ViST: A Dynamic Index Method for Querying XML Data by Tree Structures

Haixun Wang   [1]Sanghyun Park   Wei Fan   Philip S. Yu

IBM Thomas J. Watson Research Center
Hawthorne, NY 10532
{haixun, weifan, psyu}@us.ibm.com

[1]Dept. of Computer Science & Engineering
POSTECH, Pohang, Korea
sanghyun@postech.ac.kr

ABSTRACT
With the growing importance of XML in data exchange, much research has been done in providing flexible query facilities to extract data from structured XML documents. In this paper, we propose ViST, a novel index structure for searching XML documents. By representing both XML documents and XML queries in structure-encoded sequences, we show that querying XML data is equivalent to finding subsequence matches. Unlike index methods that disassemble a query into multiple sub-queries, and then *join* the results of these sub-queries to provide the final answers, ViST uses tree structures as the basic unit of query to avoid expensive join operations. Furthermore, ViST provides a unified index on both content and structure of the XML documents, hence it has a performance advantage over methods indexing either just content or structure. ViST supports dynamic index update, and it relies solely on B^+Trees without using any specialized data structures that are not well supported by DBMSs. Our experiments show that ViST is effective, scalable, and efficient in supporting structural queries.

1. INTRODUCTION
With the growing importance of XML in data exchange, much research has been done in providing flexible query mechanisms to extract data from XML documents [11, 18, 16, 9, 6, 14]. The semi-structured nature of XML data and the requirements on query flexibility pose unique challenges to database indexing methods. In this paper, we introduce a novel index structure, ViST[1], which provides solutions to a wide range of challenges, and offers better performance and usability than previous approaches in XML indexing.

XML provides a flexible way to define semi-structured data. For instance, purchase records that contain information of buyers and sellers can be described by the DTD schema shown in Figure 1. A sample XML document based on this DTD is shown in Figure 3.

```
⟨!ELEMENT purchases   (purchase*)⟩
⟨!ELEMENT purchase    (seller, buyer)⟩
⟨!ATTRIST seller      ID ID location CDATA name CDATA⟩
⟨!ELEMENT seller      (item*)⟩
⟨!ATTRIST buyer       ID ID location CDATA name CDATA⟩
⟨!ELEMENT item        (item*)⟩
⟨!ATTRIST item        name CDATA manufacturer CDATA⟩
```

Figure 1: DTD of purchase records. Sellers supply items (some contain sub-items) to buyers.

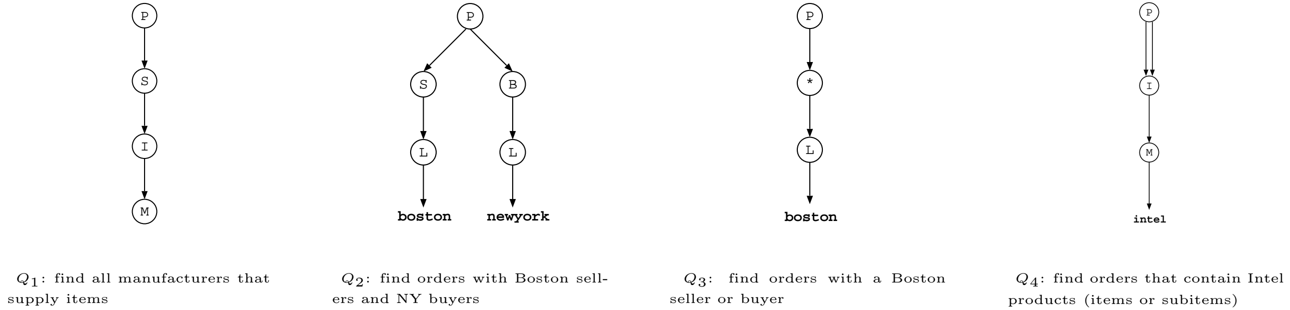
Several query languages, including XPath [7], Quilt [5], XML-QL [10], and XQuery [4], have been proposed for semi-structured XML data. The ability to express complex structural or graphical queries is one of the major focuses in XML query language design. In Figure 2, we show four sample queries in graph form. It is well known that querying XML data is equivalent to finding sub structures of the data graph that match the query structure.

Many state-of-the-art approaches create indexes on paths (e.g., "/P/S/I/M" as in Q_1) or nodes in DTD trees. Path indexes can answer simple queries such as Q_1 efficiently. However, queries involving branching structures (Q_2, for instance) usually have to be disassembled into multiple sub-queries, each corresponding to a single path in the graph. The results of these sub-queries are then combined by expensive *join* operations to produce final answers. For the same reason, these methods are also inefficient in handling '*' or '//' queries (Q_3 and Q_4, for instance), which too, correspond to multiple paths. To avoid expensive join operations, some index methods create special index entries for frequently occurring multiple-path queries (known as *refined paths*) [9, 14]. The potential disadvantages of this approach include i) we need to monitor query patterns, ii) it is not a general approach since not every branching query is optimized, and iii) the number of refined paths can have a huge impact on the size and the maintenance cost of the index.

Moreover, to retrieve semi-structured data efficiently, it is essential to have index on both structure and content of the XML data. Nevertheless, many algorithms index on structure only, or index on structure and content separately, which means, for instance, attribute values in Q_2, Q_3, and Q_4 are not used for filtering in the most effective way.

[1]ViST stands for Virtual Suffix Tree

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06...$5.00.



Q_1: find all manufacturers that supply items

Q_2: find orders with Boston sellers and NY buyers

Q_3: find orders with a Boston seller or buyer

Q_4: find orders that contain Intel products (items or subitems)

Figure 2: XML Queries in Graph Form

Another important aspect to XML indexing is whether the index structure supports dynamic data insertion, deletion, and update, and whether the index depends on specialized data structures not well-supported by database systems.

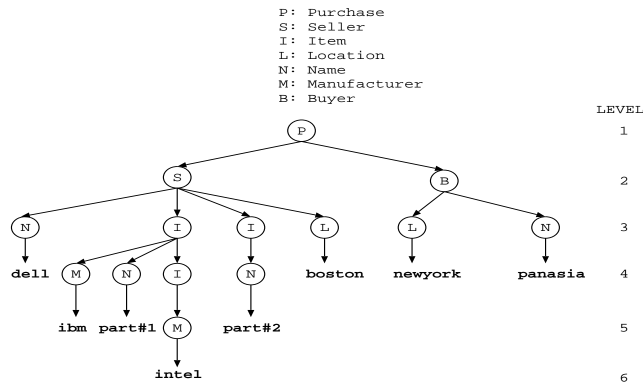


Figure 3: A Single Purchase Record

In this paper, we propose ViST, a novel index structure that addresses a wide range of challenges in indexing semi-structured data. Our objective is to provide a *general* method so that structural XML queries need not be decomposed into sub-queries, which means join operations can be avoided.

We transform XML data and XML queries into *structure-encoded sequences*. We demonstrate that XML queries, including those with branches, or wild-cards ('*' and '//'), can be expressed by structure-encoded sequences. We show that querying XML data is equivalent to finding (non-contiguous) subsequence matches, and we use a *virtual suffix tree* to organize structure-encoded sequences to speed up the matching process. Our index structure consists of two parts, the `D-Ancestor` index and the `S-Ancestor` index. The former indexes nodes by their ancestor-descendant relationships in the original XML document tree and the latter indexes nodes by their ancestor-descendant relationships in the virtual suffix tree. Together, structural XML queries can be answered in a way similar to substring matching using suffix trees.

Our approach also answers challenges in index structure design. Unlike many previous methods that index either just structure or content of the XML data, ViST unifies structural indexes and value indexes into a single index. In addition, we propose a technique called *dynamic virtual suffix tree labeling*, based on which, structural XML queries, as well as dynamic index update, can be performed directly on B^+Trees, instead of relying on specialized data structures such as suffix trees that are not well supported by DBMSs.

Our Contributions

To the best of the authors' knowledge, the index structure proposed in this paper is the first approach that provides all of the following features at the same time.

- Unlike most indexing methods that disassemble a structured query into multiple sub-queries, and then *join* the results of these sub-queries to provide the final answers, our method uses tree structures as the basic unit of query to avoid expensive join operations.
- Our approach provides a unified index on both the content and the structure of XML documents, hence it has a performance advantage over methods indexing either just content or structure.
- Unlike some XML indexing approaches that rely on specialized data structures such as the suffix tree, which is not well-supported for disk-based data, we leverage the mature disk-based B^+Tree index.
- Our index structure supports dynamic data insertion and deletion.

Paper Organization

In the next section, we introduce structure-encoded sequence, a sequential representation of XML documents and XML queries, and show that querying structured XML data is equivalent to finding subsequence matches. We present our sequence matching algorithm in Section 3. Section 4 contains experiments that show the effectiveness of our algorithms. In Section 5, we review several state-of-the-art XML indexing approaches. We conclude our work in Section 6.

2. STRUCTURE-ENCODED SEQUENCES

In this section, we introduce structure-encoded sequences, a sequential representation of both XML data and XML queries. We show that querying XML is equivalent to finding subsequence matches.

The purpose of modeling XML queries through sequence matching is to avoid as many unnecessary join operations as possible in query processing. That is, we use structure-encoded sequences, instead of nodes or paths, as the basic unit of query. Through sequence matching, we match structured queries against structured data as a whole, without breaking down the queries into sub queries of paths or nodes and relying on join operations to combine their results. Many XML databases, such as DBLP [15] and the Internet movie database IMDB [22], contain a large set of records of the same structure. Other XML databases may not be as homogeneous. A synthetic XMARK [23] dataset consists of one (huge) record. However, each sub structure in XMARK's schema, `items`, `closed_auction`, `open_auction`, `person`, etc, contains a large number of instances in the database and justifies to have an index of its own. Our sequence matching approach ensures that queries confined within the same structure are matched as a whole.

Mapping Data and Queries to Structure-Encoded Sequences

Consider the XML purchase record shown in Figure 3. We use capital letters to represent names of elements/attributes, and we use a hash function, $h()$, to encode attribute values into integers. Suppose, for instance, $v_1 = h(\text{"dell"})$ and $v_2 = h(\text{"ibm"})$. We then use v_1 and v_2 to represent "dell" and "ibm" respectively.

We represent an XML document by the preorder sequence of its tree structure. For the purchase record example, its preorder sequence is shown in Table 1.

$$\mathbf{PSN}v_1\mathbf{IM}v_2\mathbf{N}v_3\mathbf{IM}v_4\mathbf{IN}v_5\mathbf{L}v_6\mathbf{BL}v_7\mathbf{N}v_8$$

Table 1: Preorder sequence of the XML purchase record example (Figure 3)

Since isomorphic trees may produce different preorder sequences, we enforce an order among sibling nodes. The DTD schema embodies a linear order of all elements/attributes defined therein. If the DTD is not available, we simply use the lexicographical order of the names of the elements/attributes. For example, under lexicographical order, the `Buyer` node will precede the `Seller` node under `Purchase`. For multiple occurring child nodes (such as the `Item` nodes under `Seller`), we order them arbitrarily. As we shall see later, branching queries require special handling when multiple occurring child nodes are involved.

To reconstruct trees from preorder sequences, extra information is needed. Our structure-encoded sequence, defined below, is a two dimensional sequence, where the second dimension preserves the structure of the data.

DEFINITION 1. *Structure-Encoded Sequence*
A Structure-Encoded Sequence, derived from a prefix traversal of a semi-structured XML document, is a sequence of (symbol, prefix) pairs:

$$\mathcal{D} = (a_1, p_1), (a_2, p_2), \cdots, (a_n, p_n)$$

where a_i represents a node in the XML document tree, (of which $a_1, \cdots, a_n$ is the preorder sequence), and p_i is the path from the root node to node a_i.

Based on the definition, the XML purchase record (Figure 3) can be converted to the structure-encoded sequence in Figure 4. The prefixes in the sequential representation contain much redundant information; however, as we shall see, since we do not store duplicate (symbol, prefix) pairs in the index and that the prefix can be encoded easily, it will not create problems in index size or storage.

$$\begin{aligned}&\mathcal{D} = \\ &\underline{(P, \epsilon)}, \underline{(S, P)}, (N, PS), (v_1, PSN), (I, PS), (M, PSI), \\ &(v_2, PSIM), (N, PSI), (v_3, PSIN), (I, PSI), (M, PSII), \\ &(v_4, PSIIM), (I, PS), (N, PSI), (v_5, PSIN), \underline{(L, PS)}, \\ &\underline{(v_6, PSL)}, \underline{(B, P)}, \underline{(L, PB)}, \underline{(v_7, PBL)}, (N, PB), (v_8, PBN)\end{aligned}$$

Figure 4: The structure-encoded sequence of the purchase record document (Figure 3). The underlined non-contiguous subsequence of $\mathcal{D}$ matches query Q_2 (Table 2).

In the same spirit, we convert XML queries to structure-encoded sequences. The queries in Figure 2 can be transformed to the structure-encoded sequences in Table 2. The following rules are observed in the conversion:

- Just like converting XML data, we use preorder sequences to represent queries. (Example: Q_1, Q_2)
- Wild-card nodes ('*' and '//') are discarded. However, the prefix paths of their sub nodes will contain a '*' or '//' symbol as a place holder. As we shall see, '*' and '//' are handled as range queries by ViST in sequence matching. (Example: Q_3, Q_4)

Querying XML through Structure-Encoded Sequence Matching

The purpose of introducing structure-encoded sequences is to model XML queries through sequence matching. In other words, querying XML is equivalent to finding (non-contiguous) subsequence matches. We show this by queries $Q_1, \cdots, Q_4$ (Table 2).

The structure-encoded sequence of Q_1 is a subsequence of $\mathcal{D}$, and we can see Q_1 is a sub tree of the XML purchase record that $\mathcal{D}$ represents. The sequence of Q_2 is a non-contiguous subsequence of $\mathcal{D}$, and again, Q_2 is a sub tree of the XML purchase record. The same can be said for query Q_3 and Q_4, where prefix paths contain wild-cards '*' and '//' — if we simply match '*' with any single symbol in the path, and '//' with any portion of the path.

The obvious benefits of modeling XML queries through sequence matching is that structural queries can be processed as a whole instead of being broken down to smaller query units (paths or nodes of XML document trees), as combining the results of the sub queries by join operations is often expensive. In other words, we use structures as the basic unit of query.

Most structural XML queries can be performed through direct subsequence matching. The only exception occurs

	Path Expression	Structure-Encoded Sequence
Q_1 :	$/Purchase/Seller/Item/Manufacturer$	$(P,\epsilon)(S,P)(I,PS)(M,PSI)$
Q_2 :	$/Purchase[Seller[Loc=v_5]]/Buyer[Loc=v_7]$	$(P,\epsilon)(S,P)(L,PS)(v_5,PSL)(B,P)(L,PB)(v_7,PBL)$
Q_3 :	$/Purchase/*[Loc=v_5]$	$(P,\epsilon)(L,P*)(v_5,P*L)$
Q_4 :	$/Purchase//Item[Manufacturer=v_3]$	$(P,\epsilon)(I,P//)(M,P//I)(v_3,P//IM)$

Table 2: XML Queries in Path Expression and Sequence Form

when a branch has multiple identical child nodes. For instance, in $Q_5 = /A[B/C]/B/D$, the two nodes under the branch are the same: B. In this case, the tree isomorphism problem can not be avoided by enforcing sibling orders, since the two nodes are identical. As a result, the preorder sequences of XML data trees that contain such a branch can have two possible forms. In order to find all matches, we convert Q_5 to two different sequences, namely, $(A,\epsilon)(B,A)(C,AB)(B,A)(D,AB)$ and $(A,\epsilon)(B,A)(D,AB)$ $(B,A)(C,AB)$. We find matches for these two sequences separately and *union* their results. On the other hand, we may find false matches if the indexed documents contain branches with identical child nodes. Then, we ask multiple queries and compute *set difference* on their results. If, in the unlikely case, the query contains a large number of same child nodes under a branch, we can choose to disassemble the tree at the branch into multiple trees, and use *join* operations to combine their results. For instance, Q_5 can be disassembled into two trees: $(A,\epsilon)(B,A)(C,AB)$ and $(A,\epsilon)(B,A)(D,AB)$[2].

After both XML data and XML queries are converted to structure-encoded sequences, it is straightforward to devise a brute force algorithm to perform (non-contiguous) sequence matching. The rest of the paper focuses on building a dynamic index structure so that such matches can be found efficiently.

3. THE VIST APPROACH

We present ViST in three stages. The naïve algorithm, based entirely on suffix trees, requires traversal of a large portion of the tree structure for non-contiguous subsequence matching. We then present RIST, which improves the naïve algorithm by using B^+Trees to index suffix tree nodes. Finally, we present ViST, an index structure having the same functionality but relying exclusively on B^+Trees.

3.1 Desiderata

The desiderata of an XML indexing method include:

1. The index method should support structural queries directly. With structure-encoded sequences, this requirement is equivalent to having efficient support for (non-contiguous) subsequence matching.

2. Instead of relying on specialized data structures such as suffix trees, the index method should leverage well-supported database indexing techniques such as B^+Trees.

3. The index structure should allow dynamic data insertion, deletion, etc.

[2] Q_5 is a special case where each split tree is a single path.

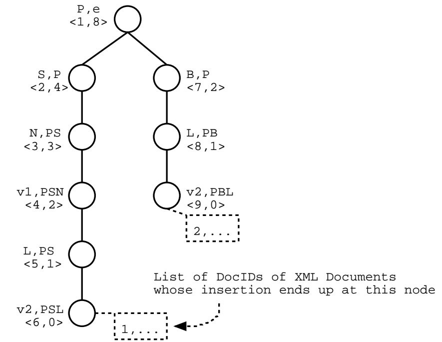
3.2 A Naive Algorithm Based on Suffix Trees

Much research has been done in the area of subsequence matching [12]. The most widely used index structure for substring matching is the *suffix tree* [17], which embodies a compact index to all the distinct, contiguous substrings of a given string.

$$Doc_1 \;:\; (P,\epsilon)(S,P)(N,PS)(v_1,PSN)(L,PS)(v_2,PSL)$$
$$Doc_2 \;:\; (P,\epsilon)(B,P)(L,PB)(v_2,PBL)$$

$$Q_1 \;:\; (P,\epsilon)(B,P)(L,PB)(v_2,PBL)$$
$$Q_2 \;:\; (P,\epsilon)(L,P*)(v_2,P*L)$$

XML docs and queries in structure-encoded sequences



A tree structure for Doc_1 and Doc_2

Figure 5: Suffix-tree-like structure for structure-encoded sequences (Labels $\langle.,.\rangle$ are described in Section 3.3)

Figure 5 shows an example of using a suffix-tree-like structure to index structure-encoded sequences for non-contiguous matching. We insert two sequences, Doc_1 and Doc_2, into the suffix tree. Originally, the elements in the sequences represent nodes in the XML document trees, from which the sequences are derived. Now, they also represent nodes in the suffix tree. Since the nodes are involved in two different trees, two kinds of ancestor-descendant relationships among the sequence elements arise: i) the ancestor-descendant relationships of the nodes that they represent in the original XML document tree, and ii) the ancestor-descendant relationships of the nodes that they represent in the suffix tree. We call the 1st relationship the **D-Ancestorship**, and say, for instance, element (S,P) is a D-ancestor of (L,PS). We call the 2nd relationship the **S-Ancestorship**, and say, for instance, element (v_1,PSN) is an S-Ancestor of (L,PS).

Input: $Q = q_1, \cdots, q_k$, a query sequence
S, a suffix tree for a set of sequences
Output: all occurrences of Q in S

/* Search begins at the root of the suffix tree */
$NaiveSearch(S \rightarrow root, 1)$;

Function $NaiveSearch(n, i)$
if $i \leq k$ **then**
  for *each node c that is a descendent of node n* **do**
    /* n is an S-Ancestor of c */
    if *c matches q_i* **then**
      /* n is a D-Ancestor of c */
      $NaiveSearch(c, i + 1)$;
    end
  end
else
  Output all document IDs attached to the nodes under node n;
end

Algorithm 1: NaiveSearch: A naïve algorithm based on suffix trees.

Algorithm 1 presents a naïve method for non-contiguous subsequence matching. Suppose node x is one of the nodes matching $q_1, \cdots, q_{i-1}$. To match the next element q_i, we check all the nodes under x, which are the nodes satisfying the S-Ancestorship. Among them, we find those that match q_i's (Symbol, Prefix) pair, which are the nodes satisfying the D-Ancestorship, as Prefix encodes D-Ancestorship in the XML document tree. For example, to match Q_2, we start with the root node, which matches the first element of Q_2, (P, ϵ). Then, we search under the root for all nodes that match $(L, P*)$, which yields (L, PS) and (L, PB). Finally, we search for (v_2, PSL) (wild-card '*' in the query is instantiated to 'S' by the previous match) under the node labeled (L, PS), and (v_2, PBL) under the node labeled (L, PB).

In essence, Algorithm 1 searches nodes first by S-Ancestorship (searching under a suffix tree node), and then D-Ancestorship (matching nodes by symbols and prefixes). Algorithm 1 supports structural query, however, there are several difficulties in using suffix tree to index structure-encoded sequences. First, searching for nodes satisfying both S-Ancestorship and D-Ancestorship is extremely costly since we need to traverse a large portion of the subtree for each match. Second, suffix trees are main memory structures that are seldom used for disk resident data [9], and most commercial DBMSs do not have support for such structures.

3.3 RIST: Indexing by Ancestor-Descendent Relationships

RIST[3] improves the naïve algorithm by eliminating costly suffix tree traversal. With RIST, when we reach node X after matching a prefix of the query, we can 'jump' directly to those nodes Y to which X is both a D-Ancestor and an S-Ancestor. Thus, we no longer need to search among the descendents of X to find such Ys one by one. More specifically, RIST is designed as follows.

[3]RIST stands for Relationships Indexed Suffix Tree

1. We index nodes in the suffix tree by their (Symbol, Prefix) pairs. This is realized by a B$^+$Tree. It enables us to search nodes by (Symbol, Prefix), that is, by D-Ancestorship, since Prefix encodes ancestor-descendant relationships in the XML document tree. We call this B$^+$Tree the `D-Ancestorship B+Tree`.

2. Among all nodes satisfying D-Ancestorship, we are interested in those satisfying S-Ancestorship as well. We create labels for suffix tree nodes so that we can tell S-Ancestorship between two nodes by their labels. We use B$^+$Trees to index nodes by labels. We call such B$^+$Trees `S-Ancestorship B+Trees`.

Index Construction

We determine the D-Ancestorship between two elements by checking their prefixes, however, to determine S-Ancestorship between two elements requires additional information. We label each suffix tree node x by a pair $\langle n_x, size_x \rangle$, where n_x is the prefix traversal order of x in the suffix tree, and $size_x$ is the total number of descendants of x in the suffix tree. Labeling can be accomplished by making a depth-first traversal of the suffix tree. An example of such labeling is shown in Figure 5. With the labeling, the S-Ancestorship between any two nodes can be decided easily: if x and y are labeled $\langle n_x, size_x \rangle$ and $\langle n_y, size_y \rangle$ respectively, node x is an S-Ancestor of node y iff $n_y \in (n_x, n_x + size_x]$.

To construct the B$^+$Trees, we first insert all suffix tree nodes into the `D-Ancestorship B+Tree` using their (Symbol, Prefix) as keys. For all nodes x inserted with the same (Symbol, Prefix), we index them by an `S-Ancestorship B+Tree`, using the n_x values of their labels as keys.

In addition, we also build a `DocId B+Tree`, which stores for each node x (using n_x as key), the document IDs of those XML sequences that end up at node x when they are inserted into the suffix tree.

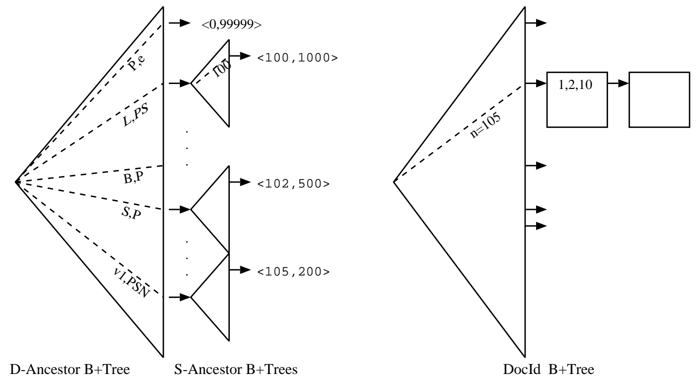


Figure 6: The RIST index structure

Figure 6 shows the index structure of RIST. In summary, the construction of the index structure takes three steps: i) adding all structure-encoded sequences into a suffix tree; ii) labeling the suffix tree by making a preorder traversal; and iii) for each node (Symbol, Prefix) labeled $\langle n, size \rangle$, inserting it to the `D-Ancestor B+Tree` using (Symbol, Prefix) as the key, and then the `S-Ancestor B+Tree` using n as the key.

Subsequence Matching

Suppose node x, labeled with $\langle n_x, size_x \rangle$, is one of the nodes matching a query prefix $q_1, \cdots, q_{i-1}$. To match the next element q_i in the query, we consult the `D-Ancestor B+Tree` using q_i as a key. The `D-Ancestor B+Tree` returns the root of an `S-Ancestor B+Tree`. We then issue a range query $n_x < n \leq n_x + size_x$ on the `S-Ancestor B+Tree` to find the descendants of x immediately. For each descendant, we use the same process to match symbol q_{i+1}, until we reach the last element of the query.

If node y is one of the nodes that matches the last element in the query, then the document IDs associated with y or any descendant node of y are answers to the query. Based on y's label, say $\langle n_y, size_y \rangle$, we know y's descendants are in the range of $(n_y, n_y + size_y]$. Thus, we perform a range query $[n_y, n_y + size_y]$ on the `DocId B+Tree` to retrieve all the document IDs for y and y's descendants.

Algorithm 2 formalizes the querying process.

Input: $Q = q_1, \cdots, q_k$, a query sequence
`D-Ancestor B+Tree`, index of (symbol,prefix) pairs
`S-Ancestor B+Trees`, index of $\langle n, size \rangle$ labels
`DocId B+Tree`, mapping between the n values in node labels and document IDs
Output: all occurrences of Q in the XML data

$Search(\langle 0, size \rangle, 1)$; /* $\langle 0, size \rangle$ is the label of the root node of the suffix tree */

Function $Search(\langle n, size \rangle, i)$
if $i \leq |Q|$ **then**
  $T \leftarrow$ retrieve, from the `D-Ancestor B+Tree`, the `S-Ancestor B+Tree` that represents q_i;
  $N \leftarrow$ retrieve from T, the `S-Ancestor B+Tree`, all nodes with range inside $(n, n + size]$;
  for *each node* $c \in N$ **do**
    Assume c is labeled $\langle n', size' \rangle$;
    $Search(\langle n', size' \rangle, i + 1)$;
  end
else
  Perform a range query $[n, n+size)$ on the `DocId B+Tree` to output all document IDs in that range;
end

Algorithm 2: Search: non-contiguous subsequence matching using B$^+$Tree

Handling Wild Cards '' and '//'*

If an element in the query sequence contains wild-card '*', more than one `S-Ancestor B+Trees` might match the element. Let $Q = (P, \epsilon)(L, P*)(v_2, P*L)$. To match $(L, P*)$, we issue a range query to the `D-Ancestor B+Tree` (the key of the `D-Ancestor B+Tree` is ordered first by the $Symbol$, then by the length of the $Prefix$, and lastly by the content of the $Prefix$). The search then continues on each `S-Ancestor B+Tree` returned by the range query. Note that we only need to handle $(L, P*)$, or elements whose prefixes end with '*', since the matching of $(L, P*)$ will instantiate the '*' in $(v_2, P*L)$ to a concrete symbol, which means $(v_2, P*L)$ is not considered as a wild-card query. Queries with wild-card '//' are handled as a series of '*' queries. Thus, the index supports wild cards '*' and '//' appearing both in the beginning and in the middle of a query sequence.

In summary, unlike the naïve algorithm, RIST does not use suffix trees for subsequence matching (Algorithm 2). From any node, instead of searching the entire subtree under the node, we can 'jump' to the sub nodes that match the next element in the query right away. Thus, RIST supports non-contiguous subsequence matching efficiently. In comparison with many other indexing approaches that break a query down to pieces and then join the results, RIST has the advantage of querying tree structures as a whole.

3.4 ViST: The Virtual Suffix Tree

RIST uses a static scheme to label suffix tree nodes, which prevents it from supporting dynamic insertions, since for any node x labeled $\langle n, size \rangle$, late insertions can change the number of nodes that appear before x (in the prefix order) as well as the size of the subtree rooted at x, which means neither n nor $size$ can be fixed.

The sole purpose of the suffix tree is to provide a labeling mechanism to encode S-Ancestorships. Suppose a node x is created for element d_i during the insertion of sequence $d_1, \cdots, d_i, \cdots, d_k$. If we can estimate i) how many different elements will possibly follow d_i in future insertions, and ii) the occurrence probability of each of these elements, then we can label x's child nodes right away, instead of waiting until all sequences are inserted. It also means i) the suffix tree itself is no longer needed, because its sole purpose of providing a labeling mechanism can be accomplished on the fly; and ii) we can support dynamic data insertion and deletion.

ViST uses a dynamic labeling method to assign labels to suffix tree nodes. Once assigned, the labels are fixed and will not be affected by subsequent data insertion or deletion.

3.4.1 Dynamic Virtual Suffix Tree Labeling

We present a dynamic method for labeling suffix tree nodes without building the suffix tree. The method relies on rough estimations of the number of attribute values, and other semantic/statistical information of the XML data. To the authors' knowledge, the only dynamic labeling scheme available was recently proposed by Cohen et al. [8] to label XML document trees. Our dynamic scheme is designed to label suffix trees built for structure-encoded sequences derived from XML document trees.

Top-Down Scope Allocation

A tree structure defines nested scopes: the scope of a child node is a sub scope of its parent node, and the root node has the maximum scope which covers the scope of each node. Initially, the suffix tree contains a single node (root), and we let it cover the entire scope, $[0, Max)$, where Max is the maximum value that the machine can represent under certain precision[4].

[4] $Max = 2^{128} - 1$ if we use 8 bytes to represent an integer. Alternatively, we can use 16 bytes for a Max as large as $2^{256} - 1$.

Semantic and Statistical Clues

Semantic and statistical clues of structured XML data can often assist sub scope allocation. Figure 7 shows a sample XML schema. We use $p(u|x)$ to denote, in an XML document, the probability that node u occurs given node x occurs. For a multiple occurring node v, $p(v|x)$ denotes the probability that at least one v occurs given x occurs in an XML document.

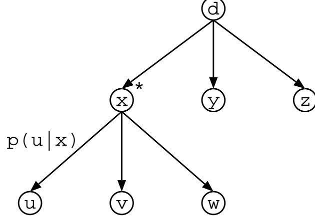


Figure 7: A Simple XML Schema

If x is the parent of u, usually it is not difficult to derive or estimate, from the semantics of the XML structure or the statistics of a sample dataset, the probability $p(u|x)$. For instance, if each Buyer has a name, then $p(Name|Buyer) = 1$. If we know that *roughly* 10% of the items contain at least a sub-item, then $p(SubItem|Item) = 0.1$.

We start with two assumptions: i) we know probability $p(u|x)$ for all u, where x is the parent of u; and ii) in XML document trees, sibling nodes occur independently of each other. We will see how assumption ii) can be relaxed. If node x appears in an XML document based on the schema in Figure 7, then each of the following symbols can appear immediately after x in the sequence derived from the document: u, v, w, y, z, and ϵ (empty, x is the last element). These symbols form the *follow set* of x.

DEFINITION 2. *Follow Set*
Given a node x in an XML scheme, we define the follow set of x as a list, i.e., $follow(x) = y_1, \cdots, y_k$, where y_i satisfies the following condition: $x \prec y_i \prec y_{i+1}$ (according to prefix traversal order) and the parent of y_i is either x or an ancestor node of x.

It is straightforward to prove that only symbols in x's follow set can appear immediately after x. Suppose $follow(x) = y_1, \cdots, y_k$, based on the assumption that sub nodes occur independently, we have:

$$p(y_i|x) = p(y_i|d), \quad where\ d\ is\ the\ parent\ of\ y_i \qquad (1)$$

Eq(1) is trivial if $d = x$. If $d \neq x$, then based on the definition of the follow set, d must be an ancestor of x, so we have $p(y_i|x) = p(y_i|x, d)$. Since x and y_i are in different branches under d, according to our assumption, they occur independently of each other, which means $p(y_i|x, d) = p(y_i|d)$.

Let $follow(x) = y_1, \cdots, y_k$. The probability that x is followed immediately by y_1 is $p(y_1|x)$, by y_2 is $(1-p(y_1|x))p(y_2|x)$. The probability that x is followed immediately by y_i is:

$$P_x(y_i) = p(y_i|x) \prod_{k=1}^{i-1} (1 - p(y_k|x)) \qquad (2)$$

We allocate subscopes for the child nodes in the suffix tree according to the probability. More formally, if x's scope is $[l, r)$, the size of the subscope assigned to y_i, the ith symbol in x's follow set, is:

$$s_i = (r - l - 1)P_x(y_i)/C \qquad (3)$$

where $C = \sum_{y \in follow(x) - \{\epsilon\}} P_x(y)$ is a normalization factor (we do not allocate any scope to ϵ).

In other words, we should assign a subscope $[l_i, r_i) \subset [l, r)$ to y_i, where:

$$\begin{aligned} l_i &= l + 1 + (r - l - 1) \sum_{j=1}^{i} P_x(y_j) \qquad (4) \\ r_i &= l_i + s_i \end{aligned}$$

In the following situations, the follow set and Eq(2) need to be adjusted.

- A same node can occur multiple times under its parent node. Let $follow(x) = y_1, \cdots, y_k$. If x occurs multiple times under its parent, then x also appears in $follow(x)$, i.e., $follow(x) = y_1, \cdots, x, \cdots, y_k$, where the symbols before x are the descendants of x. Let the probability that an XML document contains n occurrences of x under d is $p_n(x|d)$, then the probability that the (n-1)-th x is followed immediately by the n-th x is $p_n(x|d) \prod_{k=1}^{i-1}(1 - p(y_k|x))$.
- Nodes do not occur independently. Eq(2) is derived based on the assumption that nodes occur independently. However, this may not be true. Suppose for instance, in Figure 7, either u or v must appear under x, and $p(u|x) = p(v|x) = .8$. We have $follow(x) = u, v$, since if either u and v must occur, there is no possibility that any of w,y,z,ϵ can immediately follow x. Thus, we have,

$$\begin{aligned} P_x(u) &= p(u|x) = .8 \\ P_x(v) &= (1 - p(u|x))p(v|\neg u, x) = .2 \times 1 = .2 \end{aligned}$$

Dynamic Scope Allocation without Clues

Assume we do not have any statistical information of the data, or any semantic knowledge about the schema, and all that we can rely on is a *rough* estimation of the number of different elements that follow a given element. The best we can do is to assume each of these elements occurs at *roughly* the same rate. This situation usually corresponds to attributes values. For instance, in a certain dataset, we roughly estimate the number of different values for attribute `CountryOfBirth` to be 100.

Suppose node x is assigned a scope of $[l, r)$. Node x itself will then take l as its ID, and the remaining scope $[l+1, r)$ is available for x's child nodes. Assume the expected number

of child nodes of x is λ. Without the knowledge of the occurrence rate of each child node, we allocate $\frac{1}{\lambda}$ of the remaining scope to x's first inserted child, which will have a scope of size $(r-l-1)/\lambda$. We allocate $\frac{1}{\lambda}$ of the remaining scope to x's second inserted child, which will have a scope of size $\frac{(r-l-1)-\frac{r-l-1}{\lambda}}{\lambda} = (r-l-1)(\lambda-1)/\lambda^2$. The third inserted child will use a scope of size $(r-l-1)(\lambda-1)^2/\lambda^3$, and so forth.

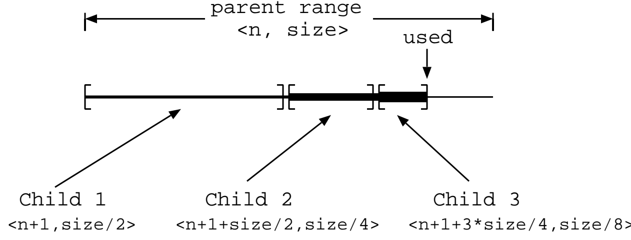


Figure 8: Dynamic Scope Allocation with Parameter $\lambda = 2$

Figure 8 demonstrates an example of dynamic range allocation with parameter $\lambda = 2$. It shows that the kth child is allocated a range that is $1/2^k$ of the parent range in size. As another example, assuming the expected number of sub nodes of node y is 100, then the ranges of those child nodes that are inserted among the first 5 occupy 1%, .99%, .98%, .97%, .96% of the parent range respectively. Apparently, the allocation method has a bias that favors nodes inserted earlier.

More formally, according to the above procedure, for a given node x with a range of $[l, r)$, the size of the subrange assigned to its kth child is $s_k = \frac{(r-l-1)-\sum_{i=1}^{k-1} s_i}{\lambda}$. It is easy to prove that

$$s_k = (r-l-1)(\lambda-1)^{k-1}/\lambda^k \tag{5}$$

In other words, we should assign a subrange $[l_k, r_k) \subset [l, r)$ to the kth child of node x, where:

$$\begin{aligned} l_k &= l+1+(r-l-1)(1-(\lambda-1)^{k-1}/\lambda^{k-1}) \quad (6) \\ r_k &= l_k + s_k \end{aligned}$$

Based on the above discussion, we introduce the following definition of dynamic scope.

Definition 3. *Dynamic Scope of a Suffix Tree Node The dynamic scope of a node is a triple $\langle n, size, k\rangle$, where k is the number of subscopes allocated inside the current scope. Let the dynamic scopes of x and y be $s_x = \langle n_x, size_x, k_x\rangle$ and $s_y = \langle n_y, size_y, k_y\rangle$ respectively. Node y is a descendant of x if $s_y \subset s_x$, i.e., $[n_y, n_y + size_y) \subset [n_x, n_x + size_x)$.*

Scope Underflow

Let $T = t_1, \cdots, t_k$ be a sequence. Each t_i corresponds to a node in the suffix tree. Assume the size of the dynamically allocated scopes decreases on average by a factor of γ every time we descend from a parent node to a child node. As a result, the size of t_i's scope comes to Max/γ^{i-1}, where Max is the size of the root node's scope. Apparently, for a large enough i, $Max/\gamma^{i-1} \to 0$. This problem is called scope underflow.

As we have mentioned, XML databases such as DBLP [15] and IMDB [22] are composed of records of small structures. For databases with large structures, such as XMARK [23], we break down the structure into small sub structures, and create index for each of them. Thus, we limit the average length of the derived sequences.

If scope underflow still occurs for a given sequence $T = t_1, \cdots, t_k$ at t_i, we allocate a subscope of size $k-i+1$ from node t_{i-1}, and label each element $t_i, \ldots, t_k$ sequentially. If node t_{i-1} can not spare a subscope of size $k-i+1$, we allocate a subscope of size $k-i+2$ from node t_{i-2}, and so forth. Intuitively, we borrow scopes from the parent nodes to solve the scope underflow problems for the descendent nodes. In order to do this, we preserve certain amount of scope in each node for this unexpected situation, so that it does not interfere with the dynamic labeling process prescribed by Eq (3)(4)(5)(6). Using this method, the involved nodes are labeled sequentially (each node is allocated a scope for only one child), and they can not be shared with other sequences, but they are still properly indexed for matching.

3.4.2 The Algorithms

In this section, we present the dynamic labeling algorithm and the index construction algorithm of ViST. ViST uses the same sequence matching algorithm as RIST (Algorithm2).

Algorithm 3 outlines the top-down dynamic range allocation method described above. The labeling is based on a virtual suffix tree, which means it is not materialized.

Input: p: parent scope
    e: symbol for which a subscope is to be created
Output: s, a subscope inside the parent scope p
    p, updated parent scope

Assume $p = \langle n, size, k\rangle$;
if *semantical/statistical clues for e is available* **then**
    Assume e is the ith symbol in the follow set of e's parent node;
    $s \leftarrow \langle l_i, s_i, 0\rangle$;   /* l_i and s_i are defined in Eq(4) and Eq(3) respectively */
else
    $s \leftarrow \langle l_k, s_k, 0\rangle$;   /* l_k and s_k are defined in Eq(6) and Eq(5) respectively */
end
$p \leftarrow \langle n, size, k+1\rangle$;
return s;

Algorithm 3: subScope(parent, e): create a sub scope within the parent scope for e

We use an example to demonstrate the process of inserting a structure-encoded sequence into the index structure. Suppose, before the insertion, the index structure already contains the following sequence:

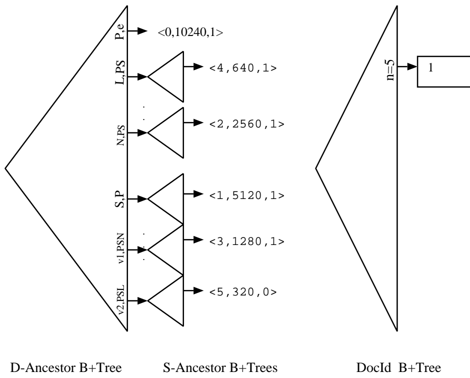
$$Doc_1 = (P, \epsilon)(S, P)(N, PS)(v_1, PSN)(L, PS)(v_2, PSL)$$

The sequence to be inserted is

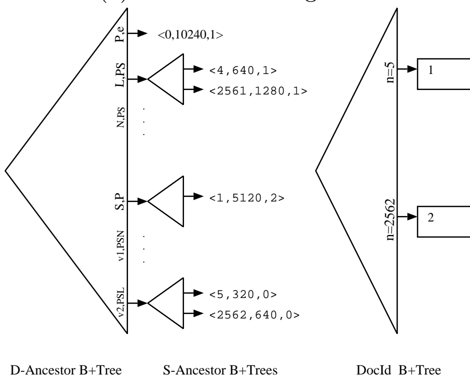
$$Doc_2 = (P, \epsilon)(S, P)(L, PS)(v_2, PSL)$$

The index before the insertion of Doc_2 is shown in Figure 9(a). For presentation simplicity, we make two assumptions: i) $Max = 20480$, that is, the root node covers a scope of $[0, 20480)$; and ii) there is no semantic/statistical clues available and the top-down dynamic scope allocation method uses a fixed parameter $\lambda = 2$ for all nodes.

The insertion process is much like that of inserting a sequence into a suffix tree – we follow the branches, and when there is no branch to follow, we create one. We start with node (P, ϵ), and then (S, P), which has scope $\langle 1, 5120, 1\rangle$. Next, we search in the `S-Ancestor B`$^+$`Tree` of (L, PS) for all entries that are within the scope of $[2, 5120)$. The only entry there, $\langle 4, 640, 1\rangle$, is apparently not an immediate child[5] of $\langle 1, 5120, 1\rangle$. As a result, we insert a new entry $\langle 2561, 1280, 1\rangle$, the 2nd child of (S, P), in the `S-Ancestor B`$^+$`Tree` of (L, PS). The scope for the (S, P) node is updated to $\langle 1, 5120, 2\rangle$ as it has a new child now. Similarly, when we reach (v_2, PSL), we insert a new entry $\langle 2562, 640, 0\rangle$. Finally, we insert key 2562 into the `DocId B`$^+$`Tree` for Doc_2. The resulting index is shown in Figure 9(b).



(a) Index containing Doc_1



(b) Changes caused by the insertion of Doc_2

Figure 9: Index structure before and after insertion

Algorithm 4 details the process of inserting an XML sequence into the index structure.

[5] We can tell the immediate parent-child relationship by Eq (4) and Eq (6).

Input: $\mathcal{T}$: a structure-encoded sequence XML document represented by $\mathcal{T}$ $\quad$ id: ID of the
Output: updated index file F

Assume $\mathcal{T} = (a_1, l_1), \cdots, (a_i, l_i), \cdots, (a_k, l_k)$;
$s \leftarrow \langle 0, Max, k\rangle$; /* s is the scope of the root node of the virtual suffix tree */
$i \leftarrow 1$;
while $i \leq k$ **do**
  Search key (a_i, l_i) in the `D-Ancestor B`$^+$`Tree`;
  if *found* **then**
    $e \leftarrow$ the `S-Ancestor B`$^+$`Tree` associated with (a_i, l_i);
  else
    $e \leftarrow$ new `S-Ancestor B`$^+$`Tree`;
    Insert e into the `D-Ancestor B`$^+$`Tree` with key (a_i, l_i);
  end
  Search in e for scope r such that r is an immediate child scope of s ;
  if *not found* **then**
    $r \leftarrow \langle n, size, k\rangle \leftarrow$ subScope(s, a_i) ;
    Insert $(n, size)$ into `S-Ancestor B`$^+$`Tree` e with n as key;
  end
  $s \leftarrow r$;
  $i \leftarrow i + 1$;
end
Assume $s = \langle n, size, k\rangle$;
Insert (n, id) into the `DocId B`$^+$`Tree`;

Algorithm 4: Index an XML document

4. EXPERIMENTS

We implemented RIST and ViST in C++ for XML indexing. We also implemented a path index method similar to Index Fabric [9], and a node index method similar to XISS [16] for comparison purposes. The implementation uses the B$^+$Tree API provided by the Berkeley DB library [20]. We carry out our experiments on a Linux machine with a 662 MHz Pentium III CPU and 256 MB main memory.

Data Sets

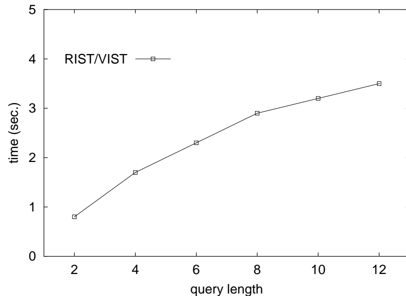
For our experiments, we use public XML databases DBLP [15], the XML benchmark database XMARK [23], and we also generate our own synthetic datasets.

- DBLP. The popular computer science bibliography database is widely used in benchmarking XML index methods. In the version we downloaded, there are 289,627 records, 2,934,188 elements, and 364,348 attributes, totaling 301,318 KBytes of data. Each record of DBLP corresponds to a publication, with a simple tree structure of maximum depth 6. The average length of the structure-encoded sequences derived from the DBLP records is around 31.

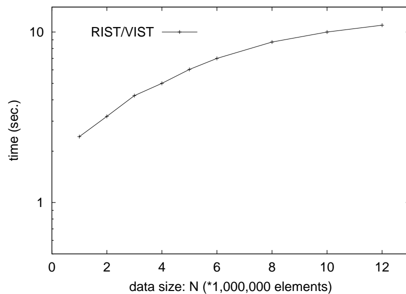
- XMARK. Unlike DBLP, an XMARK dataset is a single record with a very large and complicated tree structure. Since it is not meaningful to represent the entire dataset with a single structure-encoded sequence, we

break down its tree structure into a set of sub structures, including `item` (objects for sale), `person` (buyers and sellers), `open_auction`, `closed_auction`, etc. We convert each instance of these sub structures into a structure-encoded sequence. In our experiments, we use an XMARK dataset generated by `xmlgen` [19] with scaling factor 1.0, totaling 108 MBytes of data.

- SYNTHETIC. We also generate our own synthetic datasets for scalability tests. The data generator is based conceptually on a tree of height k where each node has j sub nodes. We generate a subtree of L nodes. First we select the root node, then we randomly select the next node x from the tree, under the condition that x has not been selected, and x is a child node of a selected node. We repeat this process N times to generate N data sequences of length L. Random queries can be generated in the same way. Since no semantic meaning is associated with this synthetic dataset, we collect statistics during data generation for dynamic labeling purpose.



(a) Synthetic: N=10^6, L=30. Query: varying lengths.



(b) Synthetic: N=?, L=60. Query: length l=6.

Figure 10: Random queries over `Synthetic` datasets

Performance of Query Processing

We first demonstrate the scalability of RIST and ViST with regard to query processing. As both approaches use the same sequence matching algorithm (Algorithm 2) and work on the same index structure[6], they exhibit the same performance in query processing. We generate synthetic datasets with parameters $k = 10$ and $j = 8$. The synthetic dataset used in Figure 10(a) has 1,000,000 sequences, which are of an average length of 30 elements. The query processing time shown in the figure does not include the time spent in data output after each range query on the `DocId B`$^+$`Tree`. Figure 10(a) indicates it takes more time to process longer queries, as longer queries require larger amount of index traversals. The synthetic datasets used in Figure 10(b) are of varying sizes, but each is composed of sequences of the same average length, 60. It shows that our index structure scales up sub-linearly with the increase of data size. We also tried synthetic datasets generated with different values of k and j, and found no significant differences in performance.

We tried various kinds of queries on the DBLP and the XMARK dataset, and compared RIST/ViST with two other index methods. One method, using XML paths as the basic unit of query, is the Index Fabric algorithm [9] (without the extra index for refined paths.) The other method is XISS [16], which uses nodes as the basic query unit.

Table 3 lists 8 queries with ascending complexity. The experimental results of using RIST/ViST, Index Fabric (raw paths), and XISS to answer these queries are summarized in Table 4. Q_1 is a single path query, and there is no attribute values involved. RIST/ViST and Index Fabric have similar response times, while it takes longer for XISS, as it *joins* the results of two 2 sub queries. An attribute value is involved in path query Q_2, and it slows down Index Fabric and XISS since value indexes require special handling in the two approaches. Q_3 and Q_4 use wild-cards, which affects the performance of Index Fabric, unless both are treated specially as *refined paths* (frequently occurring queries with additional index support). $Q_5, \cdots, Q_8$ are branching queries, and from Table 4 we can see that RIST/ViST has the most satisfactory performance. Note that each of the 8 queries is converted to one structure-encoded sequence, and RIST/ViST solves the query with one sequence matching, without using any join operations. Both Index Fabric (raw paths) and XISS have to use (multiple) join operations to answer most of the queries.

	RIST/ViST	raw path index (Index Fabric)	node index (XISS)
Q_1	1.2	0.8	10.1
Q_2	2.3	4.8	54.6
Q_3	1.7	24.8	36.8
Q_4	1.7	23.3	30.2
Q_5	1.6	6.7	19.8
Q_6	3.7	18.0	22.4
Q_7	2.5	37.2	27.6
Q_8	4.1	49.3	48.2

Table 4: Comparing RIST/ViST with path index and node index (time in seconds)

Index Size and Index Construction Time

Finally, we study the space requirement of the index structure used in the RIST/ViST approach and the time it takes to build such indexes. The index structure of ViST is realized by two B$^+$Trees, the `DocId B`$^+$`Tree` and the combined `D-Ancestor` and `S-Ancestor B`$^+$`Trees`. Both B$^+$Trees are implemented using the Berkeley DB library [20]. For a dataset with N sequences, each sequence having L elements on an average, there are a total number of N entries in the `DocId B`$^+$`Tree`. This is because for each document, we make

[6]For any given dataset, the only difference between the index structures constructed by RIST and ViST comes from the handling of scope underflow in dynamic labeling, which is insignificant under most situations.

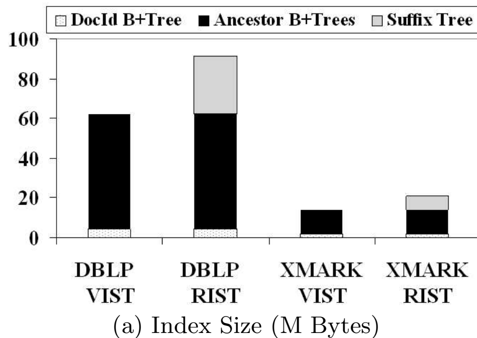
	Path Expression	Dataset
Q_1	/inproceedings/title	DBLP
Q_2	/book/author[text='David']	DBLP
Q_3	/*/author[text='David']	DBLP
Q_4	//author[text='David']	DBLP
Q_5	/book[key='books/bc/MaierW88']/author	DBLP
Q_6	/site//item[location='US']/mail/date[text='12/15/1999']	XMARK
Q_7	/site//person/*/city[text='Pocatello']	XMARK
Q_8	//closed_auction[*[person='person1']]/date[text='12/15/1999']	XMARK

Table 3: Sample queries over `DBLP` and `XMARK` datasets

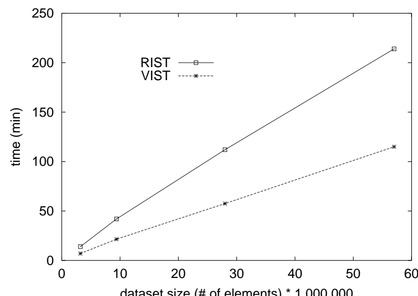
one insertion into the `DocId B`$^+$`Tree` of the following information: its DocId, together with the label of the last virtual suffix tree node it reaches.

In the most unlikely case, the `S-Ancestor` and the `D-Ancestor` `B`$^+$`Trees` will have altogether $N \times L$ elements. This occurs, of course, only if none of the sequences share any nodes in the virtual suffix tree. Thus, the entire space requirement of ViST is $O(N + NL)$. RIST takes more space than ViST, since it maintains a suffix tree, which is of size $O(NL)$ in the worst case.

Figure 11 shows the size of the index structure for DBLP (301 Mbytes of data) and XMARK (for structure `items` only, totaling 52 Mbytes of data). Figure 11(b) shows linear index construction time on synthetic datasets generated with parameters $k = 10$, $j = 8$, and $L = 32$. In both tests, we use disk pages of size 2K for Berkeley DB B$^+$Trees, and we use 8 bytes to label a virtual suffix tree node (i.e., $MAX = 2^{256} - 1$).



(a) Index Size (M Bytes)



(b) Index Construction Time

Figure 11: Index Structure

5. RELATED WORK

One of the most critical tasks in indexing XML documents is to provide efficient support for arbitrary structured queries with branches, wild-cards, etc. Most approaches find it time-consuming to answer such queries, since they rely on expensive join operations to combine results of multiple sub queries based on small graph units, such as paths or elements, that are more easily manageable.

XISS [16] uses single elements/attributes as the basic unit of query. A complex path expression is decomposed into a collection of basic path expressions. Atom expressions (a single element or attribute) are found by directly accessing the index structure. All other forms of expressions involve join operations. One of the merits of the XISS approach is its flexibility, since all kinds of structural queries, including regular expression queries, can be constructed using the most basic building block: the atom expressions.

Paths are also used as the basic query unit. DataGuide [11] provides concise summaries of path structures for a semi-structured database. DataGuide is restricted to raw paths and do not support complex path expression or regular expression queries [18]. The Index Fabric [9] is conceptually similar to the DataGuide in that it indexes all raw paths starting from the root element. In addition, it supports *refined paths*, i.e., a set of query patterns that occur frequently. Such query patterns can contain branches, wild-cards '*' and '//', etc. A tree-structured query not in the set of refined paths, however, has to rely on join operations.

APEX [6] is an adaptive path index for XML data. Unlike the traditional techniques, APEX uses data mining algorithms to summarize paths that appear frequently in the query workload. When the query workload changes, the APEX is incrementally updated. Instead of keeping all paths starting from the root, it maintains every path of length two. Therefore, it also has to rely on join operations to answer path queries with more than two elements.

Similar to Index Fabric [9], F$^+$B Index [14] optimizes a set of branching queries. It is based on the Forward and Backward Index (F&B Index [1]), which covers all branching paths but is often too big to be efficient in query evaluation. F$^+$B Index supports an index definition scheme to restrict the class of branching queries being indexed. It attains significant speedup for pre-defined query types but still has to rely on the F&B index for generic queries.

Besides the ability of answering structured queries with branches and wild-cards, it is also very important whether or not the

index structure contains value indexes in addition to structure indexes, since attribute values in the query often have the major influence on selectivity. Many index methods, including [11, 18, 6, 14], however, do not support value indexes.

Another important criterion in evaluating index methods is whether it relies on special data structures such as the suffix tree that are not well-supported in DBMSs. The XISS approach [16] is based on B^+Trees. Most other approaches, including DataGuide [11], Index Fabric [9], APEX [6], F^+B [14], etc, however, rely on specialized data structures.

Recently, tree labeling has become a focus of study as it is important in answering ancestorship queries. XISS [16], for example, uses a static labeling scheme for this purpose. Several studies [2, 3, 13] focus on the minimum label sizes. Cohen et al [8] introduced a dynamic labeling scheme, which is indispensable for dynamic index structures.

Our approach supports strutural XML queries by converting XML documents into sequeces. The indexing method supports efficient non-contiguous sequece matching. A similar technique is used for weighted-subsequece matching and pattern discovery [21]. We unify structure indexes and value indexes into a single index that relies solely on B^+Trees through a dynamic labeling method.

6. CONCLUSION

We have developed ViST, a dynamic indexing method for XML documents. We convert XML data, as well as structured XML queries to sequences that encode their structural information. Efficient sequence matching algorithms are introduced to find XML documents that contain the structured queries. While state-of-the-art XML indexing methods have difficulty in handling queries containing branches, insofar as most of them first disassemble a structured query into multiple sub-queries each handling a single path in the structured query, and then join the results of the sub-queries to provide the final answers, ViST uses the structures as the basic unit of query, which enables us to process, through sequence matching, structured queries as a whole, and as a result, to avoid expensive join operations. In addition, ViST supports dynamic insertion of XML documents through the top-down scope allocation method. Finally, the index structure of ViST is entirely based on B^+Trees, which, unlike some specialized data structures used in other approaches, are well supported by DBMSs.

7. REFERENCES

[1] 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.

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

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

[4] D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. XQuery: A query language for XML W3C working draft. Technical Report WD-xquery-20010215, World Wide Web Consortium, 2001.

[5] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB*, May 2000.

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

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0 w3c recommendation. Technical Report REC-xpath-19991116, World Wide Web Consortium, 1999.

[8] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *PODS*, pages 271–281, 2002.

[9] Brian F. Cooper, Neal Sample, Michael Franklin, Am (Bsli Hjaltason G , and Moshe Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, September 2001.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the 8th International World Wide Web Conference*, pages 77–91, May 1999.

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

[12] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[13] 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.

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

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

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

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

[18] 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.

[19] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatica, 2001.

[20] Sleepycat Software, http://www.sleepycat.com. *The Berkeley Database (Berkeley DB).*

[21] Haixun Wang, Chang shing Perng, Wei Fan, Sanghyun Park, and Philip S. Yu. Indexing weighted sequences in large databases. In *ICDE*, 2003.

[22] The internet movie database. http://www.imdb.com, 2000.

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