TWIX: Twig Structure and Content Matching of Selective Queries using Binary Labeling

S. Alireza Aghili Hua-Gang Li Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California-Santa Barbara
Santa Barbara, CA 93106
EMail: {aghili,huagang,agrawal,amr}@cs.ucsb.edu

***Abstract*— XML queries specify predicates on the content and the structure of the elements of tree-structured XML documents. Hence, discovering the occurrences of twig (tree structure) query patterns is a core operation for XML query processing. In this paper, we propose a novel technique for matching XML twig query patterns, named TWIX, which results in a substantial reduction of the search space, response time, size and structure invariance through a distributed binary labeling and tree traversal algorithm. Furthermore, TWIX benefits from an interactive graphical user interface for twig query matching.**

I. INTRODUCTION

The *rich content* and the *semi-structuredness* of XML documents demands efficient support for complex yet declarative queries. XML documents can be viewed as ordered tree structures where each tree node corresponds to document ELEMENTS (or ATTRIBUTES) and edges represent direct (element→sub-element) *relationships*. Structured queries on such ordered trees specify complex selection predicates on element *labels* (keyword search) and the *relationships* among them (structure pattern search). In general, structural relationship queries may be categorized in two different classes: $i)$ *path expression*, and $ii)$ *twig* queries. In such queries, single slash and double slash edge notations are used to require the presence of a *Parent-Child (PC)* or simply an *Ancestor-Descendant (AD)* relationship among the nodes, respectively. For instance, the query ***Q1*** = /dblp//inproceedings[/author = 'Abiteboul'] is a path expression query that matches all the proceedings $i)$ written by the author 'Abiteboul', for which $ii)$ the binary structural component (inproceedings/author) has a PC relationship, while (dblp//inproceedings) element pair imposes an AD relationship. Similarly, the twig query ***Q2*** = /article[./author = 'Abiteboul' AND /title = 'XML'] seeks the articles written by the author Abiteboul, having the term 'XML' in their titles.

The initial structural search techniques [2], [3] proposed processing the twig queries in the following order: (i) decompose the twig query structure into its binary path expression components, (ii) perform a semi top-down inspection of the XML document tree for each decomposed path component, (iii) join the potential matching instances to each of the query's binary components to form the answer to the original twig query.

The top-down traversal of the document tree results in scanning a large number of path combinations. For instance, the root node of the dblp[10] database has 3,288,858 immediate children (as of September 2005) which makes it impossible to inspect all the path combinations specially when the given query includes the root node. Despite optimizations [6], [13], it is still inevitable that a large number of binary path combinations have to be examined. For instance, the inspection of dblp database for query $Q1$ reveals 212,273 instances of (dblp/inproceedings), 491,783 of (inproceedings/author), and 150 of author instances containing the term 'Abiteboul'. Bruno et al. [6] proposed TwigStack, that uses a chain of linked stacks to compactly represent the intermediate path expression results and subsequently joins them to obtain the query twig pattern match. Lu et al. [13] extend TwigStack for optimality when PC edges are used as well. Moreover, various *numbering schemes* have been proposed [2], [6], [8], [12] which associate interval encoding with every node, based on the document order, to help identify PC or AD relationships among the nodes. For instance, each label may consist of (*start*, *end*, *level*) values for each node, acquired from the *preorder* traversal of the document, which is used to mainly determine the vertical structural relationships among the individual nodes. Alternatively, several *Indexing methods* [7], [8], [9], [12], [17] have also been proposed to index the elements and attributes of XML documents. Most of these methods represent intervals as points in a multidimensional space and utilize available indexing techniques, such as B^+-trees and R-trees, on element sets to store and query these points. Recently, several proposals have been made to encode structural information to improve the efficiency of twig query evaluation. Xu et al. [19] incorporates the notion of Lowest Common Ancestor (LCA) for a more efficient keyword search. Li et al. [11] propose Meaningful Lowest Common Ancestor (MLCAS)-embedded XQuery that provides the user with the full expressive power of XQuery while eliminating the need for the user's knowledge of the underlying schema specifications. Most similarly, Lu et al. [14] propose the use of a new labeling scheme named *extended Dewey* that facilitates access to the labels of each node of a path expression, by inspecting its leaf label alone. We argue that the *horizontal (proximity)* of the nodes may further be incorporated to help identify the relevance of the nodes to each other. Moreover, the preorder and level information associated with the nodes, and hence the

information derived from the node labels, is very *limited* and hence not capable of supporting extended functionalities. For instance, given the labels of any two nodes, the computation of their *Nearest Common Ancestor (NCA)* requires additional computations by traversing through the parental labels of the document tree nodes.

This paper proposes the TWIX technique, which shares similar ideas with structural-based twig processing methods, but further utilizes the statistical and positional characteristics of the document tree nodes. TWIX, in particular, exploits the common optimization technique of pushing down the `selection` operation down to the leaves of the query plan tree where the ***selectivity*** of the nodes is the most. Consider the query $Q1$, it is clear that $selectivity$(`Abiteboul`) $\geq$ $selectivity$(`author`), and $selectivity$(`Abiteboul`) $\geq$ $selectivity$(`inproceedings`). Hence, for any given path expression, TWIX inspects a compact inverted index for the occurrences of the query's leaves in the document and deploys an efficient bottom-up scheme which builds the twig subtree answers on top of the matched leaves. As a result, the internal nodes are *only* checked for the *final* twig candidates. Moreover, considering $Q2$ where $selectivity$(`'Abiteboul'`) = 150 and $selectivity$(`'XML'`) = 980, there are only 22 matching articles satisfying both of the query's predicates. It is clear that the `'Abiteboul'` and `'XML'` instances within each of those 22 answers, reside within the same `/article` subtree node. Hence, TWIX further utilizes the preorder values to maintain the ***horizontal proximity*** of the matching nodes, in order to focus only on those matching leaf instances which are potentially the attributes of a common object. This observation would narrow the scope of the search to only those leaf matching instances that are "*close enough*" to each other. Furthermore, we note that the information encoded within the traditional labels *limits* the functionalities that can be efficiently supported on the element nodes of the document tree. TWIX, therefore, uses a ***richer labeling scheme*** [4] that efficiently infer additional information regarding the structural relationships among the nodes, particularly finding the *Nearest Common Ancestor (NCA)* of the nodes, with *minimal* computational and space overhead. A fully functional $java$-based implementation of TWIX interactive system and a graphical user interface (GUI) has been developed, which allows the users to enter queries graphically using a tree data model [1]. TWIX implementation allows us to clearly demonstrate, on real large scale datasets, the effectiveness of the TWIX approach for processing twig queries.

The rest of the paper is organized as follows: Section 2 provides an overview of the TWIX procedure and each component of the proposed technique. Section 3 discusses the empirical performance analysis followed by Section 4 which concludes the work.

II. The TWIX Procedure

In this section, we describe the primary components of TWIX and provide in-depth descriptions and analysis of each individual unit. Figure 1 depicts a general overview of the TWIX procedure, which consists of an *offline* pre-processing and an *online* phase, described in the following sections. Furthermore, an algorithmic description of TWIX is provided in Figure 3.

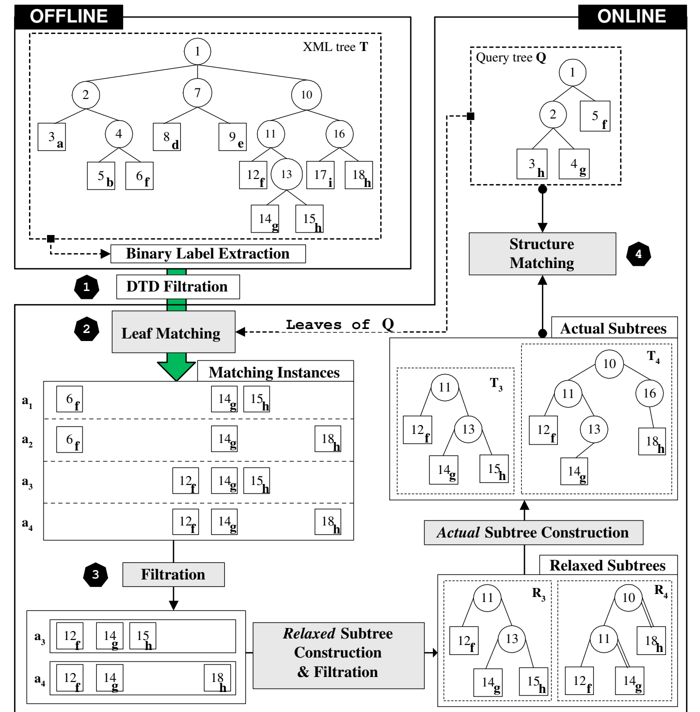
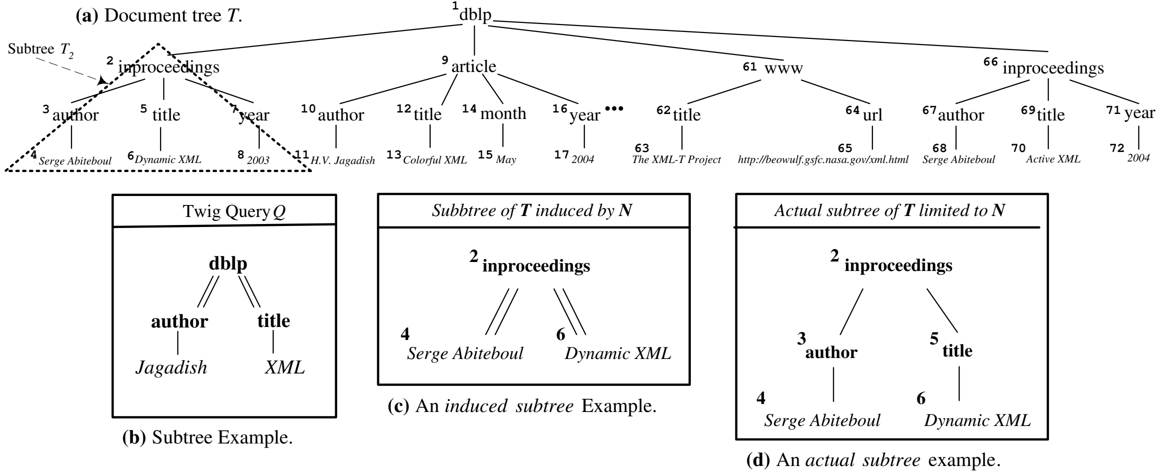


Fig. 1. TWIX procedure: a general overview.

▷**Basics.** Let $T = (V, E)$ denote an ordered XML document tree, where V and E represent the set of nodes and edges, respectively. Each internal node has a *tag* (e.g., `author`) and each leaf node has a *value* which corresponds to the tag of its parent (e.g., `'Serge Abiteboul'`) which is not necessarily unique across T (e.g., the term `'John'` may appear in the value contents of an *author* or a *title*).

A. Offline Pre-processing Phase

The offline pre-processing phase is designed to calculate and maintain the additional information needed to facilitate more efficient processing of the query in the online phase. It consists of three main components: 1) Preorder assignment, 2) Binary label assignment, and 3) Inverted index. Given a document tree T, TWIX augments each node with a preorder rank and a binary label. The document tree T is also indexed using an inverted index on all the leaves of T. The preorder traversal is performed on T and each node is associated with its unique *preorder traversal rank*. The preorder values of the nodes in a document tree impose a logical document order. Consider the sample bibliography XML document tree T of Figure 2(a), the *tags* of the internal nodes with the preorder ranks 3 and 12, named n_3 and n_{12}, are `author` and `title`. Moreover, the



(a) Document tree T.

(b) Subtree Example.

(c) An *induced subtree* Example.

(d) An *actual subtree* example.

Fig. 2. (a) An XML document tree T from DBLP, (b) A sample twig query Q, and the (c) actual and (d) induced subtrees restricted to the leaf nodes $N = \{n_4, n_6\}$.

values of the leaf nodes with the preorder ranks 4 and 13, named n_4 and n_{13}, are `'Serge Abiteboul'` and `'Colorful XML'`, respectively, which are the value contents related to the nodes n_3 and n_{12}. Using the preorder ranks of the nodes, each node is further assigned a unique $O(log\ n)$-bit *binary label*, proposed by Alstrup et al. [4], which can later be used to efficiently determine the nearest common ancestor of any two nodes in *constant time*. The details of the binary labeling algorithm can be found in [4]. Finally, an inverted index Υ is constructed on the leaf keywords of T to help efficiently locate the occurrences of any given keyword in the document tree. For each keyword k_i, there exists an entry $\Upsilon_{h(k_i)}$ in the inverted index corresponding to the hashed value $h(k_i)$ and also a link to the preorder rank list of the occurrences of k_i in T, sorted on the preorder values of the instances. For instance, the inverted index Υ for the tree T of Figure 2(a) would have one entry per unique leaf keyword of T. The entry for the term `'Serge'`, $\Upsilon_{h(Serge)}$, points to the sorted list $[n_4 \rightarrow n_{68} \rightarrow null]$. Given a leaf keyword l_i of the query (further discussed in the online phase), its corresponding entry in Υ is quickly located, where all the occurrences of l_i in T may be found by simply traversing the nodes in the associated link list at $\Upsilon_{h(l_i)}$.

B. Online Query Processing Phase

The online phase of TWIX performs the query processing phase of the system and is sequentially organized in the following four main stages: 1) DTD filtration, 2) Leaf matching, 3) Horizontal & Vertical filtration, and 4) Structure matching:

▷ **DTD Filtration of Documents.** An XML document repository may have multiple XML document sets with different schema, defined in their corresponding (and unique) DTD (Document Type Definition). It is clear that the answers to a twig query Q should reside in an XML document whose DTD potentially supports the structure imposed by Q. TWIX, therefore, starts with a *document filtration* procedure to eliminate those XML documents whose DTD does not match the given query. For instance, given a query Q = `/dblp//article[/author = 'Abiteboul']`, all the PC and AD edges of Q are extracted ($edge_1$ = [`dblp//article`] and $edge_2$ = [`article/author`]) and the search space is limited to only those XML documents whose DTD schema includes both $edge_1$ and $edge_2$ edges.

▷ **Leaf Matching.** Given a twig query Q, we are interested in identifying all its possible matches in those document trees whose DTD match with Q. Our bottom-up *leaf matching phase* approach starts by first trying to identify all leaf sets in T that match the leaves of Q, in the correct order. We therefore define the notions of leaf matching and the matching set, as follows.

Definition 1: Given a keyword s and a tree T, the **leaf matching** of s in T is $e(s,T)$, the set of all the leaf nodes of T, whose value contains the keyword s.

Example. Let T_L denote the set of leaves of T, $T_L = \{n_4, n_6, n_8, n_{11}, n_{13}, n_{15}, n_{17}, n_{63}, n_{65}, n_{68}, n_{70}, n_{72}\}$ as depicted in Figure 2(a). Given s =`'Abiteboul'`: $e(s,T)$ = $\{n_4, n_{68}\}$ = {`Serge Abiteboul`$^{(4)}$, `Serge Abiteboul`$^{(68)}$}. Moreover, given s' = `'Color'`: $e(s',T)$ = $\{n_{13}\}$ = {`Colorful XML`$^{(13)}$}.

Definition 2: Given a set of keywords $S = \{s_1, \ldots, s_m\}$, the **leaf matching set** of S on T, $E(S,T)$, is the set of all the **leaf matchings** $e(s_i,T)$. More formally, $E(S,T) = \{e_1, \ldots, e_m\}$, where $e_i = e(s_i,T)$ denotes the leaf matching of s_i in T. The nodes of each set e_i are ordered in increasing order of their corresponding preorder rank.

Hence, given the term (keyword) s_i, which corresponds to

Offline pre-processing phase, Given an XML document tree T:

1) Perform a *preorder traversal* on tree T and associate a *preorder traversal rank* to each node of T.
2) Perform the *binary labeling* procedure and attach a *binary label* to each node of T.

Online phase, Given twig query pattern Q with the set of leaves Q_L:

Notations:

$\triangleright$ $S = \{s_1, \ldots, s_m\}$: set of all the keywords of the query's leaves ($m \geq |Q_L|$).

$\triangleright$ $e(s_i, T) = e_i$: Leaf matching of s_i in T.

$\triangleright$ $a_j = (t_1, \ldots, t_m)$: a matching instance tuple, where $1 \leq j \leq \beta = \Pi_{i=1}^{m} |e_i|$.

$\triangleright$ r_u: preorder traversal rank of the node u.

$\triangleright$ τ: horizontal distance threshold.

- **[DTD Filtration]**: Filter the documents whose DTD does not embrace queries edges.

- **[Horizontal and Vertical Filtration]**
While progressively producing the a_j tuples:
```
while j ≤ β: do
  for i = 1 to m − 1: do
    if ( NCA(t_i, t_{i+1}) == root ||   //vert. distance
         |r_{t_i} - r_{t_{i+1}}| ≥ τ )   //horizontal distance
       − Prune a_j from the processing queue.
```

$\triangleright$ $\{a_{\pi_1}, \ldots, a_{\pi_s}\}$: the set of all matching instance tuples which are not yet pruned.

- **[Induced Subtree Structure Matching]**
While progressively constructing the induced subtrees of T for the leaf pairs of each $a_{\pi_p} = (t_1, \ldots, t_m)$ tuple:
```
for each i = 1 to m − 1: do
  if ( NCA(t_i, t_{i+1}) != NCA(s_i, s_{i+1}) ) ⟹ Prune a_{π_p}
from further processing.
```

- **[Full Structure Matching]**
For each of the remaining induced subtrees of the non-pruned matching instances ($a_{\pi_1} \ldots a_{\pi_s}$), construct their *actual* induced subtrees as $T_{a_{\pi_1}} \ldots T_{a_{\pi_s}}$.
```
for each j = π_1 to π_s: do
 - Match the structure of T_{a_{π_j}} against twig query
pattern Q.
```

Fig. 3. TWIX procedure formulation.

a leaf in query Q, e_i denotes the corresponding list linked to the entry $\Upsilon_{h(s_i)}$ of the inverted index Υ of all the occurrences of s_i in T.

Example. Given the set of a query's leaf keywords $S = \{s_1, s_2\} = \{$`'Abiteboul', 'color'`$\}$ results in $E(S,T) = \{e_1, e_2\}$ where $e_1 = \{n_4, n_{68}\}$ = $\{$`Serge Abiteboul`$^{(4)}$, `Serge Abiteboul`$^{(68)}\}$, and $e_2 = \{n_{13}\} = \{$ `Colorful XML`$^{(13)}\}$. From the query processing point of view, $e_1 = \Upsilon_{h(s_1)} : [n_4 \rightarrow n_{68} \rightarrow null]$, and similarly $e_2 = \Upsilon_{h(s_2)} : [n_{13} \rightarrow null]$.

Definition 3: Given a set of keywords $S = \{s_1, \ldots, s_m\}$, a **matching instance** of S in $E(S,T)$ is defined as a unique m-ary tuple $a = [t_1, \ldots, t_m]$ such that $t_i \in e_i$ for $1 \leq i \leq m$. The **tuple matching set**, $M(S,T)$, denotes the set of all the possible unique m-ary matching instance tuples of S on T, where $|M(S,T)| = \Pi_{i=1}^{m} |e_i|$.

Example. Given S=$\{$`'Abiteboul','color'`$\}$: M(S,T) = $\{[n_4, n_{13}], [n_{68}, n_{13}]\}$ = $\{$[`Serge Abiteboul`$^{(4)}$, `Colorful XML`$^{(13)}$],[`Serge Abiteboul`$^{(68)}$, `Colorful XML`$^{(13)}$]$\}$.

Note that, the *tuple matching set* is the set of all the possible candidates for the query's leaf keyword combinations. A naïve approach would generate all such pairs and perform the necessary inspection steps to extract the correct answers. However, such an approach is not practical since the size of the tuple matching set can become quite large. Hence, it is essential to incorporate an online filtration technique at the matching instance pair generation step in order not to generate irrelevant matching candidate pairs.

$\triangleright$ **Filtration Techniques.** The horizontal and vertical distances between nodes of the matching instances are used as a measure to assess the relevance of the matching instance. TWIX applies the filtration techniques in the following order *1) Horizontal*, followed by *2) Vertical* filtration. The filtration step is later followed by the *Structure matching* phase. The horizontal filtration is based on the *proximity* and the *preorder ranks* of the nodes, while the vertical filtration uses the *binary labels* of the nodes. As mentioned earlier, the *preorder ranks* of the nodes impose a logical order on the document tree. However, the combination of both techniques is essential since there are cases which are exclusively captured only by one of them.

The proposed filtration techniques are specifically tailored to the class of XML documents such as DBLP [10], where the ***node context*** (*individual meaningful entities*) starts at the immediate children level of the root (e.g., `article`). For instance, in Figure 2(a) the node contexts are the subtrees rooted at the immediate children of the root, such as `inproceedings`, `article`, and `www`. In other words, all the nodes in the subtree T_2 provide details of the same entity pertaining to a conference proceeding paper, with `author = 'Serge Abiteboul'`, `title = 'Dynamic XML'`, and `year = '2003'`. Similarly, nodes belonging to different node contexts detail the information on different entities. Such an assumption may be adaptively adjusted and pushed further down the tree using the knowledge of the general structure (DTD) of the underlying data.

• *Horizontal Filtration Technique (HFT).:* The number of potential matching instances, $|M(S,T)|$, is a function of (i) the total number of *leaf matching sets* $|E(S,T)|$, and (ii) the size of each *leaf matching* $|e_i|$. The execution of a keyword search to find the matches to the leaves of Q, $Q_L = \{$`'Abiteboul', 'XML'`$\}$ results in e(`'Abiteboul'`,dblp) = 150, and e(`'XML'`,dblp) = 957 instances. The total number of matching instances in the worst case is $|M(S, dblp)| = 150 \times 957 = 143,550$, which shows the quadratic size of the intermediate results set and the inevitable need for efficient filtration techniques. TWIX incorporates a merge-join matching instance enumeration using horizontal filtration to make sure that the number of the processed matching instances is not quadratic. This procedure is explained using an example.

Motivating Example. Consider the query Q and the document tree T of Figure 2(a), where $Q_L = \{s_1, s_2\} = \{$

'Jagadish', 'XML'$\}$ and $e(s_1,T)$ = $\{n_{11}\}$ and $e(s_2,T)$ = $\{$ n_6, n_{13}, n_{63}, n_{65}, $n_{70}\}$. A naïve enumeration of the matching instance set results in $E(Q_L,T)$ = $\{[n_{11}, n_6]$, $[n_{11}, n_{13}]$, $[n_{11}, n_{63}]$, $[n_{11}, n_{65}]$, $[n_{11}, n_{70}]\}$ consisting of 5 instances which is quadratic in size. However, it is clear that the matching instances beyond $[n_{11}, n_{13}]$ are definitely irrelevant because the nodes n_{63}, n_{65}, and n_{70} are located too far from their matching counterpart n_{11} and hence correspond to different node contexts: T_9 versus T_{61} and T_{66}. To tackle this problem, we have deployed a Sort-Merge-Join (SMJ) algorithm which, due to the need for some preliminary formalizations, is explained at the end of this section.

Property 1: Let "root" denote the root node of tree T where $|T| = n$, and let CHILDREN(root) = $\{c_1, \ldots, c_m\}$ represent the immediate children (in the left-to-right sibling order) of the root node. Given any node c_i, its preorder traversal rank r_i can be computed as,

$$r_i = |T_{c_{i-1}}| + r_{i-1}, \quad for\ 1 \leq i \leq m,$$

where r_{i-1} and $|T_{c_{i-1}}|$ denote the preorder traversal rank of the node c_{i-1}, and the size of the subtree rooted at c_{i-1}, respectively.

For instance, given the tree T of Figure 2(a) and CHILDREN($root$) = $\{c_1, c_2, \ldots, c_m\}$ = $\{$inproceedings$^{(2)}$, article$^{(9)}$, ..., inproceedings$^{(66)}\}$. The corresponding preorder ranks and subtree sizes of the immediate children of the root are $\{r_1 = 2, r_2 = 9, \ldots, r_m = 66\}$ and $\{|T_{c_1}| = 7, |T_{c_2}| = 9, \ldots, |T_{c_m}| = 7\}$, respectively. The rank of a node c_i is equal to $\{$the rank of node $c_{i-1}\}$ + $\{$the size of the subtree rooted at node $c_{i-1}\}$.

Definition 4: **(Horizontal distance threshold).** Given a tree T, the horizontal distance threshold, τ, is defined as:

$$\tau = max\ \{|T_{c_i}|\ |\forall c_i \in CHILDREN(R)\}.$$

Inspecting Figure 2(a), the subtree rooted at node c_2 = article$^{(9)}$has the maximum size and hence: τ = $|T_{c_2}| = 9$.

Proposition 2: **(Horizontal distance bound).** Let u and v denote two nodes in tree T, where $u \in T_{c_i}$ and $v \in T_{c_j}$:

$$|r_u - r_v| > \tau \Longrightarrow i \neq j,\ for\ 1 \leq i, j \leq m.$$

For instance, for the subtree rooted at the node inproceedings$^{(2)}$, T_2, the node with the minimum preorder is the root node of T_2 which has the preorder r = 2, and the node with the maximum preorder is the lowest-right node of T_2 having the preorder $|T_2| + r - 1$ = 7 + 2 -1 = 8. The difference between the preorder ranks of nodes inproceedings$^{(2)}$ and 2003$^{(8)}$ can not be larger than $|T_2|$ = 7. As a result, τ denotes the maximum difference between the preorder ranks of any two nodes that belong to the same subtree or context. Hence, if the preorder traversal rank of any two given nodes is more than the horizontal threshold τ, then it is guaranteed that they do not belong to the same subtree and hence do not entail the same entity. In the dblp example of Figure 2(a), τ = max $\{7, 9, \ldots, 7\}$ = 9 and applying the *horizontal distance bound*, the matching instance pairs $[n_{68}, n_6]$ and $[n_{68}, n_{13}]$ will be pruned from further considerations, i.e., $|r_{n_{68}} - r_{n_{13}}| = |68 - 13| = 55 > \tau = 9$. The threshold constant for each document tree is determined as an offline profile.

Sort-Merge-Join (SMJ). matching instance pair generation: It is very important to note that the SMJ generation of the matching instance pairs and the horizontal filtration are performed in a single phase. Given the query Q and the document tree T, where Q_L = $\{s_1,\ s_2\}$ from the "motivating example" of the beginning of this section. Let M = $e(s_1,T)$ and P = $e(s_2,T)$ be sorted on the preorder rank values of their respective nodes. The algorithm generates the qualifying matching instance pairs [m,p], for $m \in M$ and $p \in P$, by essentially merging the two sorted lists where the merging criteria requires the preorder ranks of the nodes m and p to be below the threshold τ, as calculated above. Therefore, the enumeration of the list for a given leaf keyword is continued as long as *merge criteria* $|preorder(m) - preorder(n)| \leq \tau$ holds. Hence, the generation/enumeration of the matching instances would stop after inspecting the matching instance $[n_{11}, n_{63}]$. For a given node n_{11}, we call the list $\Delta(n_{11})$ = [n_{63}, n_{65}, n_{70}], as the *Black List* of n_{11}.Once the sorted matching instances for the given keywords are accessed, the SMJ algorithm incorporates the τ threshold for the merge criteria. This guarantees that, for a given $m \in M$, nodes of those instances of P beyond the first element of the black list $\Delta(m)$ are never considered. The combination of HFT with SMJ avoids the exponential size of the intermediate result set. The cost of merging M and P is O $(M + N)$ considering that both M and P are already sorted in the offline phase. Moreover, more sophisticated merging techniques, such as additionally using clustered index on M and/or P, may be used to further boost the performance. These result are easily extended to the case where $|Q_L| \geq 2$ using transitivity. Following property generalizes the above arguments.

Property 3: Given query Q, document tree T and Q_L = $\{s_1,\ \ldots,\ s_K\}$. For any matching instance m $\in$ e(s_i,T), all the matching instances of Δ(m) $\in$ e(s_{i+1},T) are pruned from further consideration, where $1 \leq$ i $\leq K - 1$.

- *Vertical Filtration Technique (VFT):* Vertical filtration is used for further filtration of leaf matching instances, subsequent to the HFT. VFT is inspired by the fact that the NCA of any two given nodes u, v belonging to the same context (subtree) should be a node other than the root node. In Figure 2(a), nodes n_4 and n_6 belong to the same context node (inproceedings$^{(2)}$, which is the root node of the subtree T_2). In contrast, the nodes n_8 and n_{11} belong to different contexts (subtrees T_2 versus T_9) and as a result their nearest common ancestor is the *root* node, that is, NCA(n_4, n_6) = n_2 = inproceedings$^{(2)}$ and NCA(n_8, n_{11}) = $root$ = dblp$^{(1)}$.

However, we have further extended this notion to the context node instead of the root. For instance, given two query leaf nodes u and v with the NCA node w, as imposed by the query, we filter out all the instance pairs of (u, v) whose NCA is a node other than w. This feature is captured in the following property formulation.

Property 4: Let u and v denote two leaf nodes in the document tree T such that $u \in T_{c_i}$ and $v \in T_{c_j}$, where c_i and c_j denote the roots of the entity subtrees containing u and v, respectively.

$$NCA(u,v) = \begin{cases} c_i = c_j & \text{u, v belong to the same entity;} \\ otherwise & \text{u, v belong to different entities.} \end{cases}$$

Unfortunately, HFT can not eliminate such matching instance pairs, since nodes may be arbitrarily close in their preorder rank, and still have the root as their NCA. For instance, the nodes in the matching instance pair $[n_8, n_{11}]$ are 3 units apart in horizontal distance, which is less than the threshold bound, 9, but do not belong to the same context (meaningful subtree). We, therefore, need to efficiently identify the NCA of any two given nodes to eliminate such instances. VFT incorporates the information encoded in the binary labels of the nodes for NCA calculation. These unique O (log n)-bit binary labels facilitate the *constant time* calculation of the NCA [4]. The details of the incorporated binary labeling assignment and NCA calculation is provided in [4]. Performing VFT subsequent to HFT will further eliminate the matching instance pair $[n_8, n_{11}]$ from further considerations. The vertical filtration eliminates all those matching instances $a_i = (t_1, \ldots, t_m)$, where the NCA of at least one of its node pairs, (t_j, t_{j+1}), is the root node.

Note that, when considering a very deep document tree where the root's fanout is very small, the lower level nodes may instead be considered to define the *node contexts*. However, the presence of a domain expert to categorize different classes of XML documents based on their DTD structure might be necessary. The incorporation of an adaptive algorithm to facilitate the automated determination of the node contexts is part of our future directions on TWIX.

▷ **Structure Matching.** So far, TWIX has only been processing the leaf information of the query and has determined the candidate leaf matches. Now, given a leaf matching instance we need to construct the subtree structure confined to it in T and compare it with the query's tree structure. A naïve structure matching approach would compare every single path of each matching instance against its counterpart in the query. However, this process is very costly and for this reason we introduce the notion of the *induced subtree*. An *induced subtree* is a topological subtree of T confined to a set of leaves while preserving the PC and AD properties among the nodes. It is constructed using the binary labels associated with each leaf instance node of a matching instance pair. Given the matching leaf nodes of a matching instance, the binary labels of each node are used to progressively find the NCA of the adjacent leaf instance nodes and further connecting the nodes to their appropriate NCA. Figure 2(c) depicts the subtree of T, of Figure 2(a), induced by the set of leaf nodes $N = \{n_4, n_6\}$, which is simply constructed by finding the NCA of n_4 and n_6. The result is a much more compact subtree (hiding unnecessary intermediate nodes) and is used to further prune false positives. Figures 2(c) and 2(d) depict the *induced* and *actual* subtrees of T restricted to the set of leaf nodes N. It can be observed that the subtree in Figure 2(c) does not include the intermediate nodes on the path n_4-n_2 and n_6-n_2 which is the reason behind the compactness of the induced versus actual subtrees. The *compactness* of the induced subtree is a function of the vertical and horizontal distance of the nodes present in the twig query. That is, the resulting induced subtree would be more compact when the distance between the instance nodes is larger which, in turn, results in a longer path between the nodes and their corresponding NCA.

Algorithm 1 Subtree Construction Procedure:

Notations:
N Set of leaves in increasing order of their preorder traversal rank.
I The inverted index built on the potential parents of the nodes in N. Each node entry I[j] points to the list of its children.

```
▷ Allocate an array NODE [2×|N| − 1].
NODE [1] ⟵ N_1
j ⟵ 1
for i = 1 to |N| − 1: do
   NODE [j + 1] ⟵ NCA [N_i, N_{i+1}]
   NODE [j + 2] ⟵ N_{i+1}
   j ⟵ j + 2
end for
▷ Allocate the inverted index I [|N| − 1].
j ⟵ 1
for all i ∈ [2 . . . 2×|N| − 2], i ← i + 2: do
   I [j] ⟵ NODE [i]
   Insert CHILDREN(I [j]) to the linked list of the I [j] node entry.
   j ⟵ j + 1
end for
▷ Sort the node entries of the inverted index, I[i], on the decreasing
  order of their level value as it appears in the document tree.
for i = |N| − 1 to 1: do
   for all nodes N_j in the list I[i] do
      if N_j is not marked then
         Mark the node N_j
         Set parent(N_j) ⟵ I[i]
      end if
   end for
end for
```

Algorithm 1 depicts the detailed procedure of the *induced subtree* construction from a set of leaves. The first `for` loop calculates the NCA of every two adjacent leaf instance nodes N_i and N_{i+1} within a matching instance. Any node N_i ($2 \leq i \leq t-1$) contributes to two NCA calculations: NCA(N_{i-1}, N_i) and NCA(N_i, N_{i+1}), which results in two potential candidates for the appropriate choice of ancestor for each leaf node. Next, an inverted list is constructed for each NCA node linked to the sorted list of its descendants. These NCA nodes are sorted in increasing order of their corresponding $level$ information in T. For instance, in Figure 2(a) nodes n_9 and n_{13} belong to the level 2 and 4, respectively. Finally, the last `for` loop assigns the appropriate ancestor

for every node of the set (including leaves and the NCA nodes) in a bottom-up fashion. Moreover, since the number of leaves used in a twig query is generally very small (<10), the *induced subtree* construction procedure has an expected *constant time* due to the very small number of leaf nodes. The following definition ensures that the induced subtree construction procedure preserves the relationship among the nodes as required by T.

*Definition 5 (***Induced Subtree***):* Given a tree T with set of leaves T_L and $N \subset T_L$: The **subtree of T induced by N** is the non-empty **induced subtree** of T restricted to the leaf nodes in N. Let $R(N, T)$ denote the induced subtree of T induced by N, then for any two nodes $u, v \in N$,

$$NCA^T(u, v) = NCA^{R(N,T)}(u, v).$$

Matching of the structures is performed in two sequential stages: *1) Induced subtree matching*, and *2) Actual subtree matching*. Given each qualifying matching instance (instances of Q's leaves) S from the filtration phase, its induced subtree, $R(S)$, is constructed (e.g., Figure 2(c)). Each path of the induced subtree $R(S)$ is compared against its counterpart in $R(Q)$, and all those matching instances which do not exactly match are filtered from further consideration. The actual subtree comparison is performed for only those matching instances not yet filtered out. This comparison phase further compares the intermediate nodes of the actual subtree of the remaining matching instances versus their counterpart in Q as the final refinement step. The final result is the set of those subtrees of T which match the query structure Q.

TWIX Overview. Given a query Q, first of all, its edge relationships are compared against all the DTDs present in the database to prune irrelevant sets of documents. Next, the query's leaf set is compared against the leaf inverted index of the "*qualifying*" document trees and the leaf matching instances are reported using the combination of SMJ and HFT procedures. Furthermore, the appropriate filtration techniques are applied (VFT and Structure matching) and the complete subtree structure of the remaining matching instances are compared against the query structure. An algorithmic formulation of the main components of TWIX procedure is shown in Figure 3.

III. IMPLEMENTATION AND FUNCTIONALITY ANALYSIS

We implemented the TWIX system using *Java 1.4.2* and ran our experimentations on a *Pentium M-2GHz* processor with *2GB* of main memory. The experimental evaluations were performed on a set of both synthetic (XMark [18]) and real (dblp, acquired from the University of Washington's XML Data Repository at *http://www.cs.washington.edu/research/xmldatasets/*) XML datasets. Table I depicts the statistics of the incorporated datasets and along with corresponding sizes of the binary labeling scheme. Row 4 of Table I shows the amount of storage needed for each corresponding dataset, without using any compression/encoding techniques. Moreover, row 5 denotes the average number of bits used per node. The last row denotes the labeling storage requirement (space overhead) as a percentage of the original dataset size. The storage cost of maintaining binary labels stays at an average of 6.8% for XMark and 15.4% for the dblp datasets. The difference is due to the fact that the overall size of the binary labels is closely related to the average depth of each document tree and corresponding number of nodes. This is a very desirable feature of the binary labeling scheme because the space overhead of maintaining them is very affordable.

A. Filtration Analysis

TWIX deploys various filtration techniques to reduce the size of the intermediate result set. Figure 4 depicts the set of queries used in the filtration study, 3 queries were chosen for dblp and 6 random queries on the synthetic XMARK dataset. Each of the inspected queries investigates a general category of queries ranging from simple twig queries to those with multiple subtrees with various selectivity at the leaf levels. $Q1$ and $Q2$ are only made of PC edges while $Q3$ is made of AD edges, and the rest include a combination of both PC and AD edges. Moreover, $Q1$, $Q3$ and $Q4 - Q8$ are simple twig queries having a single subtree while $Q2$ and $Q9$ impose more than one subtree structures including different context nodes (e.g., inproceedings, and proceedings in $Q2$). In order to study the impact of selectivity on the performance of TWIX, we incorporated queries with various selectivity. The numbers associated with the nodes in Figure 4 denote the *selectivity* of the each particular ELEMENT entry in the inspected dataset. For instance, the leaf level selectivity of the query $Q4$ is equal to 34×618, which is an average-selective query in contrast to $Q6$ and $Q9$ which indicate no selectivity with the values of 780×10659 and $52 \times 1661 \times 89$, respectively.

Filtration order The first step in the filtration study is to determine the most effective order of applying the proposed filtration techniques. For this purpose, we executed the $Q1$-$Q3$ from Figure 4 against the dblp dataset. The response times of the filtration in two different orders *HFT→VFT→Structure* and *VFT→HFT→Structure* are shown in Figure 5(a). The numbers indicate the average response times of 10 different runs of the same queries against each designated dataset. In comparison, HFT, due to considering the proximity of the nodes in the tree, provides much more efficient filtration when it is performed first, while VFT has to perform more calculations (e.g., detecting NCA and context node). Hence, we concluded to apply HFT prior to VFT filtration.

Filtration Effectiveness Table II depicts the effectiveness of the proposed filtrations for *exact search* of the sample twig queries provided in Figure 4. The second column of Table II shows the total number of potential matching instance combinations (*total tuples*). However, TWIX inspects a reduced number of potential tuples resulted from the proposed

	XMLark (*scale factor*)				DBLP
	(0.1)	(0.2)	(0.6)	(1.0)	
Datasize (MB)	11.3	22.8	68.2	113	127
Number of Nodes	119,091	239,103	712,660	2,849,444	5,682,094
Max/Avg Depth	12/6	12/6	12/6	12/6	6/2.9
Binary Labels (MB)	0.79	1.66	5.16	8.97	19.6
Avg bits/Node	23	24	25	26	25
Labeling Storage Overhead	6.98%	7.2%	5.25%	7.93%	15.4%

TABLE I

XML DATASETS USED IN THE EXPERIMENTS AND THE SIZE OF THE BINARY LABELS.

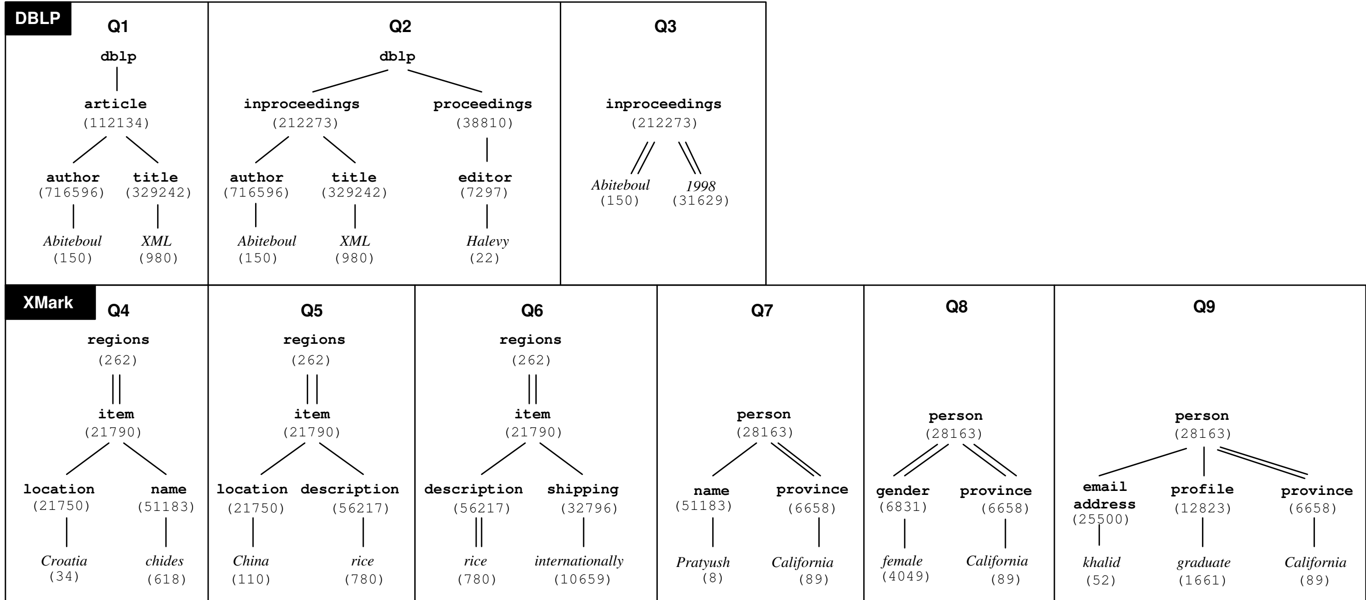


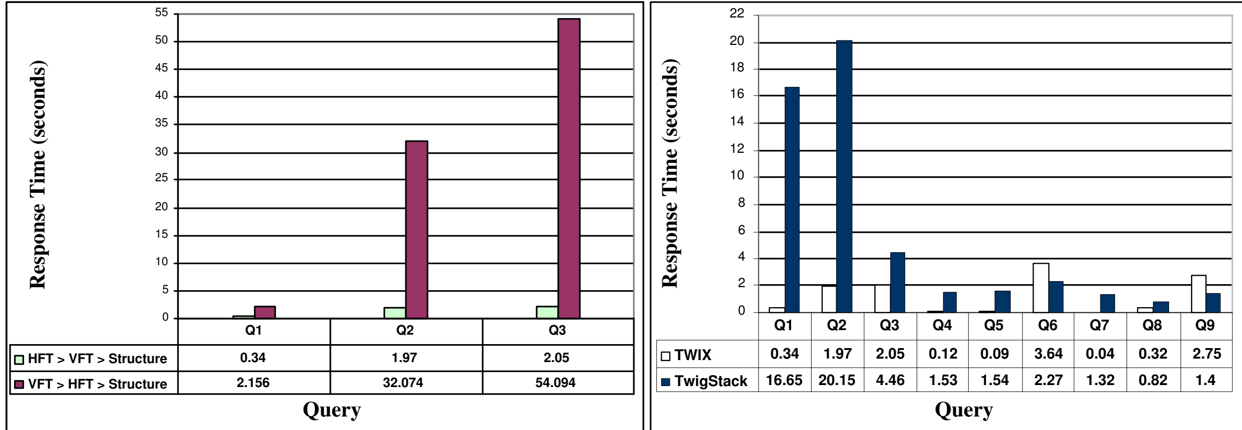
Fig. 4. Twig queries used in the filtration and timing analysis.

Query	Total Tuples	SMJ Tuples	HFT Filtration	VFT Filtration	Structure Matching
Q1	150×980 = 147,000	57,984	57,912	69	2
Q2	150×980×22 = 3,234,000	624,794	566,315	436	56
Q3	150×31629 = 4,744,350	2,530,453	1,905,540	111	0
Q4	34×618 = 21,012	12,469	12,464	2	0
Q5	110×780 = 85,800	46,331	46,239	2	0
Q6	780×10659 = 831,402	2,294,501	2,293,506	982	13
Q7	8×89 = 712	689	684	5	0
Q8	4049×89 = 360,361	178,509	178,471	28	0
Q9	52×1661×89 = 7,687,108	2,848,717	2,848,717	0	0

TABLE II

FILTRATION COMPARISON.

Sort-Merge-Join (SMJ) phase which is depicted in the third column of the table. This number shows the actual size of the search space. The comparison of the second and third columns depicts the amount of the space reduction achieved through the application of SMJ. For instance, the application of SMJ for $Q2$ results in the reduction of the search space from 3,234,000 tuples to 624,794, or in short, that reduces the search space by inspecting only the 20% of the database. The rest of the columns show the amount of filtration achieved (*the number of tuples filtered out*) using each of the proposed filtration techniques in the *HFT* $\rightarrow$ *VFT* $\rightarrow$ *Structure Matching* order. For instance, applying HFT on $Q1$ prunes 57912 out of the 57984 tuples, leaving only 57984 - 57912 = 72 tuples for further inspection. Furthermore, VFT filters out 69 of the remaining 72 tuples, leaving only 3 tuples behind. Finally, the structure matching phase compares the path structure of these 3 tuples, filters out 2 of them, and returns 1 final answer to the user. Note that, there are cases where the tuples which are delivered to the structure matching phase, are already "*answers*" to the query (e.g., on the last columns of $Q4$-$Q6$



(a) Response time of various filtration order applications.

(b) Query response time on dblp and XMark queries.

Fig. 5. Response time analysis of (a) applying filtration techniques in various orders, and (b) filtration efficiency.

and $Q7$-$Q9$ queries). This is a very desirable quality of TWIX that passes only a limited number of tuples to the elaborate structure matching phase.

B. Response Time Analysis

We compared the performance of TWIX against TwigStack[6][1] for all the queries of Figure 4. The complete results are shown in Figure 5(b), where the numbers in the table denote the actual response times (in seconds) averaged over 10 different runs of the query. TWIX outperformed TwigStack for all the queries except for queries $Q6$ and $Q9$ while still performing reasonably close. $Q6$ and $Q9$ are examples of queries with very low selectivity. These results are consistent with the main goal of TWIX which is to target *selective* to *highly-selective* queries, in all of which TWIX has substantially outperformed TwigStack. For the selective queries, TWIX's running time outperformed TwigStack by 49-times at the best ($Q1$), 2-time at the lowest ($Q3$), with the average of 18-times faster response time over all the $Q1$-$Q5$, and $Q7$-$Q8$ queries. The efficiency of TWIX for *selective* queries is an artifact of the effectiveness of the bottom-up space reduction employed by the filtration phases. The reason behind the higher response time of TwigStack versus TWIX (on selective queries) is the fact that the frequency of the internal ELEMENT nodes plays an important role in TwigStack[2], which are frequently encountered in real datasets (e.g. `author` in `dblp`).

C. Structure Scalability Analysis (Structure Invariance)

We further studied the advantages of TWIX when processing queries with various structures. Figure 6 depicts 4 queries $Q10$-$Q13$, having exactly the same set of keywords, with different structures imposed on them. This is an example of the same *identical* entity (an article written by `Abiteboul` on `XML`) being represented by the databases possibly designed by different companies. Figure 7(a) depicts the response time (in seconds) results of running queries $Q10$-$Q13$ against TWIX versus TwigStack. TwigStack demonstrates an exponential increase in the response time when adding more structure to the query, however, TWIX's response time remains "*almost invariant*" to the underlying structure. This is due to the fact that TWIX uses the proximity distance of the leaf nodes (e.g., `Abiteboul` and `XML`) and the notion of the *context node* (e.g., `inproceedings`) for the HFT and VFT filtrations, which are invariant to the other internal nodes of the query twig. Moreover, as depicted in Table II, the combination of HFT and VFT prunes most of the irrelevant tuples, leaving only very few tuples (e.g., 57984 - 57912 - 69 = 3 tuples for $Q1$ in Table II) for the structure matching phase to inspect.

D. Size Scalability Analysis

Figure 7(b) demonstrates the efficiency of TWIX when varying the sizes of the underlying dataset. We incorporated various scaling factors 0.1, 0.2, 0.6, and 1.0 on the XMark synthetic dataset, generating XML documents of sizes 11.3MB, 22.8MB, 68.2MB and 113MB, respectively. A random query was chosen to be placed against all of these datasets. The $Q5$ of Figure 4 was chosen for this purpose due to the following reasons: 1) $Q5$ is not "*too-selective*" in order not to bias TWIX significantly, 2) it includes both PC and AD edges, and 3) the depth of the twig query is larger than 2. Figure 7(b) shows the overall database size scalability of TWIX versus of TwigStack. TWIX maintains a much lower rate of increase in response time compared with TwigStack, which is due to the following reasons: i) The constructed inverted index on the keywords of the document tree provides a very efficient access method, and ii) The increase in size of the database is more likely to introduce more general "*internal nodes*" such as `item`, `location` and `description` (w.r.t. $Q5$) rather than affecting the quantity of "*each*" individual keyword such as `china` or `rice`. The scalability of TWIX with respect

[1] The source codes for this algorithm was kindly provided by Jiaheng Lu from National University of Singapore.

[2] TwigStack maintains one stack per internal node of the query.

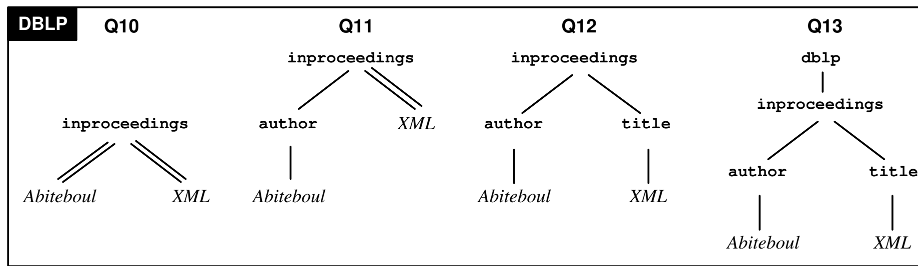


Fig. 6. Structure variation queries.

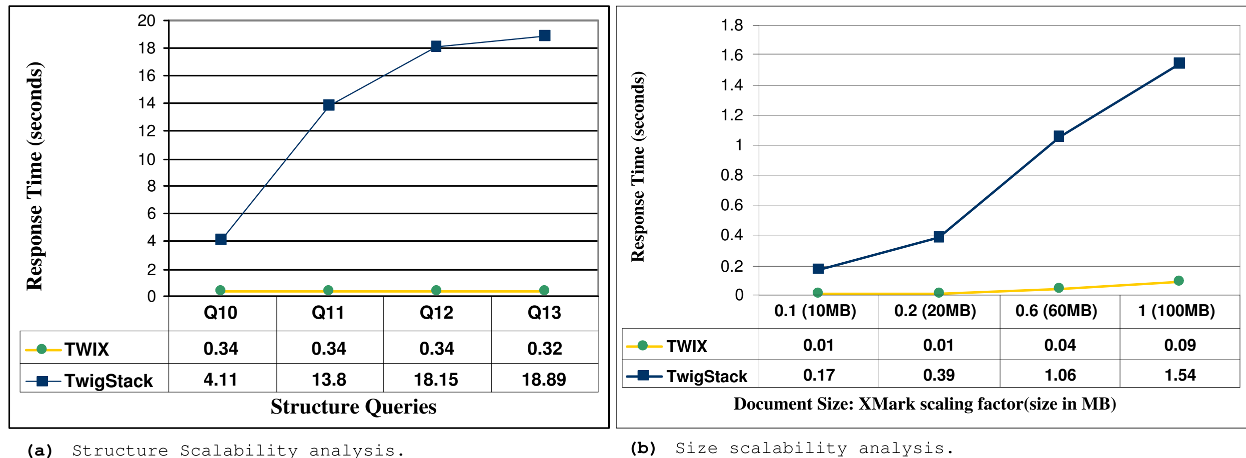


Fig. 7. Response time analysis of (a) structure scalability, and (b) size scalability of TWIX versus TwigStack.

to the size of the underlying database is a very advantageous aspect of applying bottom-up searching for selective queries.

IV. Conclusion

This paper proposed an efficient system design and implementation, named TWIX, for labeling and matching of selective twig queries in a database of XML documents. TWIX incorporates a unique bottom-up traversal of document trees by locating the matching keyword instances and progressively constructing the matching subtrees on top of the keyword matching instances using a rich binary labeling scheme [4]. Various efficient and effective filtration techniques based on DTD relevance, the horizontal and vertical proximity differences of the node extents, and structure matching were deployed. Experimental results demonstrate the promising filtration, response time, structure invariance and the size scalability features of TWIX.

References

[1] S.A. Aghili, H. Li, D. Agrawal and A. El Abbadi, MARS: A Matching and Ranking System for XML Content and Structure Retrieval. Technical Report 2005-11, UCSB (2005).
[2] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas and D. Srivastava, Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE, 141–152 (2002).
[3] S. Al-Khalifa, C. Yu and H.V. Jagadish, Querying Structured Text in an XML Database. SIGMOD, 4–15 (2003).
[4] S. Alstrup, C. Gavoille, H. Kaplan and T. Rauhe, Nearest Common Ancestors: A Survey and a New Distributed Algorithm. Theory of Computing Systems **37**, 441–456 (2002).
[5] C. Botev, J. Shanmugasundaram and S. Amer-Yahia, A TeXQuery-Based XML Full-Text Search Engine. SIGMOD, 943–944 (2004).
[6] N. Bruno, N. Koudas and D. Srivastava, Holistic twig joins: optimal XML pattern matching. SIGMOD, 310–321 (2002).
[7] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras and C. Zaniolo, Efficient Structural Joins on Indexed XML Documents. VLDB, 263–274 (2002).
[8] T. Grust, Accelerating XPath location steps. SIGMOD, 109–120 (2002).
[9] H. Jiang, W. Wang, H. Lu and J.X. Yu, Holistic Twig Joins on Indexed XML Documents. VLDB, 273–284 (2003).
[10] DBLP Bibliography Server, *http://dblp.uni-trier.de/*.
[11] Y. Li and C. Yu and H.V. Jagadish, Scheme-Free XQuery. VLDB, 72–83 (2004).
[12] Q. Li and B. Moon, Indexing and Querying XML Data for Regular Path Expressions. VLDB, 361–370 (2001).
[13] J. Lu, T. Chen and T.W. Ling, Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. CIKM, 533–542 (2004).
[14] 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. VLDB, 193–204 (2005).
[15] P. Rao and B. Moon, PRIX: Indexing And Querying XML Using Prüfer Sequences. ICDE, 288–300 (2004).
[16] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita and C. Zhang, Storing and Querying Ordered XML using a Relational Database System. SIGMOD, 204–215 (2002).
[17] H. Wang, S. Park, W. Fan and P.S. Yu, ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. SIGMOD, 110–121 (2003).
[18] A. R. Schmidt et al., The XML Benchmark Project. Technical Report INS-R0103, CWI (2001).
[19] Y. Xu and Y. Papakonstantinou, Efficient Keyword Search for Smallest LCAs in XML Databases SIGMOD, 527–538 (2005).