PRIX: Indexing And Querying XML Using Prüfer Sequences *

Praveen Rao   *Bongki Moon*

{rpraveen,bkmoon}@cs.arizona.edu

Technical Report 03-06

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.

July 2003

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

*This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037), NSF Grant No. IIS-0100436, and 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.

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 (*e.g.*, path expressions expressed in XPath, path expressions in the `for` and `let` clauses in XQuery), while values of the XML elements are used as part of selection predicates. For example, an XPath expression

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

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

Queries with path expressions 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 [10], representative objects [22], 1-index [20], approximate path summary [17], F&B index [16]). These structural indexes can help reduce the search space for processing linear path and 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 [9], extended preorder [18]) or textual positions of start and end tags (*e.g.*, containment property [30], absolute region coordinate [28]). 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, 7, 11, 18, 30]. 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 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 filtering and 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 indexing method called ViST [26], which also converts trees into sequences.

Furthermore, most of the previous approaches (*e.g.*, TwigStack [5], TSGeneric$^+$ [15]) do not attempt the problem of *ordered twig pattern matching* that is useful in applications where the twig pattern nodes follow the document order in XML. For example, an XML data model was proposed by Bow *et al.* [6] for representing interlinear text for linguistic applications, which is used to demonstrate various linguistic principles in different languages. The XML model provides a four-level hierarchical representation for the interlinear text, namely, text level, phrase level, word level and morpheme level. For the purpose of linguistic analysis, it is essential to preserve linear order between the words in the text [27]. Thus, there is a compelling need for ordered twig pattern matching. In addition to interlinear text, language treebanks have been widely used in computational linguistics. Treebanks capture syntactic structure of textual data and provide a

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

hierarchical representation of the sentences in the text by breaking them into syntactic units such as noun clauses, verb phrases, adjectives and so on. A recent paper by Müller *et al.* [21] used ordered pattern matching over treebanks for question answering systems. Our PRIX system supports ordered twig pattern matching inherently.

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 matching 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. [2]
- Our system supports *ordered twig pattern matching* that is useful for applications that require the twig pattern nodes to follow the document order in XML.
- We have developed effective optimizations to speed up the subsequence matching 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 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 describe the architecture of the PRIX system. In Section 7 we present our experimental results. Section 8 surveys the related work in XML indexing and query processing. Lastly Section 9 summarizes the contributions of this paper.

2 Background and Motivations

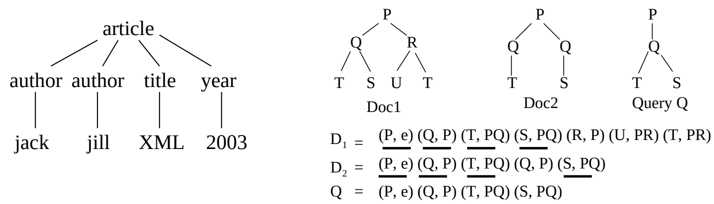
An XML document can be modeled as an ordered labeled tree as shown in Figure 1(a). Each node in this tree corresponds to an element or a value. Values are represented by character data (`CDATA`, `PCDATA`) and occur at the leaf nodes. The tree edges represent a relationship between two elements or between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

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 of the recent contributions made for XML pattern matching: TwigStack [5] and ViST [26]. We will then discuss some of their drawbacks to motivate our proposed approach.

TwigStack Algorithms Bruno *et al.* proposed optimal XML pattern matching algorithms [5]. These stack-based algorithms process input lists of element instances for tags that appear in a query twig. *TwigStack* and *PathStack* algorithms operate on the positional representation of the element instances to find twig matches. A variant of *TwigStack* (denoted hereinafter by *TwigStackXB*) uses XB-Trees to speed up the 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 *TwigStack* and *TwigStackXB* is that it suffers from sub-optimality for parent-child relationships in a query twig. The algorithm might produce a partial match of a path of a twig that cannot be combined with any other partial match of another path of the twig. For example, consider a query

[2] In contrast, ViST [26] does not guarantee linear space requirement for its tree-to-sequence transformation.



(a) An XML document (b) False alarms by ViST

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

twig with 3 nodes `P`, `Q` and `R` where nodes `Q` and `R` are child nodes of `P`. 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.* proposed a new method called *ViST* that transforms XML data trees and twig queries into structure-encoded sequences [26]. 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 the XML documents. These sequences are stored in a disk-based virtual trie built using B$^+$-trees.

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 index* is higher than linear in the total number of elements in the XML documents. For example, consider a tree with n nodes and maximum depth d. In this case, the total size of the structure-encoded sequence of the tree is $O(nd)$. Thus the D-Ancestorship index requires $O(nd)$ space to store all the (symbol, prefix) keys. Another drawback of *ViST* is that the query processing strategy by straightforward subsequence matching 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. Such false matches could be discarded by adding a post-processing step that examines the input document trees.

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 transforms trees into sequences, thereby reducing the problem of finding twig matches to that of finding subsequences and (3) to construct a method for finding twig matches with no *false alarms* or *false dismissals*.

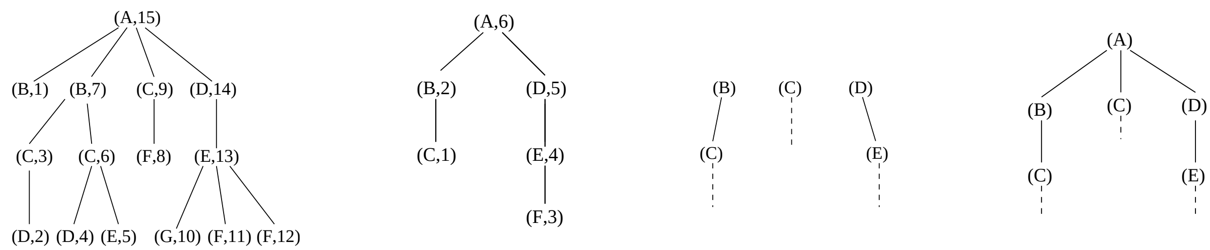
3 Overview of PRIX

In this section, we present 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. We also present an architectural overview of 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 [23]. 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 the 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.



(a) Tree T (b) Tree Q (c) Disconnected (d) Connected

Figure 2: An XML document tree, a query twig, and matches representing connected/disconnected graphs

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. In our PRIX system, the nodes of an XML document tree are numbered in postorder from one to the total number of nodes. Thus each node is associated with a distinct number.

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 the *NPS (Numbered Prüfer sequence)* of the document. If each number in this NPS is replaced by its corresponding tag, a new sequence that consists of XML tags can be constructed. We call this sequence the *LPS (Labeled Prüfer sequence)* of the document. [3]

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.* □

For each XML document in the database, its LPS and NPS are constructed. The set of LPS's are indexed for efficient query processing. The set of NPS's are stored in the database (*e.g.*, as records in a heap file) together with their unique document identifiers.

3.3 Processing Twig Queries

A query twig is transformed into its Prüfer sequence just like an XML document. Non-matches are filtered out by performing 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 that highlights the steps described in Section 3.2 and Section 3.3. For a detailed description of PRIX's architecture, refer to Section 6. 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 the presentation of concepts in this section, we shall use the notations listed in Table 1. Formally the problem for finding twig matches can be stated as follows: *Given a collection of XML documents Δ and*

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

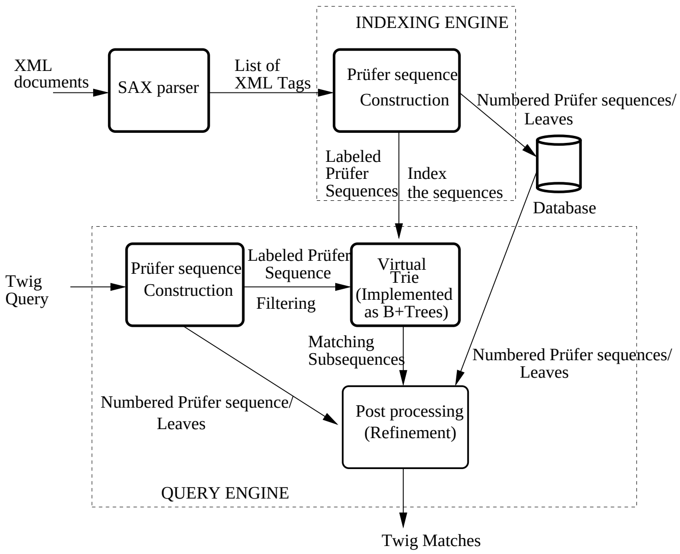
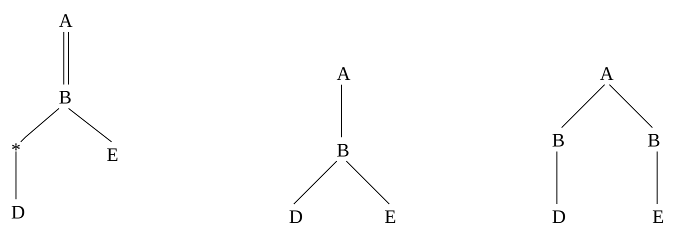


Figure 3: Architectural Overview of PRIX



(a) Twig Pattern Q_1 (b) Twig Pattern Q_2 (c) Twig Pattern Q_3

Figure 4: Twig Patterns

a query twig Q, report all the occurrences of twig Q in Δ. Note that our problem description is based on twig patterns as opposed to XPath expressions. The twig patterns that we deal with in this paper can be mapped to equivalent XPath expressions. Essentially, a twig pattern has a *defined structure* on the nodes in the pattern, and each node has a label associated with it. For example, a twig pattern Q_1 (in Figure 4(a)) can occur anywhere in the XML document, and can be expressed as '`//A//B[*/D]/E`' in XPath. However, this XPath expression has no restriction on the order of the sibling nodes '`E`' and '`*`'. Using the XPath axis `following-sibling`, Q_1 can be rewritten as '`//A//B/*[D]/following-sibling::E`' to specify that '`E`' should follow '`*`' in the matches. Note that an XPath expression can be mapped to more than one twig pattern. As an example, the XPath expression '`//A[B/D][B/E]`' can be interpreted as either of the two distinct twig patterns Q_2 and Q_3 shown in Figures 4(b) and 4(c).

We will initially deal with the problem of finding all occurrences of twig Q without the axis '`//`' and wildcard '`*`'. Later in Section 4.5, we explain how query twigs with '`//`' and '`*`' can be processed. Note that our work focuses primarily on *ordered twig pattern matching*. Hence we will first address the problem of finding ordered twig matches of Q with equality predicates. Later in Sections 5.8 and 5.9 we explain how PRIX can be extended to handle unordered twig matches and inequality predicates.

Finding twig matches in PRIX involves the following phases, namely (1) *filtering by subsequence matching*, (2) *refinement by connectedness*, (3) *refinement by structure* and (4) *refinement by leaf nodes.*

4.1 Filtering by Subsequence Matching

The filtering phase in PRIX involves subsequence matching. The classical definition of a subsequence is stated as follows.

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
Θ_C	A subset of Θ that represent trees (connected)
LPS(T)	Labeled Prüfer sequence of tree T
NPS(T)	Numbered Prüfer sequence of tree T
Label(v, T)	returns the label associated with vertex v in T
Number(v, T)	returns the number associated with vertex v in T
S	A subsequence of an LPS
N	The postorder number sequence of S

Table 1: Notations

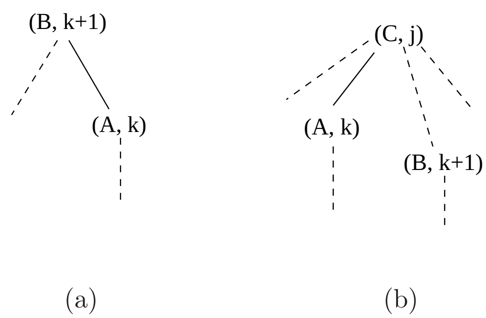


Figure 5: Nodes assigned consecutive postorder numbers

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

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 in the i^{th} step during the Prüfer sequence construction is the node numbered i.*

Proof. We prove the theorem by Induction on the number of nodes in the tree. In each step of the construction of Prüfer sequences, the leaf node with the smallest label is deleted from the tree to give a smaller tree. For simplicity, we will refer to the node numbered i as node i. Let T_{n-k} be the tree obtained after k node deletions. Let $P(k)$ denote the proposition that *the id of the node deleted in the k^{th} step of the Prüfer sequence construction is k*. Note that $1 \leq k < n$.

(i) *Basis of Induction:* $P(1)$ is true. This is because a node numbered one in postorder is a leaf and is deleted first.

(ii) *Induction Hypothesis:* Assume that $P(i)$ is true for $1 \leq i \leq k$. We will show that $P(k+1)$ is also true. Only the two following scenarios are possible for nodes numbered k and $k+1$ in postorder: (1) node $k+1$ is a leaf, (2) node $k+1$ is not a leaf and node k is the last child of node $k+1$. (Refer to Figure 5.) If node $k+1$ is a leaf in T_n, then by induction hypothesis, node $k+1$ is the smallest leaf in T_{n-k}. Thus $P(k+1)$ is true. If node k is the last child of node $k+1$, then node $k+1$ becomes a leaf after k node deletions. By induction hypothesis, all nodes i such that $i \leq k$ have been deleted. The smallest leaf node in T_{n-k} is $k+1$ and is deleted next. Thus $P(k+1)$ is true. $\square$

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 the Prüfer sequence construction. In addition, the i^{th} element in the NPS denotes the postorder number of the parent of node i.

In the subsequent discussions, we shall frequently use two notations namely $Number(\cdot)$ and $Label(\cdot)$ that are described in Table 1. 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).*

Proof. Assume that the tree nodes in T and Q are numbered in postorder. Let $v_1, v_2, v_3, \ldots, v_{n-1}$ be the order of deletion of nodes by the Prüfer sequence construction for tree Q (with n nodes). Since Q is a subgraph of T, the same set of nodes are also deleted during the Prüfer sequence construction for tree T in some order along with other nodes in T. We would like to prove that for $1 \le i < n$, v_i is deleted before v_{i+1} in tree T.

By Lemma 1, it is true that $Number(v_i, Q) < Number(v_{i+1}, Q)$ for any pair of nodes v_i and v_{i+1} in Q ($1 \le i < n$). Furthermore, for any pair of nodes v_i and v_j in Q, if $Number(v_i, Q) < Number(v_j, Q)$ is true, then $Number(v_i, T) < Number(v_j, T)$ is also true. (Note that nodes v_i and v_j are also in T.) Therefore nodes v_i and v_{i+1} satisfy the condition $Number(v_i, T) < Number(v_{i+1}, T)$ for $1 \le i < n$. By Lemma 1, v_i is deleted before v_{i+1} in tree T. Since for $1 \le i \le n$, $Label(v_i, Q) = Label(v_i, T)$, $LPS(Q)$ 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). Tree T in Figure 2(a) 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. Tree Q in Figure 2(b) 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 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 the nodes that correspond to the labels in a subsequence may not be connected (represent a tree) in the data tree. Let Θ_C denote a set of subsequences that satisfy the connectedness property. Formally, we state a **necessary** and **sufficient** condition for a subsequence $S \in \Theta_C$.

Theorem 2 *Given a tree T, let S be a subsequence of LPS(T), and let $N[i]$ denote the i^{th} element in the postorder number sequence of S. ($N[i]$ is the postorder number taken from NPS(T) for a node corresponding to the i^{th} element in S.) Then the tree nodes in T corresponding to the elements of S constitute a connected subgraph (or sub-tree) of T, if and only if, condition (iii) is true for every $N[i]$ that satisfies conditions (i) and (ii).*

(i) $N[i] \neq max(N[1], N[2], ..., N[|S|])$,

(ii) $N[i]$ *is the last occurrence of the same number in the postorder number sequence of S*

(iii) $N[i+1]$ *is equal to the* $N[i]^{th}$ *element of NPS(T).*

Proof of (If) part. For each node corresponding to the elements in S, we shall check if its parent node in T also appears in S. If $N[i]$ is the maximum in the postorder number sequence of S, then neither the parent nor the ancestor of its corresponding node is in S. Thus the node for $N[i]$ need not be checked for its parent in S. Moreover, if S represents a sub-tree of T then the node for $N[i]$ would represent the root of that sub-tree. On the other hand, if $N[i]$ is not the maximum in the postorder number sequence of S, then the node for $N[i]$ has to be connected to its parent in order for S to represent a sub-tree of T.

From Lemma 1, if the last occurrence of a node's postorder number n occurs at the i^{th} position in an NPS, then the number at the $(i+1)^{th}$ position in the NPS is the postorder number of the parent of n. This is because, during Prüfer sequence construction, the deletion of the last child of n makes n the smallest leaf node. Hence if $N[i+1]$ is equal to the $N[i]^{th}$ element of $NPS(T)$, then $N[i]$'s node is connected to its parent node in S. If this is true for every $N[i]$ in S ($N[i]$ is not the maximum), then the tree nodes in T corresponding to the elements of S represent a connected subgraph of T.

Proof of (Only If) part. It is given that the tree nodes in T corresponding to the elements in S constitute a connected subgraph. Let us call it tree $T^{'}$. First we number these $|S+1|$ nodes in postorder. Then the last occurrence of a postorder number in $NPS(T^{'})$ (except the root) is followed by its parent's postorder number. Now let us replace the postorder numbers of the nodes in $T^{'}$ with their corresponding postorder numbers from T. As a result, we can transform $NPS(T^{'})$ to the postorder number sequence of S. Since each node is assigned a unique number, the old and new postorder numbers of nodes in $T^{'}$ have a one-to-one correspondence. Without loss of generality, let n_c and n_p be the old postorder numbers of a child and parent node in $T^{'}$ respectively. Let $n_c^{'}$ and $n_p^{'}$ be their new postorder numbers. Since the last occurrence of $n_c^{'}$ is followed by $n_p^{'}$ in $NPS(T^{'})$, the last occurrence of n_c (say at $N[i]$) is followed by n_p (at $N[i+1]$) in the postorder number sequence of S. The $N[i]^{th}$ element in NPS(T) corresponds to the parent of the node numbered n_c *i.e.*, n_p. As a result, the conditions *(i)*, *(ii)* and *(iii)* are true. Hence we have proven the *Only If* case. □

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 2 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$(N_A[1], N_A[2], ..., N_A[5])$ = 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 $N_A[j] = 7$. However $N_A[2]$ is not followed by $N_T[7]$, i.e., $N_A[3] \neq 15$. Hence Theorem 2 is not satisfied. The nodes represented by elements of S_B represent a tree as shown in Figure 2(d) because 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 the details of refinement by structure, we shall first introduce the notion of *gap* between two tree nodes and *gap consistency* between two tree sequences.

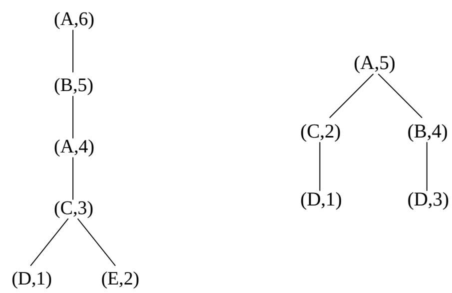
Definition 2 *The gap between two nodes **a** and **b** in a tree is defined as the difference between the postorder numbers of the nodes **a** and **b**.*

By using the NPS of a tree, the gaps between tree nodes can be computed.

Definition 3 *Tree sequence S_A is said to be gap consistent with respect to tree sequence S_B if*

1. *S_A and S_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.



(a) Tree T_g (b) Query Q_g

Figure 6: Pruning Tree Sequences using Gap Consistency

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 D A be a subsequence of LPS(T) and let* N_1 *= 7 15 13 13 14 15 be the postorder number sequence of* S_1*. Consider a query tree with LPS* S_2 *= B A E E D A, and NPS* N_2 *= 2 7 5 5 6 7. 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 2,*
- *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.*

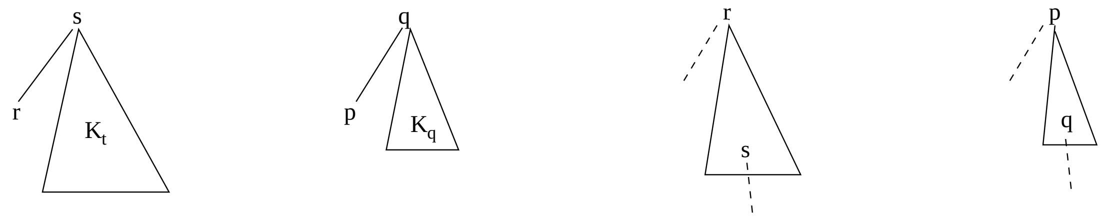
$\square$

Intuitively, the gap between two nodes in a data tree gives an idea of how many nodes occur between these two nodes during the postorder traversal. Similar is the case with the nodes of a query twig. Suppose M denotes the NPS of a tree. Let $M[i]$ and $M[i+1]$ denote the postorder number of nodes corresponding to the i^{th} and the $(i+1)^{th}$ entry in M. A negative gap between $M[i]$ and $M[i+1]$ indicates that node $M[i]$ is a child of node $M[i+1]$. However, a positive gap between $M[i]$ and $M[i+1]$ indicates that node $M[i]$ is an ancestor of $M[i+1]$. A zero gap implies that $M[i]$ and $M[i+1]$ correspond to the same node in the XML document. If more nodes are traversed in the query twig as compared to a twig matched in the data, then this indicates that there is a structural difference between the match in the data and the query twig. This concept forms the basis for Theorem 3 that states a necessary and sufficient condition for a match by twig structure.

Example 5 *We illustrate how gap consistency can be used to prune away tree sequences that are false matches. Consider a tree* T_g *and a query pattern* Q_g *as shown in Figures 6(a) and 6(b). Note that LPS(*T_g*) = C C A B A, NPS(*T_g*) = 3 3 4 5 6, LPS(*Q_g*) = C A B A, and NPS(*Q_g*) = 2 5 4 5. LPS(*Q_g*) matches a subsequence S = C A B A in LPS(*T_g*) whose postorder number sequence is 3 4 5 6. Although S is a tree sequence by Theorem 2, LPS(*Q_g*) is not gap consistent with S (by Definition 3). Therefore* Q_g *does not have a structural match in* T_g*.*

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 the above observation by defining a property called *frequency consistency*.



(a) Tree T (b) Tree Q (c) Tree T (d) Tree Q

Figure 7: Gap between nodes

Definition 4 *Tree sequences S_A and S_B are frequency consistent if*

1. *S_A and S_B have the same length n,*
2. *Let N_A and N_B be the postorder number sequences of S_A and S_B respectively. Let $N_A[i]$ and $N_B[i]$ be the i^{th} element in N_A and N_B respectively. For every i from 1 to n, $N_A[i]$ occurs k times in N_A at positions $\{p_1, p_2, ..., p_k\}$, if and only if, $N_B[i]$ occurs k times in N_B at positions $\{p_1, p_2, ..., p_k\}$.*

Note that frequency consistency is an equivalence relation.

Example 6 *In Example 4, sequences S_1 and S_2 are frequency consistent. The 1^{st} element in N_1 '7' occurs once at position 1. The 1^{st} element in N_2 '2' also occurs once at position 1. The 2^{nd} element in N_1 '15' occurs at positions 2 and 6. The 2^{nd} element in N_2 '7' also occurs at positions 2 and 6. Similar is the case with the remaining elements in N_1 and N_2.* □

It should be noted that the LPS of a tree contains only the non-leaf node labels. So 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 *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. 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 if and only if*

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 S.*

Proof of (Only If) part. Suppose Q has a partial match in tree T, then Q matches a subgraph T' of T, except that the leaf node labels of Q may not match with the leaf node labels of T'. By Lemma 1, LPS(T') is a subsequence of LPS(T). The set of nodes in T' and Q are deleted in the same relative order during Prüfer sequence construction (using postorder numbering) of trees T and Q respectively. Furthermore, since Q and T' have the same structure, the corresponding non-leaf nodes in Q and T' have the same number of child nodes. Thus LPS(Q) matches a subsequence S of LPS(T) that represents tree T' (*i.e.*, tree sequence), and S and LPS(Q) are frequency consistent.

Our goal is to show that LPS(Q) is gap consistent with S. Consider two nodes p and q in tree Q. Let $Label(p, Q)$ and $Label(q, Q)$ be two adjacent elements in LPS(Q) where $Label(p, Q)$ occurs before $Label(q, Q)$. Let n_{pQ} and n_{qQ} be the postorder numbers of nodes p and q in tree Q. Note that nodes p and q are internal nodes in Q. Let nodes r and s be nodes in T' that match p and q respectively. $Label(r, T)$ and $Label(s, T)$

(a) Nodes in tree $T^{'}$ (b) Nodes in tree Q (c) Nodes in tree $T^{'}$ (d) Nodes in tree Q

Figure 8: Possible scenarios

are adjacent in S. Let n_{rT} and n_{sT} be the postorder numbers of nodes r and s in tree T. Let $\|\cdot\|$ denote the number of nodes in a tree. Let g_T be the gap between r and s in T, and let g_Q be the gap between p and q in Q. The following are the only possible scenarios for nodes p, q, r and s.

Case 1: r is a child of s, and p is a child of q. (Refer to Figure 7(a) and (b).) Let K_t be the subtree of T rooted at s whose nodes have their postorder number n_{K_t}, $n_{rT} < n_{K_t} \leq n_{sT}$. Let K_q be the subtree of Q rooted at q whose nodes have their postorder number n_{K_q}, $n_{pQ} < n_{K_q} \leq n_{qQ}$. We know that $\|K_q\| \leq \|K_t\|$, since Q matches a tree $T^{'}$ that is a subgraph of T. Also, by virtue of postorder numbering,

$$g_T = n_{sT} - n_{rT} = \|K_t\| \quad \text{and} \quad g_Q = n_{qQ} - n_{pQ} = \|K_q\|.$$

Therefore g_Q and g_T have the same sign and $|g_Q| \leq |g_T|$.

Case 2: s is a descendant of r, and q is a descendant of p. (Refer to Figure 7(c) and (d).) Let m_Q be the number of nodes in Q whose postorder number n_Q, $n_{qQ} < n_Q < n_{pQ}$. Let m_T be the number of nodes in T whose postorder number n_T, $n_{sT} < n_T < n_{rT}$. By virtue of postorder numbering,

$$g_T = n_{rT} - n_{sT} = m_T + 1 \quad \text{and} \quad g_Q = n_{pQ} - n_{qQ} = m_Q + 1.$$

Since Q matches tree $T^{'}$ that is a subgraph of T, $m_Q \leq m_T$. Therefore g_Q and g_T have the same sign and $|g_Q| \leq |g_T|$.

Case 3: p and q are the same nodes (*i.e.*, $Label(p, Q) = Label(q, Q)$ and $Number(p, Q) = Number(q, Q)$), and r and s are the same nodes (*i.e.*, $Label(r, T) = Label(s, T)$ and $Number(r, T) = Number(s, T)$). This is a trivial case. g_Q and g_T are both zero.

Proof of (If) part. Given LPS(Q) matches a subsequence S of LPS(T) where S is a tree sequence. Also LPS(Q) is gap consistent and frequency consistent with S. Let tree $T^{'}$ be a subgraph of T that S represents (*i.e.*, LPS($T^{'}$) $= S$). From the *proof of (Only If) part*, LPS($T^{'}$) is gap consistent with S, and LPS($T^{'}$) and S are frequency consistent. We know that LPS(Q) and LPS($T^{'}$) are identical. Our goal is to show that NPS(Q) and NPS($T^{'}$) are identical. We shall use contradiction to prove this.

Q and $T^{'}$ are trees with n nodes numbered from 1 to n in postorder. Let N_Q be the NPS of Q and let $N_{T'}$ be the NPS of $T^{'}$. Let us assume that $\exists i$ where i is the largest index, such that, $N_Q[i] \neq N_{T'}[i]$. The index i will always be less than $n-1$, since $N_Q[n-1] = N_{T'}[n-1] = n$, where n is the number of nodes in trees Q and $T^{'}$ [4]. Let $N_{T'}[i] = a$ and let $N_Q[i] = b$ such that $a \neq b$. Let $N_Q[i+1] = N_{T'}[i+1] = c$. Let us refer to the node numbered i as node i. Based on Lemma 1, in tree Q, the parent of node numbered i has postorder number b, and the parent of node $i+1$ has postorder number c. Similarly, in tree $T^{'}$, the parent of node numbered i has postorder number a, and the parent of node $i+1$ has postorder number c. Consider the following relationships between a, b and c.

[4] Due to postorder numbering, the root of the tree has the highest number and its child is the last node to be deleted during Prüfer sequence construction.

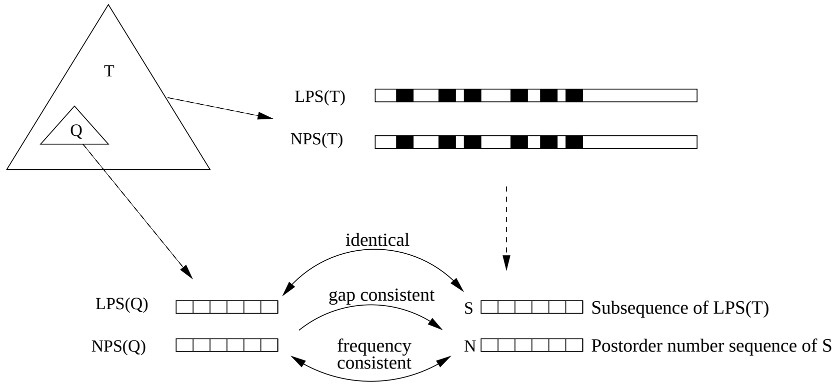


Figure 9: Data and query sequences, and the relationships between them

Case 1: $a < c < b$ (or $b < c < a$). This situation cannot occur since LPS(Q) and LPS($T^{'}$) are each gap consistent with S.

Case 2: $a = c$ (or $b = c$). Then the gap is zero. Since LPS(Q) is gap consistent with S and LPS($T^{'}$) is gap consistent with S, $b = c$ (or $a = c$) should be true. Hence our assumption that $a \neq b$ is false. Therefore $a = b = c$.

Case 3: $c > a, c > b$. (Refer to Figure 8(a) and (b).) Without loss of generality, let us assume that $a < b$. Since $T^{'}$ is numbered in postorder and $a < c$, $i + 1 = a$, and $i = a - 1$ (Figure 8(a)). Now in tree Q (Figure 8(b)), the node i (*i.e.*, $a - 1$), has its parent numbered b and the node $i + 1$ (*i.e.*, a), has its parent numbered c. Also $a < b < c$. This contradicts the fact that Q is numbered in postorder *i.e.*, node c cannot be a descendant of node b. Hence our assumption that $a < b$ is false. Using a similar argument and swapping the roles of $T^{'}$ and Q, we can show that an assumption $a > b$ is also false. Thus our assumption that $a \neq b$ is false, and $a = b$ must be true.

Case 4: $c < a, c < b$. (Refer to Figure 8(c) and (d).) Without loss of generality, let us assume that $a < b$. Since $c < a, c < b$, we have a scenario as shown in Figure 8(c, d) for the nodes i and $i+1$. The dotted line in the figure indicates that there is an ancestor-descendant relationship between the nodes. Node c is the parent of node $i + 1$. Node a in $T^{'}$ and node b in Q have at least one child with postorder number greater than i. Also node $a - 1$ is a child of a in $T^{'}$ and node $b - 1$ is a child of b in Q. As a result of Lemma 1, $N_{T'}[a - 1] = a$ and $N_Q[b - 1] = b$. We know that LPS($T^{'}$) is frequency consistent with S. This means that the i^{th} and $(a - 1)^{th}$ element in the postorder number sequence of S should be equal. Since LPS(Q) is frequency consistent with S, the i^{th} and $(a - 1)^{th}$ element in N_Q should be equal. This implies that $N_Q[a - 1] = b$ and $N_{T'}[a - 1] = a$. This contradicts the original assumption that i is the largest index such that $N_Q[i] \neq N_{T'}[i]$ since $i < c < a$. Hence our assumption that $a \neq b$ (*i.e.*, $a < b$) is false. Using a similar argument, we can show that an assumption $a > b$ is also false. Hence $a = b$ must be true.

From the above four cases, we conclude that our original assumption that $\exists i$ where i is the largest index s.t. $N_Q[i] \neq N_{T'}[i]$ is false. Thus NPS($T^{'}$) and NPS(Q) are identical. Since every Prüfer sequence corresponds to a unique labeled tree (one-to-one correspondence), Q matches $T^{'}$ except that the labels of their leaf nodes (*i.e.*, $Label(\cdot)$) may not match. Thus Q has a partial match in T. $\square$

The different relationships between the data and query sequences are illustrated in Figure 9. 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 the 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, it is evident that (1) LPS(Q) and S are identical, (2) NPS(Q) is *gap consistent* with N, and (3) 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 to find *complete twig matches.*

Example 7 *The leaf nodes of tree T in Figure 2, namely, (D,2), (D,4), (E,5), (G,10), (F,11), and (F,12) are stored in the database. Let tree Q (Figure 2(b)) be a twig query. 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 the 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 since 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. In a similar way, 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. Indeed we have 'C' and '3' as the* 2^{nd} *element in LPS(T) and NPS(T) respectively.* □

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 are examined during subsequence matching and refinement by connectedness and structure phases. This processed is explained in Section 5.7.

4.5 Processing '//' and '*' in Twig Queries

We explain how the axis '//' and the wildcard '*' can be handled in twig queries using the following example.

Example 8 *Let us find the pattern Q = //A//C/D in tree T (in Figure 2(a)). Q is transformed to its Prüfer sequences by ignoring '//' (and '*' if present). 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 '//' 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 and 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 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 be examined in the next phase.*

If the wildcard '' appears as a branch node in the query, then a few modifications are required. For example, consider the query //A/*[B/D]/C/F. In this case, B C A is the sequence representation for the query. If matching subsequences are found, then the NPS is used to test if A can be reached from B and C in one step as explained before. In addition, it should be ensured that the parent of B and C are one and the same. This can be done as follows. Let* n_B *and* n_C *represent the postorder numbers of B and C in the data respectively. If the condition* $N_T[n_B] = N_T[n_C]$ *is true, then B and C have the same parent.* □

Note that in the above example, the leaf nodes of the query patterns do not appear in the sequences. In Section 5.7, we explain how the leaf nodes can be made to appear in the sequences.

5 Implementation Issues in PRIX

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

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. An algorithm describing the sequence construction using a SAX based interface is provided in Section 6. Note that the entire XML document need not be available in memory during sequence construction and can be processed in document order. We shall show in our experiments that the bottom-up transformation is useful to process twig queries efficiently.

5.2 Indexing Sequences Using B$^+$-trees

The set of Labeled Prüfer sequences for the XML documents are indexed in order to support fast subsequence matching during 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. Note that the set of Numbered Prüfer sequences are stored in the database (*e.g.*, as records in a heap file).

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 [26]. Since the virtual trie is a dynamic index, XML documents can be added to and deleted from the database.

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 [18, 30]. 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. This index is called *Trie-Symbol index*. In addition, we store the identifier of each document tree in a separate B$^+$-tree and index it using the `LeftPos` of the node, where its 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 on the set of sequences [26], hence the name virtual trie. However, this dynamic labeling scheme suffers from *scope underflows* [26] 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. The length of the Prüfer sequence of a tree 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.)

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 a subsequence match;

procedure *FindSubsequence*$(Q_s, i, (q_l, q_r))$
begin
1: $R \leftarrow RangeQuery(T_{Q_{si}}, (q_l, q_r))$; // find the descendants of the current node in the trie
2: **for** *each r in R* **do**
3: $S_i \leftarrow Level(r)$; // note the match position
4: **if** $(i = |Q_s|)$ **then**
5: $D \leftarrow RangeQuery(DocidIndex, [r_l, r_r])$; // find the candidate documents
6: output (D, S);
else
7: *FindSubsequence*$(Q_s, i+1, (r_l, r_r))$; // recursively find the subsequences
endif
endfor
end

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 $T_{Q_{si}}$ (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 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*$(\cdot)$ 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.*, the 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.

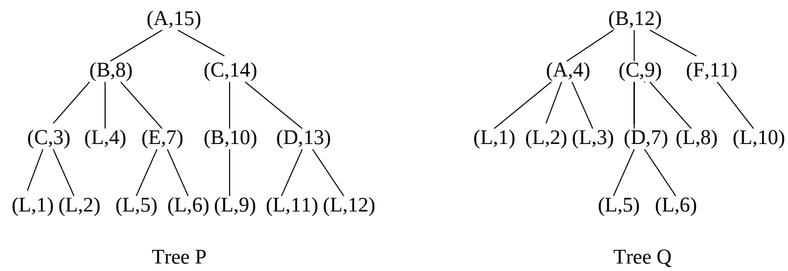
5.4 Optimized Subsequence Matching

In order to speed up subsequence matching, it is desired to reduce the number of range queries performed in Algorithm 1 without causing any false dismissals. In this regard, we propose two optimizations, namely (1) *Bi-directional subsequence matching* and (2) *Subsequence matching using MaxGap metric.*

5.4.1 Bi-directional Subsequence Matching

Nodes in a query twig may have different selectivities for range searches, and this may affect the performance of subsequence matching. The cost of subsequence matching can be higher if a node with low selectivity is in the beginning of the LPS of a query and the subsequence match is carried out from the left to the right of the sequence.

We propose an optimization called *Bi-directional subsequence matching* to handle such cases. We build a trie "*LTrie*" that indexes all the LPS's of the data trees *from the left to the right*. We build another trie "*RTrie*" that indexes the LPS's of the data trees in the reverse order (*i.e.*, *from the right to the left*). For each *LTrie* and *RTrie*, we build a separate *Docid index*. Let Q_{sj} denote the j^{th} label in Q_s. (Q_s is the LPS of Q.) Let us call Q_{sj} the *pivot*. In order to find all subsequences matching Q_s, we first invoke



LPS(P) = C C B B E E B A B C D D C A
NPS(P) = 3 3 8 8 7 7 8 15 10 14 13 13 14 15

LPS(Q) = A A A B D D C C B F B
NPS(Q) = 4 4 4 12 7 7 9 9 12 11 12

Figure 10: Examples for MaxGap

$FindSubsequence(\cdot)$ for the sequence $Q_{sj}, Q_{s(j+1)}, ..., Q_{sk}$ using *LTrie*. Next we invoke $FindSubsequence(\cdot)$ for the sequence $Q_{sj}, Q_{s(j-1)}, ..., Q_1$ using *RTrie*. The two partial results are combined to determine all the matching subsequences in the data. After this step, the regular refinement steps are performed.

A query optimizer can use the selectivity information of the labels in Q_s to determine the pivot for subsequence matching. In the current version of our system, the frequency of occurrence of a label in the collection of Prüfer sequences (constructed over the XML documents) is used to determine the pivot. Thus among the node labels in Q_s, we choose the one that has the minimum frequency of occurrence. A simple histogram can be built over the dataset to estimate the frequency of each node label (*i.e.*, elements/attributes/values) in the Prüfer sequence collection for the XML documents. Thus we can avoid low selectivity node labels being searched first during subsequence matching, thereby reducing the total I/O cost. We observed from our experiments that Bi-directional subsequence matching yielded considerable improvement in performance during query processing.

5.4.2 Subsequence Matching using MaxGap Metric

We can prune some nodes (r in line 2 of Algorithm 1) with an additional requirement on the gap between node labels in a query sequence. In this regard, we have developed an upper-bounding distance metric called *MaxGap* 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 10, the difference in the postorder numbers of the first child and the last child of node label A is $14 - 8 = 6$ in tree P and $3 - 1 = 2$ in tree Q. Hence $MaxGap(A, \{P, Q\})$ 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 showed that the i^{th} node to be deleted during Prüfer sequence construction is the node numbered i. Consider tree P in Figure 10. 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 the last child of a node e denotes how far apart the first and the last occurrence of its label 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 query twig and C, B are adjacent in the query sequence. The CB of this query has eight matches in LPS(P) (Figure 10) which are denoted by position pairs (1,3), (1,4), (1,7), (1,9), (2,3), (2,4), (2,7), and (2,9). Each number pair 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. Among the eight matches listed above, only four position pairs (1,3), (1,4), (2,3), and (2,4) may be considered for further processing. Thus this example illustrates how $MaxGap$ helps to 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 LPS's 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 $j - i \geq MaxGap(A, \Delta)$.*

Proof. Part(1): If A is a child of B in Q and in the data, then the first and the last occurrence of A in the data sequence can occur at a distance of at most $MaxGap(A, \Delta)$ apart. The element B will follow the last occurrence of A. Thus the distance between A and B (A occurs before B) in the data sequence is at most $MaxGap(A, \Delta) + 1$.

Part(2): If A is an ancestor of B in Q and in the data, then the first and the last occurrence of A in the data can occur at a distance of at most $MaxGap(A, \Delta)$ apart. Since B is a descendant of A, it occurs between the first and the last occurrence of A (Lemma 1). Then the distance between A and B (A occurs before B) in the data sequence is at most $MaxGap(A, \Delta) - 1$. □

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 conditions in Theorem 4 using $MaxGap$ of Q_{i-1}. The extended algorithm called $FindSubsequenceExt$ is described in Algorithm 2. The algorithm is invoked by $FindSubsequenceExt(Q_s, 1, (0, MAX\_INT))$.

Both *"Bi-directional subsequence matching"* and *"Subsequence matching using MaxGap metric"* can be combined together by simple modifications to the filtering algorithm.

5.5 The Refinement Phases in PRIX

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 3. The goal of the algorithm in its current form is to demonstrate the core ideas during the refinement phases in a simple way. However, the time complexity of the algorithm can be improved by using sorting. The details are deferred until Section 5.12 where we analyze the CPU and I/O costs incurred by PRIX during query processing.

The NPS's and the set of leaf nodes for the documents in D are read from the database and passed as input to this algorithm. Each input subsequence is refined by connectedness (Theorem 2) in lines 1 through 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 14. Finally, the algorithm matches the leaf nodes of the query twig in lines 15 through 17. This step can be eliminated by special treatment of leaf nodes in the query twigs and the data trees. (Refer to Section 5.7). We report a twig match in line 18. Note that this algorithm does not handle the axis `'//'` and wildcard `'*'`, but it can be easily extended (as mentioned in Section 4.5) by replacing lines 1 through 4 by the procedure $Connectedness(\cdot)$ described in Algorithm 4. In Algorithm 4, we follow a series of edges till we reach a node's ancestor in the data trees. For wildcard '*', we check if the node's ancestor can be reached by following two edges (lines 7 through 8). For the axis '//', we check if the node's ancestor can be reached by following one or more edges (lines 9 through 11). It is assumed that the procedure $NodeType(\cdot)$ returns the type of wildcard associated with a query node.

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

Algorithm 2: Filtering Algorithm Using MaxGap Metric

```
   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 the document (tree) identifier;
            S denotes the positions of a subsequence match;

  procedure FindSubsequenceExt(Q_s, i, (q_l, q_r)))
   begin
1:    R ← RangeQuery(T_{Q_si}, (q_l, q_r)); // find the descendants of the current node in the trie
2:    for each r in R do
3:       S_i ← Level(r); // note the match position
4:       if i > 1 and Q_{i-1} is a child of Q_i then
5:          if S_i − S_{i-1} > MaxGap(Q_{i-1}, Δ) + 1 then  goto 2; // discard current r
         else
6:          if i > 1 and Q_{i-1} is an ancestor of Q_i then
7:             if S_i − S_{i-1} > MaxGap(Q_{i-1}, Δ) then  goto 2; // discard current r
            endif
         endif
8:       if (i = |Q_s|) then
9:          D ← RangeQuery(DocidIndex, [r_l, r_r]); // find the candidate documents
10:         output (D, S);
         else
11:         FindSubsequenceExt(Q_s, i + 1, (r_l, r_r)); // recursively find the subsequences
         endif
      endfor
   end
```

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 appear as leaf nodes in the document tree. The document tree is transformed into a sequence after adding dummy child nodes. 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 number of root-to-leaf paths are explored in the virtual trie of `EPIndex` than in the virtual trie of `RPIndex`. 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.

5.7 Avoiding Refinement by Matching Leaf Nodes

We now explain how the final refinement phase of matching leaf nodes can be avoided by special treatments of leaf nodes in a query twig. The key idea is to allow the leaf nodes of a query twig appear in its LPS, so that all the query nodes are examined during the filtering and refinement phases.

For twig queries with value nodes (*i.e.*, `CDATA`, `PCDATA`), PRIX uses Extended Prüfer sequences. These value nodes, which are leaf nodes in the original query twig (before extending the twig by adding dummy child nodes), are examined during the subsequence matching phase and the first two refinement phases. The dummy child nodes of the query twig will match the dummy child nodes in the data trees, because the value nodes are extended in a consistent way in both the data trees and query twigs. As a result, the final

Algorithm 3: Refinement Phases

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 denotes the positions of a subsequence match in LPS(D);
Output: report twig match;

procedure *RefineSubsequence*(N_D, N_Q, L_D, L_Q, S)
begin
// test for connectedness (refinement by connectedness)
$maxN \leftarrow max(N_D[S_1], N_D[S_2], ..., N_D[S_{|S|}])$; // maximum postorder number
for $i = 1$ *to* $|S|$ **do**
if $N_D[S_i] \neq maxN$ *AND* $\nexists(j > i)$ *s.t.* $N_D[S_i] = N_D[S_j]$ **then**
if $N_D[S_i] \neq S_{i+1}$ **then return**; // check the last occurrence for connectedness
endif
endfor
// test for gap consistency (refinement by structure)
for $i = 1$ *to* $|S| - 1$ **do**
$dataGap \leftarrow N_D[S_i] - N_D[S_{i+1}]$;
$queryGap \leftarrow N_Q[i] - N_Q[i + 1]$;
if *(dataGap* $= 0$ *AND queryGap* $\neq 0$*) OR (queryGap* $= 0$ *AND dataGap* $\neq 0$*)* **then**
return;
else
if $dataGap * queryGap < 0$ **then return**;
else if $|queryGap| > |dataGap|$ **then return**;
endif
endfor
// test for frequency consistency (refinement by structure)
for $i = 1$ *to* $|S|$ **do**
for $j = 1$ *to* $|S|$ *AND* $j \neq i$ **do**
if $N_Q[i] = N_Q[j]$ *AND* $N_D[S_i] \neq N_D[S_j]$ **then return**;
endfor
endfor
// match leaves (refinement by matching leaves)
// (can be omitted when Extended Prüfer sequences are used)
for *each* l *in* L_Q **do**
if l *not found in* L_D **then**
if l *not found in LPS/NPS of D* **then return**;
endif
endfor
report twig match;
return;
end

refinement by leaf nodes phase need not be performed.

On the other hand, for element nodes (*i.e.*, element and attribute tags), which are leaf nodes in the original query twig, we do the following. We extend these nodes in the original query twig with dummy child nodes and perform subsequence matching and the first two refinement phases. However a simple additional step is required at the end of the refinement phases. Suppose we want to find all occurrences of the query pattern `//A/B/C` in tree T shown in Figure 2(a). The LPS of T is *A C B C C B A C A E E E D A* and the NPS is *15 3 7 6 6 7 15 9 15 13 13 13 14 15*. The LPS and NPS of this query pattern after extending the leaf node `C` is *C B A* and *2 3 4* respectively. Two subsequence matches are found in the data with an

Algorithm 4: Refinement by Connectedness with support for Wildcards

Input: $\{N_D, N_Q, S\}$: N_D is the NPS of tree D;
N_Q is the NPS of query Q;
S denotes the positions of a subsequence match in LPS(D);
Output: true if the subsequence passes the test, false otherwise;

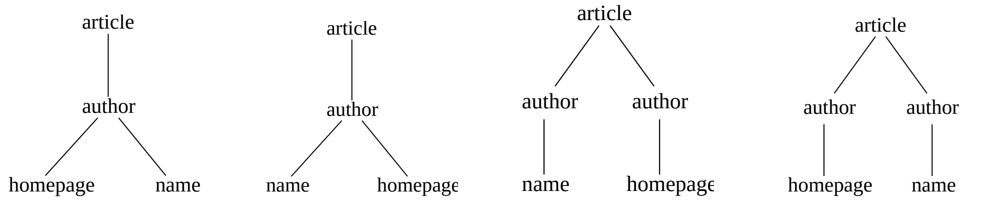
procedure *Connectedness*(N_D, N_Q, S)
begin
  $maxN \leftarrow max(N_D[S_1], N_D[S_2], ..., N_D[S_{|S|}])$; // find the maximum postorder number
  for $(i = 1$ *to* $|S|)$ **do**
    if $N_D[S_i] \neq maxN$ *AND* $\nexists(j > i)$ *such that* $N_D[S_i] = N_D[S_j]$ **then**
      q $\leftarrow N_D[S_i]$;
      if *NodeType*($N_Q[i]$) = '*' *OR* '//' **then**
        let $N_Q[k]$ be the closest ancestor of $N_Q[i]$ in Q such that $k > i$;
        if *NodeType*($N_Q[i]$) = '*' **then**
          // check if ancestor has been reached
          if $N_D[N_D[q]] \neq N_D[S_k]$ **then return** false;
        else
          if *NodeType*($N_D[S_i]$) = '//' **then**
            while $q < N_D[S_k]$ **do** q $\leftarrow N_D[q]$;
            // check if ancestor has been reached eventually
            if $q \neq N_D[S_k]$ **then return** false;
          endif
        endif
      else
        if $q \neq S_{i+1}$ **then return** false;
      endif
    endif
  endfor
end

identical postorder number sequence (*i.e.*, *6 7 15*) at the end of the refinement phases. One match occurs at positions *4, 6, 7* and the other match occurs at positions *5, 6, 7* in the document sequence. However both these matches correspond to the same set of nodes in T *i.e.*, (C,6), (B,7) and (A, 15). Hence only one of these matches should be output as a solution. A simple way to achieve this in the above example is to choose the subsequence where label `C` corresponds to the last occurrence of node 6 in NPS(T) *i.e.*, the match at positions *5, 6, 7* in the document sequence.

If the element nodes that are leaf nodes in the original query twig have parent-child edges, we assign an upper bounding distance metric of zero to these element nodes. Algorithm *FindSubsequenceExt* discards redundant matches without any modifications. If these element nodes have ancestor-descendant edges, then we discard these redundant matches by using the NPS's after the refinement phases.

5.8 Unordered Twig Matches

The PRIX system can be extended to find unordered twig matches by a simple modification of the Prüfer sequence construction. For a given twig query, Prüfer sequences should be constructed for different arrangements of the branches in the query twig, and they should be 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. This process could be implemented efficiently by identifying common prefixes among the sequences for different twig configurations in order to avoid repeated subsequence matching on common prefixes of the sequences. However, not all configurations may yield valid solutions. In the following example we explain how some twig configurations can be quickly eliminated.



(a) Configuration 1 (b) Configuration 2 (c) Configuration 3 (d) Configuration 4

Figure 11: Different query twig configurations

Example 9 *Consider a query twig Q =* `//article[@key="TR"][title="XML"]/year=2003`*. Since attribute tag name* `key` *can be considered as a child node of* `article`*, Q has 3 root-to-leaf paths (branches). We shall construct the Extended-Prüfer sequence for this query twig. In one twig arrangement, value tag* `XML` *appears before value tag* `2003` *in the LPS of Q. In another arrangement, value tag* `2003` *appears before tag* `XML` *in the LPS. If tag* `2003` *never appears before tag* `XML` *in the data sequences, we can eliminate the latter arrangement by ensuring that* `2003` *does not have* `XML` *as its descendant in the virtual trie. This check can be done faster than performing a complete subsequence match. In the case of the root-to-leaf path resulting from attribute tag* `key`*, we need not rearrange it w.r.t. other paths. This is because at the time of indexing XML documents, we can always treat the attribute tag* `key` *as the first child of* `article`*. The query twig can be transformed similarly by treating* `key` *as the first child of* `article`*. Thus in all, query twig Q has only one arrangement to be tested in which* `XML` *appears before* `2003` *in the LPS of Q.* □

Note that in order to be consistent with the XPath semantics, the twig configurations need to be carefully chosen. For example, let us consider the following XPath expression.

`//article[author/homepage][author/name]`

Then the different ordered twig pattern configurations that are possible are shown in Figure 11. If the DTD is available, some of the query configurations can be discarded. For example, if `author` element always has both `homepage` and `name` as its subelements, then Prüfer sequences need to be constructed for configurations in Figures 11(a) and (b). As a result, the query processing will output the correct matches.

5.9 Queries with Inequality and Positional Predicates

PRIX can be extended to handle inequality predicates. Consider an XPath query `//book[year = "2004"][price > "100"]`. This query should return all the `book` elements whose `price` is greater than '100' published in the `year` '2004'. To process such queries, we can first find all matches for the pattern `//book[year="2004"][price]`. Since the tag-number pair of the leaf nodes in the data trees can be stored in the database, we can fetch them for each matched document and perform the *Refinement by Matching Leaf Nodes* phase using the leaf node's tag-number pair ('100', 1) with a slight modification. Thus instead of an equality match, we need to perform an inequality match.

PRIX can also be extended to handle queries that contain positional predicates. For example, an XPath query `//book/author[n]`, returns the n^{th} `author` element under each `book` element in the XML documents. Since the Prüfer tree-to-sequence transformation preserves all information about the data trees, PRIX can process such queries with additional post-processing.

First, all occurrences of the query pattern `//book/author` can be found in the data using our proposed techniques. Let (L, N) be the LPS and NPS of a document with a match. Let (a_i, b_i) denote the a_i^{th} and b_i^{th} entry in L (and N) that match `author` and `book` respectively. Let (a_1, b_1), (a_2, b_2),..., (a_t, b_t) denote all such entry pairs for all the matches of (`author`, `book`) in the document. Note that $a_i < b_i$ for $1 \leq i \leq t$ because in the LPS of the query pattern, `author` appears before `book` (Lemma 1). Note that $N[a_i]$ and $N[b_i]$ denote the postorder numbers of the nodes labeled `author` and `book` respectively in the input document. First we group each pair (a_i, b_i) based on the value $N[b_i]$. For each group, the pair (match) with the n^{th} smallest $N[a_i]$ value is output. If the positional predicate has a descendant axis (*e.g.*, `//book//author[n]`) then the (a_i, b_i) pairs are first grouped using the value $N[b_i]$ as before. In addition, each group is further grouped into smaller groups using the value $N[N[a_i]]$ (postorder number of the parent of `author` in the document). From each smaller group, the pair (match) with the n^{th} smallest $N[a_i]$ value is output.

5.10 Alternate Strategy for Subsequence Matching

In this section, we shall discuss the use of a different strategy for subsequence matching. The subsequence matching process in PRIX is I/O bound. As the selectivity of the queries reduces, the number of range queries increases due to the increase in the number of paths in the virtual trie that need to be traversed. For such twig queries, the use of Algorithm 1 cannot guarantee a worst-case I/O that is linear in the total number of instances of the tags (of the query pattern) appearing in the Prüfer sequences of the XML data. This is due to the nature of subsequence matching that does random I/O to process the range queries. It is interesting to note that the process of subsequence matching in a virtual trie is similar to that of finding matches for simple path expressions in an XML document. For example finding all subsequences of the sequence ABC in a virtual trie, is equivalent to finding the path expression `//A//B//C`. As a matter of fact, the `PathStack` [5] algorithm proposed by Bruno *et al.* can be used for this purpose. The use of `PathStack` can guarantee a worst-case linear I/O for queries as their selectivity reduces. Furthermore, XB-tree indexes can be built to speed up the subsequence matching process.

5.11 Cost Analysis of Twig Query Processing in PRIX

During query processing in PRIX, the cost of subsequence matching is dominated by I/O. Once the NPS's are fetched from the database, all the subsequent refinement phases are performed in memory without additional I/O. Therefore, for the purpose of analyzing the query performance of PRIX, we will be focused on the I/O cost of the subsequence matching phase and on the CPU cost of the refinement phases.

It is hard to show that Algorithm 1, which performs range queries, guarantees a worst-case disk I/O cost that is linear in the size of the input lists (data pages in the *Trie-Symbol* indexes). This is because a data page may be read from the disk more than once during range searches. However, by using the alternate subsequence matching strategy explained in Section 5.10, the filtering phase can be shown to have a linear worst-case I/O cost like the TwigStack algorithm. As for the I/O pattern, the data pages accessed by TwigStackXB and TSGeneric$^+$[15] algorithms tend to be scattered, because these algorithms scan input lists through indexes potentially several times in different orders. The I/O pattern of PRIX is also random since the Trie-Symbol indexes for the elements and values are accessed in different orders during the subsequence matching phase.

Our algorithm does not guarantee the optimality of the CPU cost during the refinement phases. This is because a subsequence found in the filtering phase may not be part of the final answer. Note that these subsequences can be stored in memory using a compact stack encoding proposed by Bruno *et al.* [5]. Let l be the length of $LPS(Q)$ and k be the number of subsequences that match $LPS(Q)$. Suppose N denotes the postorder number sequence of one such matching subsequence S. For testing connectedness, sorted versions of both N and $NPS(Q)$ are created in the ascending order using [*postorder number, position in sequence*] as the key. Let m be the postorder number and i be the position in the original sequence. Then in the sorted sequence, any key $[n, j]$ to the left of $[m, i]$ satisfies the following property $(n < m) \vee ((n = m) \wedge (j < i))$. This property holds for every $[m, i]$ except the first key in the sorted sequence. So cost for sorting the k subsequences is $O(k \cdot l \cdot log(l))$. Using the sorted version of N, we can test if the last occurrence of each postorder number in N is followed by its parent (Theorem 2) in linear time (*i.e.*, $O(l)$). Gap consistency can be tested in linear time using N and $NPS(Q)$. Finally frequency consistency can be tested in linear time using the sorted versions of N and $NPS(Q)$. Overall the CPU cost is $O(k \cdot l \cdot log(l))$.

Now let us consider the case when '//' is present in the query. As explained in Section 4.5, the respective NPS's are examined by traversing a series of edges. However the fact that the NPS's are read in blocks (*e.g.*, heap files), the I/O cost is already accounted for in the d pages. Hence the additional cost for processing '//' in the query is a CPU cost of $O(k \cdot h)$ where h is the height of the document tree. In the case of '*', the query processing incurs an additional CPU cost of $O(k \cdot b)$ where b is the number of child nodes of the '*' node.

Algorithm 5: Constructing Prüfer Sequences

```
Input: {T_sax}: SAX parser output for a well-formed XML document T
Output: {(L, N)}: L - LPS of T; N - NPS of T

Global data structures: stack Stk; integer nodeID, count;
procedure RegularPrufer(T_sax)
begin
    Stk.clear(); // clean up the stack
    nodeID ← 1; // postorder numbering starts from 1
    count ← 0;
    (tagName, type) ← T_sax.next(); // read the first line of the SAX output
    GenerateSequence(T_sax, (tagName, type));
    return (L, N);
end
procedure GenerateSequence(T_sax, (tagName, type))
begin
    numChildren ← 0; // keeps track of the number of children of tagName
    while true do
        if T_sax.eof() then return;
        (tagNameNext, typeNext) ← T_sax.next();
        if typeNext = END then break; // end tag encountered
        GenerateSequence(T_sax, (tagNameNext, typeNext)); // recursively traverse the tree
        Stk.push(nodeID); // store the node's postorder number
        nodeID ← nodeID + 1;
        numChildren ← numChildren + 1;
    endw
    // If the leaf node is reached, store it
    if numChildren = 0 then
        count ← count + 1;
        store the leaf's (hash(tagName), count) pair;
        return;
    endif
    // Once a non-leaf node is numbered, update entries in the
    // LPS/NPS corresponding to the deletion of the node's children
    for k = 1 to numChildren do
        N[Stk.top()] ← count + 1;
        L[Stk.top()] ← hash(tagName);
        Stk.pop();
    endfor
    count ← count + 1;
    return;
end
```

6 Description of PRIX's Architecture

In this section, we describe the major components in PRIX. (Refer to Figure 3 for an architectural overview of the system.)

Document Parsing An XML document is first parsed using an event-driven SAX parser. The output of the SAX parser is a stream of `start` and `end` tags for each element, attribute, and value in document order. The SAX parser output is then input to the *indexing engine*.

Indexing Engine The elements, values, and attributes are first mapped to unique integers, because it is more efficient to store and process integers than raw strings. The indexing engine constructs Prüfer sequences for XML documents by reading their parsed output. Given a well-formed XML document T, we assume that its SAX parser output stream T_{sax} has functions $next(\cdot)$ and $eof(\cdot)$ associated with it. The function $T_{sax}.next()$ returns the next line in the stream T_{sax} *i.e.*, tag name and tag type (**start** or **end**). The function $T_{sax}.eof()$ checks for the end of the stream. We also assume that a function $hash(\cdot)$ uniquely maps every element, attribute and value in the input data to a number. Algorithm 5 describes the steps involved in constructing Regular Prüfer sequences. The algorithm recursively traverses the tree structure and numbers the tree nodes in postorder starting from 1. The algorithm outputs two sequences (L, N) representing the LPS and NPS of the document T. In the algorithm, once a non-leaf node is numbered, the entries corresponding to the deletion of its child nodes in the LPS and NPS are updated (lines 20 through 23). The above algorithm can be extended to construct Extended Prüfer sequences by making small modifications. Line 18 should be replaced with the following lines: `numChildren = 1; Stk.push(nodeID); nodeId = nodeId + 1;`. Depending on the query workload, the indexing engine can choose to build Regular and/or Extended Prüfer sequences and index them (*i.e.*, `RPIndex` and `EPIndex`).

The purpose of the stack in Algorithm 5 is to store the postorder numbers assigned to a node's children. Once a node is assigned a postorder number, all its children are popped out of the stack. As a result, the maximum depth of the stack is upper-bounded by the maximum fanout of the XML document tree. Moreover, once a node is numbered, the LPS and NPS for the subtree rooted at that node can be generated completely. However, only when the root node is numbered, the LPS and NPS for the entire tree can be generated. If memory is a concern, then we could first write the partial sequences for the subtree rooted at the first child of the root to disk followed by a blank entry that would be filled when the root is assigned a postorder number. This process can be repeated for the other subtrees that follow. Once the root is numbered, the blank entries in the sequences are filled by performing random disk I/O. However, this may not always be necessary. PRIX can be used to index both a collection of XML documents as well as a single large XML document (*e.g.*, DBLP [25]). In the case of a single large XML document, we need not construct a single Prüfer sequence. Rather we could split the document tree at the root and construct sequences for the subtrees rooted at the child nodes of the root. These sequences can then be indexed.

In order to support fast subsequence matching, the LPS's are indexed by building a virtual trie using B^+-trees. The nodes in the trie are assigned number ranges to support containment queries. Note that suffixes of the LPS's need not be stored since all subsequences can be found using Algorithm 1. For each unique label, a B^+-tree is built to store the number ranges of all the instances of that label in the virtual trie. The NPS's and the leaf nodes (if used) are stored in the database (*e.g.*, as records in a heap file).

Querying Engine The LPS and NPS for an XPath query are constructed. All subsequences that match the query's LPS in the data sequences are found by searching the virtual trie. The document identifiers of the data trees that have a matching subsequence are also determined. Based on the nature of the query, the querying engine can either choose the `RPIndex` or the `EPIndex` for the subsequence match phase. Post-processing is performed on the matching subsequences to discard non-matches (*i.e.*, false alarms). The NPS's corresponding to the matching document identifiers are fetched from the database and are used for the refinement-by-connectedness and refinement-by-structure phases. At the end of the refinement phases, all occurrences of the query twig are output.

7 Experimental Results

In our experiments, we compared the query performance of PRIX, TwigStack/TwigStackXB and TSGeneric$^+$ [15] for a set of high selectivity queries. We implemented PRIX, and TwigStack/ TwigStackXB in the C++ language, and used the B^+-tree implementation of GiST [13] for all their indexes. The implementation of TSGeneric$^+$ algorithm was obtained from Jiang *et al.* Since their code was developed on the Microsoft Windows platform, we compared PRIX and TSGeneric$^+$ in terms of disk I/O. Note that the disk I/O for all these algorithms is random in nature (Section 5.11). In our previous work [24], we showed that PRIX outperforms ViST considerably and therefore we do not compare them again in this paper.

PRIX and TwigStackXB/TSGeneric$^+$ are suited for two different application domains. PRIX supports ordered twig pattern matching inherently. TwigStackXB and TSGeneric$^+$ support unordered pattern match-

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

Table 2: Datasets

Query	XPath Expression	# of Matches
Q_1	//Ref/Author="Price S.R"	12
Q_2	//Ref/Author="Moss J"	38
Q_3	//Ref/Author="Vaughan M"	27
Q_4	//Entry[PFAM[@prim_id="PF00304"]][//DISULFID/Descr]	24
Q_5	//Entry[Org][PFAM[@prim_id="PF00304"]][//SIGNAL/Descr]	39
Q_6	//Entry[Species="Vicia faba"][Organelle="Chloroplast"][Org="Vicia"]	4
Q_7	//Entry[Species="Glycine max"][Organelle="Chloroplast"][Org="Glycine"]	9
Q_8	//Entry[Keyword="Ubiquitin conjugation"][Keyword="Zinc"]	7
Q_9	//Entry[Org="Piroplasmida"][Ref/Author="Kemp D.J"]	4

Table 3: XPath Queries for SWISSPROT Dataset

ing. These algorithms can be adapted for ordered pattern matching by performing a post processing step to verify order among siblings. On the other hand, PRIX can be adapted for unordered pattern matching by testing the different twig pattern configurations that result from different sibling orders. Thus PRIX and TwigStackXB/TSGeneric$^+$ are suited for different application domains.

7.1 Experimental Setup

We ran our experiments for PRIX and TwigStack/TwigStackXB on 1.8GHz Pentium IV processor with 512 MB RAM running Solaris 8. A 120GB 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 TSGeneric$^+$, the code was compiled using Microsoft Visual C++ compiler version 6.0. The experiments were run on the Microsoft Windows XP platform. For all the experiments, the buffer pool size was fixed at 2000 pages. The page size of 8 KB was used. For PRIX, 4-byte number ranges were used to label the nodes in the virtual trie. For TwigStack/TwigStackXB and TSGeneric$^+$, the same 4-byte number ranges were used to label the nodes in the XML document trees.

Datasets We experimented with the datasets shown in Table 2. These datasets were obtained from the University of Washington XML repository [25]. We chose these three datasets since each had a different characteristic. The document trees in the SWISSPROT dataset were bushy and shallow. The document trees in the DBLP dataset had high similarity in structure and were shallow. The document trees in the TREEBANK dataset were narrow and had deep recursion of element names. Table 2 provides additional information such as the maximum depth, number of elements for the datasets. PRIX can be used to index and query a collection of XML documents or a single large XML document. In the case of a single large XML document tree (*e.g.*, SWISSPROT) we remove the root element and transform the collection of subtrees into sequences. The sequences are then indexed using a disk-based virtual trie. Table 2 also shows the number of sequences constructed for each dataset.

Queries The twig queries used for our experiments are shown as XPath expressions in Tables 3, 4, and 5. To avoid the frequent use of the axes like `following-sibling` in these tables, we assume that the order between the siblings in a twig query follows the left-to-right order in the corresponding XPath expression. For example, `//phdthesis[year][number]` indicates that `year` is followed by `number`. The listed queries have different characteristics in terms of selectivity, presence of values and twig structure. For the TREEBANK dataset, since the values were encrypted, we chose queries without values (character data). Tables 3, 4, and 5 also show the number of twig matches for each query. In our work, we focus on ordered twig pattern

Query	XPath Expression	# of Twig Matches
Q_{10}	//EMPTY//S//SYM	15
Q_{11}	//S/SBARQ-1	2
Q_{12}	//S[PRT][NP]	2
Q_{13}	//S[ADVP-1/NN][NP]	0
Q_{14}	//NP/ADJP/IN_OR_RB	1
Q_{15}	//NP//NN_OR_NNS	23
Q_{16}	//NP[NEG][NN]	1
Q_{17}	//S//NP[NEG][NN]	2
Q_{18}	//S[NP-1][NN]	0
Q_{19}	//S//NP[NEG/RB][NN]	4
Q_{20}	//EMPTY/*/LS_OR_JJ	1
Q_{21}	//S/*/RB_OR_JJ	2

Table 4: XPath Queries for TREEBANK Dataset

Query	XPath Expression	# of Twig Matches
Q_{22}	//inproceedings/author="Antonin Guttman"	2
Q_{23}	//inproceedings/author="C. J. Date"	12
Q_{24}	//article/author="C. J. Date"	12
Q_{25}	//article/author="E. F. Codd"	33
Q_{26}	//phdthesis[year][series][number]	1
Q_{27}	//phdthesis[year][number]	3
Q_{28}	//inproceedings[author="Jim Gray"][year="1990"]	6
Q_{29}	//inproceedings[key][author="Jim Gray"][year="1990"]	6

Table 5: XPath Queries for DBLP Dataset

matching that is useful for applications that require the query nodes to follow the document order in XML. Note that the query processing time for PRIX, TwigStackXB and TSGeneric$^+$ increases as the number of matches increases. For fairness of comparison, the number of ordered and unordered twig matches for each twig query were the same, *i.e.*, only one query twig configuration was present in the data trees.

7.2 Performance Analysis

In this section, we analyze the query performance of PRIX, TwigStack/TwigStackXB and TSGeneric$^+$ algorithms for high selectivity queries listed above.

The TwigStack algorithm examines every node in the sorted input stream. On the other hand, TwigStackXB uses XB-Trees to skip nodes in the sorted input stream. The effectiveness of XB-Trees depends on the distribution of possible twig matches in the input streams. If the possible solutions are localized in certain regions of the input streams, then XB-Trees are effective in skipping large portions of the input streams. However, if the possible solutions are scattered in the input streams then TwigStackXB is forced to drill down frequently to the lower regions of the XB-Trees to identify true matches. As mentioned in Section 2, TwigStack algorithms are sub-optimal for query patterns with parent/child relationships.

The TSGeneric$^+$ algorithm (partly motivated by TwigStack algorithms) was proposed by Jiang *et al.* [15] to further improve the skipping of elements by using available indexes on the elements. The algorithm uses XR-trees [14] that is based on the concept of interval trees. Using XR-trees, both ancestors and descendants of a given element can be determined efficiently. The TSGeneric$^+$ algorithm takes advantage of high join selectivity edges in the query pattern to skip elements in the input lists. The algorithm uses different strategies for picking edges so that the skipping of elements in other edges of the query pattern can be maximized. In our experiments, we compared PRIX with the version of TSGeneric$^+$ strategy that incurred the minimum I/O cost for fairness. Similar to TwigStack algorithms, TSGeneric$^+$ is sub-optimal for query patterns with parent/child relationships.

Query	TwigStack		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_1	26.53 secs	4543 pages	0.14 secs	18 pages
Q_4	35.39 secs	5563 pages	2.86 secs	377 pages
Q_8	12.88 secs	1715 pages	0.38 secs	40 pages

Table 6: SWISSPROT - TwigStack vs TwigStackXB

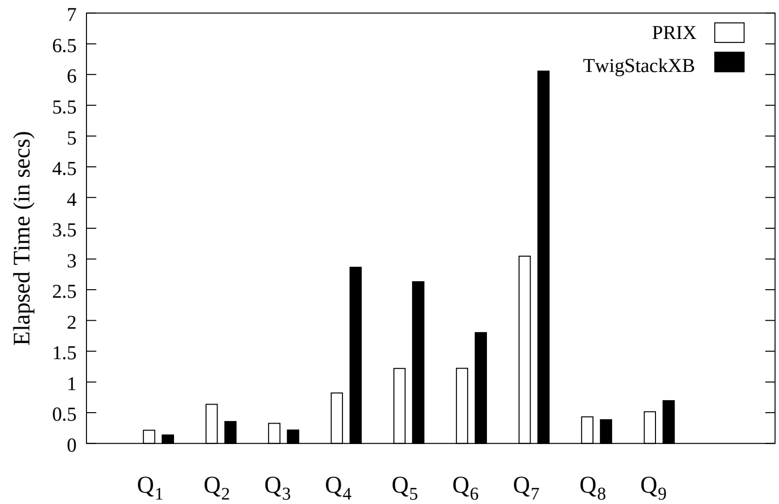


Figure 12: SWISSPROT (PRIX vs TwigStackXB)

The performance of PRIX is dominated by the I/O cost incurred during subsequence matching. The fewer paths are traversed in the virtual trie, the fewer disk pages are accessed. We have proposed optimizations (in Section 5.4) that can speed up the subsequence matching phase by reducing the number of range searches required to find all the subsequences. It should be noted that the query processing cost also depends on the number of NPS's that are read from the database.

7.2.1 SWISSPROT Dataset

PRIX vs TwigStack/XB For the SWISSPROT dataset, we observed that TwigStack performed worse than TwigStackXB for queries in Table 3. The elapsed time and the disk I/O for some of the queries are shown in Table 6.

In Figure 12, the elapsed time for processing the queries in Table 3 using PRIX and TwigStackXB are plotted. PRIX used EPIndex to process the queries. For queries Q_1, Q_3, Q_8 and Q_9, PRIX and TwigStackXB had comparable performance. PRIX performed subsequence matching beginning with nodes with high selectivity due to bottom-up transformation of the query twigs. TwigStackXB found all matches by skipping large sections of the sorted input streams using XB-Trees.

TwigStackXB was faster than PRIX for Q_2 since a large number of matches were clustered in certain regions in the sorted streams. As a result, XB-Trees were effective in reducing the I/O cost.

Query	PRIX		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_4	0.82 secs	84 pages	2.86 secs	377 pages
Q_5	1.22 secs	121 pages	2.63 secs	352 pages
Q_6	1.22 secs	127 pages	1.80 secs	183 pages
Q_7	3.04 secs	411 pages	6.05 secs	666 pages

Table 7: SWISSPROT - PRIX vs TwigStackXB

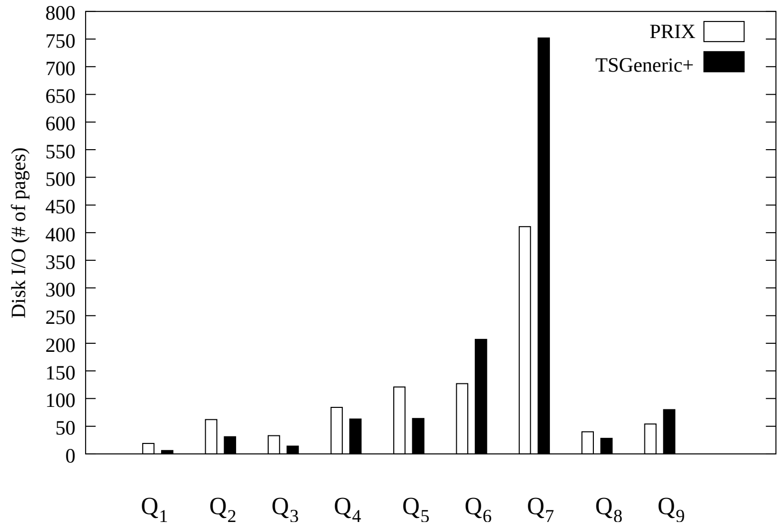


Figure 13: SWISSPROT (PRIX vs TSGeneric$^+$)

PRIX processed queries Q_4, Q_5, Q_6 and Q_7 faster than TwigStackXB. Table 7 shows the elapsed time and disk I/O for these queries. We shall first analyze the performance of queries Q_4 and Q_5. The value node "PF00304" was scattered in the input dataset. However about half the documents containing `//Entry/PFAM[@prim_id=`"PF00304"`]` did not have the pattern `//DISULFID/Descr`. However `//DISULFID/Descr` occurred very frequently in the dataset and as a result TwigStack frequently drilled down to the lower regions of the XB-Trees to eliminate false matches for processing Q_4. This resulted in an increase in disk I/O. The situation was similar for query Q_5 due to the pattern `//SIGNAL/Descr`.

PRIX on the other hand was two to three times faster than TwigStackXB for queries Q_4 and Q_5. Recall that the subsequence matching phase depends on the number of candidate ranges that are searched in the virtual trie to find all the matching subsequences. For Q_4, the subsequence matching was performed beginning with the node "PF00304" due to bottom-up transformation of the query twig. The partial matching documents were eliminated during the filtering stage. For Q_5, Bi-directional subsequence matching was performed using "PF00304" as the pivot since the tag `Org` had low selectivity. PRIX was slower processing Q_5 as compared to Q_4 due to the use of both the *LTrie* and *RTrie* which increased the disk I/O.

Next we shall analyze the performance of query Q_6. The patterns `//Entry/Organelle=`"Chloroplast" and `//Entry[Species=`"Vicia faba"`][Org=`"Vicia"`]` were scattered in the dataset. However not all documents that had the pattern `//Entry[Species=`"Vicia faba"`][Org=`"Vicia"`]` had `//Entry/Organelle=`"Chloroplast". Since these patterns occurred in nearby documents in the data, TwigStack had to access lower level regions of the XB-Trees frequently to skip such document resulting in an increase in disk I/O.

PRIX on the other hand eliminated such matches during subsequence matching by starting with high selectivity nodes due to the bottom-up transformation of the query pattern. As a result fewer paths were traversed in the virtual trie. Similar behavior was observed for query Q_7 and PRIX was nearly two times faster than TwigStackXB.

PRIX vs TSGeneric$^+$ The performance results from PRIX and TSGeneric$^+$ in terms of disk I/O for the SWISSPROT dataset are plotted in Figure 13. Recall that TSGeneric$^+$ can efficiently process twig queries by skipping both ancestors and descendants if the join selectivities of (at least one of) the twig pattern edges are high. For example, the edge `Author=‘‘Price S.R’’` in Q_1 had high join selectivity. On a similar note, PRIX can also process queries efficiently if they have high selectivities since only a few paths in the virtual trie are accessed. Since queries Q_1, Q_2, Q_3 Q_8 and Q_9 had the above characteristics, both PRIX and TSGeneric$^+$ had comparable performances for them.

Of special interest is the performance of queries Q_6 and Q_7. Let us first analyze the query Q_7. The pattern `//Entry/Organelle=‘‘Chloroplast’’` was scattered in the dataset. In addition, the patterns

Query	TwigStack		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_{10}	9.34 secs	1426 pages	0.51 secs	39 pages
Q_{11}	6.55 secs	1060 pages	0.18 secs	19 pages
Q_{15}	14.01 secs	3008 pages	0.24 secs	26 pages

Table 8: TREEBANK - TwigStack vs TwigStackXB

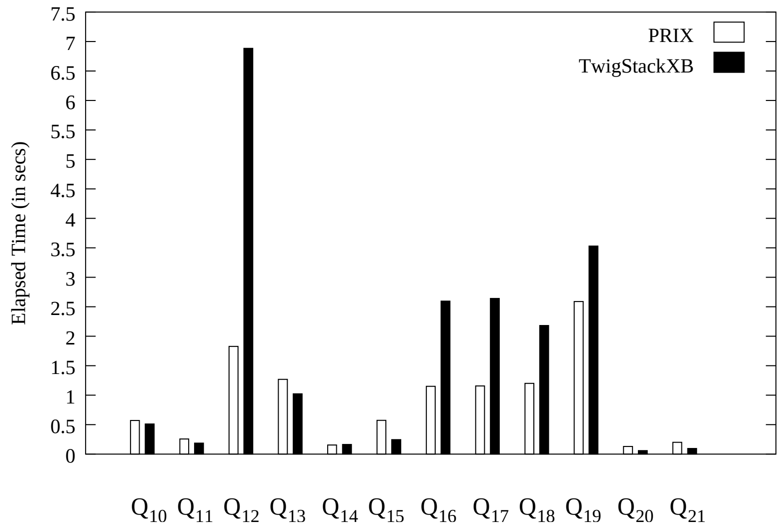


Figure 14: TREEBANK (PRIX vs TwigStackXB)

`//Entry/Species=''Glycine max''` and `//Entry/Org=''Glycine''` occurred less frequently than `//Entry/Organelle=''C`
The edge join selectivities for Q_7 was overall lower than that of queries like Q_1, Q_2, etc. As a result, the effectiveness of skipping elements using the XR-tree indexes was diminished and more elements in the input lists were scanned. Overall the I/O cost was significantly higher than that of PRIX. Similarly, for Q_6 PRIX was more efficient than TSGeneric$^+$. We can draw the conclusion that the distribution of probable solutions for different branches of the twig pattern can reduce the effectiveness of XR-trees to skip elements in the input lists. As for Q_5, due to the presence of high join selectivities in the edges (*i.e.*, `@prim_id=''PF00304''`) of Q_5, the query was processed efficiently by TSGeneric$^+$. In contrast, due to the use of Bi-directional subsequence matching, PRIX did not take advantage of the high join selectivities and was outperformed by TSGeneric$^+$.

7.2.2 TREEBANK Dataset

For the TREEBANK dataset, we observed that TwigStack performed worse than TwigStackXB for the queries in Table 4. Table 8 shows the elapsed time and disk I/O for some of the queries. Note that the TREEBANK dataset had deep recursion of element names.

In Figure 14 the elapsed time and disk I/O for the queries in Table 4 using PRIX and TwigStackXB are plotted. For PRIX, we used RPIndex to process all the queries. TwigStackXB and PRIX had comparable performance for queries Q_{10}, Q_{11}, Q_{13}, Q_{14}, Q_{20} and Q_{21}. For query Q_{13}, Bi-directional subsequence matching was performed with `ADVP-1` as the pivot. This resulted in speeding up the filtering process. Note that for query Q_{15}, PRIX was slower than TwigStackXB. Since the twig matches were localized in certain regions of the sorted input stream, TwigStackXB could skip large regions of data using XB-Trees.

As stated in Section 2, TwigStack algorithms are sub-optimal for parent-child relationships in the query twigs. The performance of queries Q_{12}, Q_{16}, Q_{17}, Q_{18} and Q_{19} demonstrate this behavior. We observed that PRIX was two to three times faster than TwigStackXB in many instances. In Table 9 the elapsed time and disk I/O are shown for these queries. Let us first analyze the performance of query Q_{12}. In query

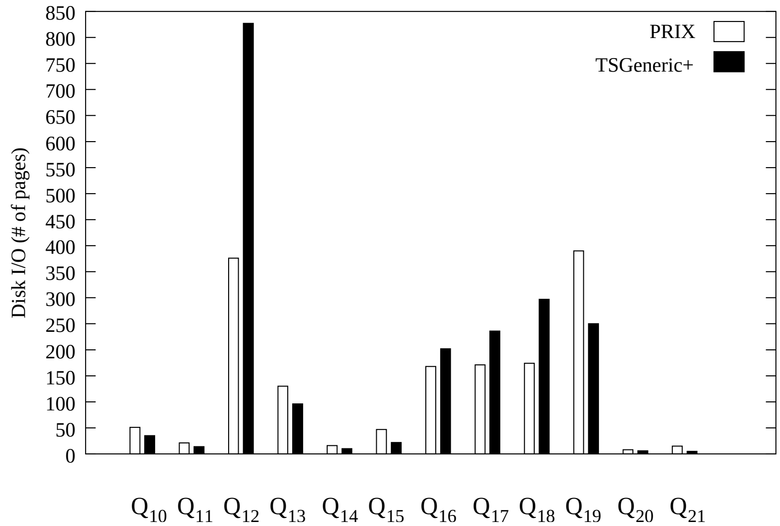


Figure 15: TREEBANK (PRIX vs TSGeneric$^+$)

Query	PRIX		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_{12}	1.82 secs	376 pages	6.88 secs	1,098 pages
Q_{16}	1.15 secs	168 pages	2.59 secs	338 pages
Q_{17}	1.16 secs	171 pages	2.64 secs	360 pages
Q_{18}	1.20 secs	174 pages	2.18 secs	356 pages
Q_{19}	2.59 secs	390 pages	3.53 secs	490 pages

Table 9: TREEBANK - PRIX vs TwigStackXB

Q_{12} the tag `PRT` appeared several times and was scattered in the input stream. However only in two such documents, tags `S` and `PRT` shared a parent-child relationship. In many other documents, `S` was an ancestor of `PRT`. As a result of sub-optimality, TwigStackXB first found all such matches and often drilled down to the leaf nodes of the XB-Trees since these matches were not localized in certain regions (but scattered) of the sorted input streams. This process increased the disk I/O. These false matches are discarded during the merge post-processing step.

PRIX on the other hand was faster than TwigStackXB and performed (optimized) subsequence matching using the high selectivity node `PRT` and used the MaxGap of the node `PRT` to discard those documents with `S` as an ancestor of `PRT`. The MaxGap of `PRT` was 0 in this case. PRIX discarded the false matches early in the subsequence matching phase and fewer disk pages were accessed overall.

Similar was the case for queries Q_{16}, Q_{17}, Q_{18} and Q_{19} and PRIX was faster than TwigStackXB. The tag `NEG` had high selectivity in these queries. Note that PRIX used Bi-directional subsequence matching for Q_{19} with `NEG` as the pivot since the tag `RB` had low selectivity. The MaxGap values for `RB` and `NEG` were 0 and 2 respectively. As expected PRIX took more time to process query Q_{19} as compared to Q_{17} because to process Q_{19} both the *LTrie* and *RTrie* indexes were used for subsequence matching.

PRIX vs TSGeneric$^+$ The I/O cost for PRIX and TSGeneric$^+$ for the TREEBANK dataset are shown in Figure 15. The queries Q_{10}, Q_{11}, Q_{13}, Q_{14}, Q_{15}, Q_{16}, Q_{20} and Q_{21} had comparable performance for both PRIX and TSGeneric$^+$. This was because for these queries, some edges had high join selectivities. For example, in query Q_{10} the edge `S//SYM` had high join selectivity. As a result, TSGeneric$^+$ was able to effectively skip elements using the XR-tree indexes. Similarly, PRIX used high selectivity nodes for subsequence matching and processed these queries efficiently.

Similar to TwigStackXB, TSGeneric$^+$ suffers from sub-optimality for parent-child edges in the queries. TSGeneric$^+$ first found all the ancestor-descendant matches and the non-matches that do not satisfy parent-

Query	TwigStack		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_{22}	26.77 secs	6409 pages	0.25 secs	24 pages
Q_{26}	15.50 secs	3036 pages	0.70 secs	71 pages

Table 10: DBLP - TwigStack vs TwigStackXB

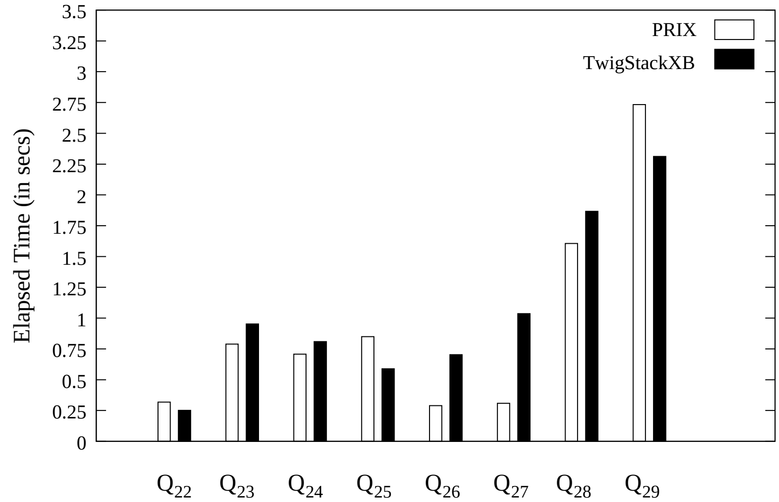


Figure 16: DBLP (PRIX vs TwigStackXB)

child relationship were eliminated during postprocessing. However, since the dataset contained many such non-matches, TSGeneric$^+$ incurred considerably more I/O than PRIX to process queries Q_{12}, Q_{17}, Q_{18}. On the other hand, PRIX performed optimized subsequence matching using the MaxGap values for query nodes and was able to prune out the non-matches early during query processing. Thus PRIX outperformed TSGeneric$^+$ for these queries.

7.2.3 DBLP Dataset

As with the other datasets, we observed that TwigStack performed worse than TwigStackXB for the queries in Table 5. Table 10 summarizes the performance for some of the queries in the DBLP dataset. Since the queries had high selectivity, TwigStackXB was effective in skipping a lot of the data in the input streams, thereby resulting in fewer disk accesses.

In Figure 16, the elapsed time for processing the queries using PRIX and TwigStackXB are plotted. PRIX used EPIndex for answering all the queries in Table 5. For queries Q_{22}, Q_{23}, Q_{24}, and Q_{28}, PRIX and TwigStackXB had comparable performance. As expected, TwigStackXB processed these queries efficiently, because the solutions for those queries were distributed 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 by performing subsequence matching beginning with the high selectivity element tags/values. The bottom-up transformation of the query twig caused higher selectivity nodes to appear in beginning of its LPS.

For queries Q_{25} and Q_{29} TwigStackXB was faster than PRIX. For Q_{25}, most solutions were clustered in small regions of the input stream allowing TwigStackXB to find all the matches faster as compared to PRIX.

For query Q_{29}, we performed the Bi-directional subsequence matching using the LTrie and RTrie with value "Jim Gray" as the pivot since the element `key` occurred in every document in the DBLP dataset. This increased the total IO as compared to processing Q_{28} which also had the same number of twig matches in the data.

For queries Q_{26} and Q_{27} PRIX was two to three times faster than TwigStackXB. The tag `phdthesis`

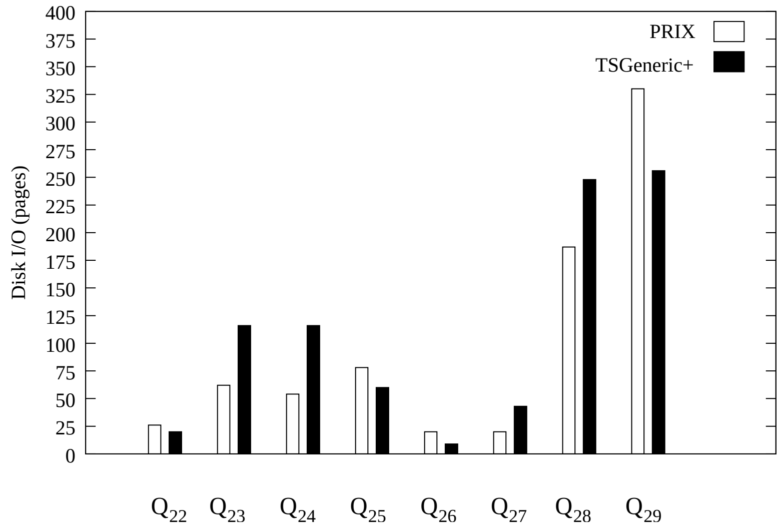


Figure 17: DBLP (PRIX vs TSGeneric$^+$)

Query	PRIX		TwigStackXB	
	Total time	Disk IO	Total time	Disk IO
Q_{26}	0.29 secs	20 pages	0.70 secs	71 pages
Q_{27}	0.30 secs	20 pages	1.03 secs	107 pages

Table 11: DBLP - PRIX vs TwigStackXB

was scattered in the sorted input stream. Also the tags `year`, `series` and `number` appeared frequently in other nearby documents that did not contain tag `phdthesis`. In order to eliminate such matches TwigStack frequently drilled down to lower regions of the XB-Trees. This increased the disk I/O. In PRIX, the cost of subsequence matching phase depends on the number of candidate ranges that are searched. Since PRIX used Bi-directional subsequence matching for processing these queries using tag `phdthesis` as the pivot, all matches were found by performing fewer disk I/O. Note that the tag `phdthesis` had high selectivity as compared to other nodes in the queries Q_{26} and Q_{27}.

PRIX vs TSGeneric$^+$ - The performance of PRIX and TSGeneric$^+$ for queries Q_{22} through Q_{29} are shown in Figure 17. The plot compares the disk I/O incurred by the PRIX and TSGeneric$^+$.

For queries Q_{22} and Q_{25}, both PRIX and TSGeneric$^+$ had comparable performance. For example, in Q_{22}, PRIX used the high selectivity node `‘‘Antonin Guttman’’` to search for matching subsequences by traversing few paths in the virtual trie. On the other hand, the high join selectivity edge `author=‘‘Antonin Guttman’’` proved useful for TSGeneric$^+$ to skip many elements in the input lists.

Next we shall analyze the performance of queries Q_{23} and Q_{24}. It is interesting to note that PRIX only incurred about half the I/O as compared to TSGeneric$^+$. The subsequence matching approach used by PRIX was far more effective than using the XR-trees indexes even though a high join selectivity edge (*i.e.*, `author=‘‘C. J. Date’’`) was present in both queries. It is also interesting to note the performance of queries Q_{26} and Q_{27}. PRIX processed the queries efficiently using Bi-directional subsequence matching with `phdthesis` as the pivot. TSGeneric$^+$ also processed the queries efficiently due to the presence of high join selectivity edges in these queries. Finally for queries Q_{28} and Q_{29}, we observed an increase in the disk I/O using PRIX for Q_{29} as compared to Q_{28} due to Bi-directional subsequence matching. Note that the performance trends for Q_{28} and Q_{29} were similar for TwigStackXB and TSGeneric$^+$.

Query	Bi-directional	Naive
Q_5	118 pages	98,645 pages
Q_{13}	130 pages	1,062 pages
Q_{19}	380 pages	444 pages
Q_{26}	19 pages	2,465 pages
Q_{27}	17 pages	2,540 pages
Q_{29}	325 pages	609,102 pages

Table 12: Improvements in Disk I/O during Subsequence Matching

Dataset	Query	CPU Time	% of Total Time
SWISSPROT	Q_7	0.27 secs	8.86%
TREEBANK	Q_{19}	0.003 secs	0.10%
DBLP	Q_{25}	0.04 secs	4.70%

Table 13: CPU Costs for the Refinement Phases

7.3 Evaluation of Bi-directional Subsequence Matching

To provide an insight to the effectiveness of Bi-directional Subsequence Matching, we compared Bi-directional subsequence matching and naive subsequence matching (from left to right) in terms of the disk I/O required to process queries Q_5, Q_{13}, Q_{19}, Q_{26}, Q_{27} and Q_{29} as shown in Tables 3, 4 and 5. The results are provided in Table 12. Note that the number of buffer pages was fixed at 2000.

Overall, it was observed that the I/O cost during subsequence matching improved drastically when the Bi-directional subsequence matching was used. This clearly shows that it is essential to start with a node (pivot) with high selectivity. For example, in Q_5, the tag `Org` had a very high frequency of 456,398 occurrences in the SWISSPORT dataset. Hence the number of paths explored in the virtual trie was very large. Similarly, the tag `key` in Q_{29} had a very high frequency of 328,858 occurrences in the DBLP dataset and showed similar trend. Overall we observed that Bi-directional subsequence matching improved the query processing performance of PRIX drastically.

7.4 CPU Costs For the Refinement Phases

We measured the CPU time for the queries listed in Tables 3, 4, and 5. For each dataset, we only show the query which had the maximum CPU cost in Table 13. The portion of the total time spent during the refinement phases is also shown in Table 13. We observed that the I/O cost dominated the total query processing time for the queries that we tested.

7.5 Summary of Performance Analysis

To summarize, we observed that PRIX yields good performance for processing queries with high selectivity. The query processing cost is dominated by the subsequence matching phase that is I/O bound. We observed that our proposed optimizations in Section 5 were effective in reducing the I/O cost during filtering. PRIX by virtue of the bottom-up tree transformation used high selectivity nodes to start searching the virtual trie for many queries. In some cases, PRIX used "*Bi-directional subsequence matching*" by choosing a pivot with high selectivity. As a result, few paths in the virtual trie were traversed thereby reducing the total processing time.

8 Related Work

Much research has been done on semistructured and XML databases in recent years. Query processing and optimization have received much attention in this context. Path join algorithms based on building *structural indexes* were proposed in this regard. The Lore system [10, 19] addressed several issues in query processing. DataGuide [10] provides concise and accurate summaries of semistructured databases. Hugh *et* *al.* [19] addressed different aspects of cost-based query optimizer for XML. The concept of representative objects was proposed by Nestorov *et al.* [22] for concise representation of the structure of semistructured, hierarchical data. Index Fabric [8] stores encodings of paths in a structure based on Patricia tries to support simple XML path queries in a single lookup. The T-index mechanism supports query processing for path expressions [20]. In addition, a family of approximate structural summaries called A(k)-indices was proposed by Kaushik *et al.* [17] for evaluating path expressions. The F&B index [16] was also proposed for evaluating branching path queries.

Several join algorithms have been developed based on numbering schemes to process path and twig queries [30, 18, 1, 5, 11, 7, 14]. Zhang *et al.* [30] addressed efficient processing of containment queries in relational database systems. The XISS system [18] decomposes a complex path expression into a collection of basic path expressions and processes them. The nodes in the XML document trees are numbered in extended preorder. Khalifa *et al.* [1] developed structural join algorithms *Tree-merge* and *Stack-tree* for matching parent-child and ancestor-descendant structural relationships. Bruno *et al.* [5] proposed *PathStack* and *TwigStack* algorithms and showed that they were I/O and CPU optimal for a large class of query twig patterns. In addition, a modification of B^+-trees called XB-Trees were developed to speed up query processing. In order to quickly find ancestors of a node in XML documents during structural joins, Jiang *et al.* [14] proposed the XR-Tree index. Subsequently, they developed the TSGeneric+ algorithm [15] that used indexes to speed up the twig pattern matching. Grust *et al.* [12] proposed *Staircase join* to speed up XPath processing using relational storage. Similar to PathStack, Staircase join exhibits optimal linear behavior during path processing. Recently, Zezula *et al.* [29] proposed the use of tree signatures for unordered XML pattern matching, and Wang *et al.* [26] proposed an indexing method called ViST that uses subsequence matching for processing twig queries by mapping trees to sequences.

Our system (PRIX) uses a novel method for XML twig pattern matching. PRIX transforms XML documents and twig queries into Prüfer sequences. PRIX's tree-to-sequence transformation requires space that is linear in the number tree nodes. By performing subsequence matching and a series of refinements phases, all occurrences of a twig pattern can be found without any false dismissals or false alarms. Our system allows holistic processing of twig queries without breaking them into root-to-leaf paths and processing them individually. PRIX inherently supports ordered twig pattern matching unlike most of the previous approaches.

9 Conclusions

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 our PRIX system.

References

[1] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing 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, February 2002.

[2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jrme Simon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, August 2002.

[3] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jrme Simon. XQuery 1.0: An XML Query Language W3C working draft 16. Technical Report WD-xquery-20020816, World Wide Web Consortium, August 2002.

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

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

[6] Baden Hughes Catherine Bow and Steven Bird. Towards a General Model of Interlinear Text. In *Proceedings of EMELD Workshop*, Lansing, MI, July 2003.

[7] 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, August 2002.

[8] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th VLDB Conference*, pages 341–350, Rome, Italy, September 2001.

[9] Paul 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.

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

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

[12] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th VLDB Conference*, pages 524–535, Berlin, Germany, September 2003.

[13] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21rd VLDB Conference*, pages 562–573, Zurich, Switzerland, September 1995.

[14] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, Bangalore, India, March 2003.

[15] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, pages 273–284, Berlin, Germany, September 2003.

[16] 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.

[17] 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, February 2002.

[18] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, Rome, Italy, September 2001.

[19] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of the 25th VLDB Conference*, pages 315–326, Edinburgh, Scotland, August 1999.

[20] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999.

[21] Karim Müller. Semi-Automatic Construction of a Question Treebank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, Lisbon, Portugal, 2004.

[22] S. Nestorov, J. Ullman 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., April 1997.

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

[24] Praveen Rao and Bongki Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, pages 288–299, Boston, MA, March 2004.

[25] UW XML Repository. Available from. http://www.cs.washington.edu/research/xmldatasets.

[26] Haixun Wang, Sanghyun Park, Wei Fan, and Philip 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.

[27] William D. Lewis. Personal communications. http://zimmer.csufresno.edu/ wlewis/.

[28] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke 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, August 2001.

[29] Pavel Zezula, Federica Mandreoli, and Riccardo Martoglia. Tree Signatures and Unordered XML Pattern Matching. In *Proceedings of the 32nd International Conference on Current Trends in Theory and Practice of Computer Science*, pages 122–139, Merin, Czech Republic, January 2004.

[30] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy 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.