Efficient Processing of Multiple XML Twig Queries

Huanzhang Liu, Tok Wang Ling, Tian Yu, and Ji Wu

School of Computing, National University of Singapore
{liuhuanz, lingtw, yutian, wuji}@comp.nus.edu.sg

Abstract. Finding all occurrences of a twig pattern in an XML document is a core operation for XML query processing. The emergence of XML as a common mark-up language for data interchange has spawned great interest in techniques for filtering and content-based routing of XML data. In this paper, we aim to use the state-of-art holistic twig join technique to address multiple twig queries in a large scale XML database. We propose a new twig query technique which is specially tailored to match documents with large numbers of twig pattern queries. We introduce the *super-twig* to represent multiple twig queries. Based on the *super-twig*, we design a holistic twig join algorithm, called *MTwigStack*, to find all matches for multiple twig queries by scanning an XML document only once.

1 Introduction

Recently, XML has emerged as a standard information exchange mechanism on the Internet. XML employs a tree-structured model to represent data. XML query languages, such as XQuery and XPath, typically specify patterns with selection predicates on multiple elements for matching XML documents. Twig pattern matching has been identified as a core operation in querying tree-structured XML data.

Many algorithms have been proposed to match XML twig pattern [3, 7, 8, 11]. [3] decomposes the twig pattern into binary structural relationships, then matching the binary structural relationships and merging these matches. Bruno et al. [7] improved the methods by proposing a holistic twig join algorithm, called *TwigStack*. The algorithm can largely reduce the intermediate result comparing with the previous algorithms. Later on, Chen et al. [8] proposed a new *Tag+Level* labeling scheme and *iTwigJoin* algorithm to improve *TwigStack*. Lu et al. [11] designed a novel algorithm, called *TJFast*, which employed *extended Dewey* to match XML twig queries.

XML query processing also arises in the scenario of information dissemination, such as publish-subscribe (pub-sub) systems. In a typical pub-sub system, many user submitted profiles are presented by XPath expressions, and an XML document is presented as input. The goal is to identify the queries and their matches in the input XML document, and disseminate this information to the users who posed the queries [4, 9].

In a huge system, where many XML queries are issued towards an XML database, we expect to see that the queries have many similarities. In traditional database systems, there have been many studies on efficient processing of similar queries using batch-based processing. Since pattern matching is an expensive operation, it would save a lot in terms of both CPU cost and I/O cost if we can process multiple similar twig queries simultaneously and only scan the input data once to get all the results. [6] has proposed *Index-filter* to process multiple simple XPath queries (no branch) against an XML document and it aims to find all matches of multiple simple path queries in an XML document. To eliminate redundant processing, it identifies query commonalities and combines multiple queries into a single structure. But *Index-Filter* does not consider how to process multiple twig queries.

Motivated by the recent success in efficient processing multiple XML queries, we consider the scenario of matching multiple XML twig queries with high similarity against an XML document.

The contributions of this paper can be summarized as follows:

- We introduce a new concept, called *super-twig*, which combines multiple twig queries into just one twig pattern. We also give the algorithm of constructing *super-twig*.
- Based on *super-twig*, we develop a new multiple twig queries processing algorithm, namely *MTwigStack*. With the algorithm, we can find all matches of multiple twig queries by scanning input data only once.
- Our experimental results show that the effectiveness, scalability and efficiency of our algorithm for multiple twig queries processing.

The rest of this paper is organized as follows. Preliminaries are introduced in Section 2. The algorithm *MTwigStack* is described in Section 3. Section 4 is dedicated to our experimental results and we close this paper by conclusion and future work in Section 5.

2 Preliminaries

2.1 Data Model

We model XML documents as ordered trees, each node corresponding to an element or a value, and the edges representing element-subelement or element-value relationships. Each node is assigned a region code (*start*:*end*, *level*) based on its position in the data tree [3, 6, 7], *start* is the number in sequence assigned to an element when it is first encountered and *end* is equal to one plus the *end* of the last element visited, *level* is the level of a certain element in its data tree. Each text value is assigned a region code that has the same *start* and *end* values. Then structural relationships between tree nodes (elements or values), such as parent-child or ancestor-descendant, whose positions are labelled with region encoding can be determined easily. Figure 1 (a) shows an example XML data tree with region encoding.

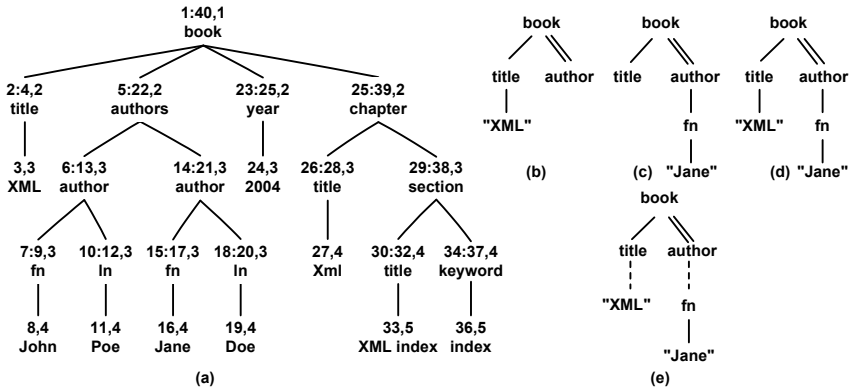


Fig. 1. An XML tree (a), three twig queries (b, c, d) and the super-twig query (e)

2.2 Super-Twig Query

When multiple twig queries are processed simultaneously, it is likely that significant commonalities between queries exist. To eliminate redundant processing while answering multiple queries, we identify query commonalities and combine multiple twig queries into a single twig pattern, which we call *super-twig*. *Super-twig* can significantly reduce the bookkeeping required to answer input queries, thus reducing the execution time of query processing. We will use q (and its variants such as q_i) to denote a node in the query or the subtree rooted at q when there is no ambiguity. We introduce the concepts *OptionalNode* and *OptonalLeafNode* to distinguish *super-twig* query from general twig queries.

In this paper, we only consider the tree patterns belonging to the fragment of XPath $XP^{\{/,//,[\ ]\}}$ [5] and the scenario that commonalities only existing in the top parts of the twigs. Given a set of twig queries against an XML document, $Q = \{q_1, \ldots, q_k\}$ belonging to $XP^{\{/,//,[\ ]\}}$, and assuming there is no repeated node in each query, we combine all the queries into a *super-twig* such that:

- The set of nodes in the super twig pattern equals the union of the sets of nodes of all individual twig queries;
- Each twig query is a subpattern (defined by [10]) of the super twig pattern;
- If the queries have different root nodes, we rewrite the queries whose root nodes are not the root of the XML document and add the document's root as the root node of the queries. Then the root node of the super twig pattern is same as the document's root;
- Suppose n is a query node which appears in q_i and q_j, P_i and P_j are the paths from the *root* to n in q_i and q_j respectively, P_i is same as P_j and m is the parent node of n in these two queries. If the relationship between m and n is Parent-Child (P-C) in q_i, Ancestor-Descendant (A-D) in q_j, then the relationship between m and n in *super-twig* is relaxed to A-D;
- Suppose n is a query node in one query q_i of Q, and m is the parent node of n in q_i. Let Q_n is the subset of twig queries of Q which contain node n, and Q_m is the subset of twig queries which contain node m (the path from its

root to m must be a prefix of the path from its root to n). If $Q_n \subset Q_m$, we call n an *OptionalNode*. And if all the relationships between m and n in Q_n are P-C relationships, then the relationship between m and n in the super twig pattern is P-C (called *optional parent-child* relationship and depicted by a single dotted line); otherwise, the relationship between m and n in the super twig is A-D (called *optional ancestor-descendant* relationship and depicted by double dotted lines);
- Following the same situations of the above item and assuming n is *OptionalNode*, let $Q_x = Q_m - Q_n$. If m is a leaf node in some queries of Q_x (so $Q_x \neq \emptyset$), then we call m an *OptionalLeafNode*.

Example 1. In Figure 1, (e) shows the *super-twig* query of three queries (b), (c) and (d). *"XML"* and *fn* are *OptionalNodes*, *title* and *author* are *OptionalLeafNodes*. The edge which connects *"XML"* to *title* represents *optional parent-child* relationship. It means that we can output path solution "book-title" whether or not the element *title* has a child whose content is *"XML"* in an XML document, or output path solution "book-title-'XML' " when the element *title* has a child whose content is *"XML"* in an XML document.

2.3 super-twig

To combine multiple twigs into a *super-twig*, we should normalize them first. It means to obtain a unique XPath query string from a tree pattern sorting the nodes lexicographically. We use the method proposed in [12], for example, the normal form of /a[q][p]/b[x[z]/y] is /a[p][q]/b[x[y][z]]. Then we design an algorithm according to the principles proposed in the last section, as shown in Algorithm 1. We input twig queries one by one and output the *super-twig* presented by XPath query.

Algorithm 1 *SuperTwig* (s, r, q)

input: s is the current *super-twig* and r is its root, q is a twig query

```
1: q = NormalizeTwig(q)
2: if s = NULL then return q
3: rewrite q and s with the root of document
4: let s_k denote each children(r) in s for k = 1, ..., n and j = 1
5: for each child q_i of the root r_q in q
6:    findmatchedNode = FALSE
7:    while j ≤ n
8:       if q_i = s_j then
9:          update the edge between r and s_j
10:         SuperTwig(subtree(s_j), subtree(q_i), s_j)
11:         let findmatchedNode = TRUE and break while
12:      else j++
13:   if findmatchedNode = FALSE then
14:      if isLeaf(r) then r is marked as OptionalLeafNode in s
15:      append subtree(q_i) to s below r and assign edge between r and q_i
16:      q_i is marked as OptionalNode in s
17: return s
```

3 Multiple Twig Queries Matching

3.1 Data Structure and Notations

Let SQ denote the *super-twig* pattern, and *root* represent the root node of SQ. In our algorithm, each node q in SQ is associated with a list T_q of database elements, which are encoded with (*start*:*end*, *level*) and sorted in ascending order of the *start* field. We keep a cursor C_q for each query node q. The cursor C_q points to the current element in T_q. Initially, C_q points to the head of T_q. We can access the attribute values of C_q by $C_q.start$ and $C_q.end$.

In *MTwigStack* algorithm, we also associate each query node q in the *super-twig* query with a stack S_q. Each data node in the stack consists of a pair: (region encoding of a element from T_q, pointer to a element in $S_{parent(q)}$). Initially, all stacks are empty. During query processing, each stack S_q may cache some elements and each elements is a descendant of the element below it. In fact, cached elements in stacks represent the partial results that could be further contributed to final results as the algorithm goes on.

3.2 The MTwigStack Algorithm

Given the *super-twig* query SQ of $\{q_1, \ldots, q_n\}$ and an XML document D, a match of SQ in D is identified by a mapping from nodes in SQ to elements and content values in D, such that: (i) query node predicates are satisfied by the corresponding database elements or content values, and (ii) the structural relationships between any two query nodes are satisfied by the corresponding database elements or content values. The answer to the *super-twig* query SQ with n twig queries can be represented as a set $R = \{R_1, \ldots, R_n\}$ where each subset R_i consists of the twig patterns in D which match query q_i.

Algorithm *MTwigStack*, for the case when the lists contain nodes from a single XML document, is presented in Algorithm 2. We execute *MTwigStack(root)* to get all answers for the *super-twig* query rooted at *root*. *MTwigStack* operates in two phases. In the first phase, it repeatedly calls the *getNext*(q) function to get the next node for processing and outputs individual root-to-leaf and root-to-*OptionalLeafNode* path solutions. After executing the first phase, we can guarantee that either all elements after C_{root} in the list T_{root} will not contribute to final results or the list T_{root} is consumed entirely. Additionally, we guarantee that for all descendants q_i of *root* in the *super-twig*, every element in T_{q_i} with *start* value smaller than the *end* value of last element processed in T_{root} was already processed. In the second phase, the function *mergeAllPathSolutions*() merges the individual path solutions for respective original twig queries.

To get the next query node q to process, *MTwigStack* repeatedly calls function *getNext(root)* and the function will call itself recursively. If q is a leaf node of the *super-twig*, the function returns q without any operation because we need not check whether there exist its descendants matching the *super-twig*; otherwise, the function returns a query node q_x with two properties: (i) if $q_x = q$, then $C_q.start < C_{q_i}.start$ and $C_q.end > C_{q_{max}}.start$ for all $q_i \in children(q)$

Algorithm 2 `MTwigStack(`*root*`)`

```
1: while NOT end(root) do
2:    q = getNext(root)
3:    if NOT isRoot(q)      cleanStack(S_parent(q), C_q.start)
4:    cleanStack(S_q, C_q.start)
5:    if isRoot(q) OR NOT empty(S_parent(q))
6:       push(C_q, S_q)
7:       if isLeaf(q)      outputSolution(S_q),   pop(S_q)
8:       else if isOptionalLeafNode(q)      outputSolution(S_q)
9:    else   advance(C_q)
10: end while
11: mergeAllPathSolutions()
```

Function getNext(q)

```
1: if isLeaf(q)   return q
2: for q_i ∈ children(q) do
3:    n_i = getNext(q_i)
4:    if n_i ≠ q_i   return n_i
5: q_min= the node whose start is minimal start value of all q_i ∈ children(q)
6: q_max= the node whose start is maximal start value of all q_i ∈ children(q)
        which are not OptionalNodes
7: while q_max ≠ NULL and C_q.end < C_qmax.start do   advance(C_q)
8: if C_q.start < C_qmin.start   return q
9: else    return q_min
```

Procedure cleanStack(S_p, $qStart$)

```
1: pop all elements e_i from S_p such that e_i.end < qStart
```

Procedure mergeAllPathSolutions()

```
1: for each q_i ∈ Q do
2:    read the path solution lists whose leaf node is a leaf node of q_i
3:    merge the path solutions and check the relationships between any two nodes
```

and q_i is not *OptionalNode* (lines 5-8 in Func. *getNext(q)*). In this case, q is an internal node in the *super-twig* and C_q will participate in a new potential match. If the maximal *start* value of C_q's children which are not *OptionalNodes* is greater than the *end* value of C_q, we can guarantee that no new match can exist for C_q, so we advance C_q to the next element in T_q (see Figure 2(a)); (ii) if $q_x \neq q$, then $C_{q_x}.start < C_{q_j}.start$, for all q_j is in siblings of q_x and $C_{q_x}.start < C_{parent(q_x)}.start$ (lines 9 in Func. *getNext(q)*). In this case, we always process the node with minimal *start* value for all $q_i \in children(q)$ even though q_i is *OptionalNode* (see Figure 2(b)). These properties guarantee the correctness in processing q.

Next, we will process q. Firstly, we discard the elements which will not contribute potential solutions in the stack of q's parent (see Figure 2(c)) and execute the same operation on q's stack. Secondly, we will check whether C_q can match the *super-twig* query. In the case that q is *root* or the stack of q's parent is not empty, we can guarantee C_q must have a solution which matches the subtree

rooted at q. If q is a leaf node, then it means that we have found a root-to-leaf path which will contribute to the final results of some or all queries; hence, we can output possible path solutions from the node to *root*; especially, if q is an $OptionalLeafNode$, we can also output the path for some queries, but we do not pop up S_q because q is an internal node and maybe will contribute to other queries in which q is not a leaf node. Otherwise, C_q must not contribute any solutions and we just advance the pointer of q to the next element in T_q (see Figure 2(d)).

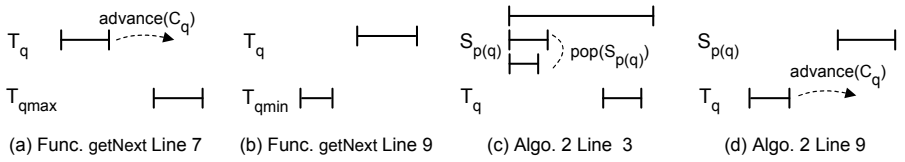


Fig. 2. Possible scenarios in the execution of *MTwigStack*

In [7], when *TwigStack* processes a leaf node, it outputs root-to-leaf solutions. However, for *super-twig*, there are leaf nodes and optional leaf nodes. Different from *TwigSack* in the first phase, *MTwigStack* will output path-to-leaf and path-to-*OptionalLeafNode* solutions if a node q of *super-twig* is leaf or *OptionalLeafNode* (it means q is a leaf node in some queries). Furthermore, in the function $getNext(q)$, q_{max} is the node whose *start* is maximal *start* value of all q's children which are not *OptionalNodes*. This restriction guarantees that some nodes in T_q are not skipped mistakenly by $advance(C_q)$ when some children of q are not necessary for all the twig queries.

After all possible path solutions are output, they are merged to compute matching twig instances for each twig query respectively. In this phase, we will not only join the intermediate path solutions for each query but also check whether P-C relationships of the queries are satisfied in these path solutions. Merging multiple lists of sorted path solutions is a simple practice of a multi-way merge join. In this paper, we do not explain the details for saving space.

MTwigStack is a modification of the *TwigStack* algorithm. The main diversification is to introduce the concept of *OptionalLeafNode*, which is treated as a leaf node when processing the *super-twig*. The algorithm will output intermediate matches when processing the *OptionalLeafNodes* as they are in fact leaf nodes of some twig queries. Hence, we can easily modify other algorithms such as *iTwigJoin* [8], *TJFast* [11], etc.

Example 2. In Figure 3, SQ is the *super-twig* of q_1, q_2, and q_3; in SQ, C is an *OptionalLeafNode*, D and E are *OptionalNodes*; *Doc*1 is an XML document. Initially, $getNext(A)$ recursively calls $getNext(B)$ and $getNext(C)$. At the first loop, a_1 is skipped and C_A advances to a_2 because a_1 has no descendant node C. Then node B is returned and $q = B$. Now the stack (S_A) for parent of B is empty, hence, $b1$ is skipped and C_B points to $b2$. In the next loop, A is returned and $a2$ is pushed into S_A; next, B is returned and ($a2$, $b2$) is output; then A is returned again and $a3$ is pushed into S_A but $a2$ will be not popped; B is returned and $b3$ is pushed into S_B, ($a3$, $b3$) and ($a2$, $b3$) are output. At the sixth

loop, C is returned and c_1 is pushed into S_C. C is an *OptionalLeafNode*, hence ($a3$, $c1$) and ($a2$, $c1$) are output but $c1$ is not popped. Next D is returned and $d1$ is pushed into S_D; Then F is returned, (a_3, c_1, d_1, f_1) and (a_2, c_1, d_1, f_1) are output. Next, $c2$ is processed, ($a3$, $c2$) and ($a2$, $c2$) are output. Finally, E is returned, then ($a3$, $c2$, $e1$), ($a3$, $c1$, $e1$), ($a2$, $c2$, $e1$) and ($a2$, $c1$, $e1$) are output. At the second phase, *mergeAllPathSolutions*() merges the path solutions of (A, B) and (A, C) for q_1, (A, B) and (A, C, D, F) for Q_2, and (A, B) and (A, C, E) for q_3. In this phase, we also check whether P-C relationships are satisfied.

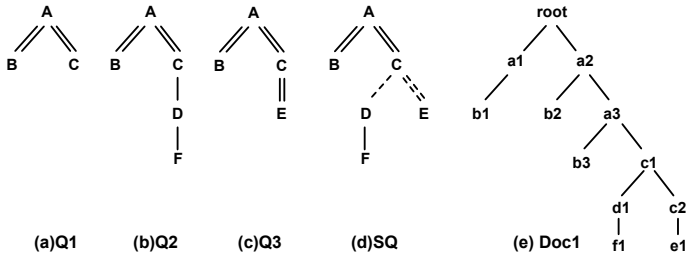


Fig. 3. Illustration to *MTwigStack*

4 Experimental Evaluation

4.1 Experimental Setup

We implemented *MTwigStack* algorithm in Java. All experiments were run on a 2.6 GHz Pentium IV processor with 1 GB of main memory, running windows XP system. We used the TreeBank [1] and XMark [2] data sets for our experiments. The file size of TreeBank is 82M bytes, and the file sizes of XMark are 128KB, 2MB, and 32MB respectively. We test our *MTwigStack* comparing with *TwigStack* [7] and *Index-Filter* [6] with different numbers of queries on these different data sets.

The set of queries consists of 1 to 10000 twig queries, with a random number of nodes between 10 to 20. The total number of distinct tags in these twig queries is less than 30% of total distinct tags (75 tags) for XMark data sets, and is less than 15% of total distinct tags (249 tags) for TreeBank data set.

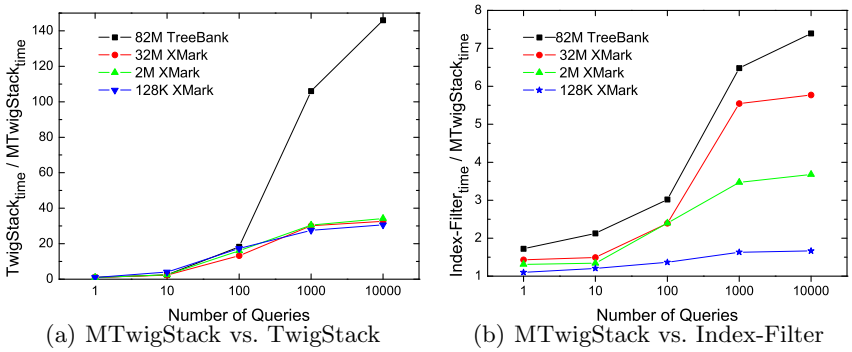
4.2 Experimental results

MTwigStack vs. TwigStack Figure 4 (a) shows the execution time of *TwigStack* to the execution time of *MTwigStack* on the four data sets when processing different numbers of queries. We find that whatever the data size is, when there is only one query, these two methods consume the same time; with the number of queries increasing, the processing time increase of *MTwigStack* is far lower than the increase of *TwigStack* (e.g. the ratio is about 60 for 1000 queries on the TreeBank data set). This is explained by the fact that *MTwigStack* process all the multiple queries simultaneously, while *TwigStack* needs to match the queries one by one.

In table 1, we show the number of elements scanned by *MTwigStack* and *TwigStack* when processing different numbers of queries. Obviously, *MTwigStack* scans far less elements than *TwigStack* does. The reason is, for the nodes which appear in multiple queries, *MTwgStack* scans them only once. But extremely, *MTwigStack* and *TwigStack* will scan the same number of elements only when there is no node that appears in all the queries repeatedly, that is, all nodes in the multiple queries are distinct.

MTwigStack vs. Index-Filter We implemented *Index-Filter* as follows: firstly, decomposing twig pattern into simple path queries for each twig query and combining these path queries into a *prefix tree*; next, executing the *Index-Filter* algorithm to get intermediate solutions for each path; finally, joining the path solutions which belong to the same query and eliminating useless solutions.

Figure 4 (b) shows the execution time of *Index-Filter* to the execution time of *MTwigStack*. With the increase of data size and number of queries, *Index-Filter* will run longer time even though it scans the same number of elements as *MTwigStack* does, as shown in Table 1. The reason is, *Index-Filter* decomposes a twig query into multiple simple paths during query processing and it will produce many useless intermediate path solutions, as shown in Table 2. Merging more path solutions also need consume more time. Furthermore, *Index-Filter* also requires more space to keep intermediate results.



(a) MTwigStack vs. TwigStack (b) MTwigStack vs. Index-Filter

Fig. 4. Execution time ratio for different data sets

Table 1. The number of scanned elements

Data Set	128K XMark		2M XMark		32M XMark		82M TreeBank	
No. of Queries	10	100	10	100	10	100	10	100
MTwigStack	286	397	5027	6059	78167	96357	1278766	1465232
Index-Filter	286	397	5027	6059	78167	96357	1278766	1465232
TwigStack	2312	18455	40337	354260	635718	6005265	11685319	106760340

5 Conclusion and Future Work

In this paper, we proposed a new twig join algorithm, called *MTwigStack*, to process multiple twig queries with a high structural similarity. Although holistic twig join has been proposed to solve single twig pattern, applying it to multiple

Table 2. The number of intermediate path solutions

Data Set	128K XMark		2M XMark		32M XMark		82M TreeBank	
No. of Queries	10	100	10	100	10	100	10	100
MTwigStack	29	33	349	459	5401	7386	646	678
Index-filter	134	157	2332	2827	37197	44797	496691	498688
TwigStack	127	1215	1237	10425	19897	172360	775	3965

twig patterns matching is nontrivial. We developed a new concept *super-twig* with *OptionalNode* and *OptionalLeafNode* to determine whether an element is in the shared structure of the XML twig patterns. We also made the contribution by processing the shared structure in the *super-twig* only once. The experimental results showed that our algorithm is more effective and efficient than the applying *TwigStack* to each individual twig quires, or applying *Index-Filter* by decomposing twig queries into many simple path queries.

In the future, we will improve the algorithm based on the following two issues: one is to design an efficient index scheme to fasten the processing speed. Another issue is our method only supports a subset of XPath queries. Some queries, such as //A[B]/C and //D[B]/C, can not be processed efficiently. We will try to process more XPath queries.

References

1. Treebank. Available from http://www.cis.upenn.edu/treebank/.
2. The xml benchmark project. Available from http://www.xml-benchmark.org.
3. S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.
4. M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB*, 2000.
5. S. Amer-Yahia, S. Cho, L. K. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceedings of ACM SIGMOD*, 2001.
6. N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proceedings of ICDE*, 2003.
7. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of ACM SIGMOD*, 2002.
8. T. Chen, J. Lu, and T. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of ACM SIGMOD*, 2005.
9. Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. In *ACM Transactions on Database Systems (TODS)*, volume 28, pages 467–516, 2003.
10. S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *Proceedings of VLDB*, 2003.
11. J. Lu, T. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *Proceedings of VLDB*, 2005.
12. B. Mandhani and D. Suciu. Query caching and view selection for xml databases. In *Proceedings of VLDB*, 2005.