Journal of Web Engineering, Vol. 14, No.3&4 (2015) 234-250
© Rinton Press

A NOVEL TWIG-JOIN SWIFT USING SST-BASED REPRESENTATION FOR EFFICIENT RETRIEVAL OF INTERNET XML

YI-WEI KUNG

Department of Computer Science and Engineering, National Sun Yat-sen University
d963040001@student.nsysu.edu.tw

HSU-KUANG CHANG

Department of Information Engineering, I-Shou University
hkchang@isu.edu.tw

CHUNG-NAN LEE

Department of Computer Science and Engineering, National Sun Yat-sen University
cnlee@cse.nsysu.edu.tw

Received February 15, 2014
Revised December 9, 2014

Compiling documents in extensible markup language (XML) plays an important role in accessing data services when both rapid response and the precise use of search engines are required. The main operation in XML query processing is to find nodes that match the given query tree pattern (QTP) in the document. An efficient query service should be based on a skillful representation that can support low complexity and high precision search capabilities. However, accessing too many useless nodes in order to match a query pattern is very time-consuming. This paper proposes a structural summary tree (SST) algorithm that is not only able to satisfy a query, but also has better time-saving efficiency compared with the existing twig-join algorithms such as the TJFast algorithm. A novel twig-join Swift (TJSwift) associated with adjacent linked (AL) lists for the provision of efficient XML query services is also proposed, in which queries can be versatile in terms of predicates. TJSwift can completely preserve hierarchical information, and the new index generated from SST is used to save semantic information. The TJSwift approach can provide template-based indexing for fast data searches. An experiment is also conducted for the evaluation of the TJSwift approach.

Key words: XML, query tree pattern, structural summary tree, TJSwift, TJFast
Communicated by: M. Gaedke & P. Fraternali

1 Introduction

With the rapid development of Internet and information technology, W3C [1] defines XML as a standard information description language widely used in the exchange of information and data storage. It is used in many applications such as science, biology, business and, particularly, web information systems using XML as their data representation format. Therefore, many XML researches, including improved data storage, transport and query XML, have been proposed in the past few years.

Because XML content is growing so fast, XML and optimization methods have gradually become a major web service research focus.

In the query method, a combination tree structure with content for the XML document provides m any query methods which are the most commonly applied Xpath [2] and XQuery [3]. The two most popular query languages in the XML domain are based on path expressions. In order to process XML queries, all portions matching the query tree pattern in the XML document must be found; this is a time-consuming task, especially if very large XML documents are involved. These patterns include the most important query axes: parent child (PC, / for short) and ancestor descendant (AD, // for short). Therefore, the XML query and optimization methods are based on path expression query methods. Accordingly, many researchers have proposed algorithms to improve the path expressions so that they more efficiently match the query; these can be categorized into string [4], [5], path [6-16] and twig [17-20] groups. The remarkable twig pattern matching methods have exceptional efficiency, much more so than the other methods overall. String representation can be derived with preorder traversal, which requires dynamic programming for editing distance measurements. This approach, with its lack of structure information, may lead to indeterminate search results. Path approaches use sub-paths as the features, and represent each XML datum as a binary vector. An element in the binary vector denotes whether the datum involves a corresponding feature, where such features can be defined as tree nodes [7], two-node sub-paths (i.e. node-pair (NP)) [8-9], or whole paths (WP) [10-11]. To improve search efficiency, several path representation modifications have been proposed. Yang et al. [12] used the content instead of the leaf node for node representation. Liu et al. [13] presented a hybrid definition that combines NP and WP for XML data description. Based on the determined finite automata, Mustafa et al. [14] and Lee et al. [15] presented a path-embedded string representation. In order to improve the efficiency of common Xpath and principal component analysis, Li [16] presented a modified WP with limited-length paths. These approaches are essentially only concerned with the service of simple queries (single path queries). Nicolas B. [17] proposed holistic twig join opinion and TwigStack algorithms in the XML path expression in order to discuss AD relations. This algorithm manipulates every node relationship by construction index structure, and utilizes multiple stacks, based on twig pattern status, to design a concatenated stack data structure to represent a matching data structure; many twig pattern algorithms have since been proposed. Chen S. [18] improved its practice further, proposing a two layer stack mode. TwigList [28] is elimination of the merge cost in the second phase of TwigStack because it achieved by using simple lists rather than the hierarchical stacks. Jiang et al. [19] proposed TSGeneric+ algorithm using the concept of XR-Tree indexes, which can pre-determine the unconnected node to be skipped. The majority of these methods for accessing all query tree patterns in the XML document must process all of the query nodes; as a result, it is very time consuming. A few methods, such as Lu et al. [20] have identified a key issue in holistic algorithms, called TreeMatch, which use concise encoding to present matching results, and then reduce unused intermediate results. TJFast [21] utilizes the Extended Dewey encodes to label each node. The encoding result is combined with Finite state transducer (FST) techniques from single node coding to derive the path of all element names from the root. However, the algorithm also generates a lot of unnecessary nodes matching the query path; the results not only require too much time to produce, but the query matching also increases the number of results. According to the Document Type Definition (DTD) for XML characteristics, the results will produce many repetitive queries matching information, and these redundant data lead to low efficiency of the query search. Moreover the query node increases

the number of comparisons.

The remainder of this paper is organized as follows: In Section 2, the optimization XML document and corresponding structural summary tree (SST) are proposed. In Section 3, twig-join swift (TJSwift) with SST is described. Section 4, shows the performance evaluation results of handling versatile queries using variant approaches. Finally, conclusions are drawn in Section 5.

2 The Proposed Structural Summary Tree for Optimal XML Document

In this section, the optimized XML tree structure is described. Any XML datum defined with the DTD can be modeled as an ordered label tree [4], shown in Figures 1 (a) and (b), respectively. The hierarchical tree information is extracted by a pre-ordered traversal process performed with a document object model (DOM). It can be seen that two effects of DTD features lead to increased unnecessary query matches and low efficiency in the tree pattern. The first originates in the nested XML structure tree with a left-recursion problem according to the DTD features. Another is the impact of duplicate query paths: often many duplicate paths are found in creating the XML tree. Some approaches have been proposed that aim to avoid nested and repeated access to the input data tree, and only consider the repeat factor, such as [24], which supports a matching algorithm called TwigVersion that combines with the Dewey ID labeling scheme. QueryGuide [25] labels DataGuide nodes with Dewey ID lists, and is part of the S^3 matching method. The optimized XML tree is intended to achieve the best possible minimization, high satisfaction and provide results corresponding to the query matching. The optimized XML tree is called a structural summary tree (SST), as in [26], [27]. The SST of an XML document formulate the problem as follows: given a XML document, find an equivalent XML of the smallest size. Formally, given a XML data tree t and a smallest size XML data tree P of size n (i.e., number of nodes), let $S = \{P_i\}$ be the set of smallest size XML data tree of size n_i contained in t (P_i t and $n_i \leq n,$ i). Minimizing t is finding a smallest size P_{min} S of size n_{min} such that

- $P_{min} \equiv t$ when matched against t;
- $n_{min} \leq n_i,$ i.

Moreover, that can be extracted by three functional processes, as follows:

Step 1. This step performs a tree conversion using Java DOM (JDOM), where the tree element values are neglected. For the DTD-formatted XML datum of Figure 1 (a), the tree conversion process result is illustrated in Figure 1 (b). Figure 1 (b) shows nested and duplicate paths.

Step 2. For efficient matching, the name of the tree node is symbolized with an abbreviated character order, as shown in Figure 2.

Step 3. Based on the pre-order traversal process, SST extraction requires two simplification procedures:

a) First reduce the nested procedure. For each node, examine whether the current node name is equal to a parent name and a child node name (the current is not a root). If it is, set the current node's sub-tree to be a child of the parent; otherwise, check the next node. The purpose of this procedure is to remove nested sub-paths, as shown in Figure 3. Figure 3 shows the tree structure, without a nested tree structure and with no left-recursion problem.

b) Next, eliminate the duplicate paths which check every path formed by a node to determine

whether the parent node and child node are the same. In addition, it is important that the comparison node be on the same level; if the condition is true, then eliminate and retain the unique path. Conversely, if the comparison node is not on the same level, then continue to the next node even though the path is the same. Figure 4 shows the repeated branch elimination result where the simplified tree is the SST. The reconstructed SST eliminates nested-duplicated paths; this satisfies minimization and high satisfaction, while providing results corresponding to the matching query. The algorithms for reducing nested nodes and eliminating duplicate nodes are given in Figures 5 and 6, respectively.

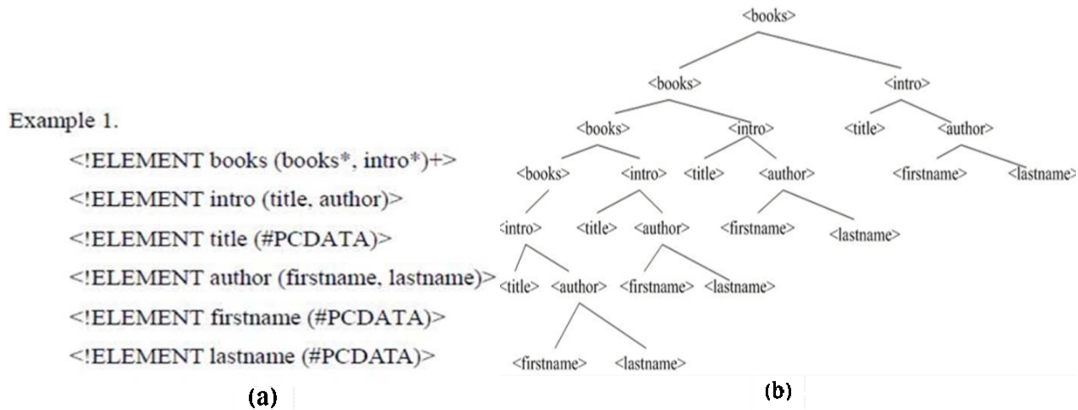


Figure 1: (a) An example based on the DOM; (b) Ordered label tree.

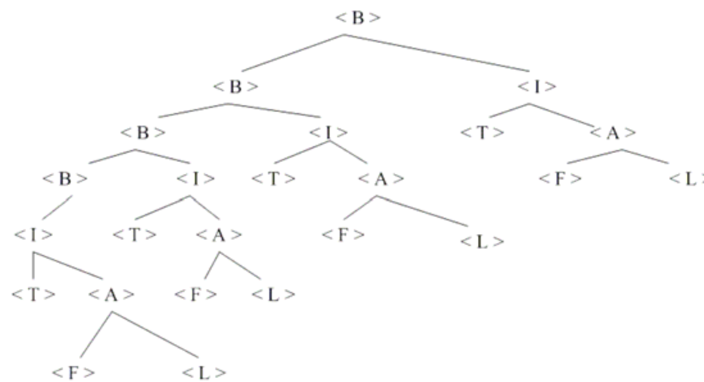


Figure 2: Symbolization process for the XML tree from Figure 1 (b).

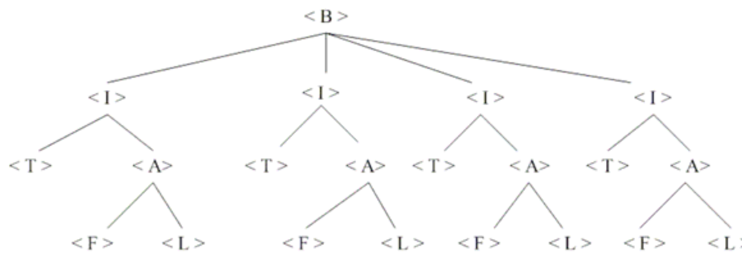


Figure 3: The XML tree of Figure 1 (b) with nested nodes removed.

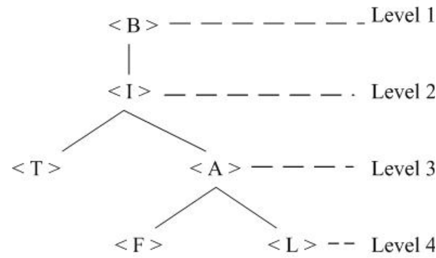


Figure 4: The SST of Figure 2 where the level of the root is defined as 1 and increased to the leaves.

$R_{educe}N_{ested}$ Algorithm (*XML*)
// *AL* :: adjacent-linked list of each XML tree node.
// δ_i :: Head Node for AL list.
// *stack* :: store the nodes for AL : δ_i.
// PUSH(), POP() :: Two operations for stack.
// Find-Parent-Child :: Collection of node info include parent and child.
// Compare-unlike :: A function checks the nested condition.
// *RNTree* :: Reduce nested XML tree.
// $Node_i$, *p*, and *c* define as the *i*th node, *parent* and *child* respectively.
1. **for** each node $\delta_i \leftarrow 1$ **to** k **do** //Each each node of *XML* in *AL* :: $\delta_i \sim \delta_k$
2. $Node_i$ = Find-Parent-Child ($AL_i : \delta_i$) //find node's child & parent
3. **if** ($Node_i$ is root) **then**
4. PUSH($Node_i$, *p*, *c*, *stack*) //push nodes info in stack
5. **else if** (Compare-unlike($Node_i$, *stack*($Node_{i-1}$))) //compare parent & child
6. PUSH($Node_i$, *p*, *c*, *stack*)
7. **end**
8. *RNTree* = POP(*stack*) //after reduced nested by POP

Figure 5: XML documents with nested nodes reduced.

```
E_liminateD_uplicate Algorithm (RNTree)
// AL :: adjacent-linked list of each RNTree.
// δi :: Head Node for AL list.
// stack :: store the nodes for AL : δi.
// PUSH(), POP() :: Two operations for stack.
// Find-Node-Level :: A function returns the node level for ALi : δi
// Find-Parent-Child :: Collection of node info include parent and child.
// compare--equivalent :: A function determines duplicate condition.
// SST :: Structural summary tree after reduce nested-duplicate method.
// Nodei , p, c and l define as the ith node, parent, child and level respectively.

1.  for each node δi ←1 to k do //Each node of RNTree in AL :: δi ~ δk
2.     l = Find-Node-Level(ALi : δi) //Find node level
3.        Nodei = Find-Parent-Child(ALi : δi) //find node's child & parent
4.    if (Nodei is root) then
5.      PUSH(Nodei , p, c, l, stack) //push nodes info in stack
6.    else if (!Compare-equivalent(Nodei , stack(Nodei - 1))) //compare p, c, l
7.      PUSH(Nodei , p, c, l, stack)
8.  end
9.  return SST = POP(stack) //after Repeat reduced by POP
```

Figure 6: XML documents with duplicate nodes eliminated.

3 The Proposed Twig-Join Swift Using Indexing AL SST

In this section, the AL list of structural summary tree (SST) and twig-join swift (TJSwift) algorithm methods are introduced.

3.1 The AL list of SST Representation

The Dewey scheme has been extended to combine node names [21] by exploiting schema information available in XML documents. Encoding node names along a path into a Dewey label provides not only the labels of the ancestors and parents of a given node, but also their names. The main difference between Dewey labeling schemes and the proposed method lies in the way structural relationships can be inferred from a label. In the proposed method, for the indexed node, it is necessary to construct the adjacent linked (AL) lists of the structural summary trees (SSTs) of all XML data. An AL list is a data structure that records the linking information of each node, containing the parent, child and level information, and then facilitates the pre-order traversal process. Although its node labeling information is different, the proposed method can also derive an original node path with this information, and thus only needs to know the current node's label. The AL list of the SST in Figure 4 is given in Figure 7 where δ_i[n] denotes the *n*th head node of the *i*th XML document. Each node in the set consists of a 3-tuple (*parent*, *child*, *level*) where *parent* and *child* comprise the relationships of the current node. The *level* is used to demonstrate whether the current node has the proper level information in the SST structure. For example, node T with δ_i [2] in Figure 7: the AL list can provide three pieces of information. First, the parent node of *T* is node *I*; second, the child node of node *T* is <*Nil*>, which means that node *T* has no child, so it belongs to the leaf node. Finally, node *T* is on *level 3* (*L3*) in the SST. For the AL lists of SST, we can label node information by indexing SST, as shown in Figure 7,

and the indexing algorithm for SST of AL lists can be denoted in Figure 8. In Figure 8, indexing of AL lists for SST, we can label each of node information by parent, child and its level. Based on the index, it can eliminate unused nodes and increase search efficiency.

		<parent>;<child>;<level>
$\delta i[0]$	B	<Nil>;<I>;L1
$\delta i[1]$	I	<B>;<T>→<A>;L2
$\delta i[2]$	T	<I>;<Nil>;L3
$\delta i[3]$	A	<I>;<F>→<L>;L3
$\delta i[4]$	F	<A>;<Nil>;L4
$\delta i[5]$	L	<A>;<Nil>;L4

Figure 7: The AL list of the SST from Figure 4.

B

$B_{uild}ALLSST(SST)$

// *AL* :: adjacent-linked list of each *SST*.
// δ_i :: Head Node for AL list.
// *stack* :: store the nodes for *AL* : δ_i .
// Find-Node-Level :: Function return current node level.
// Find-Parent-Child :: Collection of node info includes parent, child.
// Build-3tuple :: Function builds 3-tuple of SST.
// $Node_i$ :: $Node_i$ of parent and child respectively.

1. **for** each *SST* X_i , ←1 **to** *k* **do** //Each *SST* in *AL* :: $\delta_i \sim \delta_k$
2. *Node-Levels* = Find-Node-Level($AL_i : \delta_i$) //Find node level
3. $Node_i$ = Find-Parent-Child($SST_i : \delta_i$) //find node's child & parent
4. Build-3tuple($Node_i$, *Node-Levels*, δ_i : AL list) //build 3-tuple of each node
5. **end**

Figure 8: Indexing of SST for AL lists.

3.2 The Twig-Join Swift for AL lists of SST

The TJFast algorithm utilizes Extended Dewey encoding for each node, and uses the coded information to derive the results. TJFast only stores the leaf node information, and utilizes the Finite state transducer (FST) to convert from the root node to a path pattern and then compare the query path with the node label name to determine whether it can be satisfied. If it matches, as a result of partial matching output until the final result, it will merge all the corresponding results. More importantly, if the query path knows in advance what level to begin with, it will not be necessary to store all paths from the root to the leaf nodes; this will eliminate many unnecessary path nodes, thus offering high satisfaction and more precise query results. As with the query tree pattern (QTP), the origin XML is applied to the TJFast [21]. In contrast to TJFast, the AL lists for SST are requested to the matched twig pattern. The twig query searches for all satisfied query pattern tree structure results from the XML

structure tree. Based on the envisioned procedure, the twig-join swift (TJSwift) algorithm with the indexed SST applied can be denoted as in Figure 9. In line 2 of Figure 9, the leaf nodes of the query tree pattern (QTP) are obtained, and line 4 locates the nodes in the AL list that match the leaf nodes. The *SearchLeaf* () function can find all matched nodes from leaf to root. The purpose of line 5 is to locate the path matching the query tree pattern (QTP) and merge all results in line 7. The *SearchLeaf* function can be denoted as in lines 8–16 of Figure 9.

```
Twig-join swift (QTP, SST)
// QTP :: query tree pattern.
// AL :: adjacent-linked list build from BuildALLSST algorithm.
// ALδi :: δi elements in AL list.
// QTPL :: contain all query tree pattern list information.
// Parse(QTP) :: parse each path of QTP and get leaf node.
// Qi :: contains path and leaf node information in QTPL.
// SearchLeaf() :: function includes two parameter including Qi 's leaf node and AL list intent to return
                   all match node from leaf to root.
// MatchNode :: find all leaf to root nodes results from function SearchLeaf().
// GetPathSolution() :: function intent to return all match path.
// MatchPath :: find match path with all MatchNode results.
// CombineAllPath() :: receive one parameter to merge all match and return results.
// Pathresults :: get all possibly results from function CombineAllPath().
1.    AL = BuildALLSST(SST)  //build AL list structural using BuildALLSST algorithm
2.    QTPL = Parse(QTP)      //parse each path of QTP and get leaf node
3.    while(Qi in QTPL) do      //each Qi contains path and leaf node information
4.      MatchNode = SearchLeaf(Qi's leaf, AL);  //find leaf node in AL and return results
5.      MatchPath = GetPathSolution(Qi's path, MatchNode) //find match path with all MatchNode
6.    end while
7.    Pathresults = CombineAllPath(MatchPath)  //merge all match paths and return results

      function SearchLeaf (Qi, AL)
8.    for each ALδi ←1 to k  do //AL list :: δi ∼ δk
9.      node = GetNode(ALδi)       //get ALδi node
10.     if(node is leaf && node equal to Qi's leaf) then
11.       while(node's parent is not Null) do  //find node's parent until Null
12.         TjNodes.add(node)      //TjNodes store all leaf to root node
13.         node = GetParent(node)
14.       end while
15.     end if
16.   end for
```

Figure 9: Twig-Join Swift Algorithm.

In lines 8 to 16, the algorithm describes each line process shown in Figure 10 (a) - (d). Based on the B/I/T of query pattern, each Figure shows a leaf node of the tree pattern at the left side, and all tree nodes of the AL lists at the right side. In lines 8 to 9, start from the left side node to search each right side node for a match in Figure 10 (a) and (b). When both nodes are the same leaf nodes in line 10, the algorithm then starts from lines 11-14 to find the matched node's parent until a root or beginning node of the query tree pattern is found, as shown in Figure 10 (c). When the process is finished, the output result of path B/I/T can be denoted as in Figure 10 (d).

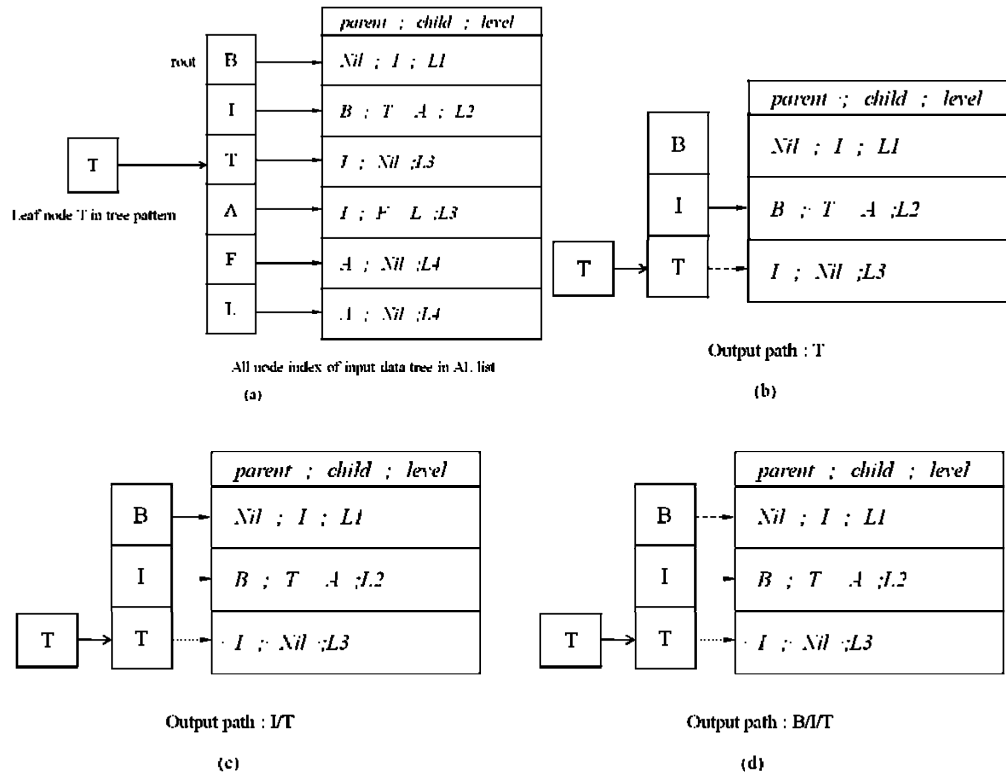


Figure 10: An example describing the search procedure between a leaf node of a tree pattern and an AL list.

When all data have been processed, all the results guarantee the completeness of the output solutions, and the TJSwift then performs the optimization XML tree intended to achieve the best possible minimization, high satisfaction and provide results corresponding to the query matching. For a clear description of the difference between these two approaches, the example query tree pattern (QTP), *B//T^B/I/A/L*, is shown in Figure 11. Nodes *T* and *L* are leaf nodes in the query. First, all information belonging to nodes *T* and *L* are read from the SST (see Figure 4). Since node *T* can direct to parent node *I* and ancestor node *B* according to the AL list, and node *T* is a descendant of *B* (see Figure 7), the first path result *B//T* is output. Subsequently, another leaf node *L* can also be derived from the AL list to output a second path result *B/I/A/L*. Finally, merging the results produces only one query result. In this scenario, TJFast may output path solutions *B//T^B/I/A/L* ten times, which are all repeated results. Note that TJFast would output ten more results than would TJSwift. Because TJFast uses the original XML tree (see Figure 1) to perform the tree pattern query, it outputs the path solution for all nodes in the query, as well as unused nodes. Therefore, TJFast entails more unused nodes than TJSwift does. Even if the available memory is large, TJFast will possibly offer many solution paths that do not contribute to final answers. However, TJSwift can efficiently use available memory to guarantee that each output contributes to final answers.

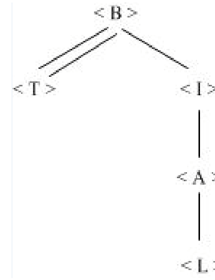


Figure 11: An example for tree pattern query.

4 Experimental Results

4.1. Experimental setup

In order to evaluate the performance of the TJSwift approach compared with that of TwigStack, TwigList and TJFast, here were evaluated in terms of total execution time, scalability and number of elements read, which indicates how many nodes must be read in a matching process. A spectrum of different and well-known datasets was chosen for this evaluation. The first are real dataset, Nasa [23] and DBLP [29], and the second is the benchmark dataset of synthetic data, XMark [22], with a scaling factor of 0.1, which means that 10 megabytes (MB) were downloaded and built with the XML documents. These three datasets were chosen because they have different characteristics. Nasa is a shallow and wide document with many repetitive structures. DBLP is not only have the same features as Nasa but also complex and the document size large than Nasa 16 times. XMark features a moderately complicated and fairly irregular schema, which makes it an ideal dataset for experiment. Table 1 summarizes acronyms and properties of these three datasets used in the experiments, including that the data size refers to the dataset in its plain text format, whereas the number of nodes and the maximum and maximum depth are computed from the tree representation of these datasets. The XMark data was used to analyze execution time concerning the XML document size, and two XMark datasets with scaling factors ranges of 0.5 and 1 (meaning 56.2 and 113 megabytes (MB) for the XML datasets, respectively) were used. The experimental results verify the effectiveness, in terms of accuracy and optimality, of TJSwift as a query tree pattern method for large XML datasets. Furthermore, the combinations of parent-child (PC) and ancestor-descendant (AD) relationships are designed in the queries for TwigStack, TwigList, TJFast and TJSwift.

Table 1: XML Datasets used in the experiments.

	Nasa	DBLP	XMark	
Data size(MB)	23.8	404	56.2	113
Nodes(Million)	0.53	31.88	1.02	2.04
Max depth	8	8	12	12

4.2. Performance Comparison and Analysis

For identify the optimal method for useful applications, the TwigStack, TwigList, and TJFast will

compared with TJSwift emphasis on three aspects in Table 2. First, number of target nodes in QTP means access how many target nodes related to the QTP nodes. Because TJFast and TJSwift process path expression only start from leaves, so except two methods have to access all target node related to the QTP nodes. Second, number of nodes means how many nodes read in XML document, the number of nodes have to be read is minimal in TJSwift, because the method reduce a lot of unnecessary nodes. Finally, amount of intermediate results are insignificant for in TJSwift that preferable than other method.

Table 2: Comparison of four methods in three aspects.

Aspects	TwigStack	TwigList	TJFast	TJSwift
# of target node in QTP	All	All	Leaves	Leaves
# of nodes	Maximum	Maximum	Maximum	Minimum
Intermediate results	Depending on results	Depending on results	Depending on results	Insignificant

Table 3 presents the collection of queries used. The single-path queries of X1, X2 and X3 on the XMark, N1 on the Nasa, and D1 and D2 on the DBLP datasets are designed in Table 3 where / and // denote the PC and AD relations. The other tree queries are varying patterns where denotes the conjunction operator. First, TJSwift and other methods were compared in terms of total execution time and number of nodes read. The total execution time is the time elapsed between the arrival of the query and the delivery of the complete result to the user, and the number of nodes read indicates how many nodes must be read in a matching process. We first choose TJFast as the basis to compare, because TJFast is the fast algorithm for processing twig-pattern matching queries with both PC and AD relations which outperforms TwigStack and TwigList. Figure 12 shows TJFast and TJSwift using the original XMark data, and TJSwift does not have an SST procedure. Although SST is not performed in Figure 12, TJSwift has the advantage of only needing to access children of path elements from the AL lists from those beginning query nodes to the end. In contrast, TJFast has to read all beginning elements. It can be clearly seen from the results that the execution time of TJSwift is 1.3 times lower, on average, than that of TJFast.

Table 3: Designed queries for Nasa, XMark and DBLP datasets.

Name	Query
X1	site/closed_auctions/closed_auction/price
X2	site/regions//item/location
X3	site/people/person/profile/gender
X4	people/person//name^people/person//city
X5	item/location^item/description//keyword
X6	people/person/address/zipcode^people/person/profile/age^people/person/profile/education
N1	textFile/description//footnote//para
N2	revisions//year^revisions//para^revisions//creator
N3	dataset/reference^dataset//keyword^dataset//description/para^dataset//description/heading^dataset/subject
D1	article/title
D2	dblp/inproceedings/booktitle
D3	article//mdate^ article//volume^ article//cite^ article//journal

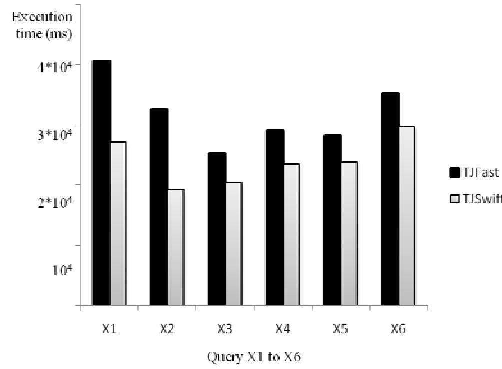
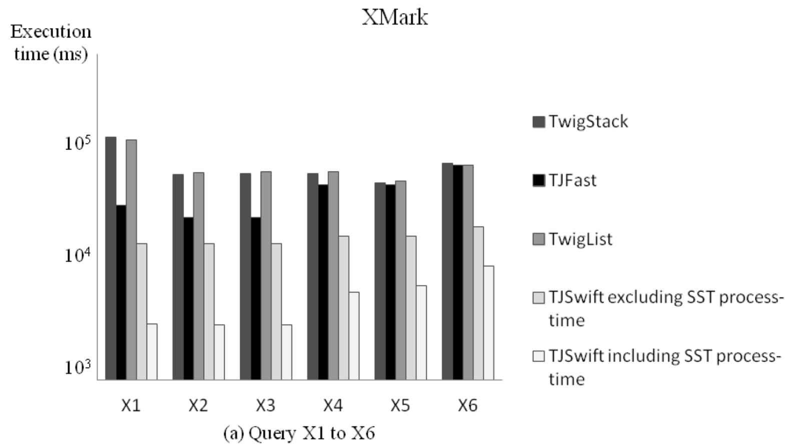


Figure 12: Execution time of TJFast and TJSwift for original XML.



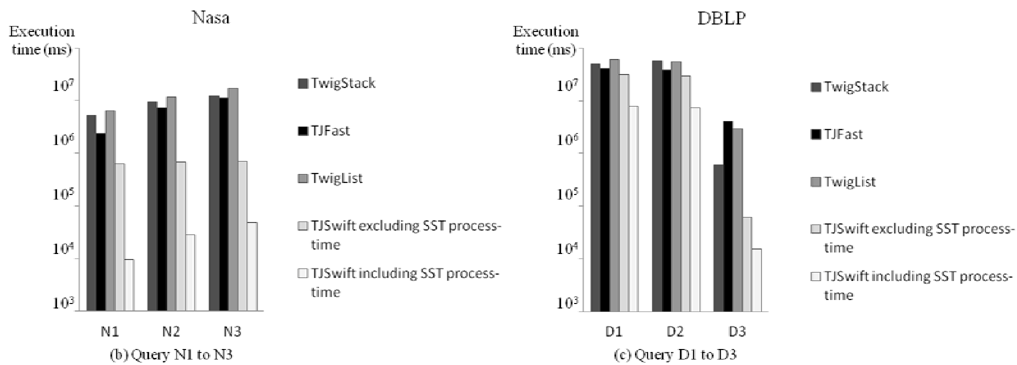


Figure 13: Execution time for queries: (a) X1 to X6 in XMark, (b) N1 to N3 in Nasa and (c) D1 to D3 in DBLP datasets.

Figure 13 shows the all queries in Table 3 with XMark, Nasa and DBLP datasets for TwigStack,

TwigList, TJFast and TJSwift including/excluding SST process-time. Because the SST needs to minimize the XML dataset, the SST process time should be added to TJSwift's performance for an indication of overall execution time. Most remarkably, the execution time of TJSwift is still faster than that of others, even with the SST process time included, as shown in Figure 13. This experiment for DBLP dataset illustrates the efficiency of using a SST in TJSwift. Although DBLP dataset are shallow and that have many nodes, TJSwift only have to read about half of the nodes than the other methods. One of the more interesting is D3; TJFast has a loss over TwigStack and TwigList that because D3 is all AD relationships and the advantage for two methods, but TJSwift still has a gain over them. These results can be inferred from Figure 14. In Figure 14, because the range of the number of nodes read is very large, a logarithmic scale was chosen. The number of nodes read in Figure 14 refers to the difference in the number of nodes between the input data using SST and the original data. Based on the number nodes to be read for the original nodes and the SST nodes, the original data require twice as many nodes, on average, to be read as for SST. From this it can be inferred that the number of nodes will affect the execution time and both of them is direct proportion relationship in the query tree pattern. TJSwift has the advantage of only needing to access a small number of nodes due to the elimination of the nested nodes and duplicates in SST. In contrast, TJFast and other methods have to read all the nodes in the original data.

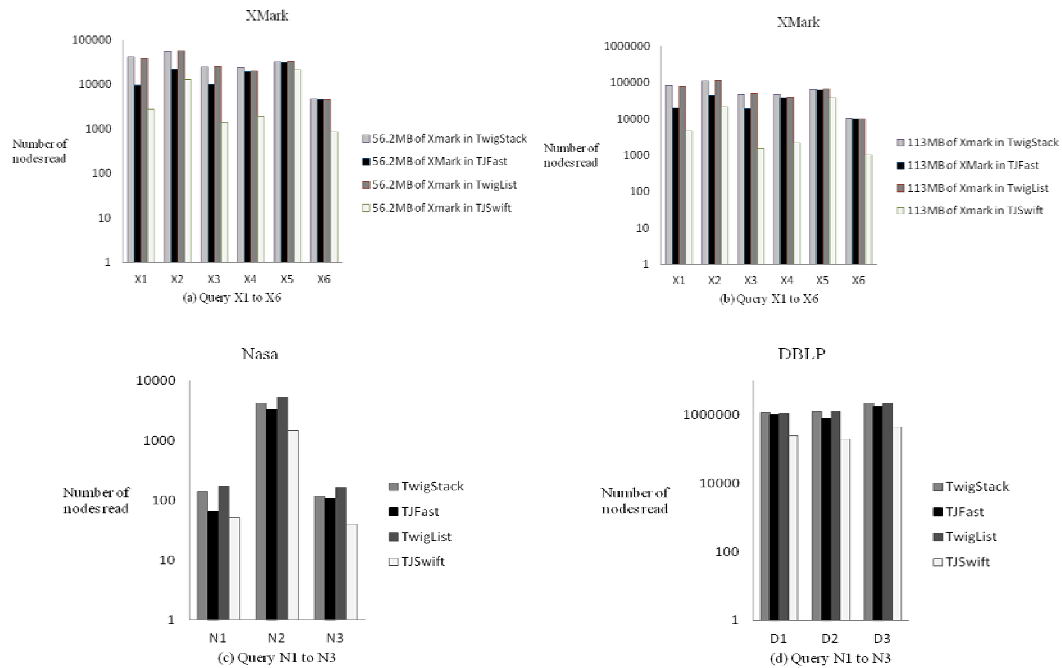


Figure 14: Number of nodes read for queries: (a) and (b) X1 to X6 in XMark in 56MB and 113MB respectively, (c) N1 to N3 in Nasa and (d) D1 to D3 in DBLP datasets.

The time complexity derived only from the TJFast and TJSwift approaches, due to the TJFast outperforms TwigStack and TwigList, and is shown in Figure 14. The TJFast approach described in

[26] enumerates all paths from root to leaf to match twigs in the tree pattern. TJFast [26] defines the time complexity as $O(d^n)$, where d is the number of nodes for the input data tree and n is number of nodes in the query tree pattern. In contrast, the TJSwift with SST procedure to minimize the original tree and generate AL lists corresponds to the node record of all relations; hence, the number of nodes in the input data tree is minimized. Meanwhile, it only enumerates paths from the beginning nodes to leaf nodes in the query pattern. So, we can sum up the complexity analysis of TJSwift and definition in the following.

Given a twig-pattern matching query Q and an XML tree T. Suppose the corresponding Q has N leave nodes as show in Figure 15 (a). The Q leaf nodes N from $L[0]$ to $L[N]$ denote L_n ($L_0, L_1, \ldots, L_n$ Q). The T through SST procedure to AL list into $\delta[n]$ can denote the totally minimized nodes size Md (Md T). In practical application, each L_n corresponds to the total Md of a leave node such that $\{ \sum_{i=0}^{n} L_i \times Md \}$, therefore the complexity $O(Md)^{Ln}$ in worst case. The matching process of the tree pattern depends on depth H, which is the distance from the leaf to the beginning node in Figure 15 (b). The totally worst case of time complexity, $O((Md)^{Ln}+H)$, can be inferred, and the Md will be far less than d because of the minimized origin tree nodes; from this, it can be implied that $O((Md)^{Ln}+H)$ is better than $O(d^n)$.

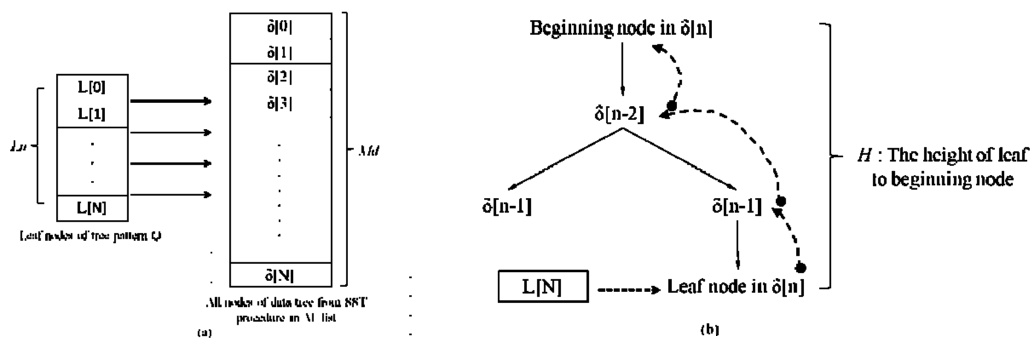


Figure15: Searched procedure: (a) leaf nodes and AL list size of tree pattern (b) finding matched paths from leaf to beginning nodes.

Figure 16 shows the number of matched paths for the query tree pattern in the two XMark data sizes, including 56.2 and 113 MB, Nasa and DBLP datasets. Based on the SST procedure with TJSwift, it is inevitable that there will be fewer matched paths than other methods, i.e., the TJSwift results are the same as those of them, but the difference between the numbers of matched paths is reduced.

The comparison of scalability corresponds to the execution times of TJSwift, TwigStack, TwigList and TJFast in terms of XML document size. For analyzed scalability concerning the XML document size, XMark datasets with scaling factors including 0.003, 0.005, 0.02, 0.04, 0.06, 0.1, 0.5 and 1 for the XML dataset were developed, and query X3 and X6 were used for implementation. Theoretically, the query X6 execution time should be higher than that of X3 because query X6 is more complex than X3. However, the X3 execution time was higher than X6 in Figure 17 (a) and (b) when XML size became large in 0.5 to 1. It can be inferred that even if X3 is a single query, the depth of the X6 query is less

than that of X3. Therefore, the depth of the query path also affects the execution time. It also can be seen in Figure 17 (a) and (b) that the execution times of X3 and X6 for TJFast with other methods scale in 0.06 to 1 rise rapidly with respect to XML document size, but the proposed TJSwift exhibits a far steadier increase in execution time. It can be deduced from the experiment that when the size of an XML document increases, TJSwift showed a more economical execution time than did TJFast. Therefore, the benefits of TJSwift become apparent in comparison to other methods. As shown in the scalability experiment, TJSwift is more efficient than the other methods.

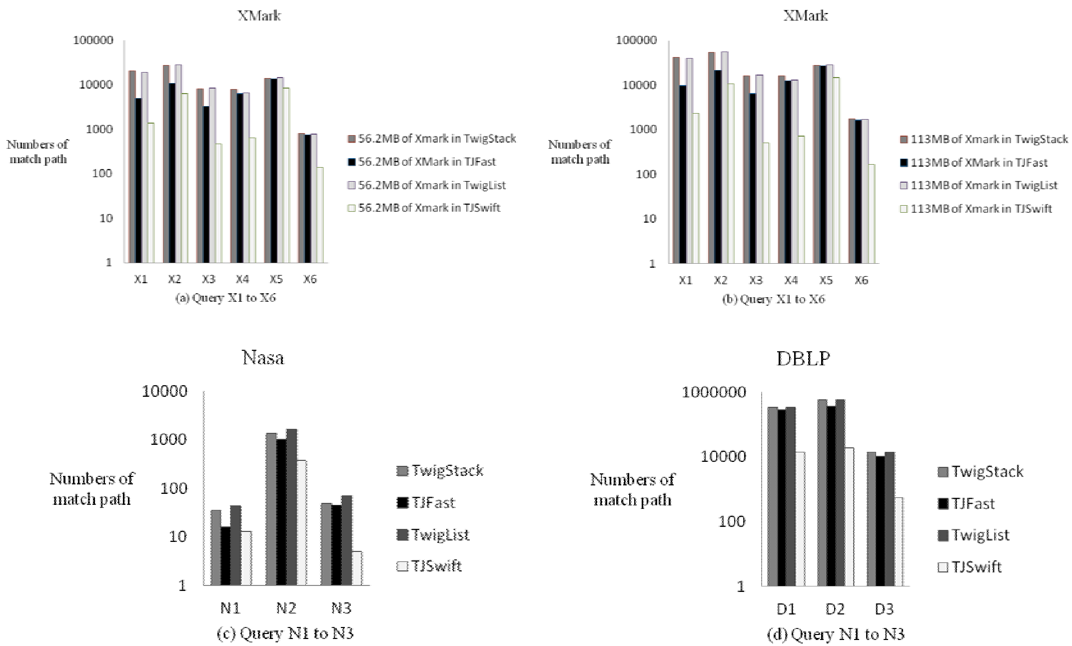


Figure 16: Numbers of matched paths for queries (a) and (b) X1 to X6 in XMark in 56MB and 113MB respectively, (c) N1 to N3 in Nasa and (d) D1 to D3 in DBLP datasets.

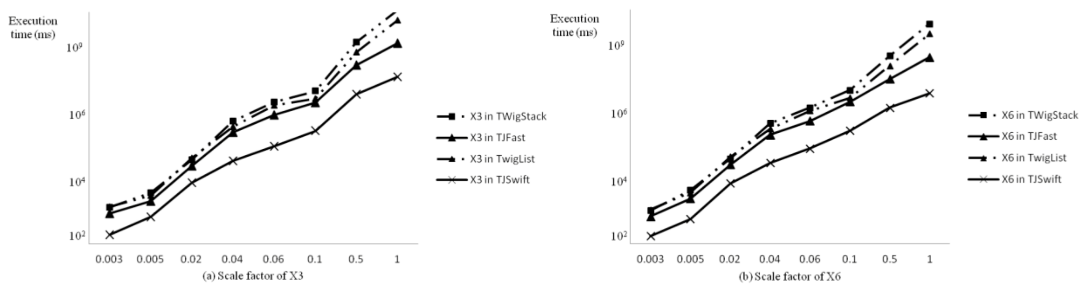


Figure17: Scalability of XML document for variant sizes in queries (a) X3 and (b) X6.

5 Conclusion

XML twig pattern matching is a core issue for XML document query processing. In this paper, a new XML document query method called TJSwift was proposed as a means of providing an efficient and versatile query service. TJSwift uses SST-base as XML data description features and association with adjacent linked (AL) list.

The TJSwift method can preserve both structure and semantic information, as well as perform the optimization XML tree intended to achieve the best possible minimization, high satisfaction and provide results corresponding to the query matching. Experimental results show that TJSwift is much more efficient than current twig pattern matching algorithms (TwigStack, TwigList and TJFast), and can answer most queries with acceptable response time and accuracy.

References

1. World Wide Web Consortium. The document object model. http://www.w3.org/DOM/
2. A. Berglund, S. Boag, and D. Chamberlin, XML Path Language (XPath) 2.0, W3C Recommendation, http://www.w3.org/TR/xpath20/, January 2007.
3. J. Robie and R. Hat, IEEE Internet Computing, "XML Processing and Data Integration with XQuery", vol. 11 no. 4, pp. 62-67, August 2007.
4. T. Dalamagas et al., "Clustering XML Documents using Structural Summaries", EDBT Work-shop on Clustering Information over the Web (ClustWeb04), Heraklion, Greece, 2004.
5. A. Nierman and H. V. Jagadish, "Evaluating Structural Similarity in XML Documents", Fifth International Workshop on the Web and Databases (WebDB 2002), 2002.
6. S. Flesca et al., "Fast Detection of XML Structural Similarity", IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 2, pp. 160-175, February 2004.
7. W. Lian et al., "An Efficient and Scalable Algorithm for Clustering XML Documents by Structure", IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 1, pp. 82-96, January 2004.
8. M. Kozielski, "Improving the Results and Performance of Clustering Bit-encoded XML Documents", Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06), 2006.
9. J. S. Yuan, X. Y. Li and L. N. Ma, "An Improved XML Document Clustering Using Path Feature", vol. 2, pp.400-404, 2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery, 2008.
10. H. P. Leung, F. L. Chung, S. C. F. Chan and R. Luk, "XML Document Clustering Using Common Xpath", Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration, Tokyo, pp. 91-96, April 2005.
11. A. Termier, M. C. Rousset and M. Sebag, "treefinder: a first step towards XML data mining", Proceedings of IEEE International Conference on Data Mining, Maebashi, pp. 450-457, December 2002.
12. J. Yang, W. K. Cheung and X. Chen, "Learning the Kernel Matrix for XML Document Clustering", Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), Hong Kong, pp. 353-358, April 2005.
13. J. Liu, J. T. L. Wang, W. Hsu and K. G. Herbert, "XML Clustering by Principal Component Analysis", Proceedings. Of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04), Boca Raton, pp. 658-662, November 2004.
14. J. W. Lee, K. Lee and W. Kim, "Preparation For Semantic-Based XML Mining", The 2001 IEEE International Conference on Data Mining, San Jose, pp. 345-352, November 2001.

15. M. H. Qureshi and M. H. Samadzadeh, "Determining the Complexity of XML Documents", Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05), vol. 02, pp. 416 - 421, 2005.
16. X. Y. Li, "Using Clustering Technology to Improve XML Semantic Search", Proceedings of the Seventh International Conference on Machine Learning and Cybernetics, vol. 5, pp. 2635-2639, July 2008.
17. B. Nicolas, K. Nick and S. Divesh, "Holistic Twig joins: Optimal XML pattern matching", In Proceedings of the SIGMOD Conference, 2002.
18. S. Chen, H. G. Li, J. Tatemura, W. P. Hsiung, D. Agrawal and K. S. Candan, "Twig2Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents", In Proceedings of the VLDB Conference, pp. 283-294, September 12-15, 2007.
19. Jiang H., Wang W., Lu H., Yu J. X., "Holistic twig joins on indexed XML documents", Proceedings of the VLDB Conference, pp. 273 – 284, 2003.
20. J. Lu, T. W. Ling, Z. Bao and C. Wang, "Extended XML Tree Pattern Matching: Theories and Algorithms," IEEE Trans. Knowledge and Data Eng., vol. 23, no. 3, pp. 402-416, Mar 2011.
21. J. Lu, T. W. Ling, C. Y. Chan and T. Chen, "From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching", In Proceedings of VLDB Conference, pp. 193-204, Norway 2005.
22. R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt and F. Waas, XMark an XML benchmark project. http://monetdb.cwi.nl/xml/index.html
23. University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets/
24. X. Wu and G. Liu, "XML Twig Pattern Matching Using Version Tree", Data and Knowledge Eng., vol. 64, no. 3, pp. 580-599, 2008.
25. S. K. Izadi, T. Harder, and M. S. Haghjoo, "S^3: Evaluation of Tree-Pattern Queries supported by Structural Summaries," Data and Knowledge Eng., vol. 68, no 1, pp. 126-145, 2009.
26. M. Hachicha and J. Darmont, "A Survey of XML Tree Patterns", IEEE transactions on knowledge and Data Engineering, vol. 25, no. 1, January 2013.
27. H. K. Chang, K. C. Hung, I. C. Jou, "Efficient XML Retrieval Service with Complete Path Representation" IEICE Transactions on Information and Systems, vol.E96-D, no. 4, pp. 906-917, 2013.
28. L. Qin, J. Xu Yu, B. Ding, TwigList: make twig pattern matching fast, in: Proceedings of the DASFAA Conference, pp. 850 – 862, 2007.
29. M. Ley, DBLP Computer Science Bibliography. http://dblp.uni-trier.de/xml/dblp.xml.