On the Sequencing of Tree Structures for XML Indexing

Haixun Wang
IBM T. J. Watson Research Center
haixun@us.ibm.com

Xiaofeng Meng
Renmin University of China
xfmeng@ruc.edu.cn

Abstract

Sequence-based XML indexing aims at avoiding expensive join operations in query processing. It transforms structured XML data into sequences so that a structured query can be answered holistically through subsequence matching. In this paper, we address the problem of query equivalence with respect to this transformation, and we introduce a performance-oriented principle for sequencing tree structures. With query equivalence, XML queries can be performed through subsequence matching without join operations, post-processing, or other special handling for problems such as false alarms. We identify a class of sequencing methods for this purpose, and we present a novel subsequence matching algorithm that observe query equivalence. Still, query equivalence is just a prerequisite for sequence-based XML indexing. Our goal is to find the best sequencing strategy with regard to the time and space complexity in indexing and querying XML data. To this end, we introduce a performance-oriented principle to guide the sequencing of tree structures. For any given XML dataset, the principle finds an optimal sequencing strategy according to its schema and its data distribution. We present a novel method that realizes this principle. In our experiments, we show the advantages of sequence-based indexing over traditional XML indexing methods, and we compare several sequencing strategies and demonstrate the benefit of the performance-oriented sequencing principle.

1 Introduction

XML is the standard language for representing and exchanging semistructured data in many commercial and scientific applications and much research has been undertaken on providing flexible indexing and query mechanisms to extract data from XML documents [7, 14, 11, 6, 5, 9].

Figure 1 is a sample XML document which describes a project hierarchy in the form of a tree structure. Due to the semi-structured nature of XML, XML documents are often modeled by tree hierarchies. For the same reason, the design of XML query languages also focuses on the ability to express complex structured queries. Such queries can also be modeled, in much the same way, by tree patterns. Thus, query answering becomes a process of finding embedded sub tree structures among a set of data tree structures.

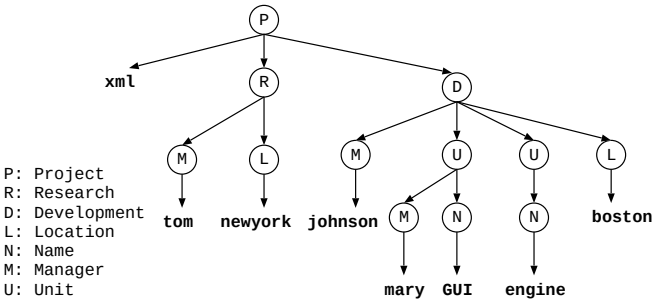


Figure 1: A Sample XML Document

The purpose of XML indexing is to provide efficient support for structured queries, which may contain values, wildcards ('*' and '//'), tree patterns, etc. However, in most indexing solutions [2, 14, 11, 7, 6, 9, 5], the tree pattern is not a first class citizen or, the most basic query unit. As a result, structured queries cannot be handled directly, and the most commonly supported query interface is instead:

$$\text{Simple Paths} \Rightarrow \mathcal{P}(\text{Node Ids})$$

That is, given a path, the index returns a set of nodes that represent such a path. Some index methods extend the above interface to support relative paths that *start with* a '*' or '//' at the cost of building a much larger index [6].

Tree patterns are not the most basic query unit because we cannot afford to maintain a separate index entry for each possible tree pattern, especially when they contain attribute values, wildcards '*' or '//'. Instead, we disassemble a tree pattern into a set of simple path queries. Then, we use join operations to merge their results to answer the original query. To avoid expensive join operations for queries that occur frequently, some index methods create special index entries for a limited set of path templates [6, 9].

Querying XML by Subsequence Matching

Sequence-based XML indexing [18] represents a major departure from previous XML indexing approaches. The new method supports a more general query interface:

$$\text{Tree Pattern} \Rightarrow \mathcal{P}(\text{Doc Ids})$$

That is, given a tree pattern, the index returns a set of XML documents/records that contain such a pattern. Instead of dis-

assembling a structured query into multiple sub queries, the tree structure is used as the basic query unit.

To do this, it transforms both XML data and queries into sequences and answers XML queries through subsequence matching [18, 13, 19]. The motivation is that, if we can represent an arbitrary tree structure by a sequence and demonstrate the equivalence between a structure match and a sequence match, then we can answer structured queries holistically, thus avoiding expensive join operations in query processing.

Challenges

The sequence-based approach opens up many new research issues. Previous works [18, 16] singled out ad hoc sequencing methods such as the depth-first traversal and the Prüfercodes. In this paper, we ask two questions:

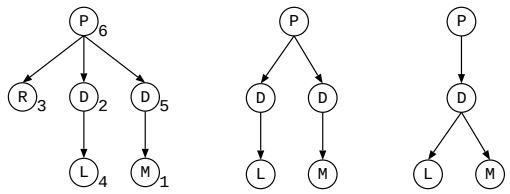
- What are the sequencing methods that preserve the query equivalence between structure and sequence match?
- Given a dataset, which sequencing method shall we use in order to maximize the performance of indexing and querying?

We elaborate the above challenges in three aspects: i) the representation of tree structures, ii) the query equivalence, and iii) the performance-oriented sequencing principle.

Tree Representation. A sequence-based approach starts with a valid sequential representation of tree structures. The ViST approach [18], for example, represents a tree node by a pair (X,Y), where X is the label of the node, and Y its path in the tree. Then, a tree structure can be represented by its depth-first traversal sequence. For instance, the tree structure in Figure 2(a) is encoded by the following sequence:

$$\langle(\mathtt{P},\epsilon),(\mathtt{R},\mathtt{P}),(\mathtt{D},\mathtt{P}),(\mathtt{L},\mathtt{PD}),(\mathtt{D},\mathtt{P}),(\mathtt{M},\mathtt{PD})\rangle \qquad (1)$$

The Prüfercode is a more succinct tree encoding method [15, 16]. Consider a tree of n nodes labeled arbitrarily from 0 to $n-1$. To encode it by a Prüfersequence, we repeatedly delete the leaf node that has the smallest label and append the label of its parent to the sequence. The Prüfersequence for the tree in Figure 2(a) is $\langle 5,6,2,6,6\rangle$.



(a) (b) (c)

Figure 2: Sample Tree Structures

Ad hoc sequencing methods such as depth-first and Prüferhave been used for XML indexing [18, 16]. However, we have not investigated the theoretical and practical implications of different tree representations.

Query Equivalence. One important implication of sequencing is the query equivalence. Assume we have transformed both XML data and queries to sequences by a particular method, and we answer XML queries by (non-contiguous) subsequence matching. Is a subsequence match always tantamount to a structure match?

Such equivalence is the foundation of sequence-based XML indexing. For certain sequencing methods, the equivalence seems to be obvious. For instance, Figure 2(b) is a sub structure in Figure 2(a), and the depth-first sequence of this embedded structure, $\langle(\mathtt{P},\epsilon),(\mathtt{D},\mathtt{P}),(\mathtt{L},\mathtt{PD}),(\mathtt{D},\mathtt{P}),(\mathtt{M},\mathtt{PD})\rangle$, is a non-contiguous subsequence of (1). This indicates that we can answer structured query through subsequence matching.

However, query equivalence between a structure match and a subsequence match has never been formally investigated. There are exceptions to this equivalence even for depth-first sequences. For instance, $\langle(\mathtt{P},\epsilon),(\mathtt{D},\mathtt{P}),(\mathtt{L},\mathtt{PD}),(\mathtt{M},\mathtt{PD})\rangle$ is the depth-first traversal sequence for Figure 2(c), and it is a non-contiguous subsequence of (1). But apparently, Figure 2(c) is not a sub structure of Figure 2(a). Thus, this match is a *false alarm*.

The false alarm problem described above is not unique to depth-first traversal. It is shared by almost all sequencing methods, including Prüfer. Current approaches get around this nonequivalence problem by resorting to join operations [18] or document by document post-processing [16], both of which are very time consuming.

Thus, the challenge here is to i) identify a class of sequencing methods that preserve query equivalence, and ii) devise an efficient algorithm that performs subsequence match under such equivalence.

Performance-Oriented Sequencing. The challenges aforementioned focus on equivalences: i) the equivalence between a tree structure and a sequence, and ii) the equivalence between a structure match and a subsequence match.

However, representation and query equivalence is just a prerequisite for sequence-based XML indexing. The ultimate challenge is to find the best sequencing method, that is, to support performance-oriented sequencing.

Sequencing has great implications to query performance. As we will demonstrate in Section 5, the performance of an index structure for a set of sequences is very specific to the distribution of the sequences, or more specifically, to the extent of sharing among the sequences. The extent of sharing is determined by the sequencing method as well as the data distribution of the original dataset of tree structures. Thus, instead of looking for a sequencing method that outperforms any other method for any dataset, we shall focus on the following challenge: given certain information (e.g. the schema and/or other data distribution statistics) of an XML dataset, how do we find a sequencing method that maximizes the performance of indexing and querying?

Problem Statement and Our Contributions

Let g be a sequencing method which maps a tree structure d to a sequence $g(d)$. For a dataset $\mathcal{D}$, let $g(\mathcal{D}) = \{g(d)|d \in \mathcal{D}\}$. The problem is the following:

First, find a class of sequencing methods $g_1, \cdots, g_n$ such that each g_i preserves the query equivalence between a structure match and a subsequence match.

Second, for a given dataset $\mathcal{D}$, find one sequencing method g from $g_1, \cdots, g_n$ so that the index structure built on $g(\mathcal{D})$ provides the best performance in terms of time and space complexity.

We solve the above problems based on a systematic study of sequence-based XML indexing. We introduce:

QUERY-EQUIVALENT SEQUENCING. We identify a class of sequencing methods that preserve query equivalence, so that structured queries can be answered through subsequence matching without join operations or document-by-document post-processing, and problems such as false alarms can be avoided.

PERFORMANCE-ORIENTED SEQUENCING. Our principle of sequencing is to maximize the performance of XML indexing and querying. We show how this principle can be realized by taking into consideration the distribution of the XML data during the process of sequencing.

ALGORITHMS FOR SEQUENCING, INDEXING AND QUERYING. We propose algorithms that transform XML data to valid sequences for high-performance indexing and querying, and we show how the index structure overcomes the false alarm problem.

Paper Organization

Next section studies tree sequencing methods. In Section 3, we discuss the issues in performing XML queries through subsequence matching. In Section 4, we present a query algorithm which overcomes the nonequivalent problem. In Section 5, we introduce the performance-oriented principle of sequencing. The experimental results are reported in Section 6 and we conclude our work in Section 7.

2 Data Representation

Most tree sequencing methods can be regarded as having two parts: i) the encoding of the tree nodes, and ii) the order of the encoded nodes in the sequence [15]. Together, they must convey enough information so that we can reconstruct the tree structure from the sequence. In this section, we study different ways to represent or encode a tree structure by a sequence.

2.1 Notation

We designate each element and attribute name in an XML document by a *designator*. For instance, in Figure 1, `P, R, D, L,` $\cdots$ are the designators.

For attribute values, we have two options. One is from ViST [18], which represent each value by a single designator derived by a hash function. For instance, assuming

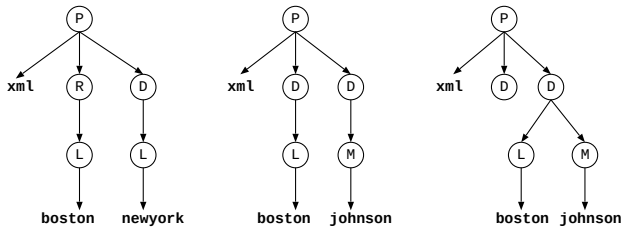
$$v_0 = h('xml'), v_1 = h('boston'), v_2 = h('newyork'), \cdots$$

we use v_1 to designate 'boston', v_2 'newyork', etc. The second option is to represent a value by a text sequence, for instance, 'boston' by b,o,s,t,o,n, which is similar to Index Fabric [6].

Thus, we can represent a simple path such as `/Project/Research[Location=boston]` by:

$$\langle \mathtt{P}, \mathtt{R}, \mathtt{L}, \mathtt{v}_1 \rangle \qquad \text{or} \qquad \langle \mathtt{P}, \mathtt{R}, \mathtt{L}, \mathtt{b}, \mathtt{o}, \mathtt{s}, \mathtt{t}, \mathtt{o}, \mathtt{n} \rangle$$

The first representation treats each value as an atomic item and the second representation will allow subsequence matching inside the attribute values. For presentation simplicity, we use the first option, but the concepts and algorithms described in this paper can be applied to the second option with easy adaptation.



(a) (b) (c)

Figure 3: Tree Structure and Set Representation

2.2 Sequencing

As we will show in Section 5, in order to achieve the best performance, sequencing shall reflect the data distribution of the XML dataset. In other words, the order of the nodes in the sequence cannot be solely bound by the tree structure it represents. Our approach is to include as much structural information as possible in node encoding, then we can relieve the burden on sequencing and achieve flexibility.

Node Encoding. We encode each node n in the tree by the path leading from the root node to n. For instance, in Figure 3(a), the nodes on the middle branch are encoded by `P`, `PR`, `PRL`, and $\mathtt{PRLv}_1$ respectively. We also use $\subset$ to denote the prefix relationship among the paths. This representation is similar to that of ViST [18], but instead of using a pair, we use paths only.

Arbitrary Sequencing. The path-encoded nodes have already included much structural information, to the extent that the tree structure in Figure 3(a) can be derived directly from the encoded nodes:

$$\{\mathtt{P}, \mathtt{Pv}_0, \mathtt{PR}, \mathtt{PD}, \mathtt{PRL}, \mathtt{PDL}, \mathtt{PRLv}_1, \mathtt{PDLv}_2\}.$$

Here, the order of the encoded nodes is irrelevant. In other words, an arbitrary sequence of the nodes can represent the tree structure. We have achieved the maximum flexibility.

However, when there are *identical sibling nodes* under a parent node, set representation is no longer valid. For instance, in the tree structures of Figure 3(b) and 3(c), under node `P`, there are two identical sibling nodes `D`, and the two structures have the same multi-set representation:

$$\{\mathtt{P}, \mathtt{Pv}_0, \mathtt{PD}, \mathtt{PD}, \mathtt{PDL}, \mathtt{PDM}, \mathtt{PDLv}_1, \mathtt{PDMv}_3\}$$

This means, with the presence of identical sibling nodes, path-based node encoding alone is insufficient. We must resort to sequencing to supplement the missing information.

Depth-first Sequencing. The order of the nodes in a sequence can supply the structural information missing in the set representation.

For instance, one possibility is to represent the tree structures of Figure 3(b) and 3(c) by depth-first traversal sequences:

Fig. 3(b): $\langle \mathtt{P}, \mathtt{Pv}_0, \mathtt{PD}, \mathtt{PDL}, \mathtt{PDLv}_1, \mathtt{PD}, \mathtt{PDM}, \mathtt{PDMv}_2 \rangle$
Fig. 3(c): $\langle \mathtt{P}, \mathtt{Pv}_0, \mathtt{PD}, \mathtt{PD}, \mathtt{PDL}, \mathtt{PDLv}_1, \mathtt{PDM}, \mathtt{PDMv}_2 \rangle$

Table 1: Depth-first traversal sequences

It is easy to see that the depth-first traversal sequences represent unique tree structures. Depth-first sequencing is certainly not the only choice, many ad hoc sequencing methods including the Prüfercodes work as well.

Nevertheless, we do not want to confine ourself to a particular way of ordering the nodes. Because, besides encoding the structure, sequencing shall play a more important role: it shall order nodes based on the distribution of the XML data, so that we can achieve the best index and query performance (Section 5).

2.3 Constraint Sequences

Given a tree structure, ad hoc sequencing methods such as depth-first traversal and Prüfercodes lead to unique orderings of the tree nodes. However, unique ordering is far more restrictive than necessary, and it is in conflict with our intention to support performance-oriented sequencing, for we are interested in finding a many-to-one relationship between sequences and tree structures.

The reason we cannot use arbitrary sequencing to represent the tree structures in Figure 3(b) and Figure 3(c) is because of the two identical sibling nodes encoded as PD. They cause ambiguity when we try to determine the ancestor-descendant relationships among the sequenced nodes.

If we can introduce a constraint that eliminates such ambiguity, we will be able to map any sequence of path-encoded nodes into a unique tree structure. Within the constraint, we can still have the freedom to order the nodes arbitrarily. Thus, a tree structure will have multiple sequential representations. Such a constraint actually defines a sequencing method.

Let $T = \langle p_1, \cdots, p_n \rangle$ be a sequence of path-encoded nodes. We define constraint as follows.

Definition 1. Constraint
A constraint is a boolean function $f(\cdot, \cdot)$ that satisfies the following condition: $\forall p_j \in T$ and $\forall t \subset p_j$, there exists one and only one $p_i \in T$ such that $f(p_i, p_j) =$ true and $p_i = t$.

Intuitively, $f(p_i, p_j)$ embodies the ancestor-descendant relationship between p_i and p_j, that is, $f(p_i, p_j)$ evaluates to true if p_i is an ancestor of p_j. The definition of constraint ensures that, $\forall p_j \in T$, i) each and every of p_j's ancestor exists; and ii) none of its ancestors can have the same path encoding (no ambiguity).

For tree structures such as the one in Figure 3(a) that contain no identical sibling nodes we can define the following constraint:

$$f_1(p_i, p_j) \equiv p_i \subset p_j \tag{2}$$

Function f_1 is a constraint because each p_i is unique (the tree structure does not have identical sibling nodes). Since the definition of f_1 does not rely on the relative positions of p_i and p_j in the sequence, it does not place any constraint on the ordering, which means nodes can form arbitrary sequences.

When identical sibling nodes are present, we need a constraint that can eliminate ambiguity. The *forward prefix* defined below is one way to eliminate ambiguity. Intuitively, among the ambiguous ancestors of x, we choose the one that appears earlier than x in the sequence, and if none or more than one such node exists, we choose the one that is closest to x.

Definition 2. Forward Prefix
Let $\langle p_1, \cdots, p_n \rangle$ be a sequence of path-encoded nodes. We say $p_k \subset p_i$ is a forward prefix of p_i if $\forall p_j = p_k$, $i < k < j$ and $\nexists p_j = p_k$, $k < j < i$.

As an example of forward index, in sequence $\langle \mathtt{P}, \underline{\mathtt{PD}}, \mathtt{PDL}, \mathtt{PDLv}_1, \underline{\mathtt{PD}}, \mathtt{PDM}, \mathtt{PDMv}_3 \rangle$, the second PD is a forward prefix of $\mathtt{PDMv}_3$ while the first PD is not.

Thus, the new constraint can be defined as:

$$f_2(p_i, p_j) \equiv p_i \text{ is a forward prefix of} p_j \tag{3}$$

It is clear that i) f_2 introduces extra constraints to f_1, and ii) f_2 relies on the order of the path-encoded nodes to determine the ancestor-descendant relationship among them.

$\langle \mathtt{P}, \mathtt{Pv}_0, \mathtt{PD}, \mathtt{PD}, \mathtt{PDL}, \mathtt{PDLv}_1, \mathtt{PDM}, \mathtt{PBMv}_3 \rangle$
$\langle \mathtt{P}, \mathtt{PD}, \mathtt{Pv}_0, \mathtt{PD}, \mathtt{PDM}, \mathtt{PBMv}_3, \mathtt{PDL}, \mathtt{PDLv}_1 \rangle$
$\langle \mathtt{P}, \mathtt{PD}, \mathtt{PDL}, \mathtt{Pv}_0, \mathtt{PDLv}_1, \mathtt{PDM}, \mathtt{PBMv}_3, \mathtt{PD} \rangle$
$\langle \mathtt{P}, \mathtt{PD}, \mathtt{PDM}, \mathtt{PBMv}_3, \mathtt{Pv}_0, \mathtt{PDL}, \mathtt{PDLv}_1, \mathtt{PD} \rangle$
$\langle \mathtt{P}, \mathtt{PD}, \mathtt{PDM}, \mathtt{PBMv}_3, \mathtt{PDL}, \mathtt{Pv}_0, \mathtt{PDLv}_1, \mathtt{PD} \rangle$
... ...

Table 2: Constraint sequences of Figure 3(c)

Despite the constraint, given a tree structure, we can still represent it by more than one sequence, and we can reconstruct the tree structure from any of them. The tree structure in Figure 3(c) and its sequence representations in Table 2 are such an example. We call such sequences *constraint sequences*.

Theorem 1. *A constraint sequence maps to a unique tree structure.*
Proof. (Sketch) In a constraint sequence, x is an ancestor to y only if $f(x, y)$ is true. Since the constraint guarantees that for any y only one of the identical sibling nodes can be y's ancestor, the position of any node in the tree is uniquely defined. □

2.4 Constraint Sequencing

How do we sequence a tree structure into multiple sequences that satisfy a constraint f? What are the implications of choosing different constraints?

Constraint sequencing is controlled by i) a constraint f, and ii) a user strategy g. That is, the generated sequences must

satisfy f, but within the constraint, we can use a user-provided strategy g to order the nodes. For constraint f_1, sequencing is totally controlled by user strategy g. For constraint f_2, let us consider the following procedure.

First, we select the root node. Then, we repeatedly invoke user strategy g to select a node whose parent node has already been selected. However, if a selected node x has identical sibling nodes, we must not select any of its identical siblings until all the descendents of x have been selected.

The above simple procedure actually enforces a stronger constraint than f_2, because in the generated sequences, an ancestor always appear earlier than its descendents. In Section 5, following the performance-oriented sequencing principle, our strategy g selects nodes based on their occurrence probabilities. Since a parent node always has a larger occurrence probability than its child nodes, we will always have x's ancestors appearing before x. Thus, the above simple procedure will suffice.

As we know, f_2 places an constraint on the order of the nodes in the sequence to handle identical sibling nodes. But certainly f_2 is not the only possible constraint for this situation. When deciding which node among a set of identical sibling nodes should be x's ancestor, the forward prefix rule favors the one that appears earlier than x and closer to x. We can of course using other criteria.

Our focus, however, is whether the constraint leaves enough freedom for applying a user strategy g. Compared with ad hoc sequencing methods (e.g., depth-first traversal and Prüfercodes), constraint sequencing has the potential to offer this flexibility which will make a big difference to the performance of XML indexing and querying.

3 Query Equivalence

In this section, we define query equivalence based on constraint sequences. In Section 4, we introduce a subsequence matching algorithm that ensure query equivalence between structure match and subsequence match.

3.1 Query by Sequence Matching

Before the 'best' user strategy is introduced in Section 5, let us use depth-first traversal as our strategy. This is only for presentation purpose, for the issues discussed in this section apply to any constraint and any user strategy.

We can represent a tree structure by constraint sequences. For example, the tree structure in Figure 1 can be represented by the following sequence that satisfies constraint f_2.

$$\langle \underline{\mathtt{P}}, \mathtt{Pv}_1, \underline{\mathtt{PR}}, \mathtt{PRM}, \mathtt{PRMv}_2, \underline{\mathtt{PRL}, \mathtt{PRLv}_3, \mathtt{PD}}, \mathtt{PDM}, \mathtt{PDMv}_4, \mathtt{PDU}, \mathtt{PDUM}, \mathtt{PDUMv}_5, \mathtt{PDUN}, \mathtt{PDUNv}_6, \mathtt{PDU}, \mathtt{PDUNv}_7, \underline{\mathtt{PDL}, \mathtt{PDLv}_8} \rangle \quad (4)$$

In the same spirit, we can represent XML structural queries by constraint sequences. For instance, the XPath query

/Project[Research[Loc=newyork]]/Develop[Loc=boston]

can be represented by constraint sequence:

$$\langle \mathtt{P}, \mathtt{PR}, \mathtt{PRL}, \mathtt{PRLv}_3, \mathtt{PD}, \mathtt{PDL}, \mathtt{PDLv}_8 \rangle$$

It is easy to see that the above query sequence is a non-continuous subsequence (the underlined part) of the document sequence in Eq (4). Moreover, queries with wildcard ('*' or '//') can also be converted to sequences. For instance, we can represent XPath query /Project/*[Loc $= \mathtt{v}_8$] by constraint sequence $\langle \mathtt{P}, \mathtt{P*}, \mathtt{P*L}, \mathtt{P*Lv}_8 \rangle$, which is also a non-contiguous subsequence in (4) once '*' is instantialized to symbol D.

However, due to the existence of identical sibling nodes, XML structure match is not equivalent to the naïve subsequence match described above.

3.2 False alarms and false dismissals

Identical sibling nodes under a parent node causes problem not only to data representation but also to queries.

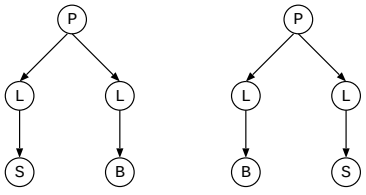
$\mathcal{D}$: $\langle \underline{\mathtt{P}, \mathtt{PL}, \mathtt{PLS}}, \mathtt{PL}, \underline{\mathtt{PLB}} \rangle$ $\mathcal{Q}$: $\langle \underline{\mathtt{P}, \mathtt{PL}, \mathtt{PLS}, \mathtt{PLB}} \rangle$

(a) (b)

Figure 4: False Alarm

Figure 4(a) and 4(b) are apparently different tree structures, however, there is a non-contiguous subsequence match between their constraint sequence representations, that is, $Q \subseteq \mathcal{D}$. That is, naïve subsequence matching triggers false alarms in answering structural queries. This problem is not unique to constraint sequencing. However, previous approaches [18, 16] handle this problem through expensive join operations or document-by-document post-processing.

The second problem, false dismissal, is due to tree isomorphism. In Figure 5, we show the same XML structures in two different forms.



(a) (b)

Figure 5: False Dismissal

However, the two forms can have different sequence representations. For instance, their constraint sequences (based on f_2 and the depth-first traversal strategy) can be the following:

$$\langle \mathtt{P}, \mathtt{PL}, \mathtt{PLS}, \mathtt{PL}, \mathtt{PLB} \rangle \text{ and } \langle \mathtt{P}, \mathtt{PL}, \mathtt{PLB}, \mathtt{PL}, \mathtt{PLS} \rangle$$

Thus, if the data sequence is in one form and the query sequence is in the other, we will have the false dismissal problem.

3.3 Constraint Match

The false dismissal problem is easy to avoid. Given a query structure, we regard each of its isomorphism structures as a different query, and union the results of these queries. The false alarm problem is more difficult, and previous methods [18, 16] cannot handle it without using expensive join operations or post-processing. In contrast, constraint sequences can handle this problem directly.

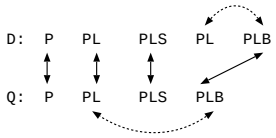


Figure 6: Sequence Match

Let us consider the example in Figure 4. We represent the match between $\mathcal{D}$ and $\mathcal{Q}$ by solid arrow lines in Figure 6.

Let function $m(\cdot)$ maps an element in $\mathcal{Q}$ to its matched element in $\mathcal{D}$. We define the concept of *constraint match* as follows:

Definition 3. Constraint Match
Given a match $m(\cdot)$ between sequences $\mathcal{Q}$ and $\mathcal{D}$, which are based on constraint f, it is a constraint match if the following criteria are satisfied:

1. $m(a) = b \Rightarrow a = b$
2. $f(a, b) \Leftrightarrow f(m(a), m(b))$

It is easy to see that naïve subsequence match only guarantees the 1st criterion, that is, $m(a) = b \Rightarrow a = b$, and leaves the 2nd criterion unchecked. For instance, although there is a sequence match in Figure 6, the 2nd criterion is violated there: as indicated by the arrows with dotted lines, element PL is an ancestor of element PLB in $\mathcal{Q}$, however, $m(\texttt{PL})$ is not an ancestor of $m(\texttt{PLB})$ in $\mathcal{D}$.

Note that if no identical sibling nodes exist in the documents, then the 2nd condition is implied by the 1st. This is so because, without identical sibling nodes, the relative positions of two nodes are uniquely defined by their paths.

Theorem 2. *Constraint match preserves query equivalence.*

Proof. By Theorem 1 and Definition 3. □

However, Theorem 2 does not apply to isomorphic trees. This is not critical because false dismissals caused by the isomorphic tree problem can be handled by simply asking multiple queries and combining their results — there is no need to use expensive join operations or document-by-document post-processing.

4 Algorithm

In this section, we present an algorithm for indexing and querying XML data. The query algorithm performs constraint subsequence matching, which preserves query equivalence between structure match and subsequence match.

4.1 Index Construction

The algorithm for Index construction takes the following three steps.

SEQUENCE INSERTION. We represent each XML document by a constraint sequence, and insert the sequence into a trie-like tree structure [12, 17]. For instance, suppose we have a document with the following constraint sequence representation:

$$\langle \mathtt{p}_1, \mathtt{p}_{10}, \mathtt{p}_2, \mathtt{p}_7, \mathtt{p}_9, \mathtt{p}_8 \rangle$$

Figure 7 shows the tree structure corresponding to the insertion of the above sequence. Supposing the insertion ends up at node x, we append the id of this document into the (document) *id list* of x. If we are indexing static data, instead of inserting sequences one by one, we can 'bulk load' the index by sorting the sequences first to improve performance.

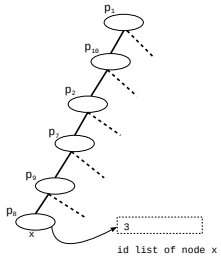


Figure 7: Insertion of a single sequence

TREE LABELING. In the second step, we label the nodes of the tree. Each node n is labeled by a pair of integers $(n^{\vdash}, n^{\dashv})$, where $n^{\vdash}$ is n's serial number (derived from a depth-first traversal of the index tree, which assigns 0 to the root node), and $n^{\dashv}$ is the largest serial number of n's descendents. This labeling scheme is used in many XML indexing algorithms [11, 3, 4, 8]. Given the labels of two nodes x and y, we know x is y's descendent if $x^{\vdash} \in (y^{\vdash}, y^{\dashv}]$. In contrast to previous algorithms using the same numbering scheme, we apply the labeling scheme on a trie-like structure built upon the constraint sequences derived from the XML documents, instead of on the original XML document trees.

PATH LINKING. In the third step, we create horizontal path links for each unique path that appears in the sequences. Each path link consists of labels of tree nodes represented by the same path-encoding. Figure 8 shows the index structure, which has the headers of the linked list on the left hand side. If we do not consider the identical sibling issue (which is discussed in detail in the next subsection), we shall find that the labels of the nodes in a single link are in ascending order of their

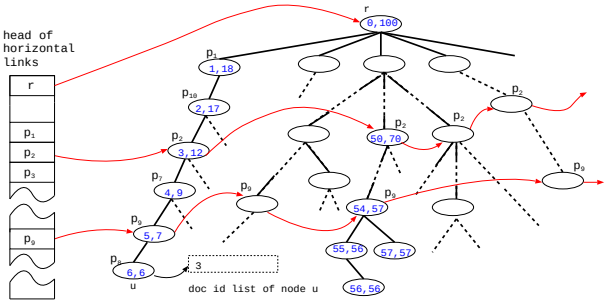


Figure 8: The index structure

serial number $n^{\vdash}$. The linked lists in Figure 8 are just for presentation purpose; they can easily be implemented by a more efficient structure that supports binary search.

4.2 Query XML by Subsequence Matching

First, we demonstrate naïve subsequence matching using the index structure. On top of that, we describe constraint subsequence matching, which preserves query equivalence between structure match and subsequence match.

Naïve Subsequence Matching. We perform naïve subsequence matching using the index shown in Figure 8. Suppose we have the following query:

$$\langle p_0, p_2, p_9, p_8 \rangle$$

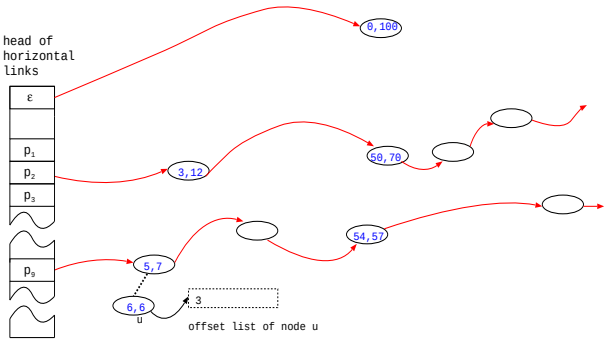


Figure 9: Index Structure

It turns out that in order to answer such a query, we can safely ignore the tree structure and focus on the path links (Figure 9) only. The query proceeds as follows. We start with the first element, p_0, in the query sequence. Through the path links, we find that it corresponds to one label, which represents a range of $[0, 100]$. Then, we check the second element, p_2. It corresponds to a list of labels, but we are only interested in those within the range of $[0, 100]$, since only those nodes are descendents of p_0. So we perform an efficient binary search $[0, 100]$ on the link for p_2, thanks to the fact that the labels there are in ascending order. We repeat the above process until the end of the query sequence. Suppose one of the nodes we finally reach is x. Then the id of the documents that satisfy the query are in the id lists of x's descendents. This naïve subsequence matching process is similar to that used in ViST [18].

Constraint Subsequence Matching. As we have mentioned, XML query is more than naïve subsequence matching: there is the false alarm problem caused by identical sibling nodes.

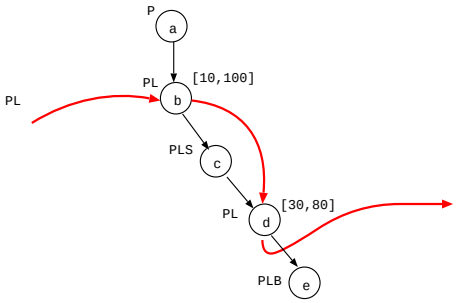


Figure 10: Identical Sibling Nodes

Input: a query $\mathcal{Q}$
Output: docs $\in \mathcal{D}$ that contain query structure $\mathcal{Q}$

Let $\mathcal{Q} = \langle p_1, \cdots, p_i, \cdots \rangle$;
Let r = root node of the index tree;
$search(r, 0, \{\})$;

Function $search(v, i, ins)$
if $i < |\mathcal{Q}|$ **then**
  $i \leftarrow i + 1$;
  $I \leftarrow$ horizontal link of p_i ;
  /*
  Perform binary search in I to find nodes $\in [v_s, v_m]$
  */
  for *each node $r \in I$ whose ID $\in [v_s, v_m]$* **do**
    if *$\nexists\, x \in ins$ such that x sibling-covers r* **then**
      if *r embeds identical siblings* **then**
        $ins \leftarrow ins \cup \{r\}$
      end
      $search(r, i, isn)$;
    end
  end
else
  output $L[v_s ... v_m]$, document id lists of node v and all nodes under v;
end

Algorithm 1: Subsequence Matching

Here, we show how constraint subsequence matching works. When there are no identical sibling nodes, the labels in the path links are in strict ascending order. This is no longer true when identical sibling nodes are involved. In Figure 10, we insert the data sequence $\mathcal{D}$ with two PL elements into an index tree. In the tree, the first PL is an S-ancestor of the second, which means the range of the first PL covers that of the second. However, since they are both PL, they reside in the path link of PL, which is also shown in Figure 10.

Assume we have matched the first 3 elements in query sequence Q to nodes b, d and e in Figure 10, and we have reached

node e that matches PLB. Does the 2nd criterion in Definition 3 hold for this match?

The answer is it does not. In query sequence Q, PL is an ancestor of PLB, that is, $f(\texttt{PL}, \texttt{PLB})$=true, where f is the constraint. However, in Figure 10, node b cannot be an ancestor of node e because of the existence of node d, that is, $f(b, e)$=false. In other words, assuming node b and d has range [10,100] and [30,80] respectively, the descendants of b cannot have labels inside [30,80]. Thus, we cannot match PLB to node e.

More formally, let $Q = \langle \cdots p_x \cdots p_y \cdots \rangle$ be a query sequence, where p_x is a forward prefix of p_y, and let the following be the path link for p_x.

$$\cdots, [u^{\vdash}, u^{\dashv}], \underbrace{[v_1^{\vdash}, v_1^{\dashv}], \cdots, [v_k^{\vdash}, v_k^{\dashv}]}_{identical\ sibling\ nodes}, [w^{\vdash}, w^{\dashv}], \cdots \quad (5)$$

In the path link, $[v_1^{\vdash}, v_1^{\dashv}], \cdots, [v_k^{\vdash}, v_k^{\dashv}]$ represents a series of identical sibling nodes such that $[v_1^{\vdash}, v_1^{\dashv}] \supseteq [v_j^{\vdash}, v_j^{\dashv}]$, $1 \leq j \leq k$. We define the concept of *sibling-cover* as follows.

Definition 4. Sibling-cover
Assume p_x matches node v_i and p_y matches node y. Node y is sibling-covered by v_i if $[y^{\vdash}, y^{\dashv}] \subseteq [v_{i+1}^{\vdash}, v_{i+1}^{\dashv}]$.

During the matching process, it is easy to check if the current node is sibling-covered by a previously matched node. All we need to do is storing each previously matched node who has identical siblings. When we try to match a new node, we check previously matched nodes that are its potential ancestors, and make sure the range of the new node is not inside any of the sibling range of the potential ancestor. Theorem 3 shows the correctness of this process.

Theorem 3. *Let query $\langle q_1, \cdots, q_k \rangle$ match $\langle v_1, \cdots, v_k \rangle$ in the index tree. The match is valid if no v_j, $j = 1, \cdots, k$, is sibling-covered.*

Proof. We only need to prove the 2nd criterion of Definition 3 is satisfied, i.e., $f(q_i, q_j) \Leftrightarrow f(v_i, v_j)$. Assume v_i is not an ancestor to v_j, then v_i is not v_j's forward prefix. Let v be v_j's forward prefix. Then v_i and v must be in the same path link. Since v is v_j's closest prefix node, we have $[v_i^{\vdash}, v_i^{\dashv}] \supseteq [v^{\vdash}, v^{\dashv}] \supseteq [v_j^{\vdash}, v_j^{\dashv}]$, which means v_j is sibling-covered by v_i. □

Algorithm 1 outlines the procedure that performs XML query through constraint subsequence matching, and it takes care of the problem of identical sibling nodes.

5 Performance-Oriented Sequencing

We have transformed XML data into constraint sequences and answered XML queries by a special subsequence matching algorithm that preserves query equivalence.

In addition to preserving query equivalence, constraint sequences allow a user defined strategy g in sequencing. In this section, we study the impact of different strategies on the performance of indexing and querying XML, and we show how to choose a strategy to maximize the performance.

5.1 Impacts of Sequencing

Many sequencing strategies are available, for instance, depth-first traversal, breadth-first traversal, or even Prüfersequencing. Then, what is the principle in choosing from these strategies? We argue that the principle should be based on whether a particular sequencing method can maximize the performance of XML indexing and querying.

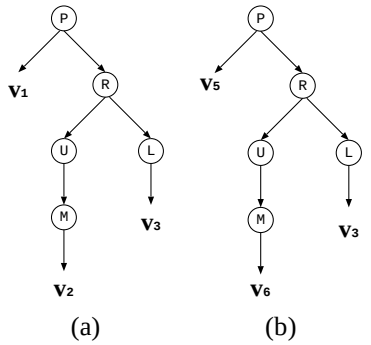


Figure 11: Sample XML Documents

To understand the principle, we must first understand the impacts of different sequencing strategies on the performance.

Impact 1. Figure 11 shows two sample XML documents that conform to a same imaginary DTD schema. We show the depth-first, breadth-first, and constraint sequences (based on constraint f_2 in Eq 3 and an unknown user strategy g_{best}) of the two documents in Table 3.

DF:	(a)	$\langle \underline{\texttt{P}}, \texttt{Pv}_1, \texttt{PR}, \texttt{PRU}, \texttt{PRUM}, \texttt{PRUMv}_2, \texttt{PRL}, \texttt{PRLv}_3 \rangle$
	(b)	$\langle \underline{\texttt{P}}, \texttt{Pv}_5, \texttt{PR}, \texttt{PRU}, \texttt{PRUM}, \texttt{PRUMv}_6, \texttt{PRL}, \texttt{PRLv}_3 \rangle$
BF:	(a)	$\langle \underline{\texttt{P}}, \texttt{Pv}_1, \texttt{PR}, \texttt{PRU}, \texttt{PRL}, \texttt{PRUM}, \texttt{PRUMv}_2, \texttt{PRLv}_3 \rangle$
	(b)	$\langle \underline{\texttt{P}}, \texttt{Pv}_5, \texttt{PR}, \texttt{PRU}, \texttt{PRL}, \texttt{PRUM}, \texttt{PRUMv}_6, \texttt{PRLv}_3 \rangle$
CS:	(a)	$\langle \underline{\texttt{P}, \texttt{PR}, \texttt{PRU}, \texttt{PRL}, \texttt{PRUM}, \texttt{PRLv}_3}, \texttt{Pv}_1, \texttt{PRUMv}_2 \rangle$
	(b)	$\langle \underline{\texttt{P}, \texttt{PR}, \texttt{PRU}, \texttt{PRL}, \texttt{PRUM}, \texttt{PRLv}_3}, \texttt{Pv}_5, \texttt{PRUNv}_6 \rangle$

Table 3: Different Sequencing Methods

The value node on the left-most branch (v_1 and v_5) corresponds to the 2nd element in the depth-first and the breadth-first sequences. Thus, the 2nd element in these sequences will have many different values. Note this variety is not due to our path-based node encoding. As a document can take any possible value at this position, any node encoding method will encounter this situation.

The problem comes when these sequences are inserted into the index tree. The variety of the 2nd element in the sequences prevents path sharing at the 2nd level of the tree. This leads to a very bushy index. In the extreme case where there is no sharing at all among the inserted sequences, the number of the nodes in the index tree equals to the number of nodes in the XML documents. The index becomes useless since querying on the index is tantamount to scanning the documents one by one.

The flexibility of constraint sequences enables us to avoid this problem. In Table 3, the two particular constraint sequences share a much longer prefix, and the index built on such sequences will have a much smaller size.

Impact 2. Let $\langle p_1, p_2, p_3, p_4 \rangle$ be a query sequence, where p_1, p_2, p_3 are the most common paths, and p_4 is among the least common paths. For instance, p_1, p_2, p_3 are encodings for such XML elements as `Project`, `Unit` and `Manager`, and p_4 stands for a value node, say 'Johnson'.

Based on Algorithm 1, the querying process starts with p_1, then p_2 and p_3. Since they are so common, when we reach p_3, we have already traversed a large amount of the index tree. However, few of them will finally lead to p_4, which is among the least common paths.

It is clear that p_4 has high selectivity, and if it appears frequently in queries, then we should make elements such as p_4 appear earlier in the sequences so that the search space can be reduced.

The principle of sequencing Sequencing must take two things into consideration. First, we want to build a *compact* index. In other words, we need a sequencing method that maximizes sharing. Second, we want to build a *tunable* index, so that we can make certain frequently queried and highly selective elements appear earlier in the sequences. In the rest of this section, we show how constraint sequences realize this principle.

5.2 Node Occurrence Probability

Constraint sequences provide the flexibility that can be leveraged to secure a sequencing strategy that observes the aforementioned principle.

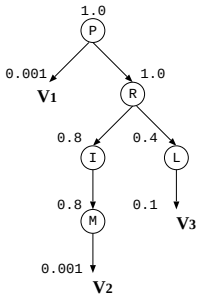


Figure 12: Existence probabilities given the parent

Let $p(C|P)$ be the probability that C exists given that node P exists, where C can be either a node or a value. Figure 12 shows such probabilities in an XML document tree. Next, we show how these probabilities are derived.

If we assume all XML documents have P as their root node, we have $p(P|\epsilon) = 1$. If node P, as prescribed by the schema, always has a child node R, then $p(R|P) = 1$. Otherwise, the value should reflect the true probability of node P having R as a child. We can either derive or estimate $p(R|P)$ from the semantics in the schema, or approximate it by data sampling.

If C is a value, for example $C = v_1$, then probability $p(C = v_1|P)$ is a combination of two factors: i) the probability that the value node exists under P, and ii) the probability that the value is v_1. The 1st factor can be derived in the same way as for the non-value nodes; for the 2nd factor, we consider the range and the distribution of the values. For instance, if v_1 is one of the states in US, whose occurrences in the XML dataset follow uniform distribution, then the 2nd factor is 1/55. If, however, v_1 is a person's name, and we used hash function with a range of 1000 to handle such values, then the 2nd factor is 1/1000.

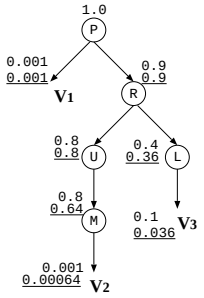


Figure 13: Existence probabilities given the root

Next, we derive $p(C|root)$ for each C. The computation is straightforward. For instance, based on the tree structure in Figure 13, we can derive $p(L|root)$ by

$$p(L|root) = p(L|R) \times p(R|root) = 0.4 \times 0.9 = 0.36.$$

Figure 13 shows the probability $p(C|root)$ for each C.

The probabilities of the nodes and values are associated with a DTD schema, not with any particular XML document, although we may need to sample the documents in order to find out some of the probabilities. Given a set of documents based on a schema, we sequence each of them in such a way that the common prefixes of the outcome sequences are as long as possible. With the probabilities, this is easy to realize. From an XML document, our strategy g_{best} always select nodes whose counterparts in the schema tree have higher probabilities, so that these nodes appear earlier in the sequences.

For instance, based on f_2 and g_{best}, the XML document of Figure 13 is sequentialized into the following:

$$\langle \mathtt{P}, \mathtt{PR}, \mathtt{PRU}, \mathtt{PBUM}, \mathtt{PRL}, \mathtt{PRLv}_3, \mathtt{Pv}_1, \mathtt{PRUMv}_2 \rangle.$$

As another example, the probability-based sequences for the two tree structures in Figure 11 share a common prefix of length 6 (out of total length 8), while depth-first, breadth-first sequences share a common prefix of length 1.

It is not difficult to prove that g_{best} maximizes sequence sharing and leads to the most compact index. However, two issues must be taken into considerations.

First, when identical siblings are present, the constraint has priority over user strategy g_{best}, for we must ensure that we can reconstruct the tree structures from the sequences. Algorithm 2 shows the procedure of generating constraint sequences with respect to constraint f_2 and user strategy g_{best}. The algorithm generates sequences where for any node x, its ancestor nodes always precedes x. This is not enforced by constraint f_2 but by strategy g_{best}. Because, according to the way the probabilities are generated, a parent node's probability is at least as high as any of its child nodes.

Second, the sequencing method shall support a tunable mechanism to favor frequently queried patterns. With the probability framework, this is easy to realize. For each node C in

an XML schema, we assign a weight $w(C)$, which reflects the query frequency and selectivity of node C. Then, in sequencing, while observing the constraint in use, we arrange nodes by descending order of $p'(C|root)$, which is given by:

$$p'(C|root) = p(C|root) \times w(C). \quad (6)$$

Input: $\mathcal{D}$: an XML document
$\mathcal{S}$: XML schema for $\mathcal{D}$
$w(\cdot)$: weight of the nodes in $\mathcal{S}$
$p(\cdot|\cdot)$: existence probability of the nodes in $\mathcal{S}$

Output: a sequence representation of $\mathcal{D}$

for *each node* $d \in \mathcal{D}$ **do**
  derive $p'(d|root)$ by Eq (6);
end

Let r be the root node of $\mathcal{D}$;
$sequentialize(r)$;

Function $sequentialize(r)$
$T \leftarrow$ the subtree whose root node is r;
output r;
while T *is not empty* **do**
  $c \leftarrow$ a node in T with largest $p'(c|root)$;
  if c *has identical siblings* **then**
    $sequentialize(c)$;
  else
    output c;
  end
  remove c from T;
end

Algorithm 2: Sequencing based on constraint Eq (3)

6 Experiments

We implemented algorithms for sequencing, indexing, and querying XML data based on constraint sequences in C++. For comparison purposes, we also implemented a path index method similar to Data Guide [6], and a node index method similar to XISS [11]. We conduct our experiments on a Windows machine with a 1.8 GHz CPU and 256 MB main memory.

6.1 Datasets

We tested our algorithms on synthetic datasets, the DBLP dataset [10], and the XMark dataset [1].

Synthetic. The generation of synthetic tree structures takes three steps. First, we generate a random DTD schema based on the following user-provided parameters:

L	maximum tree height
F	maximum fanout of a node
A	percentage of value child nodes
I	percentage of identical siblings nodes

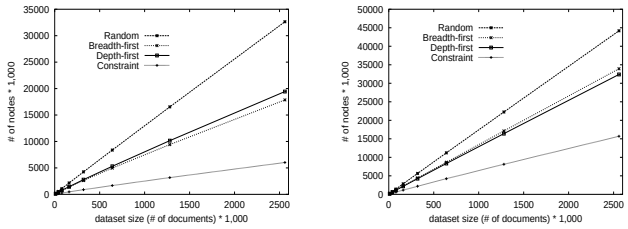
Second, we assign an occurrence probability with a uniform distribution in the range of $[P\%, 1.0]$ to each node, where P is another user-provided parameter. Finally, we generate N tree structures based on the schema, and determine the existence of their tree nodes by the occurrence probabilities. Also, we name a synthetic dataset by its generating parameters, for instance, $L_3F_5A_{25}I_0P_{40}$.

DBLP. The popular computer science bibliography database is widely used in benchmarking XML index methods. In the version we downloaded, there are 407,417 records, 8,537,681 nodes (elements, attributes, and values). Each record of DBLP corresponds to a publication, with a simple tree structure of maximum depth 6. The average length of the constraint sequences derived from the DBLP records is around 21.

XMark. An XMARK document consists of sub structures such as `item` (objects for sale), `person` (buyers and sellers), `open_auction`, `closed_auction`, etc. We convert each instance of these sub structures into a constraint sequence.

6.2 Index Size

We sequence synthetic tree structures using different strategies: random, depth/breadth-first, and probability-based constraint sequencing. The size of the index structure built on top of the sequences is determined by the extent of sharing among the sequences, that is, by their common prefixes.



(a) $L_3F_5A_{25}I_0P_{40}$ (b) $L_5F_3A_{40}I_0P_5$

Figure 14: Index Size (Synthetic Tree Structures)

Figure 14 shows the number of nodes in index structures built on sequences generated by different sequencing strategies. Combined with the index for the document id list, the size of the final disk-based index comes to $4n + cN$ bytes, where n is the input size (number of indexed XML documents/records), c is a constant, which in our implementation is close to 8, and N is the number of nodes such as those shown in Figure 14.

The average length of sequences in synthetic dataset $L_3F_5A_{15}I_0P_{40}$ and $L_5F_3A_{32}I_0P_5$ is around 25 and 32 respectively. Each sequence represents a compressed XML document. We find that the ratio of the index size to the compressed data size is around 1:1 for probability-based constraint sequencing, and 3-6:1 for random sequencing.

Figure 14 also shows that depth- and breadth-first sequencing result in indices of smaller size than random sequencing. It is because both depth- and breadth-first sequencing guarantee that a parent node appears earlier than its child nodes in the sequence and a parent node always has a larger occurrence probability than its child nodes.

	Path Expressions	Datasets
Q_1	`/site//item[location='United States']/mail/date[text='07/05/2000']`	XMark
Q_2	`/site//person/*/age[text='32']`	XMark
Q_3	`//closed auction[seller/person='person11304']/date[text='12/15/1999']`	XMark

Table 4: Sample Queries on XMark

Among the many factors that contribute to index size, we find that the average sequence length is the dominating one. When the average length increases from 25 in Figure 14(a) to 32 in Figure 14(b), the node number increases significantly for all sequencing methods. The increase is also contributed by the change of parameter A (value node ratio) from 25% to 40%. This indicates that sequence-based indexing is most favorable for large sets of small, homogeneous records, such as the DBLP dataset. However, for large XML documents (e.g. XMark), we can always decompose its DTD into multiple small, homogeneous structures and create separate index for each of them.

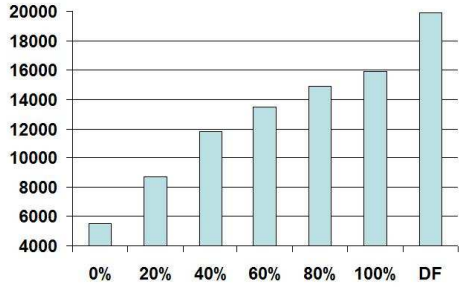


Figure 15: Impact of Identical Sibling Nodes

The experimental results shown in Figure 14 are about tree structures containing no identical sibling nodes ($I = 0$). When identical sibling nodes are present, we must adopt a stronger constraint such as f_2. Figure 15 shows the impact of identical sibling nodes on index size (dataset in use is $L_3F_5A_{25}I_?P_{40}$, where I goes from 0% to 100%). When percentage of identical sibling nodes goes up, the flexibility of ordering nodes according to their occurrence probabilities goes down. When $I = 100\%$, all siblings are identical, and as a result, constraint sequencing based on f_2 almost degrades to depth-first (DF) sequencing. The difference is that attribute values are still ordered by their occurrence probabilities. Thus, the index built on constraint sequencing is still smaller.

Records	Nodes	DF	CS
41,666	1,000,000	900,534	463,943
50,000	1,200,000	1,080,991	556,756
58,333	1,400,000	1,259,909	649,288
75,000	1,800,000	1,600,725	834,216
83,333	2,000,000	1,765,556	926,664

Table 5: XMark Index Size (Identical Sibling Nodes)

We perform similar tests on the XMark dataset. Table 5 and 6 show the size of the XMark datasets (number of records and number of XML nodes) and the size of the index structures corresponding to depth-first sequencing (DF) and probability-based constraint sequencing (CS). We used XMark sub structures with and without identical sibling nodes. In both cases, probability-based constraint sequencing outperforms depth-first sequencing.

Records	Nodes	DF	CS
20,000	680,000	658,921	347,862
30,000	1,020,000	989,515	501,448
40,000	1,360,000	1,318,699	702,915
50,000	1,700,000	1,648,693	864,414
65,250	2,218,500	2,149,701	1,157,109

Table 6: XMark Index Size (No Identical Sibling Nodes)

6.3 Query Performance

The sample queries in Table 4 are highly representative as they contain branching patterns, values, and wild cards '*', '//'. We ask the three queries against a 115,775 KB XMark dataset produced by `xmlgen` with factor 1.

	query length	result size	# disk accesses	time (s)
Q_1	6	1	23	0.10
Q_2	3	167	5	0.02
Q_3	5	6	9	0.07

Table 7: Query Performance on XMark

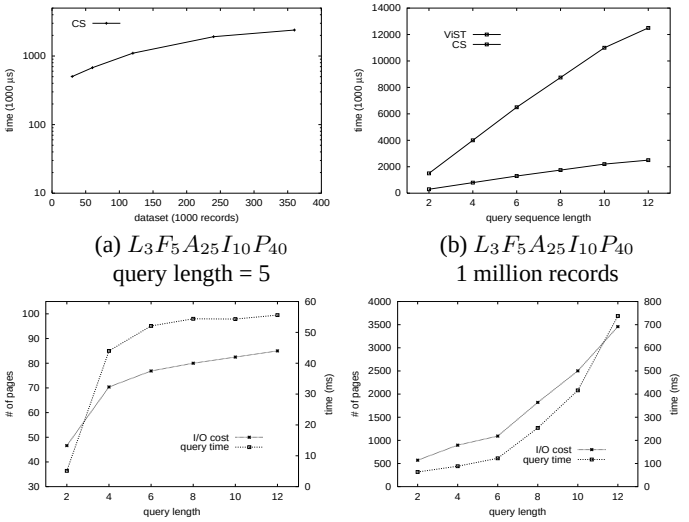
Table 7 shows the number of disk accesses and the elapsed time of running the three queries. The number of disk accesses is heavily influenced by the query length, as there is less node sharing deep down the tree, and as a result, each path link will contain larger amount of node labels. Overall, each query takes no more than 0.1 second, which shows that constraint sequencing provides a very efficient solution to querying XML by tree structures.

	path expressions	query by paths	query by nodes	CS
Q_1	/inproceedings/title	0.01	1.4	0.02
Q_2	/book/[key='Maier]/author	2.1	2.5	0.30
Q_3	/*/author[text='David']	1.9	4.9	0.31
Q_4	//author[text='David']	1.8	4.2	0.31

Table 8: Query Performance on DBLP

In Table 8, we compare sequence-based XML indexing with traditional query-by-path and query-by-node approaches, which are used in systems such as DataGuild [7] and XISS [11]. The advantage of constraint sequencing is most obvious when queries contain tree patterns and attribute values.

We also performed tests on synthetic datasets and random query sequences. Figure 16(a) shows how constraint sequencing scales when the size of the dataset increases. Among the state-of-the-art XML indexing approaches, ViST's query performance comes closest to that of constraint sequencing. We compare constraint sequencing with the depth-first sequencing



(a) $L_3F_5A_{25}I_{10}P_{40}$ query length = 5

(b) $L_3F_5A_{25}I_{10}P_{40}$ 1 million records

(c) with no identical sibling nodes

(d) with identical sibling nodes

Figure 16: Performance on Synthetic Datasets

used in the ViST approach [18]. Figure 16(b) shows the result of comparison. The performance difference is due to the following reasons. First, indices built on depth-first sequences are usually three times larger than those built on constraint sequences. Second, ViST's sequencing and query algorithms do not guarantee query equivalence between a structure match and a subsequence match, and expensive join operations are used to remedy this problem. In Figure 16(c) and 16(d) we compare performance on two fixed size datasets (100K records), one of which contains identical sibling nodes. It shows that the presence of identical sibling nodes has a huge impact on query performance.

7 Conclusion

We introduced an XML indexing infrastructure which makes tree patterns a first class citizen in XML query processing. Unlike most indexing methods that directly manipulate tree structures, we build our indexing infrastructure on a much simpler data model: constraint sequences. Previous sequence-based indexing approaches relied on ad hoc sequencing methods, such as the depth-first traversal, the breadth-first traversal, and the Prüfercodes. The performance of sequence-based indexing is hampered for the following reasons: i) ad hoc sequencing does not necessarily preserve the query equivalence a structure match and a subsequence match, and this nonequivalence is often remedied by expensive join operations or document-by-document post-processing; and ii) ad hoc sequencing is not schema aware, or it does not take into consideration the distribution of the data. In other words, ad hoc sequences are not optimized for index and query performance. In contrast, constraint sequencing maps a tree structure to a set of sequential representations, each of which satisfies query equivalence. This, in turn, enables us to form a sequencing strategy based on the data schema and the distribution of the entire dataset, and eventually choose the best sequential representation for a tree structure. Our experiments show that constraint sequencing outperforms previous approaches in index and query performance.

References

[1] XMARK: The XML-benchmark project. http://monetdb.cwi.nl/ xml, 2002.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[3] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2001.

[4] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2002.

[5] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for XML data. In *ACM SIGMOD*, June 2002.

[6] B. F. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, September 2001.

[7] R. Goldman and J. Widom. DataGuides: Enable query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, August 1997.

[8] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2002.

[9] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *ACM SIGMOD*, June 2002.

[10] M. Ley. DBLP database web site. http://www.informatik.uni-trier.de/ ley/db, 2000.

[11] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, September 2001.

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[13] X. Meng, Y. Jiang, Y. Chen, and H. Wang. XSeq: an indexing infrastructure for tree pattern queries (demo). In *SIGMOD*, pages 941–942, 2004.

[14] T. Milo and D. Suciu. Index structures for path expression. In *Proceedings of 7th International Conference on Database Theory (ICDT)*, pages 277–295, January 1999.

[15] S. Picciotto. *How to Encode a Tree*. PhD thesis, University of California, San Diego, 1999.

[16] P. R. Raw and B. Moon. PRIX: Indexing and querying XML using prüfer sequences. In *ICDE*, 2004.

[17] E. Ukkonen. Constructing suffix-trees on-line in linear time. *Algorithms, Software, Architecture: Information Processing*, pages 484–92, 1992.

[18] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD*, 2003.

[19] H. Wang, C.-S. Perng, W. Fan, S. Park, and P. S. Yu. Indexing weighted sequences in large databases. In *ICDE*, 2003.