TWIX: Approximate and Exact Twig Structure and Content Matching over XML Document Collections using Binary Labeling

S. Alireza Aghili Hua-Gang Li Divyakant Agrawal Amr El Abbadi

Compute Science Department
University of California, Santa Barbara
Santa Barbara, CA 93106, USA.
{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. Prior works have typically applied top-down decomposition of the twig patterns into (i) binary (parent-child or ancestor-descendant) relationships, or (ii) path expression queries, followed by a join operation to reconstruct matched twig patterns. However, most of these methods ***(i)*** rely on the user's knowledge of the underlying database to pose well-formed queries, and ***(ii)*** suffer from inspecting too many irrelevant results. In this paper, we propose a novel heuristic for matching of XML twig query patterns, named TWIX, which imposed minimal restrictions on the user and causes substantial reduction of the search space through a distributed binary labeling technique. The algorithm incorporates a holistic ranking scheme of *structure* and *content*, named TRANK, to rank and report the top-k results. Furthermore, TWIX benefits from an interactive graphical user interface twig query matching. Experimental results on real datasets depict the ranking semantics and efficient filtration of the search space.

1 Introduction

XML (Extensible Markup Language) has become a key technology gaining wide acceptance as the standard mechanism to facilitate the portability and integration of data across the Internet. The *rich content* and the *semi-structuredness* of XML documents demands 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 an ordered tree specify complex patterns of selection predicates on element *labels* (keyword search) and their corresponding inherent *structural relationships* (structure pattern search). The most simple form of structural relationship is a *path expression*. A single slash in the path expression query (`journal/author`) requires the presence of a *Parent-Child (PC)* edge between the corresponding nodes, while a double slash (`dblp//journal`) simply requires the presence of a path from the first to the second element of the query (*Ancestor-Descendant* relationship, AD).

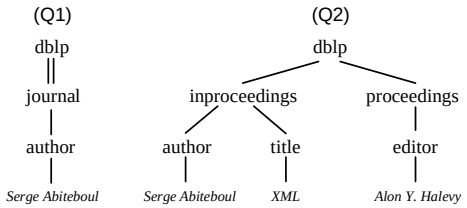

Figure 1: Query Samples.

In general, structural relationship queries may be categorized in two different classes: $i)$ *path expression* query (e.g., `Q1` in Figure 1), and $ii)$ *twig query* (e.g., `Q2` in Figure 1). In particular,

```
Q1 =/dblp//journal[/author = 'Serge Abiteboul']
```

matches all the *path expressions* of `journal` articles $i)$ written by `author` (element) named `'Serge Abiteboul'`, for which $ii)$ (`journal`, `author`) elements have a PC relationship, and (`dblp`, `journal`) elements have an AD relationship.

```
Q2=/dblp/inproceedings[/author ≈ 'Abiteboul'
                        AND /title ≈ 'XML']
      /proceedings[/editor = 'Alon Y. Halevy']
```

matches all the twig tree patterns of conference `proceeding` papers $i)$ whose `author` and `title` fields include the keywords `'Abiteboul'`and `'XML'`; in a conference where $ii)$ `'Alon Y. Halevy'` served as an `editor` and $iv)$ all the edges correspond to the constrained PC relationships. (`inproceedings`, `author`), (`inproceedings`, `title`), (`proceedings`, `editor`), (`dblp`, `inproceedings`),and (`dblp`, `proceedings`).

XML query languages [7, 8, 11, 13] and the underlying relational [15, 31] or native [26, 34] databases must provide efficient and effective support for querying both the *content* and the inherent *structure* of XML documents. Meanwhile, the support for twig queries is usually facilitated by augmenting a layer of structural search on top of the traditional path expression query language [1, 2, 6, 10]. These structural search techniques perform the following steps: (i) decompose the twig query structure into its corresponding path expression components, For instance, the query (`/dblp//journal[/author ='Serge' AND /title≈'XML']`) is decomposed into the path queries (`/dblp//journal[/author='Serge']`) and (`/dblp//journal [/title≈'XML']`), (ii) perform a semi top-down inspection of the XML document tree for each decomposed path component, (iii) join the results of each of the query's path expressions to form the result to the original twig query. However, the top-down traversal of the document tree results in scanning a large number of path combinations. The number of intermediate results is a direct function of the fan-out and size of the document tree which imposes serious scalability and efficiency issues. For instance, the root node of the `dblp` [22] database has 3,288,858 immediate children (as of Sept. 2004) which makes it impossible to inspect all the path combinations if the given query include the root node. While some optimizations may be applied, it is inevitable that quite a large number of path expressions have to be generated and scanned.

Suppose one wishes to execute a simple query `Q =/dblp /inproceedings[/author ≈ 'Abiteboul']` on the `dblp` bibliography database. The `dblp` contains 212,273 occurrences of (`dblp/inproceedings`) edges, 491,783 of (`inproceedings/ author`) edges, and 1,818 of `author` instances containing the term `'Abiteboul'`. In the worst case, the top-down algorithms need to inspect a space of 212,273 $\times$ 491,783=1.04×10^{11} potential path combinations for path join before even reaching the leaf level constraint. However only 1,818 of those combinations result in the term `'Abiteboul'`. This artifact depends on the query and the statistical characteristics of the underlying data however, in the above example the worst case scenario without having any information about the user query patterns, results in an additional inspection of $\frac{1.04\times10^{11}-1,818}{1.04\times10^{11}}$ tuples or the 99.9% of the database.

Various techniques [6, 28, 29] have been proposed to reduce the size of the intermediate result set, however they still suffer from a large number of potential inspections. TWIX deploys an efficient bottom-up technique for matching, filtration, and ranking of twig query patterns, avoiding the decomposition of the twig query and inspecting the large number of intermediate results (path combinations). For short, TWIX only inspects a compact inverted index for the occurrences of the query's keywords and builds the twig subtree on top of the potential matched keywords. As a result only a small portion of those potential keyword occurrences are inspected. Given a query tree Q, the incorporated binary labeling of TWIX reconstructs the subtree, restricted to the given leaf keywords of Q, in O($|Q|$) time. One other important property of TWIX is that, it imposes minimal restrictions on the user's knowledge of the underlying database schema by incorporating both approximate keyword and structure search/ranking.

Due to the heterogeneous nature of XML documents (i.e. schema and representation), it is crucial to incorporate approximate structure matching in conjunction with appropriate ranking of both the structure and content of the elements. The need for approximate structure matching arises in a variety of applications motivated by the following issues: (i) *Privacy*, *Scalability*: Due to access policy issues, the users might not have access to the schema of one or more of the XML documents of the target database, (ii) *Inconsistency:* The nature of the information on the web is responsible for the *semi-structuredness* of the underlying XML documents. This implies that (1) the data might be incomplete or irregular, (2) its structure may change, or (3) the data may not fully conform with the imposed structure.

The main contributions of the TWIX structure and content matching are summarized as follows:

1. TWIX deploys *proximity-based* primitives for structure matching of query tree using a polynomial-time dynamic programming alignment algorithm. It further devises an IR-style content search scheme at the fine level of attribute/element contents.

2. An efficient *binary labeling* scheme based on the notion of *Nearest Common Ancestor (NCA)* is introduced for fast construction of a subtree from its leaf content keywords. Given a set S of leaves/keywords of an XML document tree T, TWIX employs a bottom-up algorithm for the construction of the *"subtree of T induced by S"* in *constant time*, without the need to decompose the query.

3. Effective filtration techniques based on DTD (Document Type Definition) similarity and the horizontal/vertical structural extent of the nodes are integrated to avoid the exponential blow-up of the content-matched intermediate results. TWIX only searches within a compact inverted index for the occurrences of the query's keywords and builds the twig subtree on top of the matched keywords. As a result filtration, only a small portion of those matched keyword instances are inspected.

4. A fully functional $java$-based implementation of TWIX interactive system and a graphical user interface is developed to facilitate visual and intuitive twig pattern matching,

5. A novel *content* and *structure* ranking scheme (TRANK) for answering top-*k* queries.

6. The experimental evaluation on the efficiency and functionality of TWIX filtration and ranking.

The rest of the paper is organized as follows: Section 2 discusses the background and related work. Section 3 provides an overview of the TWIX procedure and each component of the proposed technique. Section 4 introduces the incorporated filtration techniques followed by Section 5 which discusses the concept of TRANK. Section 6 provides the empirical performance analysis followed by Section 7 which concludes the work.

2 Background and Related Work

The support for structural queries introduces a challenge for both relational [15, 31] and native [26, 34] implementations of XML databases. Recently, much research has been conducted to support structural similarity. These efforts may be classified into four main categories:

The *structural join methods* [1, 23, 24] typically decompose the query twig query pattern into a set of *parent-child (PC)* or *ancestor-descendant (AD)* binary components. Consequently, each binary component of the query pattern is matched against the XML database and the matched intermediate results are joined to obtain the final query twig pattern match. One major drawback of these methods is the size of the intermediate result set which can potentially be very large, for which the identification of an efficient cost-based join ordering (based on the selectivity and join size estimation) is inevitably required. Polyzotis et al. [28] propose methods to estimate the selectivity of the XML twig based on the notion of synopses which can be useful as a filtration step for reducing the number of the intermediate results. They further propose the TREESKETCH and TWIG-XSKETCH synopses model to facilitate query approximate answers. However, this approach focuses on the structural part of the query and ignores the content and the keyword distributions in the XML document. Bruno et al. [6] propose `TwigStack`, a holistic twig join algorithm which 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. [24] extend `TwigStack` for optimality when PC edges are used as well. TWIX is different from this class of techniques since it does not decompose the path expressions into binary relations and hence does not suffer from the explosion of the size of the intermediate result set.

The *numbering scheme methods* [1, 14, 16, 23] associate interval encoding with every node to help identify PC or AD relationships among the nodes. There are two main categories incorporating the numbering scheme: (i) In one approach [1, 14, 16, 23], given a node u, the algorithms assign an interval signature as $[preorder(u), postorder(u), level(u)]$. Given any two nodes u and v: node u is an *ancestor* of node v, iff $preorder(u) < preorder(v)$ and $postorder(u) > postorder(v)$. (ii) There are also a number of approaches [6, 23] which deploy an interval scheme of the form $[begin(u), end(u), level(u)]$. Similarly, node u is called an *ancestor* of node v, if $start(u) < start(v)$ and $end(u) > end(v)$. One interesting property of these methods is the fact that testing for the *ancestor-descendant* relationship is no more costly than testing for the *parent-child* relationship which is determined by a simple check for interval inclusion. However, these methods can not handle updates efficiently. As a result, more *dynamic* value assignment techniques (by either leaving some empty space between the interval values, or using *real precision* numbers) have been proposed to alleviate this problem [23]. TWIX uses only *pre-order traversal numbering* and additionally supports *dynamic* pre-order assignment which introduces gaps between pre-order values for support on future updates. The reduction in the amount of augmented information (interval vs. single pre-order value) makes it more difficult to detect AD relationships however, TWIX does not decompose the path expressions and hence does not need the AD information directly. TWIX maintains a parent pointer attached to each node of the inverted index to facilitate the discovery of PC relationships.

Several *Indexing methods* [10, 12, 16, 19, 21, 23, 33, 35, 36] have been proposed to index the document elements and attributes of XML documents. Most of these methods represent intervals as points in a multidimensional space and utilize available indexing techniques (e.g. B^+-trees, R-trees, etc) on element sets to store and query these points. Similarly, TWIX incorporates inverted index and hashing-based mappings for the fast retrieval of leaf keywords, element sets and path labels however, the way it handles the query is substantially different.

The *String matching methods* [30, 35, 36] propose encoding techniques to *flatten* tree structures to a string. The problem of tree structure similarity transforms into a substring matching problem. These techniques include: (i) Transforming the XML document and the query twig pattern into labeled sequences using the *Prüfer* method [30]. The occurrences of the query twig pattern in the database is inspected by performing subsequence matching on the corresponding *Prüfer* sequences of the query and the document, (ii) Zezula et al. [35, 36] propose a novel technique, called the *tree signature*, to represent tree structures as ordered sequences of *pre-order* and *post-order* ranks of the nodes. Similarly, the algorithm applies string matching and indexing techniques to answer ordered and unordered tree inclusion queries, as well as the structure search. TWIX does not flatten the tree structures, however, employs approximate substring matching for leaf keywords matched instances and uses dynamic programming string alignment for matching and ranking of the path expressions.

Most recently, Amer-Yahia et al. [4] and Marian et al. [25] propose FlexPath and Whirlpool systems. These methods incorporate the notion of query relaxation on the structure of the query and include the answers to all relaxations of the query. An IR-based $tf \times idf$ paradigm is employed for scoring and full-text search at the element level. FleXPath [4] studies the properties of the proposed ranking schemes for answering top-k queries. Whirlpool [25] devises an adaptive algorithm for scoring and top-k matching while studying the query plan evaluations. In contrast, the support for approximate match in TWIX is inherent in

NOTATION	DESCRIPTION
T_L	Set of leaf nodes of tree T.
$\|T\|$	Number of nodes in tree T.
T_u	Subtree of T rooted at node u.
$\|T_u\|$	Size of the subtree rooted at u.
$h(u)$	*heavy node*: heavy child of the node u.
$\|\|u\|\|$	$\|\|u\|\| = \|T_u\| - \|T_{h(u)}\|$.
r_u	Pre-order traversal rank of node u.
CHILDREN(u)	Set of immediate children of node u.
L_u	The binary label assigned to a node u.
$NCA(u, v)$	Label of the NCA of nodes u and v.
$R(N, T)$	The relaxed subtree of T restricted to the set of nodes/leaves in N.

Table 1: Basic Notations of TWIX and labeling procedures.

the bottom-up subtree construction from the successfully matched keyword instances. This reduces the search space, while the prunning of the irrelevant matches is performed at the earliest stage of structure matching, which is an inspection *only* within those intermediate results succeeding the keyword matching phase.

3 The TWIX Procedure

In this section, we decompose the TWIX into its primary components and provide a general overview of the proposed techniques. The section is later followed by in-depth descriptions and analysis of each individual unit.

3.1 Binary Labeling Procedure

This seciton introduces the distributed NCA binary labeling scheme. Alstrup et al. [3] propose a technique which assigns unique *binary labels*[1] to each node of an arbitrary tree. We generalized and extended the algorithm to handle the construction of subtrees from their respective leaves for trees with arbitrary fan-out (which is *linear* in the number of leaves of the query twig), and further implemented the entire algorithm[2].

Theorem 1 (ALSTRUP AND KAPLAN [3]) *Given a rooted tree T of n nodes, there exists a* ***linear time*** *preprocessing algorithm that assign labels of size O (log n) bits to each node, such that from the labels of any two nodes x_i, $x_j \in T$ alone, the label of their nearest common ancestor, NCA(x_i, x_j), can be determined in* ***constant time****.*

The algorithm performs an *offline* top-down preprocessing of the document tree. Given any node u, the child of u whose subtree size is maximum, is called *heavy* node and all the other children of u are referred as *light* nodes. Root node is a *light* node by default. Following this procedure the tree is segmented into disjoint individual paths consisting of *heavy* nodes (*heavy paths*) as depicted in Figure 2.a. The preliminary binary labels (*heavy* and *light* labels) are assigned in two individual stages: (i) assigning

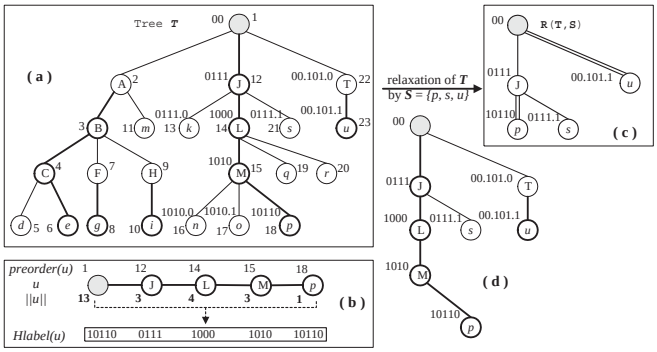

Figure 2: (a) Tree T with some of the corresponding *binary labels* and the *pre-order traversal ranks* of the nodes. The *heavy paths* are highlighted with bold edges. (b) Extracting *heavy labels* for a particular *heavy path* of T. (c) The subtree of T induced by the set of keywords $S = \{p, s, u\}$ (*Tree Relaxation*). (d) The actual minimal subtree of T restricted to the leaf nodes in set S.

binary labels to the nodes on the *heavy path (heavy label, HLabel)*, (ii) assigning binary labels to the *light* children of the nodes *(light label, Llabel)*. Given a sequence of nodes $u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_m$, depending on being a *heavy path* or the sequence of *light children* of a node, the binary *heavy* and *light* labels are extracted from $||u_1|| \rightarrow ||u_2|| \rightarrow ... \rightarrow ||u_m||$ or $|T_{u_1}| \rightarrow |T_{u_1}| \rightarrow ... \rightarrow |T_{u_m}|$, respectively[3]. For instance, the binary labels of the nodes on the *heavy path* that starts from the root ($root \rightarrow J \rightarrow L \rightarrow M \rightarrow p$) are assigned by first separating this path from the rest of the tree (Fig. 2.b) and then assigning the labels on the sequence $||root|| \rightarrow ||J|| \rightarrow ||L|| \rightarrow ||M|| \rightarrow ||p||$ which is the sequence $13 \rightarrow 3 \rightarrow 4 \rightarrow 3 \rightarrow 1$. The *heavy* and *light* labels of each node are concatenated to construct the final ***binary label*** (L) as follows: for the root node, $L_{parent(root)} = Llabel_{root}$=`null`, and for all other nodes $u \in T$: $L_u = L_{parent(anc(u))}.Llabel_{anc(u)}.Hlabel_u$. For all the nodes u of the tree, L_u consists of the concatenation of alternating *heavy* and *light* labels: L_u=$h_1.l_1.h_2.l_2 \ldots$.

Lemma 1 (Nearest Common Ancestor (NCA) Query.) *Given any two nodes or vertices $u, v \in T$:*

- *if $L_u = h_1.l_1 \ldots h_i.l_i.s$ and $L_v = h_1.l_1 \ldots h_i.l'_i.s'$, where $l_i \neq l'_i$, or $l_i.s = null$ or $l'_i.s' = null$: $NCA_{(u,v)} = h_1.l_1 \ldots h_{i-1}.l_{i-1}.h_i$.*
- *if $L_u = h_1.l_1 \ldots h_i$ and $L_v = h_1.l_1 \ldots h'_i$, where $h_i \neq h'_i$: $NCA_{(u,v)}$=$h_1.l_1 \ldots h_{i-1}.l_{i-1}.min_{lex}\{h_i, h'_i\}$.*

Example. Given the nodes k, p, s with labels $L_k = 0111.0$, $L_p = 10110$, and $L_s = 0111.1$. First case of Lemma 1 applies when calculating $\text{NCA}_{(k,s)}$ = $\text{NCA}_{(0111.0,0111.1)}$ = 0111 which is the binary label of the node J. Such a case occurs when the prefix matching of two labels end in a *heavy* region. The second case occurs for $\text{NCA}_{(p,s)}$ = $\text{NCA}_{(10110,0111.1)}$ =

[1] In this paper, the term *binary label* is truncated to simply *label* which should not be confused with the element labels or tag names of the nodes (e.g. `author, title, ...`).

[2] Alstrup et al. [3] did not provide any experimental analysis.

[3] For details on extracting heavy and light binary labels, refer to [3].

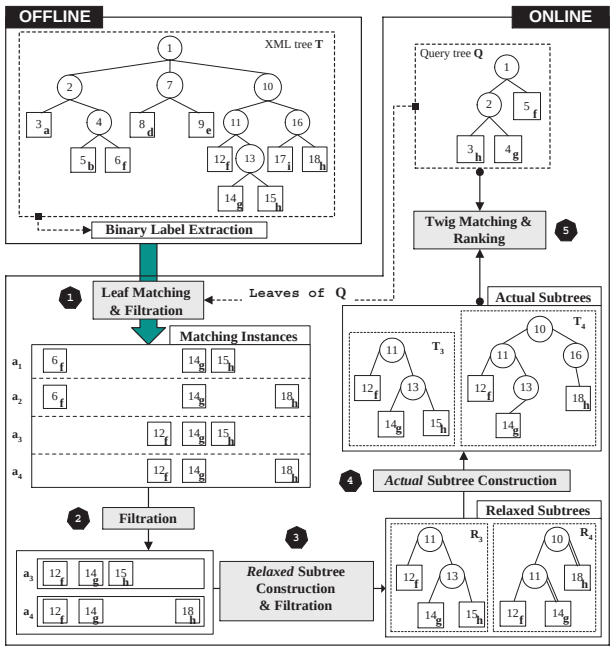

Figure 3: TWIX procedure: a general overview.

$min_{lex}\{10110, 0111\} = 0111$ which is similarly the binary label of the node J which occurs when the prefix matching of the labels end in a *light* region. The min_{lex} denotes the preceding binary string in lexicographic order[4]. Figures 5.c-d depict the *actual* and *relaxed* subtrees of T induced by $N = \{p, s, u\}$.

Given a tree of size n, the following are some of the properties of the binary labels assigned to each node: (i) The process of assigning labels to each node of the tree takes O(n) computation time which is performed *offline*, (ii) Given a tree T and any subset S of its leaves in order of their *pre-order traversal rank*, the subtree of T restricted to the nodes of S can be constructed in O($|S|$) time.

3.2 TWIX Overview

Given an XML document tree T, Figure 3 depicts a general overview of the TWIX procedure. Offline, T is analyzed and each node is associated with a unique *binary label* (not shown in the Figure 3) and its corresponding *pre-order traversal rank* (the integer value in each node). However, only leaf nodes have actual alphabetic keywords associated with them. Figure 4 depicts a more detailed algorithmic formulation of the TWIX procedure. The algorithm starts with an offline stage which associates unique O($log\ n$)-bit binary labels to each node of the tree which facilitates the detection of the nearest common ancestor (*NCA*) of any two given nodes in *constant time*. Given the twig query pattern Q and document tree T in Figure 3, the online phase of TWIX is performed in five stages, summarized as follows:

1) ***Leaf Matching, and Filtration***: Let $Q_L = \{h^{(3)}, g^{(4)}, f^{(5)}\}$ denote the set of leaves of the twig query Q where each leaf is associated with its *label* and *pre-order traversal rank*. Leaf matching, finds all occurrences of the nodes of Q_L in the leaf nodes of T. For instance, the node $h \in Q_L$ occurs twice in the leaves of T: $h^{(15)}$ and $h^{(18)}$. Given that Q_L consists of three nodes, leaf matching results in ordered triplets a_i (*matching instance*), where the j^{th} entry of a_i denotes an occurrence of the j^{th} node of Q_L in the leaves of T. TWIX incorporates a very efficient early *structure filtration* procedure when dealing with multiple XML document databases each having a unique DTD (Document Type Definitions). It is much more likely that the answers to query Q originate from those XML documents whose DTD resembles the structure imposed by Q. TWIX employs a DTD-based Filtration step (DFT) for early pruning of irrelevant documents which drastically reduces the search space.

2) ***Filtration***: This is the second filtration step in TWIX which removes most of the irrelevant matching instances (a_1 and a_2 in this example) using the extracted *NCA* binary labels. For instance, in $a_1 = [f^{(6)}, g^{(14)}, h^{(15)}]$, the nodes $g^{(14)}$ and $h^{(15)}$, belong to the same subtree of T (rooted at node n_{10}), however the node $f^{(6)}$ belongs to a relatively

[4]Due to space limitations, we have not included the implementaiton details of the binary labeling algorithm.

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

1. Perform a *pre-order traversal* on tree T and associate a *pre-order 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:

- $S = \{s_1, \ldots, s_m\}$: set of all the keywords of the query's leaves ($m \geq |Q_L|$).
- $e(s_i, T) = e_i$: Leaf matching of s_i in T.
- $a_j = (t_1, \ldots, t_m)$: a matching instance tuple from $E(K, T)$, where $1 \leq j \leq \beta = \Pi_{i=1}^{m} |e_i|$.
- r_u: pre-order traversal rank of the node u.
- τ: horizontal distance threshold.
- $R(S)$: The *relaxed* subtree induced by S.

- 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.
```

  - Let $\{a_{\pi_1}, \ldots, a_{\pi_s}\}$ denote the set of all those matching instance tuples which are not yet pruned.

- While progressively constructing the induced relaxed twig 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.
```

- For each of the remaining relaxed 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
 - Rank the relevance (ref.  figure 7) of
   T_{a_{π_j}} to query twig pattern Q.
```

Figure 4: TWIX procedure formulation.

distant subtree (rooted at node n_2). This implies that the matching instance a_1 covers more than one main meaningful entity (assuming that different subtrees of the root node entail different entities) and hence should be pruned from further considerations. This filtration step also uses the *horizontal* and *vertical* distances of the tree nodes to determine the relevance of the nodes and further pruning of the matching instances, i.e. reducing false positives.

3) ***Relaxed Subtree Construction, and Filtration***: For any remaining matching instance a_i, this stage performs a bottom-up approach to construct a *virtual* subtree of T (*relaxed subtree*) restricted to the nodes of a_i, using the *NCA* binary labels of the nodes. For instance, the relaxed subtree of a_4, R_4, is progressively constructed through finding $nca(f^{(12)}, g^{(14)})$ and $nca(g^{(14)}, h^{(18)})$ which result in the nodes n_{11} and n_{10}, respectively. In this construction procedure, all the *ancestor-descendant*(AD) and most of the direct *parent-child*(PC) relationships are preserved, however some of the PC edges are replaced by valid AD edges (*virtual edges*)[5]. This explains why these subtrees (R_i) are called *virtual* or *relaxed* subtrees. The resulting relaxed subtrees are more compact than the actual subtrees of T restricted to the nodes of a_i. This is because some PC edges are replaced by AD edges (virtual edge) which ignores all the corresponding intermediate nodes. For instance, the edge $n_{11} \Longrightarrow g^{(14)}$ in the relaxed subtree R_4 ignores the intermediate node n_{13}. The *compactness* of the constructed relaxed subtrees depends on the nodes involved in each virtual edge, which is a function of the vertical distance between the corresponding nodes. For instance, the extent of the vertical distance between n_{11} and $g^{(14)}$ indicates their contribution to the compactness of R_4. Moreover, the extent of the horizontal distance of the nodes (e.g. $f^{(12)}$ and $g^{(14)}$ in R_4) indicates the likelihood of the *NCA* of these nodes introducing a virtual edge. As a result, the horizontal distance between two nodes implies how high their corresponding *NCA* node is in the document tree T. This construction and filtration stage, benefits from the *compactness* of the relaxed subtrees and further uses the path matching on the potential results.

4) ***Actual Subtree Construction***: Given each remaining relaxed subtree from the previous stage, the corresponding virtual edges on each path are further extended to cover the required intermediate nodes. The outcome is the *actual* subtrees of T, T_i, restricted to the nodes in the matching instance a_i. While constructing the actual subtrees, the *path* information along each extended virtual edge is compared against its counterpart in Q and mismatching subtrees are either pruned or pushed to the bottom of the result set.

5) ***Twig Matching and Ranking***: The resultant actual subtrees are compared against the query twig for similarity comparison and ranking. A novel combination of an IR-based keyword and structure relevance ranking technique, and the path-level minimum tree edit distance [9, 38], called TRANK, is employed to compare each T_i against twig query Q and ranking of the T_i subtrees among each other. This measure captures the *similarity* and *significance* of the structural results, by incorporating both the *content* and *structure* of the nodes into account.

[5]These *virtual AD edges* are showed using double lines.

3.3 Matching Terminology and Formulation

This section introduces the definitions and notations used in the matching phase. The corresponding notations are summarized in Table 2. Matching finds the potential subtrees of the document XML tree by inspecting and comparing its leaves against those of the twig query.

Notation 1 *Suppose $T = (V, E)$ is an ordered tree, where V and E denote the set of nodes and edges. Each of the nodes are assigned a numerical value which is its unique **pre-order traversal rank**. The ranks of the nodes in a document tree impose a logical document order. Let $I = \{n_1, \ldots, n_{|I|}\}$ and $L = \{l_1, \ldots, l_{|L|}\}$ denote the set of internal nodes and the leaves of tree T, where $I \cup L = V$. Each of the nodes in V is assigned a **label (tag name)**. The label of a leaf is also referred to as its **value**, which may not necessarily be unique.*

For instance, in the tree T of Figure 5, the labels of the nodes with rank 2 and 4 (n_2 and n_4) are `inproceedings`, and `Serge Abiteboul`, respectively.

Definition 2 *Given a keyword s and a tree T, the **leaf matching** of s in T, $e(s, T)$, is the set of all the leaf nodes of T, wherein s appears as a keyword or substring.*

Example. Let T_L denote the set of leaves of T, $T_L = \{n_4, n_6, n_8, n_{11}, n_{13}, n_{15}, n_{17}, n_{68}, n_{70}, n_{72}\}$ as depicted in Figure 5. Given $s =$ `'Abiteboul'`: $e(s, T) = \{n_4, n_{68}\}$ = {`Serge Abiteboul`$^{(4)}$, `Serge Abiteboul`$^{(68)}$}.

Definition 3 *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 matching** sets $e(s_i, T)$. 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 pre-order traversal rank in T.*

Example. Given the set of a query's leaf keywords $S = \{s_1, s_2\}$ = {`'Abiteboul'`, `'XML'`} results in $E(S, T) = \{e_1, e_2\}$ where $e_1 = \{n_4, n_{68}\}$ = {`Serge Abiteboul`$^{(4)}$, `Serge Abiteboul`$^{(68)}$}, and $e_2 = \{n_6, n_{13}\}$ = { `Dynamic XML`$^{(6)}$, `Colorful XML`$^{(13)}$}.

Definition 4 *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 set $M(S, T) = \{a_1, \ldots, a_k\}$ denotes the set of all the possible unique m-ary matching instance tuples of S on T_L, where $|M(S, T)| = \Pi_{i=1}^{m} |e_i|$.*

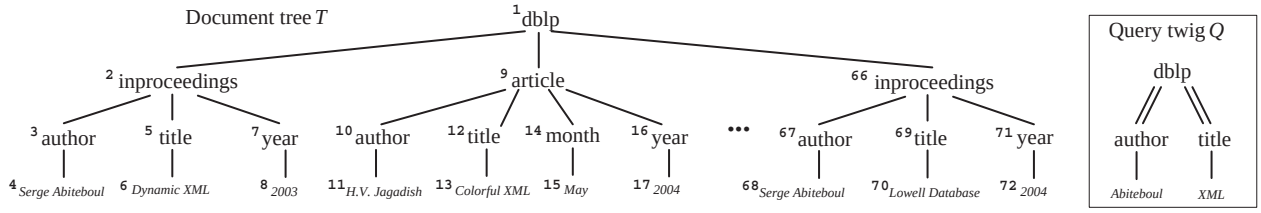

Figure 5: An XML document tree from DBLP database.

For instance, given $S = \{$'Abiteboul', 'XML'$\}$: M(S,T) = $\{[n_4, n_6], [n_4, n_{13}], [n_{68}, n_6], [n_{68}, n_{13}]\}$ = {[Serge Abiteboul$^{(4)}$, Dynamic XML$^{(6)}$], [Serge Abiteboul$^{(4)}$, Colorful XML$^{(13)}$], [Serge Abiteboul$^{(68)}$, Dynamic XML$^{(6)}$], [Serge Abiteboul$^{(68)}$, Colorful XML$^{(13)}$]}.

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* $|e_i|$. Section 4 introduces various effective filtration techniques to avoid and alleviate the exponential blow-up of the intermediate results set.

3.4 Bottom-up Subtree Construction

This section introduces the main bottom-up algorithm to reconstruct the relaxed subtree, restricted to a subset of leaves, which is used for filtration of the exact queries. The extracted subtree is further passed into the filtration step for further refinement and elimination of false positives. Algorithm 1 depicts the detailed procedure of the subtree construction. Note that, |N| denotes the number of actual leaf keywords used in the twig query which is generally very small (<10). As a result, the tree construction which is an O(|N|) algorithm appears to depict an expected *constant time* due to the very small value of |N|. The following definition captures the principle of the relaxed subtrees.

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 topological subtree or* ***relaxed subtree*** *of T restricted to the nodes of N. Let $R(N,T)$ denote the relaxed subtree of T induced by N, then for any two nodes $u, v \in N$,*

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

For instance, Figures 2.c-d depict the *relaxed* and *actual* subtrees of T induced by $N = \{p, s, u\}$, respectively. The *relaxed* subtree of T induced by N, $R(N,T)$, may have some edges not originally present in T (*virtual edges*), as depicted by double lines in Figure 2.c. The virtual edges capture the notion of AD relationships in the relaxed subtrees. The relaxed tree construction results in a topological subtree of T restricted to the nodes of N. $R(N,T)$ is a much more *compact* version of the *actual* subtree of T induced by N (preserving all the AD relationships by using *virtual edges*, and maintaining the PC relationships through *direct edges*). The *compactness* of the relaxed subtree is a function of the vertical and horizontal distance of the nodes present in the query twig. The distance information is determined by NCA lookup and inspecting the *depth* of the nodes, which drastically improves the efficient filtration of irrelevant candidates.

Algorithm 1 Subtree Construction Procedure:

Notations:
N Set of leaves in increasing order of their pre-order 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 $i \in [2 \ldots 2\times|N|-2]$, $i \leftarrow i+2$: **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
NOTE: Any node N_i ($2 \le i \le t-1$) 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 parent for each node.

4 Filtration Techniques

This section provides a detailed description of the proposed filtration techniques and how the constructed relaxed trees are used for filtration. Consider the following running example: the execution of a keyword search for $S = \{$'Abiteboul', 'XML'$\}$ on the dblp database results in e('Abiteboul',dblp) = 1818, and e('XML',dblp) = 957. The total number of matching instances in the worst case is $|M(S, dblp)| = 1818 \times 957 = 1,739,826$, which shows the inevitable need for efficient filtration techniques. TWIX employs several efficient filtration techniques to reduce the size of the intermediate result set. It employs two classes of structural filtration by i) deploying *DTD-conscious* pruning for document filtration, and ii) *vertical* and *horizontal*

proximity of the nodes for filtration based on the similarity of matching instances to the query.

Example. Let S denote the set of query's leaf keywords as defined above. The set $M(S,T)$ = $\{[n_4,n_6]$, $[n_4,n_{13}]$, $[n_{68},n_6]$, $[n_{68},n_{13}]\}$ represents the corresponding matching instances of the occurrence of S keywords in T of Figure 5. It is clear by looking at Figure 5 that the matching instance $[n_4,n_{13}]$ should not be in the final answer set because nodes n_4 and n_{13} belong to different entities[6]. Similarly, $[n_{68},n_6]$ and $[n_{68},n_{13}]$ should also be eliminated. The only valid matching instance is $[n_4,n_6]$ where both nodes n_4 and n_6 entail the same entity. As a result, given any two nodes of a document tree, the horizontal and vertical distances between them are used as a measure to assess the relevance of the nodes.

4.1 DTD Filtration Technique (DFT)

Suppose one wishes to find all the top-k answers to a given twig query Q from a collection of XML documents. DTD provides a general overview of the building structure of an XML document. It is much more likely that answers to the twig query Q originate from those XML documents whose DTD resembles the structure required by Q. For instance, given `Q =/dblp/article[/author = 'Serge Abiteboul']`, the search can be simply limited to only those XML documents whose DTD schema shares Ω edge ratio with the twig query's edges (e.g. [`dblp//article`] and [`article/author`]) to prune the irrelevant documents. TWIX uses $\Omega = 1$ and $\Omega = 3/4$ for *exact* and *approximate* structure match, respectively. For instance, in the case of approximate match, all those documents with Ω value less than 75 % are pruned from further inspection.

4.2 Horizontal Filtration Technique (HFT)

Proposition 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 pre-order traversal rank r_i can be computed as,*

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

where r_{i-1} and $|T_{c_{i-1}}|$ denote the pre-order 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 5 and CHILDREN($root$) = $\{c_1,c_2,\ldots,c_m\}$ = $\{$`inproceedings`$^{(2)}$, `article`$^{(9)}$, ..., `inproceedings`$^{(66)}\}$. The corresponding pre-order 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}}.

[6]They are attributes of two distinct `articles` published by different `authors` in 2003 and 2004.

Definition 6 ***(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 5, the subtree rooted at node c_2 = `article`$^{(9)}$ has the maximum size and hence: $\tau = |T_{c_2}| = 9$.

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

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

To clarify this property, consider a document tree T with nodes $\{n_1,\ldots,n_{|T|}\}$ where the subscripts denote the rank of the nodes, and CHILDREN(root) = $\{c_1,\ldots,c_m\}$. For any subtree T_{c_i}, the node with the minimum pre-order rank is node c_i and the node with the maximum pre-order rank, n_k, is the lowest-right node of T_{c_i} which has the rank $r_k = |T_{c_i}| + r_i - 1$. The difference between the pre-order ranks of nodes c_i and n_k can not be larger than $|T_{c_i}|$. As a result, τ denotes the maximum difference between the pre-order ranks of any two nodes that belong to the same subtree. Hence, if the pre-order 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 do not entail the same entity. This observation may also be deployed in applications where the subtrees store information on various types of items (e.g. DVD's, books, magazines, ...). In such cases, clustering the nodes on their relative *concepts* or *types* is essential in locating relevant nodes. In the `dblp` example of Figure 5, $\tau = max\ \{7,9,\ldots,7\} = 9$ 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., $|r_{n_{68}} - r_{n_{13}}| = |68 - 13| = 55 > \tau = 9$. Note that, HFT depends on the existence of main entity sub-trees at the root's children level however, this requirement is adaptively pushed further down the tree depending on the general structure of the underlying data.

4.3 Vertical Filtration Technique (VFT)

Vertical filtration is inspired by the fact that the nearest common ancestor of any two given nodes u,v belonging to the same entity (subtree) should be a node other than the root. In Figure 5, nodes n_4 and n_6 belong to the same entity (subtree rooted at n_2 = `inproceedings`) and hence their nearest common ancestor is a node other than the root. In contrast, the nodes n_4 and n_{13} belong to different subtrees (entities) 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_4,n_{13}) = $root$ = `dblp`$^{(1)}$. Performing VFT subsequent to HFT will further eliminate the matching instance pair $[n_4,n_{13}]$ 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. TWIX further develops a more adaptive method in using the notion

of root node reference. For instance, if the fanout of the root element is very small, then the second level elements may also be considered to prune non-related entities.

5 Ranking of the Twigs (TRANK)

XML document databases are heterogeneous in schema (DTD) and their representation. The structural relationship among the same set of keywords in different documents may be different. The XML documents in a collection may either ***(i)*** have different DTDs, or ***(ii)*** have a common DTD but not conforming to it. Executing a query against such different documents may result in possibly *"correct"* answers, though bearing different structures. Moreover, the notion of ranking *granularity* should be extended from *term-document* to further capture *structure*, *terms*, and *documents*. This extension introduces a great challenge since the notion of *relevance* (*scoring*) is much more complex in the hierarchical data model. The *structural proximity* (context) of the keywords (the path coming into/out of them) introduces an additional dimension of complexity to the scoring scheme. As a result, the notion of ranking should encompass a two-dimensional distance measure. For instance given two occurrences of a keyword, a rational distance measure should incorporate both the *vertical distance* (*level* difference) and the *horizontal distance* (*tree width* difference) with respect to the other keywords of the query, in addition to the traditional IR notion of *relevance*. Meanwhile, due to the XML heterogeneity, the proposition of a single general ranking scheme is not plausible. This section studies the general principles and semantics of a meaningful ranking scheme.

Weigel et al. [32] provided a concise comparative survey of recent proposals for ranking the results of structured queries. There have been several proposals in the literature to incorporate the information retrieval notion of relevance into the ranking schemes [2, 4, 5, 17, 18, 20, 25]. However most of these methods either consider the relevance ranking on the content [18], or simply consider ranking of path expressions [2, 5, 20]. Moreover, Chawathe et al. [9] and Shasha et al. [38] propose methods to calculate the distance between two ordered trees, and even add some additional edit operations [9] such as *move* and *copy* of the entire subtrees to better utilize the tree edit distance. However, the significance of each edit operation and the proposition of a general cost/weight assignment scheme is not a clear-cut problem and introduces challenges in designing a meaningful ranking scheme.

Given the simple path expression query[7] of Figure 6, Figures 6.a-g depict some of the possible results, which range from exact to approximate matches. TWIX requires the mandatory presence of the query's leaf keywords in the matching instances[8]. More precisely, for any of the matched subtrees, the presence of the query's leaves is

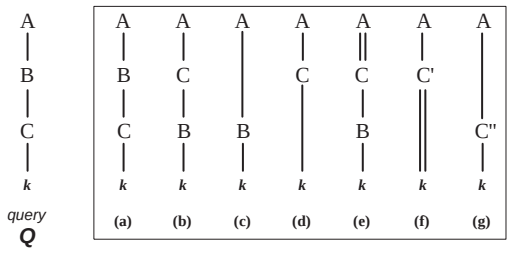

Figure 6: Ranking Samples.

mandatory while the exact conformation of the path characteristics leading to those leaves is *optional*.

5.1 TRANK Scheme, Properties and Components

Let ***Edit Distance (ED)*** denote the minimum number of *edit operations*, node *deletion (DEL)*, *insertion (INS)* and *replacement (REP)*, to transform one tree structure to another. Given the query Q and matching instances of Figure 6, we study a set of questions to clarify the individual components and desirable fearures of *content* and *structure* scoring schemes in TRANK:

▷ **Keyword Relevance Ranking.** The *relevance score* value r for a given keyword (*term*) k with respect to an XML document d, is computed as $r_{kd} = tf_{kd} \times idf_k$. The score r_{kd} depends on two parameters: (i) tf_{kd} denotes the number (*frequency*) of elements where keyword k occurs in the XML document d normalized with the frequency of the most frequent item in d, and (ii) the inverted document frequency idf_k which is calculated by dividing the total number of documents (D) in the database by the total number of documents including the keyword k (df_k). That is:

$$tf_{kd} = \frac{freq(k) \text{ in } d}{max\{freq(k) | \forall k \in d\}}, \quad idf_k = \frac{D}{df_k}.$$

▷ **Structure Relevance Ranking**.

- ***Tree Distance Score (TDS).*** *Compared to Q of Figure 6, which of the* (b) *and* (c) *structures should be ranked higher?* The structure (c) entails a higher score compared to (b) even though (b) contains all the desired internal nodes of the query. Case (c) shares one PC relationship (A/B) as required by Q while (b) shares none!

  Theorem 2 ***Tree Edit Distance Property.*** *Let Q′ and Q″ denote two matching instances of the path Q, where ED (Q, Q′) = k and ED (Q, Q″) = k+1, for some k≥0. The matching instance Q′ shares at most two more PC edges of Q, compared with Q″.*

  Proof. The proof can be easily driven from the fact that, employment of any single DEL, INS and REP operations adds [0-1], [1-2] and [1-2] new edges, respectively. □

  Hence, TRANK reports as follows:

[7] Note that, we rank the twig queries in a similar fashion. The provided path expression examples are only used for the ease of explanation.

[8] This mandatory requirement may be further relaxed.

Notations:

▷ Given any node u of a path, let d_u denote the distance (number of edges) of u to its closest reaching leaf.
▷ Let $|P|$ denote the total number of nodes P_i in a path P.
▷ Let $edge(P, i)$ denote the i^{th} edge of the path P.
▷ Let $||P||$ denote the total number of edges in a path P.
▷ Given any path P and query Q, let P' denote the new version of P after applying the minimum number of edit operations.

```
01 PROCEDURE TRANK (path P, path T, path Q){
02   IF ( TDS(P', Q) = TDS(T', Q) ), THEN
03     LS ← LS(P', Q) - LS(T', Q)
04     EMS ← EMS(P', Q) - EMS(T', Q)
05     score ← f_agg(LS, EMS).
06     IF ( score > 0 ),   RETURN ( P )
07     ELSE,   RETURN ( T )
08   ELSE
09     EMS ← EMS(P', Q) - EMS(T', Q)
10     IF ( EMS > 0 ),   RETURN ( P )
11     ELSE,   RETURN ( T )
12 }
13
14 PROCEDURE LS (path U, path V){
15   LS ← 0.
16   FOR i = 1 TO |U|: DO
17     d ← |d_Ui − d_Vi|
18     IF ( D ≠ 0 AND U_i ≠ V_i ), THEN LS ← LS + (d^-1)
19   RETURN ( LS / |U| )
20 }
21
22 PROCEDURE EMS (path U, path V){
23   Count ← 0.
24   FOR i = 1 TO ||U||: DO
25     IF ( edge(U, i) = edge(V, i) ), THEN Count ← Count + 1
26   RETURN ( Count / ||U|| )
27 }
28
29 PROCEDURE TDS (path U, path V, float α, float k) {
30   L = new float[U.length][V.length];
31
32   ▷ Initialization of the edit distance array L
33   L[0][0] ← 0.
34   FOR i = 1 TO U.length: DO L[i][0] ← L[i - 1][0] + α;
35   FOR i = 1 TO V.length: DO L[0][i] ← L[0][i - 1] + α;
36
37   ▷ Fill in the edit distance array L
38   FOR j = 1 TO V.length: DO
39     FOR i = 1 TO U.length: DO
40       L[i][j] ← min ( L[i - 1][j] + α,
41         L[i - 1][j - 1] + ( (U_i ≠ V_j) ? (α / k) : 0),
42         L[i][j - 1] + α );
43   RETURN (L[U.length - 1][V.length - 1])^-1;
44 }
```

Figure 7: TRANK Scoring Algorithm.

$$\begin{aligned} ED(Q,b) &= DEL + INS = 2 \times REP \\ ED(Q,c) &= INS \\ &\Longrightarrow TDS(Q,c) > TDS(Q,b). \end{aligned}$$

- ***Level Specificity Score (LS).*** *How about* (c) *versus* (d)*?* It is clear that the minimum cost (MTD) of transforming the given query structure Q to either (c) or (d) structures is equal to:

$$ED(Q,c) = ED(Q,d) = INS$$

which is the cost of a single insertion operation[9]. The traditional IR-based keyword relevance ranking fails to effectively distinguish the internal ELEMENTS such as B and C, which is due to the potential large frequency such internal nodes (e.g. author, year, ...). The observation is that, given any particular path in a tree, as traversal gets closer to the leaves the specificity of the nodes increase. For instance, in Figure 5, the month node is clearly much more specific compared to the node dblp. TRANK incorporates the *vertical distance* to the leaf node as the measure of *level specificity*. In any given path, those nodes which are *closer to the leaf* are assigned a higher *level specificity score LS* which is used collectively to reflect the contribution of each internal node to the overall LS score of the matching path expression. In the case of twig structures, where a node might be part of multiple paths to different leaves (e.g. node n_{14} in Figure 2.a), the nearest approaching leaf is used to measure the level specificity Hence, TRANK assigns a higher rank to (c) as compared to (d):

$$LS(Q,c) > LS(Q,d).$$

- ***Edge Matching Score (EMS).*** *How about* (b) *versus* (e)*?* The only difference between (b) and (e) is the existence of an AD edge in place of a PC edge. TRANK associates a higher score to those structures whose edge relationships follows the exact constraints as required by Q. That is:

$$EMS(Q,b) > EMS(Q,b).$$

- *How about* (f) *versus* (g)*, where* C = <author>, C′ = <authors> *and* C″= <author_name>*?* The case (g)'s PC edge C″/k has higher *LS* compared with AD edge C′//k of case (f). Moreover, Q requires a PC edge between C′ or C″ and k which again results in higher *EMS* of (g) compared to (f). However, the label content of node C′ appears to be more *similar* to C compared with C″. TRANK employs the IR notion of *concept* by clustering related ELEMENT nodes together (e.g. author, authors, and author_name). As a result, such cases are conveniently scored using *LS* and *EMS* schemes.

5.2 TRANK Procedure

Figure 7 depicts the details of the TRANK scoring procedure. Given path expressions P, T and query Q, TRANK removes all the PC or AD edge details from the original path expressions before passing them to the ***TDS*** phase. When *TDS* is called, it starts (lines 30-44) by finding the minimum number of edit operations needed to transform $P \rightarrow Q$ and $T \rightarrow Q$. At the same time it aligns them into *equal-length* P' and Q' paths by introducing necessary *gaps* in both paths to represent the implied edit operations[10]. TRANK checks if the *TDS* score of transforming P or T to Q is equal (line 2), if so, that means *TDS* has failed to provide and effective score to distinguish P and T from each other. As a result, it calculates and combines the normalized *LS* and *EMS* scores to further distinguish

[9] $[B/k] \longrightarrow [B/\mathbf{C}/k]$ in Fig.6(c), or $[C/k] \longrightarrow [C/\mathbf{B}/k]$ in Fig.6(d)

[10] Due to space constraints, the details of path modifications is not included in this work. More details may be found in Needleman et al. [27].

the given paths. An aggregate function, f_{agg}, is incorporated to combine the scores of *LS* and *EMS*, which may be chosen as sum, average, median, or a more sophisticated weighting function. For simplicity, TWIX uses linear combination of the *normalized scores* of *LS* and *EMS*. Finally, the path with the higher overall *score* (lines 6-7) is returned by TRANK. If the *TDS* scores of P and T against Q are not equal (lines 8-12), then TRANK combines the normalized *TDS* and *EMS* scores using the same aggregate function, f_{agg}, as before. Once all the matching instances of the document tree T to the twig query Q, $M(Q,T)$, are found, TRANK procedure applied to sort the set $M(Q,T)$ and output the results.

6 System Design and Functionality Analysis

We implemented the TWIX system using *Java 1.4.2* and ran our experimentations on an *Intel Xeon 2.4GHz* processor with *2GB* of main memory. The experimental evaluations were performed on synthetic and real XML datasets[11] to assess and evaluate the performance of TWIX. The synthetic datasets are generated by using XMark [37] using a scaling factor of 1. The XMark synthetic dataset is 113MB in size having 2,849,444 nodes with an average depth of 5, named XMark-1. We also used the `dblp` dataset which is 127 MB in size, including 5,682,094 nodes with an average depth of 2.9. For our experiments, we applied various randomly distributed modifications (subset and schema modification) on the DTD and document tree of the acquired datasets to create a collection of versioned XML documents. We first divided `dblp` into its corresponding entity categories: `<inproceedings>`, `<article>`, ..., `<www>`. Furthermore, three different random element modifications (e.g. renaming some of the `title` elements by `paper_title`) were performed on the corresponding `<inproceedings>` and `<article>` subsets of `dblp`. Table 2 provides the details of the incorporated modifications and the corresponding characteristics of each individual document. The following subsections describe each particular component of the system and analyze its functionality, studying the filtration *effectiveness* and *ranking accuracy*.

6.1 Effectiveness of Filtration Techniques

TWIX deploys various filtration techniques to reduce the size of the intermediate result set. Each of the inspected queries investigate a category of general queries as mentioned in the introduction section, including both PC and AD relationships as well as targeting various parts of the document tree. Table 3 depicts the effectiveness of the proposed filtrations for *exact search* of the sample twig queries of Figure 8 on the full `dblp` XML document. The second column of Table 3 shows the total number of potential matching instance combinations which is the size of the search space. Each filtration technique column entails the number filtered tuples and ration of the overall database left for further inspection. For instance, the second column of

[11] Available from University of Washington's XML Data Repository at *http://www.cs.washington.edu/research/xmldatasets/*

Dataset	Subsets	#	Replacement	Size
dblp	inproceedings	0	×	19.1 MB
		1	`$a` ⟶ `$a_$n`	20.3 MB
		2	`$t` ⟶ `$p_$t`	20.3 MB
		3	`$t` ⟶ `$p_$t` `$a` ⟶ `$a_$n`	20 MB
	article	0	×	9.63 MB
		1	`$a` ⟶ `$a_$n`	10.1 MB
		2	`$t` ⟶ `$p_$t`	9.87 MB
		3	`$t` ⟶ `$p_$t` `$a` ⟶ `$a_$n`	10 MB
	proceedings		×	1.36 MB
	incollection		×	326 KB
	phdthesis		×	16.9 KB
	mastersthesis		×	1.37 KB
	book		×	390 KB
	www		×	4.77 KB

Table 2: *Subset selection* and *modification* performed on `dblp` dataset. The × sign denotes *no modifications*, while `$a = author, $p = paper, $t = title,` and `$n = name`.

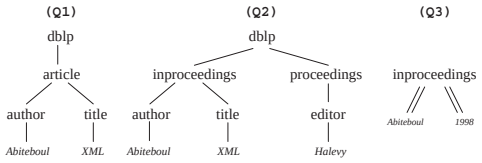

Figure 8: Twig queries used for filtration study.

HFT on `Q3` denotes that 52.29 % of the search space is pruned by simply using HFT for query `Q3`. In any of the cases, at least 47 % of the database is effectively pruned. Table 3 asserts the superior effectiveness of NCA relaxation filtration which prunes the database for a minimum filtration ratio of 2×10^{-3}% and up to 1.15×10^{-5}%.

6.2 Top-k Ranking Quality Assessment

Figure 9 depicts a visual perspective and analysis into the highly ranked matching instances returned by TWIX on the collection of 14 versioned `dblp` documents. The matching instances are listed in the decreasing order of their corresponding relevance ranks (rank 1 denoting the *best match*). A visual legend on the upper-right corner of each twig match depicts the parts of the result, *edges (structure: line)* and *nodes (content: circle)*, which match with the query and hence contributing to the ranking computation. Moreover, TRANK clusters the keywords into *conceptual groups* (e.g. `author`, `author_name`, ..., `authors`) covering the semantics of the keywords into account. This feature facilitates a flexible ranking scheme, not imposing much constraints on the user's knowledge of underlying database. The Queries are shown in the left portion of the figure followed by their corresponding top-5 ranked results. The internal nodes of the queries were particularly chosen to capture the *element label* differences among the XML documents of the collection. The first answer to `Q1`

Query		Horizontal Filtration (HFT)		Vertical Filtration (VFT)		NCA Relaxation	
	Total Combinations	Filtered Tuples	Ratio	Filtered Tuples	Ratio	Filtered Tuples	Ratio
Q1	1,605,990	779,603	48.54 %	826,347	51.45 %	32	2.00×e-3 %
Q2	35,331,780	26,545,044	75.13 %	8,786,220	24.86 %	9	2.54×e-5 %
Q3	52,027,684	27,204,742	52.29 %	24,822,926	47.70 %	6	1.15×e-5 %

Table 3: Filtration Comparison.

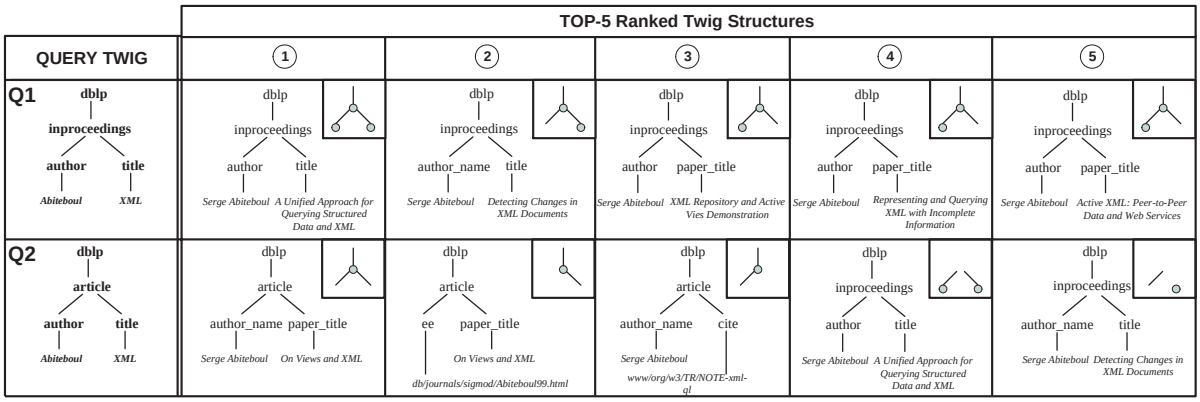

Figure 9: TOP-5 ranked twig answers to the given queries.

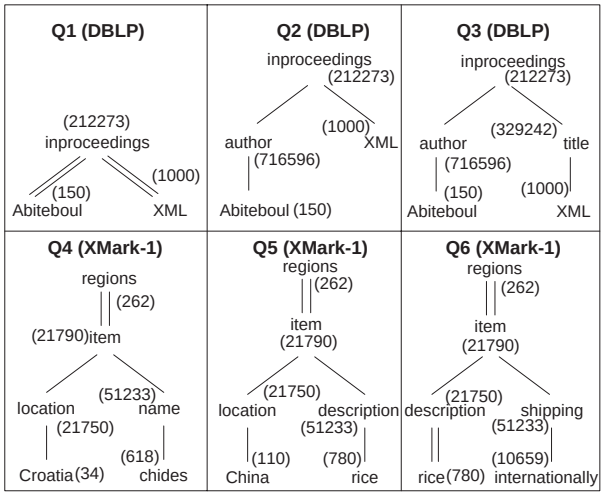

Figure 10: Twig queries used in timing analysis.

is an exact match and the similarity of the matches decrease traversing through the ranked matching list. However, for Q2, the first answer is an approximate match. Going down the ranking results, validates the quality of TRANK in distinguishing matching instances from each other.

6.3 Response Time Analysis

We compared the performance of TWIX against TwigStack[6][12]. The queries and experimental results are shown in Figure 10 and Figure 11, respectively. The numbers associated with the nodes in Figure 10 denote the selectivity of corresponding ELELEMTN entries. As

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

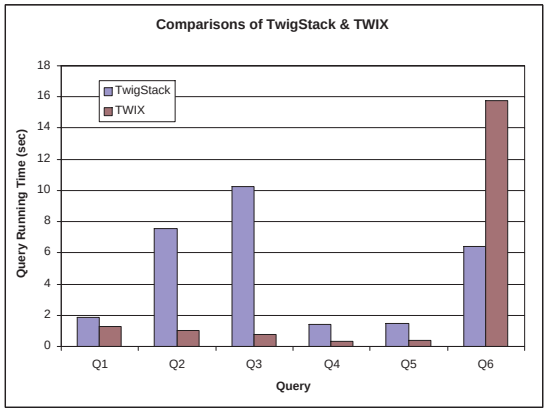

Figure 11: Query Response Time of TwigStack vs. TWIX.

depicted in 11, TWIX outperforms TwigStack specially when the query is highly selective for up 5 times faster running time (on Q3). The efficiency of TWIX for *highly selective* queries is because of the effectiveness of the bottom-up space reduction employed by its filtration phases. As a result, a smaller number of intermediate results would be left for further examination. In contrast, the frequency of the internal ELEMENT nodes plays an important role in TwigStack. High-frequency internal structure element node are frequently encountered in real datasets (e.g. author in dblp).

7 Conclusion

This paper proposed an efficient system design and implementation, named TWIX, for labeling, matching and ranking of 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 induced subtree on top of each matched instance. Various filtration techniques based on DTD relevance and horizontal/vertical differences of the node extents were deployed. TWIX also proposes a holistic ranking technique for twig queries, named TRANK. TRANK inspects each individual path of the potential twig patterns and combines keyword relevance ranking along with minimum twig tree distance for effective ranking of the twig matches. TWIX also provides maximum flexibility on the user side by pushing the complexity of the process to the core of the TWIX system. TWIX system is an ongoing effort and is augmented with improvements progressively. Experimental results demonstrate promising filtration ratio incorporating the proposed filtration techniques to eliminate false positives and semantically meaningful ranking of the results.

References

[1] 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).

[2] S. Al-Khalifa et al., Querying Structured Text in an XML Database. SIGMOD, 4–15 (2003).

[3] S. Alstrup et al., Nearest common ancestors: A survey and a new distributed algorithm. Theory of Computing Systems **37**, 441–456 (2004).

[4] S. Amer-Yahia, L.V.S. Lakshmanan and S. Pandit, FleXPath: Flexible Structure and Full-Text Querying for XML. SIGMOD, 83–94 (2004).

[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] D. Chamberlin, J. Robie and D. Florescu, Quilt: An XML query language for heterogeneous data sources. WebDB (Informal Proceedings), 53–62 (2001).

[8] D. Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon and Mugur Stefanescu, XQuery: A Query Language for XML. W3C Working Draft, http://www.w3.org/TR/xquery (2001).

[9] S. Chawathe and H. Garcia-Molina, Meaningful Change Detection in Structured Data. SIGMOD, 26–37 (1997).

[10] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras and C. Zaniolo, Efficient Structural Joins on Indexed XML Documents. VLDB, 263–274 (2002).

[11] J. Clark, XSL Transformations (XSLT), Version 1.1 W3C Recommendations, http://www.w3.org/TR/xslt11 (2001).

[12] B. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon, A Fast Index for Semistructured Data. VLDB, 341–350 (2001).

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, XML-QL: A query language for XML. QL, http://www.w3.org/TR/NOTE-xml-ql (1998).

[14] P.F. Dietz et al., Two Algorithms for Maintaining Order in a List. In Proc. of 19th Annual ACM Symp. on Theory Comput. (STOC), 365–372 (1987).

[15] D. Florescu and D. Kossman, Storing and querying XML data using an RDBMS. IEEE Data Engineering Bulletin **22(3)**, 27–34 (1999).

[16] T. Grust, Accelerating XPath location steps. SIGMOD, 109–120 (2002).

[17] L. Guo, J. Shanmugasundaram, K.S. Beyer and E.J. Shekita, Efficient Inverted Lists and Query Algorithms for Structured Value Ranking in Update-Intensive Relational Databases. ICDE, (2005).

[18] L. Guo, F. Shao, C. Botev and J. Shanmugasundaram, XRANK: Ranked Keyword Search over XML Documents. SIGMOD, 16–27 (2003).

[19] H. Jiang, W. Wang, H. Lu and J. Xu Yu, Holistic Twig Joins on Indexed XML Documents. VLDB, 273–284 (2003).

[20] R. Kaushik et al., On the Integration of Structure Indexes and Inverted Lists. SIGMOD, 779–790 (2004).

[21] R. Kaushik, P. Shenoy, P. Bohannon and E. Gudes, Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. ICDE, 129–140 (2002).

[22] DBLP Bibliography Server, *http://dblp.uni-trier.de/*.

[23] Q. Li and B. Moon, Indexing and Querying XML Data for Regular Path Expressions. VLDB, 361–370 (2001).

[24] 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).

[25] A. Marian, S. Amer-Yahia, N. Koudas and D. Srivastava, Adaptive Processing of Top-K Queries in XML. ICDE, (2005).

[26] J.F. Naughton et al., The Niagara Internet query system. IEEE Data Eng. Bulletin **24(2)**, 27–33 (2001).

[27] S.B. Needleman and C.D. Wunsch, A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of two Proteins. J. Mol. Biol. **48**, 443–453 (1970).

[28] N. Polyzotis, M. Garofalakis and Y. Ioannidis, Approximate XML Query Answers. SIGMOD, 263–274 (2004).

[29] N. Polyzotis, M.N. Garofalakis and Y.E. Ioannidis, Selectivity Estimation for XML Twigs. ICDE, 264–275 (2004).

[30] P. Rao, B. Moon, PRIX: Indexing And Querying XML Using Prüfer Sequences. ICDE, 288–300 (2004).

[31] J. Shanmugasundaram et al., Efficiently Publishing Relational Data as XML Documents. VLDB, 133–154 (2000).

[32] F. Weigel, H. Meuss, K.U. Schulz and F. Bry, Content and Structure in Indexing and Ranking XML. WebDB, 67–72 (2004).

[33] 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).

[34] Xyleme, Available from *http://www.xyleme.com*.

[35] P. Zezula, F. Mandreoli and R. Martoglia, Tree Signatures and Unordered XML Pattern Matching. SOFSEM, 122–139 (2004).

[36] P. Zezula, G. Amato, F. Debole and F. Rabitti, Tree Signatures for XML Querying and Navigation. Xsym, 149–163 (2003).

[37] A. R. Schmidt et al., The XML Benchmark Project. Technical Report INS-R0103, CWI (2001).

[38] K. Zhang and D. Shasha, Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM J. Comput. **18(6)**, 1245–1262 (1989).