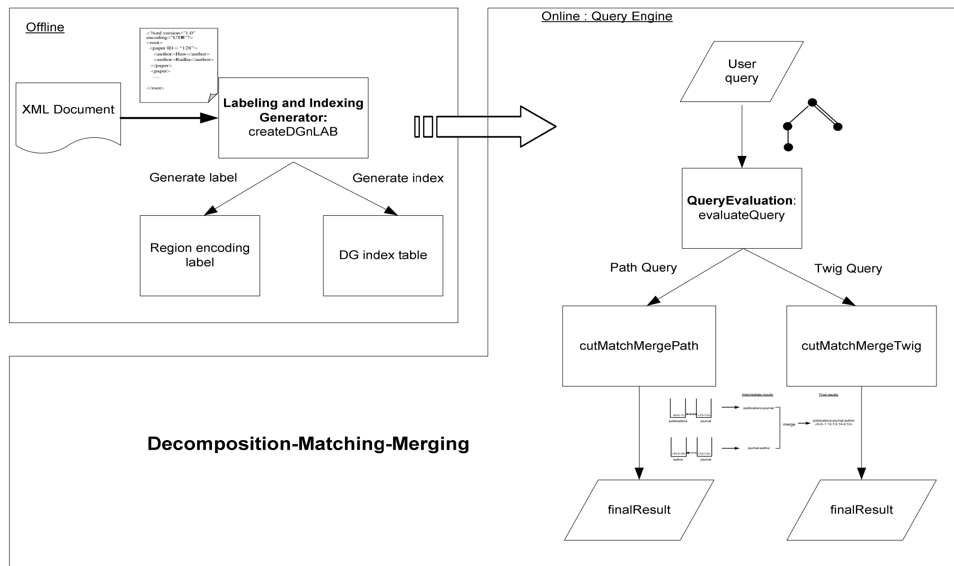


Fig. 2. TwigX-Guide system architecture.

During the offline processing, we pre-process the XML tree into a set of streams labeled with <*start* – *end*> for each node and path occurrence as depicted in Algorithm 1. Thus, instead of checking for matches against the whole XML tree, only the "qualified" streams are presented as input.

Some basic data structure operations involved in this algorithm (and all the following algorithms) 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 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 belongs 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 and *size*() to return the total entry in a vector.

For each SAX event, if the start tag is found (lines 11-22), the *createDGnLAB* function pushes the tag into the *DG* index table if the tag is not found in DG (lines 12-15). At the same time, an instance vector *vExtent* is created. Next, the label for each attribute such as *start* and *end* is generated as the current record, *curRec* in lines 16-22. *curRec* is then inserted into the *vExtent* and pushed into *eleStack* (to keep track of each element's sequence to form node label) and *pStack* (to keep track of path element sequence to form path label) as shown in lines 20-21. However, if the end tag is encountered (lines 23-42), the path label is formed in the *formPathLabel* function. If the path is not found in *DG*, it is entered as new entry. Else, it will be updated by adding another entry to the *vExtent*.

Fig. 3 depicts the fragment of XML streams stored in the *DG* hashtable during the *createDGnLAB* encoding process.

Algorithm 1 Create DataGuide and region encoding labeling

```
 1. function createDGnLAB {
 2.     Input: an XML file X
 3.     Output: encoded streams and DataGuide hashtable
 4.     /* A stack eleStack to keep track of element sequence to form node label
 5.     A stack pStack to store tag sequence to form path label
 6.     A vector vExtent to store the occurrence of each node occurrence in stream
 7.     A hashtable DG to store node label and path label
 8.     A record with <start-end> */
 9.     int ptr = 0, start = 0, string curRec = null
10.     while (! eof (X)) do {
11.         if SAX event = a start tag <T> {
12.             if (tag is not found in DG) {
13.                 create new instance of vector, vExtent
14.                 DG.put(tag, ptr++)
15.             }
16.             create new instance of record, curRec
17.             curRec.start = start++
18.             int i = DG.get(tag).intValue
19.             vExtent[i].addElement(curRec)
20.             eleStack.push(curRec)
21.             pStack.push(tag)
22.         }
23.         if SAX event = an end tag </T> {
24.             index = pStack.size() – 1 – 1
25.             current = pStack.peek()
26.             while (index > 0) {
27.                 path = formPathLabel (pStack.elementAt(index), current)
28.                 if (path is not found in DG) {
29.                     create new instance of vector, vExtent
30.                     DG.put(tag, ptr++)
31.                 }
32.                 curRec.start = eleStack.peek()
33.                 int i = DG.get(tag).intValue
34.                 vExtent[i].addElement(curRec)
35.                 index—
36.             }
37.             pStack.pop()
38.             eleStack.pop()
39.             curRec.end = start++
40.             curRec = eleStack.peek()
41.         }
42.     }
43. }   // end function
44.
45. function formPathLabel (source, destination) {
46.     Input: source and destination node to form the path
47.     Output: Path label
48.     form path label from source to destination with '/' separator for each node
49. }
```

DG

//publications	<0-25>
//book	<1-16> <17-24>
//title	<2-3> <5 –6>
//book/title	<2-3>
//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/captio	<11-12>
…	…

Fig. 3. Fragment of the *DG* index table created from the *createDGnLAB* algorithm.

Algorithm 2 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
```

3.2 Online Processing: Query Evaluation

The input query is analyzed in the function *evaluateQuery* (depicted in Algorithm 2) to determine the type of query for processing. If it is a path query, the algorithm *Cut-MatchMergePath* is invoked. Else if the input query is a twig query, the algorithm *Cut-MatchMergeTwig* will be executed.

3.2.1 CutMatchMergePath algorithm

The algorithm (depicted in Algorithm 3) 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 only in the function *partitionTwig*. Lines 14-20 are the matching and merging processes. If the path query only contains P-C edges (lines 14-15), the final results can be obtained directly from the *DG* index table. Conversely, if the path query only contains A-D edges (lines 16-17), 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 18-19), the function *connectPQ* (lines 23-32) will be invoked to match-merge the final results.

Algorithm 3 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.     /* A vector temp_result to store each suffix path expression
 5.     A vector vPathQuery to store path query after partition process */
 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.     if ((temp_result.size()==0) && (vPathQuery.size()==1))
15.         finalSolution = hashDG to get path label
16.     else if ((temp_result.size() > 0) && (vPathQuery.size()=0))
17.         finalSolution = TwigStack(temp_result)
18.     else if ((temp_result.size() > 0) && (vPathQuery.size()==1))
19.         finalSolution = connectPQ(temp_result, vPathQuery)
20.     return finalSolution
21. }   // end function
22.
23. function connectPQ(descendantAxis, vPathQuery) {
24.     Input: descendant axis path/node and path query
25.     Output: connected path query
26.     /* A counter ti to iterate all occurrences in descendantAxis
27.     A counter pi to iterate all occurrences in vPathQuery */
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. }
```

3.2.2 CutMatchMergeTwig algorithm

The CutMatchMergeTwig algorithm is presented in Algorithm 4. This algorithm takes an input query, does a pre-order traversal and cuts any nodes and its subtree that are reached by the descendant axis (lines 7-11). Besides, it decomposes any branch axis of the form *p*[[[/*q*1]/*q*2]/…]/*r* into *p*/*q*1, *p*/*q*2, …, *p*/*r* as in lines 12-20. 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 23-24. Finally, all these results are joined holistically using the TwigStack algorithm.

Example 3: Let Q3: *book*/*chapter*[/*title*]//*figure*/*caption*. Since Q3 contains branching nodes and an A-D edges, 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.

Algorithm 4 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.     /* A vector temp_result to store each suffix path expression
 5.     A vector vPathQuery to store path query after partition process */
 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 nearer to the root of the twig
24.         finalSolution = twigStack(temp_result, vPathQuery, TopBranchNode)
25.         return finalSolution
26. }   // end function
```

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

We have implemented TwigX-Guide using Java API for XML Processing. Experiments have been carried out on two datasets, *i.e.*, one synthetic and one real datasets. The synthetic dataset, Standard, is obtained from the XMark benchmark project [13]. The real dataset, Mondial is obtained from the University of Washington XML repository [14]. Table 2 depicts the data environment. All our experiments are performed on 1.7GHz Pentium IV processor with 512 MB SDRAM running on Windows XP.

Table 2. Characteristics of Standard and Mondial datasets.

	Standard	Mondial
Size	114 MB	1.4 MB
Nodes	9.24×10^6	80,560
Depth	9	4
Distinct path	66	60

4.2 Queries

For each dataset, the queries are chosen based on the classification as follows: Q1 – Query with only P-C relationship, Q2 – Query with only A-D relationship and Q3 –

Query with mixed relationships. Suffix 'P' is used to represent path query and 'T' is used to represent twig query. For instance, PQ1 means path query with P-C relationship only.

4.3 TwigX-Guide Performance

To test the performance of TwigX-Guide, we benchmark TwigX-Guide with respect to TwigStack [12], TwigStackXB [12], TwigINLAB [15] and TwigStackList [16] on the Standard and Mondial datasets. To normalize the benchmarking to our computer environment, we have also coded these algorithms into JAVA. The performance result findings concur with that observed by Lu *et al.* [16] and Rao & Moon [17] respectively on the performance of TwigStack, TwigStackXB and TwigStackList.

4.3.1 Using the standard dataset

In test cases TC1 and TC2, we firstly benchmark the execution time of TwigX- Guide with the other approaches by using the set of queries listed in Table 3 over the Standard dataset.

Table 3. Queries over the standard dataset.

Test Case	Type of query	Query notation
TC1	PQ1	description/parlist/listitem/text/keyword
	PQ2	description//listitem//keyword
	PQ3	item/description/parlist//keyword
TC2	TQ1	item[/description/parlist/listiem]/mailbox/mail/from
	TQ2	lisitem[//keyword]//emph
	TQ3	item[[/location]//parlist/listiem/text]/shipping

Fig. 4. (a) Results for TC1; (b) Results for TC2 on the standard dataset.

Figs. 4 (a) and (b) show the execution time of several path and twig queries with single type and mixed types of relationships on the Standard dataset. From these figures, we draw several observations and conclusions:

- When the path query contains only P-C edges (as shown in PQ1), TwigX-Guide per-

forms significantly faster; about 71% faster as compared to TwigStack, 60% faster as compared to TwigINLAB, about 40% faster as compared to TwigStackXB and about 35% faster as compared to TwigStackList. TwigX-Guide outperforms the rest of the other approaches because the answers can be obtained directly from the *DG* index table and thus, no join is required. All the other approaches require a total of four joins. Moreover, path matching is high selectivity compared to node matching. However, TwigStackList performs better compared to TwigStack, TwigINLAB and TwigStackXB because it caches 'potential' elements to the main memory for faster processing when the query is of P-C edges. TwigStackXB is faster than TwigStack and TwigINLAB because of its ability to skip large portions in the input streams. TwigStack performs worse than TwigINLAB because TwigStack examines every node in the input stream.

- For twig query with P-C edges only (as shown in TQ1), TwigX-Guide outperforms TwigStack by 78%, TwigINLAB by 71%, TwigStackXB by 47% and TwigStackList by 45%. This is because all the other approaches require six joins with a lot of disk access, while our approach requires only one join. The performances of TwigStackXB and TwigStackList are comparable. Since the distributions of the matches are concentrated in certain region, TwigStackXB uses XB-trees to skip nodes in the input streams effectively. 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 path and twig query contain A-D edges only (as shown in PQ2 and TQ2), the performance of all approaches are comparable. This is because these approaches are based solely on region encoding labeling to retrieve "qualified" streams for further processing. Besides, all these approaches require the same number of joins. Although TwigStackXB has the ability to skip nodes in the input streams, queries PQ1 and TQ2 are low selectivity. Therefore, the performance of TwigStackXB is only slightly better than the rest of the approaches.
- For path query with mixed edges (containing both P-C and A-D edges) as shown in PQ3, TwigX-Guide is 24% faster than TwigStack, 20% faster than TwigINLAB and comparable to TwigStackList and TwigStackXB approaches. This is because the number of joins required in TwigX-Guide is less (for this case, only one join is required) as compared to the other approaches, which need more joins (for this case, three joins are required).
- On average, TwigX-Guide outperforms TwigStack by around 39%, TwigINLAB by around 34%, TwigStackList by around 15% and TwigStackXB by around 13% for the two test cases conducted on the Standard dataset.

To test for scalability, we used XMARK to generate the first ten Standard datasets based on a scaling factor of 0.1 to 1. Performance results for each approach using twig query TQ1 in TC2 are shown in Fig. 5. As illustrated in Fig. 5, TwigX-Guide increases less drastically compared to other approaches. The main reason for this is that uses fewer joins (in this case only one join). The other three approaches require a total of six joins. As the number of joins reduces, the number of disk access is fewer and henceforth the execution time is faster. This shows that TwigX-Guide is more scalable in processing large-scale datasets efficiently.

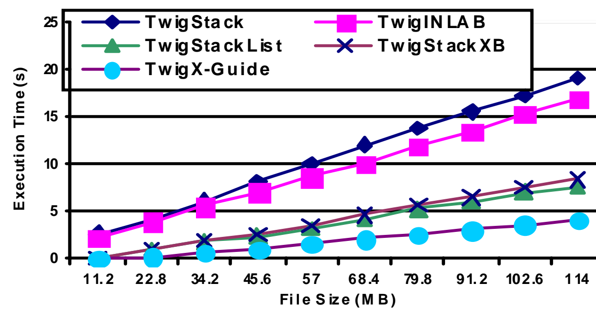
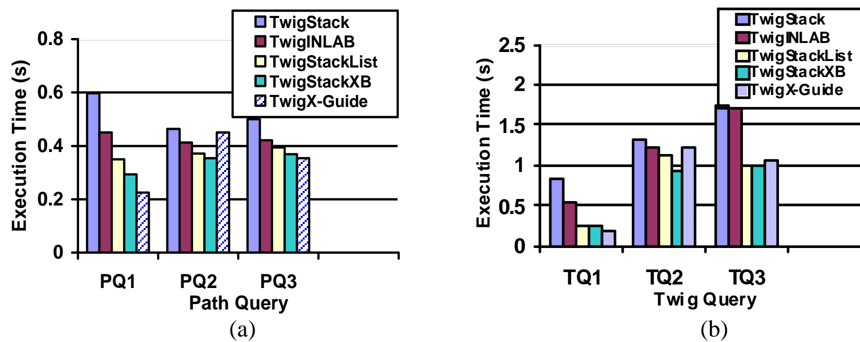


Fig. 5. Scalability test on various approaches.

Table 4. Queries over the mondial dataset.

Test Case	Type of query	Query notation
TC3	PQ1	country/province/city
	PQ2	country//city
	PQ3	mondial/country//population
TC4	TQ1	country[/religions]/city/population
	TQ2	country[[//city]//name]//population
	TQ3	mondial[/country//population]/organization/members



(a) (b)

Fig. 6. (a) Results for TC3; (b) results for TC4 on the Mondial dataset.

4.3.2 Using the Mondial dataset

In test cases TC3 and TC4, we benchmark the execution time of TwigX-Guide with the other approaches using the set of queries listed in Table 4 over the Mondial dataset.

Figs. 6 (a) and (b) shows the execution time of TwigX-Guide with respect to the other approaches on the real dataset, Mondial. Similar patterns were observed using Mondial dataset. From these figures, we see that TwigX-Guide, on average, outperforms TwigStack by around 36%, TwigINLAB by around 26%, TwigStackList by around 5% and is comparable to TwigStackXB. Although TwigX-Guide still outperforms the Twig-Stack, TwigINLAB and TwigStackList approaches (but comparable to TwigStackXB), the percentage is less significant as compared to the Standard dataset. This shows that TwigX-Guide performs best in supporting large dataset.

5. RELATED WORK

5.1 Path and Node Indexing

There are three types of structural indices, namely Path indexing, Node indexing and Sequence-based indexing [18].

Path indexing [3-7] creates a path summary from the XML tree starting from the root to the respective node. Among some of indexing schemes are DataGuide [3], T-index [4], APEX [5], A(k)-index [6] and D(k)-index [7]. However, all these approaches suffer from large index size growth. In addition, although path indexing greatly speeds up the evaluation of single-path queries, but it needs expensive join operations for processing queries with multiple branches. Our TwigX-Guide borrows the idea of labeling a path from DataGuide, but focuses on the optimization of twig query processing.

Node indexing approaches [8, 18-20] create indices on each node by its positional information (based on labeling schemes) within the XML tree. A compact and robust labeling scheme is essential to allow quick determination of the hierarchical relationships between pair of nodes. Any tree-structure query can be processed by matching each node in the query tree based on the hierarchical relationships, and by structurally joining these matches. In this context, many structural join algorithms [10-12, 15, 16] have been proposed to support efficient query processing (see section 5.2).

5.2 Twig Query Pattern Processing

MPMGJN [10], Stack-Tree [11], TwigStack [12], TwigStackXB [12] and TwigStackList [16] algorithms are based on region encoding <*start*, *end*, *level*> labeling of XML elements. MPMGJN and Stack-Tree 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. TwigStack evaluates twig query as a whole without decompositing it into binary relationships. Hence, memory is not used unwisely to store irrelevant nodes. TwigStackXB improves TwigStack by using XB-tree to skip nodes in the input streams. Nevertheless, the effectiveness of TwigStackXB depends on the distribution of possible matches in the input streams. If the possible solutions are concentrated to certain region, TwigStackXB is effective. However, if the possible solutions are distributed, TwigStackXB is frequently forced to drill-down to the lower regions to search for possible matches. Nevertheless, the main limitation of TwigStack and TwigStackXB are that they may produce large 'useless' intermediate results when queries contain any P-C relationships. Lu *et al*. [16] extend TwigStack and propose TwigStackList, which can support both P-C and A-D relationships efficiently. Their technique is to look-ahead by reading some elements in input data streams and cache 'potential' elements to the main memory. The main problem with all these approaches is that it is based on region encoding labeling schemes. In region encoding schemes, there is no path information of nodes. As a result, all nodes whose tags appear in the query must be evaluated for potential edge at a time, hence leading to a large number of joins required. Our TwigX-Guide overcomes the limitation of TwigStack by using DataGuide index to 'label' the path information.

6. CONCLUSION

We have presented the TwigX-Guide system 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 from DataGuide in terms of borrowing the path index to annotate the hierarchical relationships among individual nodes.

Also, we have proposed and implemented algorithms to decompose a complex query into path queries, subsequently evaluating them individually and finally joining the results using TwigStack. Experimental results indicate that TwigX-Guide can process twig queries on an average 38% better than the TwigStack algorithm, 30% better than Twig- INLAB, 10% better than TwigStackList and about 5% better than TwigStackXB in terms of execution time.

REFERENCES

1. W3C, "XQuery 1.0 and XPath 2.0 full-text," http://www.w3.org/TR/xquery-full-text.
2. W3C, "XPath, XML path language," http://www.w3.org/TR/xpath.
3. R. Goldman and J. Widom, "Data guides: Enabling query formulation and optimization in semistructured databases," in *Proceedings of Very Large Data Bases*, 1997, pp. 436-445.
4. T. Milo and D. Suciu, "Index structures for path expression," in *Proceedings of the 7th International Conference on Database Theory*, Vol. 1540, 1999, pp. 277-295.
5. C. W. Chung, J. K. Min, and K. Shim, "APEX: An adaptive path index for XML data," in *Proceedings of ACM Special Interest Group on Management of Data*, 2002, pp. 121-132.
6. R. Kaushik, D. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity to efficiently index paths in graph-structured data," in *Proceedings of International Conference on Data Engineering*, 2002, pp. 129-140.
7. Q. Chen, A. Lim, K. Ong, and J. Tang, "D(k)-index: an adaptive structural summary for graph-structured data," in *Proceedings of ACM Special Interest Group on Management of Data*, 2003, pp. 134-144.
8. E. Cohen, H. Kaplan, and T. Milo, "Labeling dynamic XML trees," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002, pp. 272-281.
9. P. Zezula, F. Mandreoli, and R. Martoglia, "Tree signatures and unordered XML pattern matching," in *Proceedings of Software Seminar*, 2004, pp. 122-139.
10. C. Zhang, J. Naughton, D. DeWitty, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *Proceedings of ACM Special Interest Group on Management of Data*, 2001, pp. 425-436.
11. S. A. Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural joins: A primitive for efficient XML query pattern matching," in *Proceedings of International Conference on Data Engineering*, 2002, pp. 141-152.
12. N. Bruno, D. Srivastava, and N. Koudas, "Holistic twig joins: optimal XML pattern

matching," in *Proceedings of ACM Special Interest Group on Management of Data*, 2002, pp. 310-321.
13. XMARK, "The XML-benchmark project," http://monetdb.cwi.nl/xml/.
14. University of Washington XML Repository, http://www.cs.washington.edu.edu/research/xmldatasets/.
15. S. C. Haw and C. S. Lee, "Stack-based pattern matching algorithm for XML query processing," *Journal of Digital Information Management*, 2007 (in press).
16. J. Lu, T. Chen, and T. W. Ling, "Efficient processing of XML twig patterns with parent child edges: A look-ahead approach," in *Proceedings of Conference on Information and Knowledge Management*, 2004, pp. 533-542.
17. P. Rao and B. Moon, "Sequencing XML data and query twigs for fast pattern matching," *ACM Transactions on Database Systems*, Vol. 31, 2006, pp. 299-345.
18. Q. Zou, S. Liu, and W. Chu, "Ctree: A compact tree for indexing XML data," in *Proceedings of Web Information and Data Management*, 2004, pp. 39-46.
19. X. Wu, M. L. Lee, and W. Hsu, "A prime number labeling scheme for dynamic ordered xml trees," in *Proceedings of International Conference on Data Engineering*, 2004, pp. 66-78.
20. P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORDPATHS: Insert-friendly XML node labels," in *Proceedings of ACM Special Interest Group on Management of Data*, 2004, pp. 903-908.

Su-Cheng Haw received her B.S. degree in 1999, and M.S. (IT) degree in 2001 from University Putra Malaysia, Malaysia. She is currently pursuing her Ph.D. (Information Technology) in Multimedia University, Malaysia. Her research interests include XML database, data modeling, query optimization, database tuning, data warehousing, entity-relationship approach and web programming.

Chien-Sing Lee is a senior lecturer in the Faculty of Information Technology, Multimedia University, Malaysia. Her research interests are in data mining, agents, knowledge representation and management, ontology mapping and merging, e-commerce, e-learning and quality assurance.