PRIX: Indexing And Querying XML Using Prüfer Sequences *

Praveen Rao *Bongki Moon*

Department of Computer Science
University of Arizona, Tucson, AZ 85721
E-mail: {rpraveen,bkmoon}@cs.arizona.edu

Abstract

We propose a new way of indexing XML documents and processing twig patterns in an XML database. Every XML document in the database can be transformed into a sequence of labels by Prüfer's method that constructs a one-to-one correspondence between trees and sequences. During query processing, a twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching on the set of sequences in the database, and performing a series of refinement phases that we have developed, we can find all the occurrences of a twig pattern in the database. Our approach allows holistic processing of a twig pattern without breaking the twig into root-to-leaf paths and processing these paths individually. Furthermore, we show in the paper that all correct answers are found without any false dismissals or false alarms. Experimental results demonstrate the performance benefits of our proposed techniques.

1 Introduction

Since the extensible markup language XML emerged as a new standard for information representation and exchange on the Internet [4], the problem of storing, indexing and querying XML documents has been among the major issues of database research. As the relationships between elements in an XML document are defined by nested structures, XML documents are often modeled as trees whose nodes are labeled with tags, and queries are formulated to retrieve documents by specifying both their structures and values. In most of the XML query languages (*e.g.*, XPath [2] and XQuery [3]), structures of XML documents are typically expressed by linear paths or twig patterns, while values of XML elements are used as part of selection predicates. For example, a path expression given in XPath syntax

```
book[author//name="John"]/title
```

qualifies XML documents by specifying a twig pattern composed of four elements `book`, `author`, `name` and `title` in an XML document, and a value-based selection predicate `name="John"`.

Queries with a path expression have been one of the major foci of research for indexing and querying XML documents. In the past few years, there have been two main thrusts of research activities for processing path join queries for retrieving XML data, namely, approaches based on *structural index* and *numbering schemes*. The approaches based on the structural index facilitate traversing through the hierarchy of XML documents by referencing the structural information of the documents (*e.g.*, dataguide [8], representative objects [16], 1-index [15], approximate path summary [12], F&B index [11]). These structural indexes can help reduce the search space for processing path or twig queries.

The other class of approaches are based on a form of numbering scheme that encodes each element by its positional information within the hierarchy of an XML document it belongs to. Most of the numbering schemes reported in the literature are designed by a tree-traversal order (*e.g.*, pre-and-postorder [7], extended preorder [13]) or textual positions of start and end tags (*e.g.*, containment property [21], absolute region coordinate [20]). If such a numbering scheme is embedded in the labeled trees of XML documents, the structural relationship (such as ancestor-descendant) between a pair of elements can be determined quickly without traversing an entire tree. Several join algorithms have been developed to take advantage of this extraordinary opportunity to efficiently process path and twig queries [1, 5, 6, 9, 13, 21]. In particular, it has been shown that *PathStack* and *TwigStack* algorithms [5] are optimal for processing path and twig queries in that the processing cost is *linearly* proportional to the sum of input data and query results.

Most of the previous approaches based on numbering schemes, however, process a twig query by first processing each of the root-to-leaf paths in the twig separately and then merging the results from the individual paths. In an effort to further optimize twig query processing without breaking a twig and merging the results, we propose a new way of indexing XML documents and finding twig patterns in an XML database. We have developed a system called **PRIX** (**PR***üfer sequences for* **I***ndexing* **X***ML*) for indexing XML documents

*This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037), NSF Grant No. IIS-0100436, and NSF Research Infrastructure program EIA-0080123. It was also supported by the Prop 301 Fund from the State of Arizona, and Korea Science and Engineering Foundation (KOSEF). The authors assume all responsibility for the contents of the paper.

and processing twig queries.[1] In our PRIX system, every XML document in the database is transformed into a sequence of labels by Prüfer's method that constructs a one-to-one correspondence between trees and sequences. During query processing, a twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching against the indexed sequences in the database, and by performing a series of refinement phases that we have developed, we can find all the occurrences of a twig pattern in the database. Our work was developed independently of and differs considerably from the new indexing method called ViST [19], which also converts trees into sequences.

The main contributions of this paper are summarized as follows.

- We propose a new idea of transforming XML documents into sequences by Prüfer's method. We show that twig matches can be found by performing subsequence match on the set of sequences and by performing a series of refinement phases. We also show that our approach returns correct answers without false alarms and false dismissals.
- Our approach allows holistic processing of twig queries without breaking a twig into root-to-leaf paths and processing them individually. Additionally, our tree-to-sequence transformation guarantees a worst-case bound on the index size that is linear in the total number of nodes in the XML document trees.
- We have developed effective optimizations to speed up the subsequence match phase during query processing.

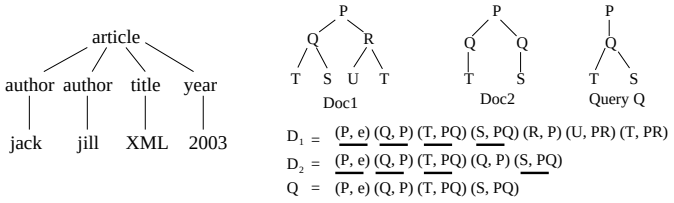
The rest of this paper is organized as follows. In Section 2 we discuss the background and motivations of our work. In Section 3 we present an architectural overview of the PRIX system. Section 4 and Section 5 provide the necessary theoretical background and describe the implementation issues of the PRIX system. In Section 6 we present our experimental results. Lastly Section 7 summarizes the contributions of this paper.

2. Background and Motivations

XML documents can be modeled as ordered labeled trees as shown in Figure 1(a). Each node in a tree corresponds to an element or a value. Values are represented by character data (CDATA, PCDATA) and occur at the leaf nodes. The tree edges represent a relationship between two elements or between an element and a value. Each element can have a list of (attribute, value) pairs associated with it. An attribute is usually represented as a subelement of an element. Hence, no special distinction will be made between elements and attributes in subsequent discussions in this paper.

Recently, much research effort has been focused on indexing and querying XML documents. Finding all occurrences of a query pattern in XML documents is one of the core operations in XML databases. Below we will briefly describe two

[1] PRIX is pronounced without the 'x' like French word Grand Prix.



(a) An XML document (b) False Alarms By ViST

Figure 1. A sample XML document and illustration of false alarms by ViST

of the recent contributions made for XML pattern matching: TwigStack [5] and ViST [19]. We will then discuss some of their drawbacks to motivate our proposed approach.

TwigStack Algorithms Bruno *et al.* have proposed optimal XML pattern matching algorithms [5]. These stack-based algorithms process input streams of element instances whose tag appears in the query twig. TwigStack and PathStack algorithms operate on the positional representation of the element instances to find twig matches. A variant of TwigStack algorithm (denoted hereinafter by TwigStackXB) uses XB-Trees to speed up processing when the input lists are long. The XB-Trees are useful in skipping sections of the input lists without missing any matches.

However, there are some limitations of TwigStackXB. The effectiveness of skipping data depends on the distribution of the matches in the input lists. If the matches are scattered all over the dataset, then the TwigStackXB algorithm drills down to lower regions of the tree (including leaves) in order to avoid missing matches. Another drawback of the TwigStack and TwigStackXB algorithms is that it suffers from sub-optimality for parent-child relationships in the query twig. The algorithm might produce a partial match of a path of the twig that cannot be combined with any other partial match of another path of the twig. For example, consider a query twig with 3 nodes and 2 branches containing parent-child relationships between (P, Q) and between (P, R). The algorithm will match a pattern in the data where P is a common ancestor of Q and R but is not their parent. This match will be discarded in the merge post-processing step of the algorithm. However, the cost of post-processing may not always be trivial.

ViST Wang *et al.* have proposed a new method called ViST that transforms XML data trees and twig queries into structure-encoded sequences [19]. The structure-encoded sequence is a two-dimensional sequence of (symbol, prefix) pairs $\{(a_1, p_1), (a_2, p_2), ..., (a_n, p_n)\}$ where a_i represents a node in the XML document tree, and p_i represents the path from the root node to node a_i. The nodes $a_1, a_2, ..., a_n$ are in preorder. ViST performs subsequence matching on the structure-encoded sequences to find twig patterns in XML documents. These sequences are stored in a trie.

One of the imminent drawbacks of the tree transformation used by ViST is that the worst-case storage requirement for a B^+-tree index named D-Ancestorship is higher than linear in

the total number of elements of the XML documents. For example, consider a unary tree with n nodes. In this case, the total size of the structure-encoded sequence is $O(n^2)$. Thus the D-Ancestorship index requires $O(n^2)$ space to store all the (symbol, prefix) keys. Another drawback of ViST is that the query processing strategy may result in *false alarms*. Figure 1(b) illustrates such a case. The structure-encoded sequence of the query twig Q is a subsequence of the structure-encoded sequence of Doc_1 and Doc_2. However, the twig pattern Q occurs only in Doc_1, and the match detected in Doc_2 is a false alarm.

Our Motivations The key motivations of our work are (1) to develop a method that allows *holistic processing* of twig queries without breaking a twig into root-to-leaf paths and processing them individually, (2) to construct a tree-to-sequence transformation such that the total storage requirement is *linear* in the total number of tree nodes, and (3) to transform trees to sequences and index them so that *similarity in documents* can be taken advantage of to reduce the total amount of data that needs to be searched during query processing.

3. Overview of PRIX Approach

In this section, we present the Prüfer's method that constructs a one-to-one correspondence between trees and sequences, and describe how Prüfer's sequences are used for indexing XML data and processing twig queries in the PRIX system.

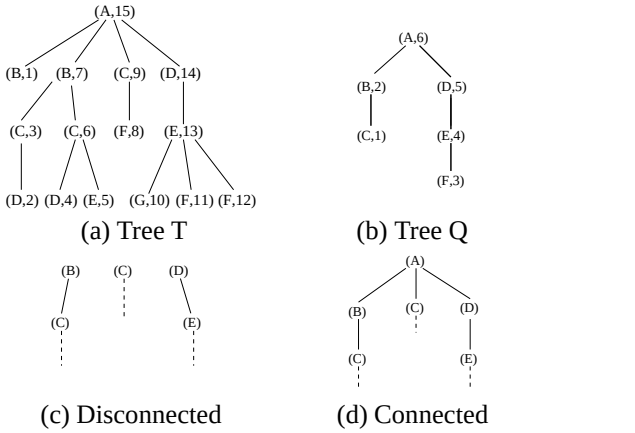
3.1. Prüfer Sequences for Labeled Trees

Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labeled tree and a sequence by removing nodes from the tree one at a time [17]. The algorithm to construct a sequence from tree T_n with n nodes labeled from 1 to n works as follows. From T_n, delete a leaf with the smallest label to form a smaller tree T_{n-1}. Let a_1 denote the label of the node that was the parent of the deleted node. Repeat this process on T_{n-1} to determine a_2 (the parent of the next node to be deleted), and continue until only two nodes joined by an edge are left. The sequence $(a_1, a_2, a_3, ..., a_{n-2})$ is called the Prüfer sequence of tree T_n. From the sequence $(a_1, a_2, a_3, ..., a_{n-2})$, the original tree T_n can be reconstructed.

The length of the Prüfer sequence of tree T_n is $n - 2$. In our PRIX approach, however, we construct a Prüfer sequence of length $n - 1$ for T_n by continuing the deletion of nodes till only one node is left. (The one-to-one correspondence is still preserved). This modified construction simplifies the proofs of the lemmas and theorems presented in Section 4.

3.2. Indexing by Transforming XML Documents into Prüfer Sequences

In the discussions to follow, each XML document is represented by a labeled tree such that each node is associated with its element tag and a number. For example, in Figure 2(a), the root element of the XML document has $(A, 15)$ as its tag-number pair. Any numbering scheme can be used to label an XML document tree as long as it associates each node in the tree with a unique number between one and the total number of nodes. This guarantees a one-to-one mapping between the tree and the sequence. In our PRIX system, without loss of generality, we have chosen to use postorder to uniquely number tree nodes, and will continue further discussions based on the postorder numbering scheme.



(a) Tree T (b) Tree Q

(c) Disconnected (d) Connected

Figure 2. XML document tree and query twig

With tree nodes labeled with unique postorder numbers, a Prüfer sequence can be constructed for a given XML document using the node removal method described in Section 3.1. This sequence consists entirely of postorder numbers and is called *NPS (Numbered Prüfer sequence)*. If each number in an NPS is replaced by its corresponding tag, a new sequence that consists of XML tags can be constructed. We call this sequence *LPS (Labeled Prüfer sequence)*. [2] The set of NPS's are stored in the database together with their unique document identifiers.

Example 1 *In Figure 2(a), tree T has LPS(T) = A C B C C B A C A E E E D A, and NPS(T) = 15 3 7 6 6 7 15 9 15 13 13 13 14 15.* □

3.3. Processing Twig Queries by Prüfer Sequences

A query twig is transformed into its Prüfer sequence like XML documents. Non-matches are filtered out by subsequence matching on the indexed sequences, and twig matches are then found by applying a series of refinement strategies. These filtering and refinement phases are described in Section 4.

Figure 3 shows an architectural overview of the indexing and query processing units in PRIX as described in Section 3.2 and Section 3.3. With this high level overview of our system, we shall now move on to explain the process of finding twig matches.

4. Finding Twig Matches

To simplify our presentation of concepts in this section, we shall use the notations listed in Table 1. Formally the problem

[2] Occasionally we will refer to an NPS as a *postorder number sequence* of an LPS

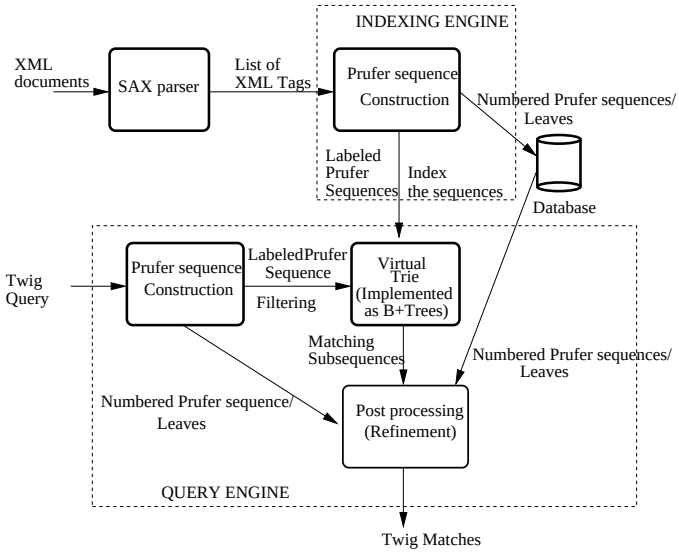


Figure 3. Architectural Overview of PRIX

Symbol	Description
Q	Query twig
Δ	A collection of XML documents
Γ	A set of Labeled Prüfer sequences of Δ
Θ	A set of subsequences in Γ that are identical
LPS(T)	Labeled Prüfer sequence of tree T
NPS(T)	Numbered Prüfer sequence of tree T

Table 1. Notations used

of finding twig matches can be stated as follows: *Given a collection of XML documents Δ and a query twig Q, report all the occurrences of twig Q in Δ.* In this paper, we restrict to handling twig Q with equality predicates only.

We will initially deal with the problem of finding all occurrences of twig Q without wildcards `'//'` and `'*'`. Later in Section 4.5, we explain how query twigs with wildcards can be processed. In addition, we will first address the problem of finding ordered twig matches. Later in Section 5.7, we explain how unordered twig matches can be found.

Finding twig matches in the PRIX system involves a series of filtering and refinement phases, namely (1) *filtering by subsequence matching*, (2) *refinement by connectedness*, (3) *refinement by structure* and (4) *refinement by leaf nodes*. Due to the space limitations, proofs of lemmas and theorems are omitted and provided in the extended version of this paper [18].

4.1. Filtering by Subsequence Matching

The filtering phase involves subsequence matching. The classical definition of a subsequence is stated below.

Definition 1 *A subsequence is any string that can be obtained by deleting zero or more symbols from a given string.*

In this phase, given a query twig Q, we find all the subsequences in Γ (the set of LPS's) that match LPS(Q). We shall discuss the significance of subsequence matching using the following lemma and theorem.

Lemma 1 *Given a tree T with n nodes, numbered from 1 to n in postorder, the node deleted the i^{th} time during Prüfer sequence construction is the node numbered i.*

As a result, if a and b are two nodes of a tree such that a has a smaller postorder number than b, then node a is deleted before node b during Prüfer sequence construction. Based on Lemma 1, we can state the following theorem.

Theorem 1 *If tree Q is a subgraph of tree T, then LPS(Q) is a subsequence of LPS(T).*

From Theorem 1, it is evident that by finding every subsequence in Γ that matches LPS(Q), we are guaranteed to have no **false dismissals**.

Example 2 *Consider trees T and Q in Figure 2(a) and Figure 2(b). T has LPS(T) = A C B C C B A C A E E E D A and NPS(T) = 15 3 7 6 6 7 15 9 15 13 13 13 14 15. Q has LPS(Q) = B A E D A and NPS(Q) = 2 6 4 5 6. Q is a (labeled) subgraph of T, and LPS(Q) matches a subsequence S of LPS(T) at positions (6, 7, 11, 13, 14). The postorder number sequence of subsequence S is 7 15 13 14 15. Note that there may be more than one subsequence in LPS(T) that matches LPS(Q).* □

4.2. Refinement by Connectedness

The subsequences matched during the filtering phase are further examined for the property of *connectedness*. This is because, only for some of the subsequences, all the labels in the subsequence correspond to nodes that are connected (representing a tree) in the tree. Formally we state a **necessary condition** for any subsequence S to satisfy the connectedness property.

Theorem 2 *Given a tree T, let N_T be the NPS of T. Let S be a subsequence of LPS(T) and let N be the postorder number sequence of S. Then the tree nodes in T corresponding to the labels of S are connected (representing a tree)* **only if** *for every element of N, i.e., N_i, if $N_i \neq max(N_1, N_2, ..., N_{|N|})$ and $\neg\exists(j > i)$ s.t. $N_j = N_i$ then $N_{i+1} = N_T[N_i]$.*

The intuition for the above theorem is as follows. Let i be the index of the last occurrence of a postorder number n in an NPS. This last occurrence is a result of deletion of the last child of n during Prüfer sequence construction. Hence the next child to be deleted (based on Lemma 1) is the node n itself. Hence the number at the $(i+1)^{th}$ index in the NPS, say m, is the postorder number of the parent of node n. Thus n followed by m indicates that there is an edge between node m and node n.

Example 3 *Consider two subsequences S_A and S_B of LPS(T) where T is the tree in Figure 2(a). Let S_A be C B C E D whose postorder number sequence N_A is 3 7 9 13 14. Let S_B be C B A C A E D A whose postorder number sequence N_B is 3 7 15 9 15 13 14 15. Let N_T be the NPS of T. Then N_T is 15 3 7 6 6 7 15 9 15 13 13 13 14 15. The nodes represented by labels of S_A form a disconnected graph as shown in Figure 2(c). In*

this case, max($N_{A1}, N_{A2}, ..., N_{A5}$) = 14. The last occurrence of postorder number 7 in N_A is at the 2^{nd} position since there is no index $j > 2$ such that $N_{Aj} = 7$. However N_{A2} is not followed by $N_T[7]$, i.e., $N_{A3} \neq 15$. Hence the necessary condition of Theorem 2 is not satisfied. The nodes represented by elements of S_B represent a tree as shown in Figure 2(d) because the necessary condition of Theorem 2 is satisfied. □

We shall refer to sequences that satisfy Theorem 2 as *tree sequences*.

4.3. Refinement by Twig Structure

The tree sequences obtained in the previous refinement phase are further refined based on the query twig structure. In this phase we would like to determine if the structure of the tree represented by a tree sequence matches the query twig structure.

4.3.1. Notion of Gaps Between Tree Nodes. Before we delve into details of refinement by structure, we shall first introduce the notion of *gap* between two tree nodes and *gap consistency* and *frequency consistency* between two tree sequences.

Definition 2 *The gap between two nodes **a** and **b** in a tree is defined as the difference between the postorder numbers of the nodes **a** and **b**.*

The gap between tree nodes can be computed using the NPS of the tree.

Definition 3 *Tree sequence A is said to be gap consistent with respect to tree sequence B if*

1. *A and B have the same length n,*
2. *For every pair of adjacent elements in A and the corresponding adjacent elements in B, their gaps, g_A and g_B have the same sign, and if $|g_A| > 0$ then $|g_A| \leq |g_B|$, else $g_A = g_B = 0$.*

Note that gap consistency is not a symmetric relation.

Example 4 *Consider the tree T in Figure 2(a). LPS(T) = A C B C C B A C A E E E D A, and NPS(T) = 15 3 7 6 6 7 15 9 15 13 13 13 14 15. Let S_1 = B A E E A be a subsequence of LPS(T) and let N_{S_1} = 7 15 13 13 15 be the postorder number sequence of S_1. Let S_2 = B A E E A, and let N_{S_2} = 2 7 6 6 7 be the postorder number sequence of S_2. Then S_2 is gap consistent with S_1 because the gap between*

- *the 1st pair of elements in S_2 is -5,*
- *the 1st pair of elements in S_1 is -8,*
- *the 2nd pair of elements in S_2 is 1,*
- *the 2nd pair of elements in S_1 is 2,*
- *the 3rd pair of elements in S_2 is 0,*
- *the 3rd pair of elements in S_1 is 0, and so on.* □

Intuitively, the gap between two nodes in a data tree gives an idea of how many nodes are encountered during postorder traversal between these two nodes. Similar is the case with the nodes of a query twig. If more nodes are traversed in the query twig as compared to the data twig, then this indicates that there is a structural difference between the data and the query twig. This concept forms the basis of Theorem 3 that states a necessary and sufficient condition for match by twig structure.

Another key observation that will be used in Theorem 3 is the following. The number of times a number n occurs in an NPS indicates the number of child nodes of n in the tree, and the positions that n occurs in the NPS depend on the subtrees rooted at node n. We formalize this observation by defining a property called *frequency consistency*.

Definition 4 *Tree sequences A and B are frequency consistent if*

1. *A and B have the same length n,*
2. *Let N_A and N_B be the postorder number sequences of A and B respectively. Let n_{Ai} and n_{Bi} be the i^{th} element in N_A and N_B respectively. For every i from 1 to n, n_{Ai} occurs k times in N_A at positions $\{p_1, p_2, ..., p_k\}$, iff n_{Bi} occurs k times in N_B at positions $\{p_1, p_2, ..., p_k\}$.*

Note that frequency consistency is an equivalence relation.

Example 5 *In Example 4, sequences S_1 and S_2 are frequency consistent. The 1^{st} element in N_{S_1} (7) occurs once at position (1). The 1^{st} element in N_{S_2} (2) also occurs once at position (1). The 2^{nd} element in N_{S_1} (15) occurs at positions (2, 5). The 2^{nd} element in N_{S_2} (7) also occurs at positions (2, 5). Similar is the case with the remaining elements in N_{S_1} and N_{S_2}.* □

It should be noted that the LPS of a tree contains only the non-leaf node labels. Thus, in addition to the LPS and NPS, the label and postorder number of every leaf node should be stored in the database. Since the LPS of a tree contains only non-leaf node labels, filtering by subsequence matching followed by refinement by connectedness and structure can only find twig matches in the data tree whose tree structure is the same as the query tree and whose non-leaf node labels match the non-leaf node labels of the query twig. Let us call such matches as *partial twig matches*. To find a *complete twig match*, the leaf node *labels* of a partially matched twig in the data should be matched with the leaf node *labels* of the query twig. This is explained in Section 4.4.

We now state a necessary and sufficient condition for a *partial twig match*.

Theorem 3 *Tree Q has a partial twig match in tree T iff*

1. *LPS(Q) matches a subsequence S of LPS(T) such that S is a tree sequence, and*
2. *LPS(Q) is gap consistent and frequency consistent with subsequence S.*

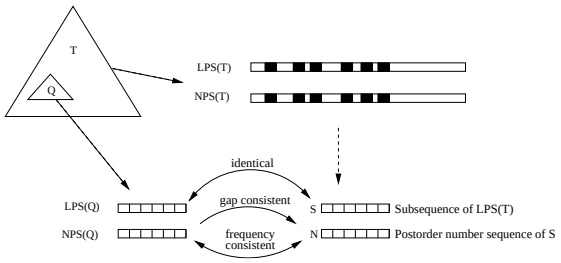


Figure 4. Data and Query Sequences

The different relationships between the data and query sequences as described in this section are illustrated in Figure 4. Consider the tree T (XML document) and its subgraph tree Q (query twig) in the figure. The dark regions in LPS(T) and NPS(T) correspond to the deletion of nodes in T during Prüfer sequence construction that are also in Q (except root of Q). The dark regions in LPS(T) and NPS(T) form sequences S and N respectively. From the lemmas and theorems described in Section 4.1, Section 4.2 and Section 4.3, we can conclude the following: LPS(Q) and S are identical, NPS(Q) is *gap consistent* with N, and NPS(Q) and N are *frequency consistent*.

4.4. Refinement by Matching Leaf Nodes

In the final refinement phase, the leaf node labels of the query twig are tested with the leaf node labels of partially matched twigs in the data to find *complete twig matches*.

Example 6 *The leaf nodes of tree T in Figure 2 i.e.,*$(D, 2), (D, 4), (E, 5), (G, 10), (F, 11), (F, 12)$ *are stored in the database. Let tree Q (Figure 2(b)) be the query twig. LPS(Q) matches a subsequence S = B A E D A in LPS(T) at positions P = (3, 7, 11, 13, 14). The postorder number sequence of S is N = 7 15 13 14 15. LPS(Q) is gap consistent and frequency consistent with S. We can match leaf (F, 3) in Q as follows. Since the leaf has postorder number 3, its parent node matches the node numbered* 13 *(i.e., the* 3^{rd} *element of N) in the data tree. Also because this node numbered* 13 *occurs at the* 11^{th} *position (*3^{rd} *element in P) in LPS(T), it may have a leaf (F, 11). And indeed, we have (F, 11) in the leaf node list of T. Similarly we can match the leaf (C, 1) of Q. The parent of (C, 1) in Q matches node* 7 *(*1^{st} *element in N) at position* 3 *in NPS(T). Hence the child of node* 7 *in T, i.e., node* 3 *matches leaf (C, 1), except that the labels may not match (partial twig match). Since there are no nodes with number* 3 *in the leaf list of T, we search LPS(T) and NPS(T) to find (C, 3) in T. And indeed we have this pair at the* 2^{nd} *position in LPS/NPS of T.* □

However, this refinement phase can be eliminated by special treatment of leaf nodes in the query twig and the data trees. The key idea is to make the leaf nodes of the query twig and the data trees appear in their LPS's, so that all the nodes of the query twig are examined during subsequence matching and refinement by connectedness and structure phases. Due to lack of space, we do not discuss the details in this paper and refer our readers to the extended version [18].

4.5. Processing Wildcards

We shall explain the processing of wildcards `'//'` and `'*'` with the following example.

Example 7 *Let us find the query pattern Q=//A//C/D in tree T (in Figure 2(a)). Q is transformed to its Prüfer sequences by ignoring the wildcards. As a result, LPS(Q) = C A, and NPS(Q) = 2 3. The wildcard at the beginning of the query is handled by our current method as it allows finding occurrences of a query tree anywhere in the data tree. To process the wildcard in the middle of the query, we do a simple modification to the refinement-by-connectedness phase. LPS(Q) matches a subsequence S = C A at positions (2, 7) in LPS(T). The postorder number sequence of S is N = 3 15. Based on Theorem 2, this subsequence would be discarded as the last occurrence of 3 in N is not followed by 7 (parent of node numbered 3 in T). To avoid this, we check instead if the last occurrence of node 3 in N can lead to node 15 (15 follows 3 in N) by following a series of edges in T. Recall that the i^{th} element in an NPS is the postorder number of the parent of node i in a tree (Lemma 1). Let $n_0 = 3$ and let N_T be NPS(T). We recursively check if n_1 $(= N_T[n_0])$ equals 15, then if n_2 $(= N_T[n_1])$ equals 15 and so on until for some i, n_{i+1} $(= N_T[n_i])$ equals 15. In the above example, we find a match at i = 2. For processing wildcard '*', we simply test whether the match is found at i = 2. Thus all the subsequences that pass the above test will move to the next phase.* □

5. Implementation Issues in the PRIX System

Given the theoretical background in Section 4, we shall move on to explain the implementation aspects of the PRIX system.

5.1. Building Prüfer Sequences

In the PRIX system, Prüfer sequences are constructed for XML document trees (with nodes numbered in postorder) using the method described in Section 3.1. Our proposed tree-to-sequence transformation causes the nodes at the lower levels of the tree to be deleted first. This results in a bottom-up transformation of the tree. We shall show in our experiments that the bottom-up transformation is useful to process query twigs efficiently.

5.2. Indexing Sequences Using B$^+$-trees

The set of Labeled Prüfer sequences of the XML documents are indexed in order to support fast subsequence matching for query processing. Maintaining an in-memory index for the sequences like a trie is unsuitable, as the index size grows linearly with the total length of the sequences. In essence, we would like to build an efficient disk-based index.

In fact, Prüfer sequences can be indexed using any good method for indexing strings. In the current version of our PRIX system, we index Labeled Prüfer sequences using B$^+$-trees in the similar way that Wang *et al.* build a virtual trie using B$^+$-trees [19].

5.2.1. Virtual Trie. We shall briefly explain the process of indexing sequences using a virtual trie. Essentially, we provide positional representations for the nodes in the trie by labeling them with ranges. Each node in the trie is labeled with a range `(LeftPos, RightPos)` such that the containment property is satisfied [13]. Typically, the root node can be labeled with a range $(1, MAX\_INT)$. The child nodes of the root can be labeled with subranges such that these subranges are disjoint and are completely contained in $(1, MAX\_INT)$. This containment property is recursively satisfied at every non-leaf node in the trie. We can then obtain all the descendants of any given node A by performing a range query that finds nodes whose `LeftPos` falls within the `(LeftPos, RightPos)` range of node A.

In the PRIX system, for each element tag e, we build a B$^+$-tree that indexes the positional representation of every occurrence of element e in the trie using its `LeftPos` as the key. We call this index *Trie-Symbol index*. In addition, we store each document (tree) identifier in a separate B$^+$-tree and index it using the `LeftPos` of the node where the LPS ends in the virtual trie as the key. This index is called *Docid index*. Note that it is sufficient to store only the LPS's in the virtual trie. The suffixes of the LPS's need not be indexed at all, since all the subsequences can be found by performing range queries on the *Trie-Symbol indexes* as described in Section 5.3.

ViST proposed a dynamic labeling scheme that can assign number ranges without building a physical trie (hence the name virtual trie) on the set of sequences [19]. However, this dynamic labeling scheme suffers from *scope underflows* [19] for long sequences and large alphabet sizes, which makes it difficult to implement. In order to reduce the scope underflows, we *pre-allocate* the number ranges for a small subset of nodes in the trie. The remaining nodes are assigned ranges using the dynamic labeling scheme. In order to do so, we build an in-memory trie for all the prefixes of the sequences of length α (where α is a small number compared to the actual length of the sequences). A node in this in-memory trie is allocated a number range based on the *frequency* and *length* of the sequences whose prefixes share that node.

5.2.2. Space Complexity. The size of a trie grows linearly with the total length of the sequences stored in it. In the PRIX system, the length of a Prüfer sequence is linear in the number of nodes in the tree. Hence the index size is linear in the total number of tree nodes, while ViST does not guarantee a linear worst-case bound on the index size. (Refer to Section 2.)

5.3. Filtering by Subsequence Matching

Let $Q_s = Q_{s1}Q_{s2}...Q_{sk}$ (a sequence of length k) denote the LPS of a query twig Q. The process of finding all occurrences of Q_s using the Trie-Symbol indexes is shown in Algorithm 1. The algorithm is invoked by *FindSubsequence*$(Q_s, 1, 0, MAX\_INT)$. A range query in the open interval (q_l, q_r) is performed on the $T_{Q_{si}}$ (Trie-Symbol index of Q_{si}) (line 1). For every node id r returned from the range query (line 1), if the sequence Q_s is found then all the documents in the closed interval $[r_l, r_r]$ are fetched from

Algorithm 1: Filtering Algorithm

Input: $\{Q_s, i, (q_l, q_r)\}$: Q_s is a query sequence; index i; (q_l, q_r) is a range;
Output: (D, S); D is a set of document (tree) identifiers; S denotes the positions of subsequence match;

procedure *FindSubsequence*$(Q_s, i, (q_l, q_r))$
1: $R = RangeQuery(T_{Q_{si}}, (q_l, q_r))$;
2: **foreach** *r in R* **do**
3: $S_i = Level(r)$;
4: **if** $(i = |Q_s|)$ **then**
5: $D = RangeQuery(DocidIndex, [r_l, r_r])$
6: output (D, S);
7: **else** *FindSubsequence*$(Q_s, i + 1, (r_l, r_r))$
end

the Docid index (line 5). (r_l, r_r) is the positional representation of node id r. (In this case $r_l = r$.) Otherwise, *FindSubsequence*(.) is recursively invoked for the next element $Q_{s(i+1)}$ in the sequence using the range (r_l, r_r). In line 3, the position of match of the i^{th} element of Q_s (*i.e.*, level of node r in the trie) is stored in S. The solutions of the range query in line 1 are the ids of nodes $Q_{s(i+1)}$ that are descendants of nodes Q_{si} in the virtual trie. In line 4, the algorithm outputs a set of document (tree) identifiers D and a list S. S contains the positions in the LPS's of trees corresponding to tree identifiers in D where Q_s has a subsequence match.

It should be noted that the subsequence matching phase is I/O bound. The total number of range queries issued in this phase depends on the length of the sequence Q_s and $|R|$ in Algorithm 1. Our goal is to reduce the number of paths explored in the virtual trie to find all the subsequences. From our experiments, we observed that PRIX, by virtue of its bottom-up tree transformation, performed fewer range queries than ViST to process query patterns.

5.4. Optimized Subsequence Matching

In order to speed up subsequence matching further, it is desired to reduce the number of range queries to be performed by Algorithm 1 without causing any false dismissals. We can achieve this by pruning some nodes (r in line 2 of Algorithm 1) with an additional requirement on the gap between elements corresponding two adjacent nodes in the query sequence. In this regard, we have developed an upper-bounding distance metric based on the property of Prüfer sequences.

Given a collection Δ of XML document trees and node label e in Δ, we define the distance metric on the pair (e, Δ) as follows.

Definition 5 (MaxGap(e, Δ)) *Maximum postorder gap of a node label e is defined as the maximum of the difference between the postorder numbers of the first and the last children of the node labeled e in Δ.*

For example, in Figure 5, the difference in the postorder numbers of the first and last children of node label A is $14-8 = 6$ in tree P and is $3-1 = 2$ in tree Q. Hence $MaxGap(A, \{P, Q\})$

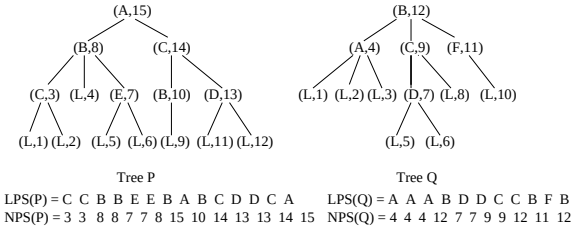


Figure 5. Examples for MaxGap

is 6. If every occurrence of label e in Δ has at most one child, then $MaxGap(e, \Delta) = 0$.

We shall now explain the usefulness of this distance metric for subsequence matching. Recall that in Lemma 1 we have shown that the i^{th} node to be deleted during the Prüfer sequence construction is the node numbered i. Consider tree P in Figure 5. The deletion of node 1 (the first child of node 3) corresponds to the first C in LPS(P). The deletion of node 2 (last child of node 3) corresponds to the second C in LPS(P). As can be observed in this example, the postorder gap between the first and last children of a node e denotes how far apart the first and last occurrences of its label (*i.e.,* e) can be in the sequence. Furthermore, the last occurrence of a node's label is always followed by its parent node label.

Suppose that a node with label B is the parent of a node with label C in a given query twig and C, B are adjacent in the query sequence. The CB of this query has eight matches in LPS(P) of Figure 5 at positions $(1,3), (1,4), (1,7), (1,8), (2,3), (2,4), (2,7), (2,8)$. Each of such number pairs represents an instance of CB match in the data sequence. Since $MaxGap(C, \{P, Q\})$ is $13 - 10 = 3$, the gap between the first and last occurrences of C in the sequence cannot be more than 3, and the gap between the first occurrence of C and its parent B cannot be more than 4. Thus, among the eight matches above, only four $(1,3), (1,4), (2,3), (2,4)$ may be considered for further processing. This example illustrates how $MaxGap$ helps discard certain subsequences that will definitely not be part of the final result.

The following theorem summarizes the use of the $MaxGap$ as an upper-bounding distance metric for pruning the search space and shortening the subsequence matching process.

Theorem 4 *Given a query twig Q and the set Θ of LPSs for Δ, let A and B denote adjacent labels in LPS(Q) such that A occurs before B.*

1. *In case node A is a child of node B in Q, any subsequence AB in Θ cannot result in a twig match, if its position pair (i, j) is such that $j - i > MaxGap(A, \Delta) + 1$.*
2. *In case node A is an ancestor of node B in Q, any subsequence AB in Θ cannot result in a twig match, if its position pair (i, j) is such that $j - i \geq MaxGap(A, \Delta)$.*

It is straightforward to extend Algorithm 1 to incorporate the upper-bounding distance metric by computing $(S_i - S_{i-1})$ (after line 3) and testing the appropriate condition in Theorem 4 using $MaxGap$ of label Q_{i-1}. Note that the $MaxGap$ metric can be defined at different levels of granularity. Finer-grained $MaxGap$ values can be stored in every occurrence of a symbol in the virtual trie.

5.5. The Refinement Phases

The set of ordered pairs (D, S) returned by Algorithm 1 are further examined during the refinement phases. The steps for the refinement phases are shown in Algorithm 2. The NPS and the set of leaf nodes of D are read from the database and passed as input to this algorithm. The input subsequence is refined by connectedness (Theorem 4.2) in lines 1 through 4. Note that this algorithm does not handle wildcards, but can be easily extended (as mentioned in Section 4.5) by modifying line 4. Next, the subsequence is refined by structure by testing for gap consistency (Definition 3) in lines 5 through 11. The subsequence is then tested for frequency consistency (Definition 4) in lines 12 through 15. Finally, the algorithm matches leaf nodes of the query twig in lines 16 through 18. This step can be eliminated by special treatment of leaf nodes in the query twigs and the data trees [18]. In line 19 we report a twig match.

5.6. Extended Prüfer Sequences

The Prüfer sequence of a tree as described in Section 3.1 contains only the labels of non-leaf nodes. We call this sequence *Regular-Prüfer sequence*. If we extend the tree by adding a dummy child node to each of its leaf nodes, the Prüfer sequence of this extended tree will contain the labels of all the nodes in the original tree. We shall refer to this new sequence as *Extended-Prüfer sequence*. In the case of XML, all the value nodes (strings/character data) in the XML document tree are extended by adding dummy child nodes before transforming it into a sequence. Similarly, query twigs are also extended before transforming them into sequences. We refer to the index based on Regular-Prüfer sequences as `RPIndex` and the index based on Extended-Prüfer sequences as `EPIndex`.

Indexing Extended-Prüfer sequences is useful for processing twig queries with values. Since queries with value nodes usually have high selectivities, Extended-Prüfer sequences provide higher pruning power than Regular-Prüfer sequences during subsequence matching. As a result, during subsequence matching, a fewer root-to-leaf paths are explored in the virtual trie of `EPIndex` than in the virtual trie of `RPIndex` for queries with values. If twig queries have no values, then indexing Regular-Prüfer sequences is recommended. Note that Extended-Prüfer sequences are longer than Regular-Prüfer sequences and the increase in length is proportional to the number of leaf nodes in the original tree.

In the PRIX system, both `RPIndex` and `EPIndex` can coexist. A query optimizer can choose either of the indexes based on the presence or absence of values in twig queries. It is easy for a query optimizer to detect values in queries since SAX parsers already have separate callback routines for values, attributes and elements.

5.7. Ordered and Unordered Twig Matches

In PRIX, the Prüfer sequence constructed after numbering a query twig in postorder, can be used to find all the ordered twig matches. In order to find unordered matches, Prüfer sequence for different arrangements of the branches of the query

Algorithm 2: Refinement Algorithms

Input: $\{N_D, N_Q, L_D, L_Q, S\}$: N_D is the NPS of tree D;
N_Q is the NPS of query twig;
L_D is a list of leaves in tree D;
L_Q is a list of leaves in Q;
S is the positions of a subsequence match in LPS(D);
Output: report twig match;

procedure *RefineSubsequence*(N_D, N_Q, L_D, L_Q, S)
// Test for connectedness (Refinement By Connectedness)
1: $maxN = max(N_D[S_1], N_D[S_2], ..., N_D[S_{|S|}])$;
2: **for** $i = 1$ *to* $|S|$ **do**
3: **if** $N_D[S_i] \neq maxN$ *AND* $\neg\exists(j > i)$ *s.t.* $N_D[S_i] = N_D[S_j]$ **then**
4: **if** $N_D[S_i] \neq S_{i+1}$ **then** return;
end
end
// Test for gap consistency (Refinement By Structure)
5: **for** $i = 1$ *to* $|S| - 1$ **do**
6: $dataGap = N_D[S_i] - N_D[S_{i+1}]$;
7: $queryGap = N_Q[i] - N_Q[i+1]$;
8: **if** *(*$(dataGap = 0$ *AND* $queryGap \neq 0$*) OR* *(*$queryGap = 0$ *AND* $dataGap \neq 0$*))* **then**
9: return;
end
10: **else if** $dataGap * queryGap < 0$ **then** return;
11: **else if** $|queryGap| > |dataGap|$ **then** return;
end
// Test for frequency consistency (Refinement By Structure)
12: **for** $i = 1$ *to* $|S|$ **do**
13: **for** $j = 1$ *to* $|S|$ *AND* $j \neq i$ **do**
14: **if** $N_Q[i] = N_Q[j]$ *AND* $N_D[S_i] \neq N_D[S_j]$ **then**
15: return;
end
end
// Match leaves (Refinement By Matching Leaves)
16: **foreach** l *in* L_Q **do**
17: **if** *l not found in* L_Q **then**
18: **if** *l not found in LPS/NPS of D* **then** return;
end
end
19: report twig match; return;

twig should be constructed and tested for twig matches. Since the number of twig branches in a query is usually small, only a small number of configurations (arrangements) need to be tested. For more discussion in this regard, we refer our readers to the extended version of this paper [18].

6. Experimental Results

In our experiments, we compared the query performance of PRIX, ViST and TwigStack/TwigStackXB. We implemented all the algorithms in PRIX, ViST and TwigStack/TwigStackXB in C++, and used the B$^+$-tree implementation of GiST [10] for all their indexes. For ViST, the symbol-prefix pairs in the structure-encoded sequences were directly stored in the D-Ancestorship B$^+$-tree.

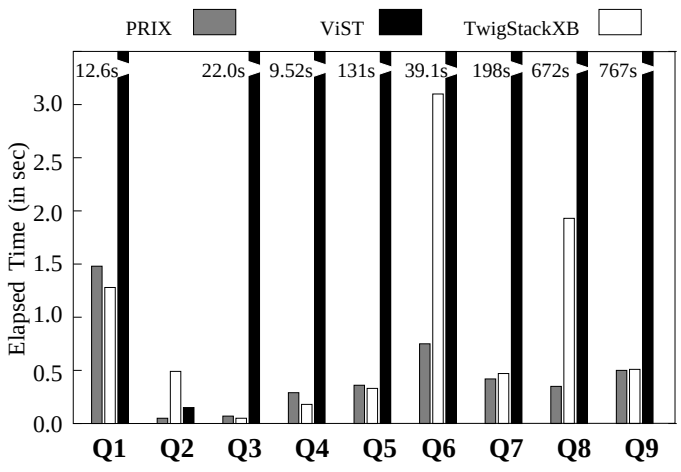


Figure 6. Elapsed time for XPath Queries in Table 3

6.1. Experimental Setup

We ran all our experiments on 1.8GHz Pentium IV processor with 512 MB RAM running Solaris 8. A 40GB EIDE disk drive was used to store the data and indexes. The code was compiled using the GNU g++ compiler version 2.95.3. Direct I/O feature available on Solaris was enabled to avoid operating system's cache effects. For all the experiments, the buffer pool size was fixed at 2000 pages. The page size of 8K was used. For PRIX and ViST, 8-byte number ranges were used to label the nodes in the virtual trie. For TwigStack/TwigStackXB, 4-byte number ranges were used to label the nodes in the XML document trees.

6.2. Data Sets

We experimented with the datasets shown in Table 2. These datasets were obtained from the University of Washington XML repository [14]. We chose these three datasets since each had a different characteristic. The document trees in the DBLP dataset had good similarity in structure and were shallow. The document trees in the SWISSPROT dataset were bushy and shallow. The document trees in the TREEBANK dataset were skinny and had deep recursions of element names. Table 2 provides additional information such as the maximum depth, the number of elements and so on for the datasets. We constructed Prüfer sequences and ViST's structure-encoded sequences for the collection of XML document trees on each dataset. Table 2 shows the number of sequences constructed for each dataset.

6.3. Queries

The XPath queries listed in Table 3 were tested in our experiments. These queries have different characteristics in terms of selectivity, presence of values and twig structure. Table 3 also shows the number of twig occurrences for each query. For

Dataset Name	Size in MBytes	# of Elements	# of Attributes	Max-depth	# of Sequences
DBLP	134	3332130	404276	6	328858
SWISSPROT	115	2977031	2189859	5	50000
TREEBANK	86	2437666	1	36	56385

Table 2. Datasets

	Query	Dataset	# of Twig Matches
Q_1	//inproceedings[./author="Jim Gray"][./year="1990"]	DBLP	6
Q_2	//www[./editor]/url	DBLP	21
Q_3	//title[text()="Semantic Analysis Patterns"]	DBLP	1
Q_4	//Entry[./Keyword="Rhizomelic"]	SWISSPROT	3
Q_5	//Entry/Ref[./Author="Mueller P"][./Author="Keller M"]	SWISSPROT	5
Q_6	//Entry[./Org="Piroplasmida"][.//Author]//from	SWISSPROT	158
Q_7	//S//NP/SYM	TREEBANK	9
Q_8	//NP[./RBR_OR_JJR]/PP	TREEBANK	1
Q_9	//NP/PP/NP[./NNS_OR_NN][./NN]	TREEBANK	6

Table 3. XPath Queries

Query	PRIX		ViST	
	Total time	Disk IO	Total time	Disk IO
Q_1	1.48 secs	185 pages	15.28 secs	3543 pages
Q_2	0.05 secs	7 pages	0.15 secs	15 pages
Q_3	0.07 secs	9 pages	22.07 secs	2280 pages

Table 4. DBLP - PRIX vs ViST

Query	PRIX		ViST	
	Total time	Disk IO	Total time	Disk IO
Q_4	0.29 secs	23 pages	9.52 secs	1757 pages
Q_5	0.36 secs	49 pages	131.67 secs	128,150 pages
Q_6	0.75 secs	86 pages	39.12 secs	6967 pages

Table 5. SWISSPROT - PRIX vs ViST

the TREEBANK dataset, since the values were encrypted, we chose queries without values (character data).

6.4. Performance Analysis

In Figure 6 we summarize the performance results in total time elapsed for the queries listed in Table 3. We first discuss the benefits of PRIX over ViST.

6.4.1. PRIX vs ViST. In this section we compare the performance between PRIX and ViST. We tested queries Q_1, Q_2 and Q_3 for the DBLP dataset. Q_1 and Q_2 are twig queries with five nodes and two branches, and with three nodes and two branches, respectively. Q_3 is a single path query with two nodes. Q_1 and Q_3 have values but Q_2 does not have any value.

PRIX performed significantly better than ViST for queries Q_1, and Q_3, and had comparable performance for query Q_2. Table 4 shows the total time taken and physical I/O (pages read from disk) to process queries Q_1, Q_2 and Q_3. The presence of values in ViST's structure-encoded sequences reduces the sharing of root-to-leaf paths in the trie. In the worst-case, every sequence could cause a separate root-to-leaf path in the trie. Furthermore, the presence of the root-to-node prefix in each node of the structure-encoded sequences further reduces the sharing in the trie.

Similarly, the presence of values in Extended-Prüfer sequences reduces the sharing of root-to-leaf paths in the trie. However, the *bottom-up* transformation of the query twig and data in PRIX plays a crucial role in reducing the query processing time. Since the selectivity of value nodes is usually higher than that of element nodes, the labels at the beginning of the LPS of a query twig may occur less frequently in the virtual trie than the labels found later in the sequence. In such cases, only a few paths in the virtual trie need to be examined to find all the subsequences. This implies that a smaller number of range queries are processed by Algorithm 1. In contrast, ViST's *top-down* transformation of a twig resulted in a large number of nodes (paths) in the virtual trie being examined during subsequence matching for commonly occurring tag names. For example, tag names `author` in Q_1 and `title` in Q_3 suffered from this behavior. PRIX used `EPIndex` to process queries Q_1 and Q_3, and clearly outperformed ViST by up to a few orders of magnitude. ViST processed query Q_2 comparably, because there were only a few occurrences of tag name `www` in the DBLP dataset and hence only a few `editor` descendants in the trie. PRIX used `RPIndex` for processing Q_2 and had comparable performance.

For the SWISSPROT dataset, PRIX again clearly outperformed ViST for all queries Q_4, Q_5 and Q_6. Query Q_4 is a simple path query with three nodes, Q_5 is a twig query with six nodes and two branches. Q_6 is a twig query with five nodes and three branches. These queries have values in them.

Table 5 shows the performance results for queries Q_4, Q_5 and Q_6. As mentioned earlier, ViST's top-down transformation of the twig deteriorated the query processing considerably. Tag names `Ref` in Q_5 and `Org` in Q_6 resulted in many range queries during subsequence matching. This increased the disk I/O and slowed down the query processing. On the other hand, PRIX used `EPIndex` to process Q_4, Q_5 and Q_6 and processed them efficiently. This demonstrates the advantage of bottom-up transformation of PRIX once again.

Another drawback of ViST that we would like to point out is the processing of queries with wildcards like `'//'` for datasets with recursions of elements. We tested queries Q_7, Q_8 and

Query	PRIX		ViST	
	Total time	Disk IO	Total time	Disk IO
Q_7	0.42 secs	46 pages	198.40 secs	40,827 pages
Q_8	0.35 secs	35 pages	672.20 secs	94,505 pages
Q_9	0.50 secs	55 pages	767.24 secs	121,928 pages

Table 6. TREEBANK - PRIX vs ViST

Query	TwigStack		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_1	20.74 secs	8756 pages	1.28 secs	201 pages
Q_2	7.25 secs	2310 pages	0.49 secs	63 pages
Q_3	6.17 secs	2271 pages	0.05 secs	8 pages

Table 7. DBLP - TwigStack vs TwigStackXB

Q_9 on the TREEBANK dataset. These queries do not have any values. Query Q_7 is a single path query with three nodes and two `//`'s, Query Q_8 is a twig with two branches and three nodes. Query Q_9 has two branches and five nodes. Table 6 shows the performance results for queries Q_7, Q_8 and Q_9.

ViST processed wildcards in Q_7 the following way. The D-Ancestorship index was first searched for all (`S`, `//`) keys. Thus, every key with `S` as its symbol was matched. The tag name `S` occurred at different levels in the TREEBANK dataset, in addition to occurring frequently in the dataset. This resulted in many unique (symbol, prefix) key matches to begin with. In all, 515 unique (symbol, prefix) keys were matched in the D-Ancestorship index during the processing of Q_7. In addition, there were several occurrences of each (symbol, prefix) key. Thus many paths in the virtual trie were searched for subsequences. Similar was the case for query Q_8 since the tag name `NP` occurred frequently and at different levels in the TREEBANK dataset. In this case, $46,355$ unique (symbol, prefix) keys were matched in the D-Ancestorship index during the processing of Q_8.

PRIX used `RPIndex` to process queries Q_7, Q_8 and Q_9 and outperformed ViST once again. The bottom-up transformation of the query twig resulted in only a few paths being searched in the virtual trie. Note that in our PRIX system, the presence of wildcards does not add extra overhead during subsequence matching. (Refer to Section 4.5.)

6.4.2. PRIX vs TwigStack/TwigStackXB. In this section, we compare the performance of PRIX and TwigStackXB. TwigStackXB uses XB-Trees to skip nodes in the sorted input stream. Note that for all the queries in Table 3 that we tested, TwigStack performed worse than TwigStackXB. Table 7 shows the performance results for TwigStack and TwigStackXB for the DBLP dataset. Other results for TwigStack have been omitted due to lack of space.

Query	PRIX		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_1	1.48 secs	185 pages	1.28 secs	201 pages
Q_5	0.36 secs	49 pages	0.33 secs	59 pages
Q_7	0.42 secs	46 pages	0.47 secs	51 pages

Table 8. PRIX vs. TwigStackXB for Q_1, Q_5, Q_7

Query	PRIX		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_2	0.05 secs	7 pages	0.49 secs	63 pages
Q_6	0.75 secs	86 pages	3.10 secs	485 pages
Q_8	0.35 secs	35 pages	1.93 secs	310 pages

Table 9. PRIX vs TwigStackXB for Q_2, Q_6, Q_8

For queries Q_1, Q_3, Q_4, Q_5, Q_7, and Q_9, both PRIX and TwigStackXB yielded comparable performance. Table 8 shows the performance results for queries Q_1, Q_5, Q_7. Similar trend in performance was observed for queries Q_3 and Q_4. (Refer to Figure 6.) As expected, TwigStackXB processed these queries efficiently, because the solutions for those queries were clustered in certain regions of the data and the XB-Trees were effective in skipping nodes in the input streams. On the other hand, PRIX also processed these queries efficiently using its *bottom-up* processing strategy.

We now analyze the query performance of PRIX and TwigStackXB for queries Q_2, Q_6 and Q_8 shown in Table 9. We compared PRIX and TwigStackXB under two different scenarios namely *distribution of possible solutions in the dataset* and *sub-optimality for parent/child relationships*.

Distribution of Possible Solutions in the Data Set The effectiveness of skipping input data using XB-Trees is dependent on the distribution of matches in the dataset. Furthermore, if the nodes in different branches of a query twig occur in different but nearby documents in the input data, then the TwigStackXB algorithm is forced to drill down to the lower regions of the XB-Trees (and possibly leaves) to verify whether these nodes represent a match. Queries Q_2 and Q_6 were tested for the behavior.

In the case of Q_2, tag name `www` was scattered in the DBLP dataset. The other two tag names in Q_2 namely `editor` and `url` occurred frequently in the dataset and were present around the documents with `www` elements in the input data. This caused TwigStackXB to drill down to the lower regions of the XB-Trees several times, in order to eliminate these nodes from the solution set. This process increased the disk I/O.

On the other hand, PRIX (using `RPIndex`) processed Q_2 several times faster than TwigStackXB. The XML documents in the DBLP dataset had good similarity in terms of tree structure. This resulted in sharing of root-to-leaf paths in the virtual trie by several Regular-Prüfer sequences. For example, one root-to-leaf path in the virtual trie was shared by $31,864$ Regular Prüfer sequences. As a result, the total number of nodes in the virtual trie was reduced considerably. Thus only a few range queries were required to find all the subsequence matches for processing query Q_2.

PRIX (using `EPIndex`) also processed Q_6 several times faster than TwigStackXB. In the SWISSPROT dataset that we used, documents with pattern `Entry/Org/Piroplasmida` were scattered in the input data. But only a few of these documents had both `Author` and `from` tags as descendants of `Entry`. In addition, the tags `Author` and `from` occurred frequently and were present in other documents near the the documents containing `Piroplasmida` in the input data. As a

result, TwigStackXB was forced to drill down to lower regions (including leaves) of the XB-Trees several times in order to eliminate such partial matches. This caused an increase in IO and slowed down query processing. However, PRIX was able to quickly eliminate these partial matching documents during subsequence matching as they shared a root-to-leaf path in the virtual trie.

Sub-Optimality for Parent/Child Relationships We tested query Q_8 to demonstrate the fact that TwigStack/TwigStackXB can suffer from sub-optimality for parent/child edges (`'//'`) in a query twig. (Refer to Section 2.) Query Q_8 has parent-child edges in it. As expected, PRIX (using `RPIndex`) processed Q_8 several times faster than TwigStackXB.

In the TREEBANK dataset that we used, there were several similar documents scattered in the input data that had tag name `NP` as an ancestor (but not the parent) of tag names `PP` and `RBR_OR_JJR`. TwigStackXB matched these documents as possible solutions due to sub-optimality. On the other hand, PRIX (using `RPIndex`) was able to quickly eliminate these false alarms during subsequence matching by using the upper-bounding distance metric associated with tag name `RBR_OR_JJR`. (The $MaxGap$ of `RBR_OR_JJR` was zero in this case.) Tag name `NP` did not occur immediately after `RBR_OR_JJR` in the LPS of those documents that were falsely matched by TwigStackXB. TwigStackXB discards such matches during the merge post-processing step.

7. Conclusions and Future Work

In this paper, we have presented a new paradigm for XML pattern matching. We transform XML documents into Prüfer sequences. To find all occurrences of a query twig, subsequence matching is performed on the set of sequences followed by a series of refinement phases. We also provide theoretical background to show the correctness of our approach. Unlike most state-of-the-art techniques, our approach processes twig queries without breaking them into root-to-leaf paths and processing them individually. We also provide empirical results to demonstrate the efficient processing of twig queries by the PRIX system. As part of future work, we would like to explore the behavior of the PRIX system for different query characteristics such as the cardinality of result sets, and analyze the complexity of query processing time.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, San Jose, California, Feb. 2002.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, Aug. 2002.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language W3C working draft 16. Technical Report WD-xquery-20020816, World Wide Web Consortium, Aug. 2002.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 second edition W3C recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, Oct. 2000.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, Aug. 2002.

[7] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 122–127, San Francisco, California, May 1982.

[8] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, Aug. 1997.

[9] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[10] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21rd VLDB Conference*, pages 562–573, Zurich, Switzerland, Sept. 1995.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[12] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, San Jose, California, Feb. 2002.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, Rome, Italy, Sept. 2001.

[14] G. Miklau. UW XML Repository. http://www.cs.washington.edu/research/xmldatasets.

[15] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295, Jerusalem, Israel, Jan. 1999.

[16] S. Nestorov, J. U. Wiener, and S. Chawathe. Representative objects: concise representations of semistructured, hierarchical data. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, Birmingham, U.K., Apr. 1997.

[17] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.

[18] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. Technical Report TR 03-06, University Of Arizona, Tucson, AZ 85721, July 2003. http://www.cs.-arizona.edu/research/reports.html.

[19] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 2003 ACM-SIGMOD Conference*, San Diego, CA, June 2003.

[20] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, Aug. 2001.

[21] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM-SIGMOD Conference*, pages 425–436, Santa Barbara, California, May 2001.