Tree Signatures for XML Querying and Navigation

Pavel Zezula
Masaryk University
Brno, Czech Republic
zezula@fi.muni.cz

Giuseppe Amato
ISTI-CNR
Pisa, Italy
g.amato@iei.pi.cnr.it

Franca Debole
ISTI-CNR
Pisa, Italy
f.debole@iei.pi.cnr.it

Fausto Rabitti
ISTI-CNR
Pisa, Italy
f.rabitti@cnuce.cnr.it

Abstract

In order to accelerate the performance of various matching and navigational operations on collections of XML documents, new indexing structures, called tree signatures, are proposed. We show that XML tree structures can be efficiently represented as ordered sequences of preorder and postorder ranks. Two proposed versions of tree signatures differ in the amount of information they contain, and extensive performance evaluation demonstrates the tradeoff between the space and performance of executing different XPath axes. We also show how to apply tree signatures in query processing and demonstrate that a speedup of up to one order of magnitude can be achieved with respect to the containment join strategy. Other alternatives of using the tree signatures in intelligent XML searching are outlined in the conclusions.

Categories and Subject Descriptors: E.1 [Data Structures]: Records, H.2.2 [Database Management]: Physical Design- Access Methods, H.3.3 [Information Storage and Retrieval]: Information Searching and Retrieval- Search process

Free Keywords: XML searching, Tree signatures, XML query processing

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a labelled-tree data model. A variety of languages have been proposed for this purpose, all of which can be viewed as consisting of a pattern language and construction expressions. Since the data objects are typically trees, tree pattern matching is the central issue.

The idea behind evaluating tree pattern queries, sometimes called the *twig queries*, is to find all the ways of embedding the pattern in the data. Because this lies in the core of most languages for processing XML data, efficient evaluation techniques for these languages require relevant indexing structures.

Contrary to previous approaches, trying to accelerate retrieval through the application of joins [ZND+01, BKS02, CVZ02], we apply the *signature file* approach. In general, signatures are compact (small) representations of important features extracted from actual documents in such a way that query processing can be performed on the signatures instead of the documents. In the past, see e.g. [TZ95] for a survey, such principle has been suggested as an alternative to the *inverted file* indexes. Recently, it has been successfully applied to indexing of multi-dimensional vectors for similarity-based searching [WSB98], image retrieval [NTC02], and data mining [NM02].

We define the *tree signature* as a compact sequence of tree-node entries, containing node names and their structural relationships. Though other possibilities are also discussed, we show how these signatures can be used for efficient tree navigation and twig pattern matching. In Section 2, we explain the motivations for developing tree signatures. The necessary background is surveyed in Section 3. The tree signatures are specified in Section 4. In Section 5, we show the advantages of tree signatures for XPath navigation, and in Section 6 we elaborate on the XML query processing application. Extensive performance evaluation is discussed in Section 7. Conclusions and a discussion on alternative search strategies is in Section 8.

2 Motivation

XML employs a tree-structured model for representing data. Queries in XML query languages typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. For example, the XPath expression:

```
project[title = "XML"]//partner[institution
= "ISTI" and leader = "Fausto"]
```

selects `partner` elements that have child elements `institution` with content "ISTI" and `leader` with content "Fausto". The selected `partner` elements must be descendants of a `project` with a child element `title` containing text "XML". This ex-

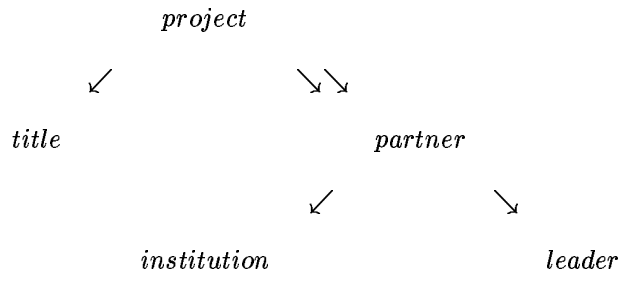


Figure 1: Query twig

pression can be represented as a node-labelled small tree pattern (or a twig) with elements and strings as node labels. See Figure 1 for an illustration of the elements' structure, where the single arrows represent parent-child relationships while the double arrows indicate the more general ancestor-descendant relationships between tree nodes.

Given a query twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) query node predicates are true, and (ii) the structural (ancestor-descendant and preceding-following) relationships between query nodes are satisfied by the corresponding database nodes. In principle, there are two ways to evaluate such queries [MW99]:

- we first find all qualifying predicates in a document and determine their position in the structure – for example by using traditional attribute-based indexes. Then we check to see if their structural relationships are satisfied;
- we first find relevant structural instances of the twig in a document and then check their predicates for qualification.

Naturally, these two approaches can also be combined. For example, consider the query above and assume that we have only an index on the `institution` attribute. If in this document there is only one instance of the `institution` "ISTI", then we have to find the query twig in the document with this `institution` as constant, and if such a twig exists, we have to check all the other predicates for qualification. Such an approach consequently implies a form of tree navigation.

Though the predicate evaluation and the structural control are closely related, in this article, we mainly consider the process of evaluating the structural relationships, because indexing techniques to support efficient evaluation of predicates already exist.

3 Preliminaries

Tree signatures are based on a sequential representation of tree structures. In the following, we briefly survey the necessary background information.

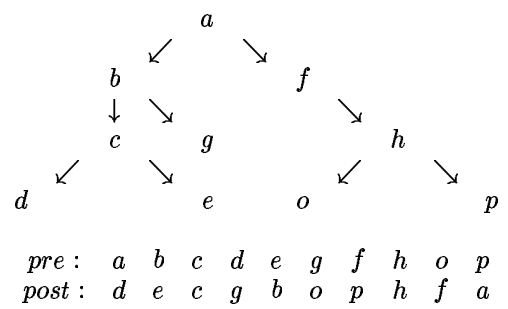


Figure 2: Preorder and postorder sequences of a tree

3.1 Labelled ordered trees

Let Σ be an alphabet of size $|\Sigma|$. Let ϵ, not in Σ, represent the null symbol. An ordered tree T is a rooted tree in which the children of each node are ordered. If a node $i \in T$ has k children then the children are uniquely identified, left to right, as $i_1, i_2, \ldots, i_k$. A labelled tree T associates a label $t[i] \in \Sigma$ with each node $i \in T$. If the path from the root to i has the length n, we say that the node i is on the level n, i.e. $level(i) = n$. Finally, $size(i)$ denotes the size of subtree rooted at i – the size of any leaf node is zero. In the following, we consider ordered labelled trees.

3.2 Preorder and postorder sequences and their properties

Though there are several ways of transforming ordered trees into sequences, we apply the *preorder* and the *postorder* ranks, as recently suggested in [Gr02]. The *preorder* and *postorder* sequences are *ranked* lists of all nodes of a given tree T. In a preorder sequence, a tree node v is traversed and assigned its (increasing) preorder rank, $pre(v)$, before its children are recursively traversed from left to right. In the postorder sequence, a tree node v is traversed and assigned its (increasing) postorder rank, $post(v)$, after its children are recursively traversed from left to right. For illustration, see the preorder and postorder sequences of our sample tree in Figure 2 – the node's position in the sequence is its preorder/postorder rank.

Given a node $v \in T$ with $pre(v)$ and $post(v)$ ranks, the following properties are of importance to our objectives:

- all nodes x with $pre(x) < pre(v)$ are either the *ancestors* of v or nodes *preceding* v in T;
- all nodes x with $pre(x) > pre(v)$ are either the *descendants* of v or nodes *following* v in T;
- all nodes x with $post(x) < post(v)$ are either the *descendants* of v or nodes *preceding* v in T;
- all nodes x with $post(x) > post(v)$ are either the *ancestors* of v or nodes *following* v in T;

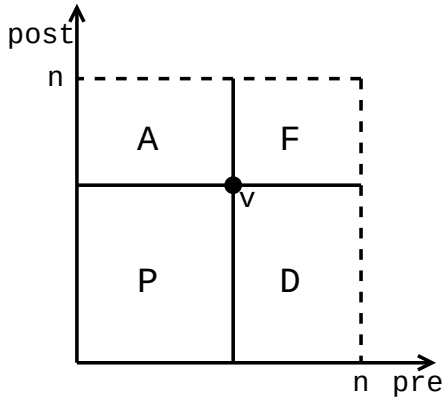


Figure 3: Properties of the preorder and postorder ranks.

- for any $v \in T$, we have $pre(v) - post(v) + size(v) = level(v)$.

As proposed in [Gr02], such properties can be summarized in a two dimensional diagram, as illustrated in Figure 3, where the *ancestors* (A), *descendants* (D), *preceding* (P), and *following* (F) nodes of v are clearly separated in their proper regions.

3.3 Longest common subsequence

The *edit distance* between two strings $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ is the minimum number of the *insert*, *delete*, and *modify* operations on characters needed to transform x into y. A dynamic programming solution is defined by a $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$ that is filled so that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ is the minimum number of operations to transform $x_1, \ldots, x_i$ into $y_1, \ldots, y_j$.

A specialized task of the edit distance is the *longest common subsequence*. In general, a *subsequence* of a string is obtained by taking a string and possibly deleting elements. If $x_1, \ldots, x_n$ is a string and $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ is a strictly increasing sequence of indices, then $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is a subsequence of x. For example, **art** is a subsequence of **a**l**g**o**r**i**t**h**m**. In the *longest common subsequence* (l.c.s.) problem, given strings x and y we want to find the longest string that is a subsequence of both. For example, **art** is the longest common subsequence of **a**l**g**o**r**i**t**h**m** and p**a**r**a**ch**u**t**e**.

By analogy to edit distance, the computation uses an $(n+1) \times (m+1)$ matrix M such that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ contains the length of the l.c.s. between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. The matrix has the following formal definition:

- $M[i,0] = M[0,j] = 0$, otherwise
- $M[i,j] = max\{M[i-1,j]; M[i,j-1];$
  $M[i-1,j-1] + eq(x_i, y_j)\}$,
  where $eq(x_i, y_j) = 1$ if $x_i = y_j$, $eq(x_i, y_j) = 0$ otherwise.

The following is the content of the matrix for the words "algorithm" and "parachute".

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3

Obviously, the matrix can be filled in $O(n \cdot m)$ time. But algorithms such as [HS77] can find l.c.s. much faster.

3.4 The sequence inclusion

A string is *sequence-included* in another string, if their longest common subsequence is equal to the shorter of the strings. Assume strings $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ with $n \leq m$. The string x is sequence-included in the string y if the l.c.s. of x and y is x.

Note that sequence-inclusion and string-inclusion are very different concepts. String x is included in y if characters of x occur contiguously in y, whereas characters of x might be interspersed in y with characters not in x. If string x is string-included in y, it is also sequence-included in y, but not the other way around.

For example, if we search for the l.c.s. of the strings "art" and "parachute", we obtain the matrix

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3

Using the l.c.s. approach, one string is sequence-included in the other if $M[n,m] = n = min\{m,n\}$. But because we do not have to compute all elements of the matrix, the complexity is $O(m) \mid m = max\{m,n\}$.

4 Tree Signatures

The idea of the tree signature is to maintain a small but sufficient representation of the tree structures, able to decide the tree inclusion problem as needed for XML query processing. Intuitively, we use the preorder and postorder ranks to linearize the tree structures and apply the sequence inclusion algorithms for strings.

4.1 The signature

The tree signature is a list of pairs. Each pair contains a tree node name along with the corresponding

postorder rank. The list is ordered according to the preorder rank of nodes.

Definition 4.1 *Let T be an ordered labelled tree. The signature of T is a sequence, $sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots t_n, post(t_n)\rangle$, of $n = |T|$ entries, where t_i is a name of the node with $pre(t_i) = i$. The $post(t_i)$ is the postorder value of the node named t_i and the preorder value i.*

For example, $\langle a, 10; b, 5; c, 3; d, 1; e, 2; g, 4; f, 9; h, 8; o, 6; p, 7\rangle$ is the signature of the tree from Figure 2, and $\langle a, 5; a, 1; a, 4; a, 2; a, 3\rangle$ is the signature of the tree from Figure 5. By analogy, tree signatures can also be constructed for query trees, so $\langle h, 3; o, 1; p, 2;\rangle$ is the signature of the query tree form Figure 4.

4.1.1 Tree inclusion evaluation

Suppose the data tree T specified by signature

$$sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots t_m, post(t_m)\rangle,$$

and the query tree Q defined by its signature

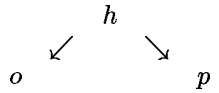
$$sig(Q) = \langle q_1, post(q_1); q_2, post(q_2); \ldots q_n, post(q_n)\rangle.$$

Let $sub\_sig_Q(T)$ be the *sub-signature* (i.e. a subsequence) of $sig(T)$ determined by sequence inclusion (see Section 3.4) of node names of $sig(Q)$ in $sig(T)$. A specific query signature can determine zero or more data sub-signatures. Clearly, if we only consider node names, any $sub\_sig_Q(T) \equiv siq(Q)$, because $q_i = t_i$ for all i. However, the entries of $sub\_sig_Q(T)$ can have postorder values different than corresponding entries in $sig(Q)$. It is also important to understand that the sequence positions of entries in sub-signatures need not correspond to sequence positions (i.e. the preorder values) of the corresponding entries in $sig(T)$.

Lemma 4.1 *The query tree Q is included in the data tree T if the following two conditions are satisfied: (1) on the level of node names, $sig(Q)$ is sequence-included in $sig(T)$ determining $sub\_sig_Q(T)$, (2) for all pairs of entries i and $i+1$ in $sig(Q)$ and $sub\_sig_Q(T)$, $i = 1, 2, \ldots |Q| - 1$, $post(q_{i+1}) > post(q_i)$ implies $post(t_{i+1}) > post(t_i)$.*

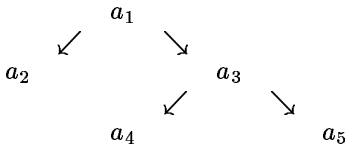
Proof 4.1 *Because the index i increases according to the preorder sequence, node $i+1$ must be either the descendant or the following node of i. If $post(q_{i+1}) < post(q_i)$, the node $i+1$ in the query is a descendant of the node i, thus also $post(t_{i+1}) < post(t_i)$ is required. By analogy, if $post(q_{i+1}) > post(q_i)$, the node $i+1$ in the query is a following node of i, thus also $post(t_{i+1}) > post(t_i)$ must hold.*

For example, consider the data tree T in Figure 2 and the query tree Q in Figure 4. Such query



$$sig(Q) = \langle h, 3; o, 1; p, 2\rangle$$

Figure 4: Sample query tree Q



$$sig(T) = \langle a_1, 5; a_2, 1; a_3, 4; a_4, 2; a_5, 3\rangle$$

Figure 5: Sample tree T with redundant names

qualifies in T, because $sig(Q) = \langle h, 3; o, 1; p, 2\rangle$ determines $sub\_sig_Q(T) = \langle h, 8; o, 6; p, 7\rangle$ that is identical to the query signature on the level of names. Besides, the trends in postorder directions between neighboring entries of $sig(Q)$ and $sub\_sig_Q(T)$ are the same. If we change in our query h for f, we get $sig(Q) = \langle f, 3; o, 1; p, 2\rangle$. Such a modified query tree is also included in T, because Lemma 4.1 does not insist in the strict parent-child relationships, and implicitly considers all such relationships as ancestor-descendant. However, the query tree with the root g, resulting in $sig(Q) = \langle g, 3; o, 1; p, 2\rangle$, does not qualify, even though it is also sequence-included (on the level of names) as the sub-signature $sub\_sig_Q(T) = \langle g, 4; o, 6; p, 7\rangle$. The reason is that the query requires the postorder to go down from g to o (from 3 to 1) , while in the sub-signature it actually goes up (from 4 to 6). That means that o is not a descendant node of g, as required by the query, but a node that follows g, which can be verified in Figure 2.

Multiple nodes with common names may result in multiple tree inclusions. In order to make our approach work correctly also for redundant names, we have to distinguish between the node names, which can be redundant, and their unique occurrences in trees. For motivation, see the tree in Figure 5, where all 5 nodes are labelled with the same name a. To distinguish between individual node occurrences of the tree, we use numeric subscripts. Note that the indices are only important and valid locally within a specific tree to distinguish among individual (redundant) node name instances. Whenever the tree sequence representations are compared regarding the names, indices are not considered, so on the name level, a_1 is the same label as, for example, a_5.

Consider the tree from Figure 5, which is defined by signature $sig(T) = \langle a_1, 5; a_2, 1; a_3, 4; a_4, 2; a_5, 3\rangle$. As-

$sig(Q)$	$a_1, 4$	$a_2, 1$	$a_3, 2$	$a_4, 3$
$sub\_sig^1_Q(T)$	$a_1, 5$	$a_2, 1$	$a_3, 4$	$a_4, \mathbf{2}$
$sub\_sig^2_Q(T)$	$a_1, 5$	$a_2, 1$	$a_3, 4$	$a_5, \mathbf{3}$
$sub\_sig^3_Q(T)$	$a_1, 5$	$a_2, 1$	$a_4, 2$	$a_5, 3$
$sub\_sig^4_Q(T)$	$a_1, 5$	$a_3, 4$	$a_4, \mathbf{2}$	$a_5, 3$
$sub\_sig^5_Q(T)$	$a_2, 1$	$a_3, \mathbf{4}$	$a_4, 2$	$a_5, 3$

Table 1: Sub-signature inclusions

sume a query tree consisting of 4 nodes identically labelled a from which one is the root and the others are independent descendants of the root. This tree results in signature $sig(Q) = \langle a_1, 4; a_2, 1; a_3, 2; a_4, 3\rangle$. The sequence $sig(Q)$ is, on the level of names, sequence included in $sig(T)$ 5 times. Table 1 reports a summary of the query tree signature and the corresponding 5 sub-signatures of the data tree. The bold values of postorder indicate discrepancies between the required (sequence) trends of these values defined by the query signature with respect to the proper sub-signature. Accordingly, only the 3rd sub-signature qualifies, because whenever the postorder in $sig(Q)$ goes up (down), the postorder of $sub\_sig^3_Q(T)$ moves in the same direction. This means that the query tree is included in the data tree from Figure 5 as the pattern consisting of nodes a_1, a_2, a_4, a_5.

If we change the implicit relationships between the query root and the leaves from the ancestor-descendant to the parent-child relationships, we have to check if a_1 is actually the parent of a_2, a_4 and a_5. In general, given a node with preorder position i, defined by the i-th entry of the corresponding signature, the parent node of $i = j \mid post(j) = min\{post(j) \mid j < i \wedge post(j) > post(i)\}$, which is a natural application of the basic properties of the preorder and postorder ranks. If we apply the parent definition on $sig(T) = \langle a_1, 5; a_2, 1; a_3, 4; a_4, 2; a_5, 3\rangle$, we see that a_2 has a_1 as the parent, because $post(a_1) > post(a_2)$ and a_1 is the only node before a_2. But the node before a_4 with the minimum postorder greater than $post(a_4)$ is a_3, so the node a_3 rather than a_1 is the parent of a_4. The same applies for a_5 with $post(a_5) = 3$. That means that for such modified query, the data tree does not qualify.

4.1.2 Extended signatures

In order to further increase the efficiency of various matching and navigation operations, we also propose the *extended signatures*. For motivation, see the sketch of a signature in Figure 6, where A, P, D, F represent areas of ancestor, preceding, descendant, and following nodes with respect to the generic node v. Observe that all descendants are on the right of v before the following nodes of v. At the same time, all ancestors are on the left of v, acting as separators of subsets of

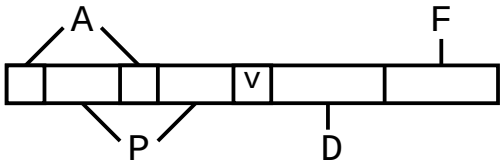


Figure 6: Signature structure

preceding nodes. This suggests extending entries of tree signatures by two preorder numbers representing pointers to the *first following*, ff, and the *first ancestor*, fa, nodes. The general structure of the extended signature of tree T is

$$sig(T) = \langle t_1, post(t_1), ff_1, fa_1; t_2, post(t_2), ff_2, fa_2; \ldots$$

$$t_n, post(t_n), ff_n, fa_n\rangle,$$

where ff_i (fa_i) is the preorder value of the first following (ancestor) node of that with the preorder rank i. If no terminal node exists, the value of the first ancestor is zero and the value of the first following node is $n+1$. For illustration, the extended signature of the tree from Figure 2 has the form

$$sig(T) = \langle a, 10, 11, 0; b, 5, 7, 1; c, 3, 6, 2; d, 1, 5, 3; e, 2, 6, 3;$$

$$g, 4, 7, 2; f, 9, 11, 1; h, 8, 11, 7; o, 6, 10, 8; p, 7, 11, 8\rangle$$

Given a node with index i, the cardinality of the descendant node set can now be computed as

$$size(i) = ff_i - i - 1,$$

and the level of the node with index i is

$$level(i) = i - post(i) + ff_i - i - 1 = ff_i - post(i) - 1$$

Observe that the level of a node can be determined by using only the node's tree signature data.

4.2 Partial signatures

Traditional data value indexes typically process information specific to individual tree nodes, e.g. titles or institutions, so the following problem is relevant:

> Given a query tree Q and a set the signature entries for only some nodes of T, can T qualify for Q or not?

The answer depends on how much and what kind of information about T is available. If one of the node names of the partial data tree specification is known and if it matches a name of Q, the evaluation is MAYBE, and additional information is needed. If there are $2 < |Q|$ matching names, we can never be sure about the positive qualification, but we may decide that the data tree does not qualify for Q. Specifically, if the relative positions, determined by the preorder

and postorder numbers, of the nodes in Q do not comply with the positions in the data tree, the answer is NO, because we are sure about the not possible qualification. Such an approach can easily be extended for more than two node names.

Another way to increase the number of NO cases is to take into account levels on which nodes appear in the data tree. For example, we may know that a specific node must be on a certain level, or that a difference between levels of two nodes is equal to a certain value. As we have already demonstrated, levels are easy to compute from isolated signature entries by means of the extended signatures, while with the short signatures we can only compute the levels of leaf nodes as $level(i) = i - post(i)$.

5 Evaluation of XPath expressions

XPath [WWW99] is a language for specifying navigation within an XML document. The result of evaluating an XPath expression on a given XML document is a set of nodes stored according to document order, so we can say that the result nodes are selected by an XPath expression. An XPath expression has the following syntax:

$$\texttt{Path} ::= /\texttt{Step}_1/\texttt{Step}_2/\ldots/\texttt{Step}_N$$

where each XPath `Step` is defined as follows:

$$\texttt{Step} ::= \texttt{Axis} :: \texttt{Node} - \texttt{test}(\texttt{Predicate})^*$$

An XPath expression is evaluated sequentially, "step" by "step", and an XPath `Step` is applied to a single node, called the *context* node, and selects a set of result nodes. Each node of the result node set is then used as the context node to evaluate the following `Step`. The initial context node is the root of the input document. The result of evaluating an XPath expression is the union of node sets selected by the last `Step`. In principle, we can say that the evaluation of an XPath expression is a recursive top-down process.

Within an XPath `Step`, an `Axis` specifies the *direction* in which the document should be explored. Given a context node v, XPath supports 12 axes for navigation. Assuming the context node is at position i in the signature, we describe how the most significant axes can be evaluated in the extended signatures, using the tree from Figure 2 as reference:

Child The first child is the first descendant, that is a node with index $i + 1$ such that $post(i) > post(i + 1)$. The second child is indicated by pointer ff_{i+1}, provided the value is smaller than ff_i, otherwise the child node does not exist. All the other children nodes are determined recursively until the bound ff_i is reached. For example, consider the node b with index $i = 2$. Since $ff_2 = 7$, there are 4 descending nodes, so the node with index $i+1 = 3$ (i.e. node c) must be the first child. The first following pointer of c, $ff_{i+1} = 6$, determines the second child b (i.e. node g), because $6 < 7$. Due to the fact that $ff_6 = ff_i = 7$, there are no other child nodes.

Descendant The descendant nodes (if any) start immediately after the reference object, that is at position $i+1$, and the last descendant object is at position $ff_i - 1$. If we consider node c (with $i = 3$), we immediately decide that the descendants are at positions starting from $i+1 = 4$ to $ff_3 - 1 = 5$, i.e. nodes d and e.

Parent The parent node is directly given by the pointer fa. The **Ancestor** axis is just a recursive closure of **Parent**.

Following The following nodes of the reference at position i (if they exist) start at position ff_i and include all nodes up to the end of the signature sequence. All nodes following c (with $i = 3$) are in the suffix of the signature starting at position $ff_3 = 6$.

Preceding All preceding nodes are on the left of the reference node as a set of intervals separated by the ancestors. Given a node with index i, fa_i points to the first ancestor (i.e. the parent) of i, and the nodes (if they exist) between i and fa_i precede i in the tree. If we recursively continue from fa_i, we find all the preceding nodes of i. For example, consider the node g with $i = 6$: following the ancestor pointer, we get $fa_6 = 2, fa_2 = 1, fa_1 = 0$, so the ancestors nodes are b and a, because $fa_1 = 0$ indicates the root. The preceding nodes of g are only in the interval from $i - 1 = 5$ to $fa_6 + 1 = 3$, i.e. nodes c, d, and e.

Following-sibling In order to get the following siblings, we just follow the ff pointers while the following objects exist and the fa pointers are the same as fa_i. For example, given the node c with $i = 3$ and $fa_3 = 2$, the ff_3 pointer moves us to the node with index 6, that is the node g. The node g is the sibling following c, because $fa_6 = fa_3 = 2$. But this is also the last following sibling, because $ff_6 = 7$ and $fa_7 \neq fa_3$.

Preceding-sibling All preceding siblings must be between the context node with index i and its parent with index $fa_i < i$. The first node after the i-th parent, which has the index $fa_i + 1$, is the first sibling. Then use the **Following-sibling** strategy up to the sibling with index i. Consider the node f ($i = 7$) as the context node. The first sibling of the i-th parent is b, determined by pointer $fa_7 + 1 = 2$. Then the pointer $ff_2 = 7$

leads us back to the context node f, so b is the only preceding sibling node of f.

`Node-test` specifies a simple test on the XML nodes found along the step's axis. The most commonly used `Node-test` examines node names. For example, the step `child::leader` would select child nodes that have the name "leader". Since tree signatures also contain node names, node tests can easily be implemented. Naturally, the often used `Node-test "*"`, which evaluates to true for all element nodes, is implicitly determined by the tree signatures.

An XPath step can also include a sequence of predicates. The predicates are applied to the node set selected by the step. Only nodes for which all predicates evaluate to true, are returned. Since tree signatures do not contain the actual data values, we limit our discussion to predicates of the form `[position() = n]`. Such a predicate selects all nodes whose position (index) within the context node set equals to n. For example, `child::leader[position()=2]` selects the second child named "leader" whereas `descendant::leader[position()=2]` selects second such a descendant. Since signatures are representations of ordered trees, the implementation of predicates on position is straightforward.

6 Query Processing

Processing a query Q on a collection of XML documents represents a process of finding sub-trees for which content predicates and structural relationships, defined by the query, are satisfied. Query execution strategies determine the ways the query's predicates are evaluated. In principle, a predicate can be decided either by accessing a specific part of the document or by means of an index. So a specific strategy depends on the availability of indexes. We assume that tree signatures are used to support verification of required structural relationships.

A query processor can also exploit tree signatures to evaluate *set-oriented* primitives similar to the XPath axes. For instance, given a set of elements R, the evaluation of $Parent(R, \texttt{article})$ gives back the set of elements named `article`, which are parents of elements contained in R. We suppose that elements are identified by their preorder values, so sets of elements are in fact sets of element identifiers.

Verifying structural relationships can easily be integrated with evaluating content predicates. If indexes are available, a preferable strategy is to use these indexes to obtain sets of elements satisfying the predicates first, and then verify the structural relationships using signatures. Consider the following XQuery [XQ02] query:

```
for $a in //people
where
      $a/name/first="John" and
      $a/name/last="Smith"
return
      $a/address
```

Suppose that content indexes are available on the `first` and `last` elements. A possible efficient execution plan for this query is:

1. let $R_1 = ContentIndexSearch(last - idx, \texttt{Smith})$;
2. let $R_2 = ContentIndexSearch(first - idx, \texttt{John})$;
3. let $R_3 = Parent(R_1, \texttt{name})$;
4. let $R_4 = Parent(R_2, \texttt{name})$;
5. let $R_5 = Intersect(R_3, R_4)$;
6. let $R_6 = Parent(R_5, \texttt{people})$;
7. let $R_7 = Child(R_6, \texttt{address})$;

First, the content indexes are used to obtain R_1 and R_2, i.e. the sets of elements that satisfy the content predicates. Then, tree signatures are used to navigate through the structure and verify structural relationships.

Now suppose that a content index is only available on the `last` element, the predicate on the `first` element has to be processed by accessing the content of XML documents. Though the specific technique for efficiently accessing the content depends on the storage format of the XML documents (plain text files, relational transformation, etc.), a viable query execution plan is the following:

1. let $R_1 = ContentIndexSearch(last - idx, \texttt{Smith})$;
2. let $R_2 = Parent(R_1, \texttt{name})$;
3. let $R_3 = Child(R_2, \texttt{first})$;
4. let $R_4 = FilterContent(R_3, \texttt{John})$;
5. let $R_5 = Parent(R_4, \texttt{name})$;
6. let $R_6 = Parent(R_5, \texttt{people})$;
7. let $R_7 = Child(R_6, \texttt{address})$.

Here, the content index is first used to find R_1, i.e. the set of elements containing `Smith`. The tree signature is used to produce R_3, that is the set of the corresponding `first` elements. Then, these elements are accessed to verify that their content is `John`. Finally, tree signatures are used again to verify the remaining structural relationships.

7 Experimental evaluation

We have experimentally evaluated the tradeoff between the space and the efficiency of executing different XPath axes by means of short and extended signatures (Section 7.2). We have also conducted experiments to demonstrate the efficiency of the extended signatures for query processing (Section 7.3).

7.1 Implementation

The length of a signature $sig(T)$ is proportional to the number of the tree nodes $|T|$, and the actual length de-

pends on the size of individual signature entries. The postorder (preorder) values in each signature entry are numbers, and in many cases even two bytes suffice to store such values. In general, the tag names are of variable size, which can cause some problems when implementing the tree inclusion algorithms. But also the domain of tag names is usually a closed domain of known or upper-bounded cardinality. In such case, we can use a dictionary of the tag names and transform each of the names to its numeric representation (a code) of fixed length. For example, if the number of tag names and the number of tree nodes are never greater than $65,536$, both entities of a signature entry can be represented on 2 bytes, so the length of the signature $sig(T)$ is $4 \cdot |T|$ for the short version, and $8 \cdot |T|$ for the extended version. With a stack of maximum size equal to the tree hight, signatures can be generated in linear time.

In our implementation, the signature of an XML file was maintained in a corresponding signature file consisting of a list of records. Each record contained two (for the short signature) or four (for the extended signature) integers, each represented by four bytes. Accessing signature records was implemented by a seek in the signature file and by reading in a buffer the corresponding two or four integers (i.e. 8 or 16 bytes) with a single read. No explicit buffering or paging techniques were implemented to optimize access to the signature file. Everything was implemented in Java, JDK 1.4.0 and run on a PC with a 1800 GHz Intel pentium 4, 512 Mb main memory, EIDE disk, running Windows 2000 Professional edition with NT file system (NTFS).

7.2 Performance of XPath navigation steps

To study the tradeoff between the space and the efficiency of executing different axes by means of the short and extended signatures, we conducted the following experiments. We generated three signatures, representing one small (2,000 nodes) and two kinds of large (20,000 nodes) trees – one large high tree with a small branching factor, and one large low tree with a high branching factor. By randomly choosing reference nodes in sufficient quantity, we have measured the execution time of the individual axis needed by algorithms operating on the short and extended signatures. For reference, we also implemented the navigation algorithms on a DOM structure, obtained by the Xerces parser [Xer]. Results of the experiments are summarized in Table 2, where S-E represents the average speedup of the extended signature with respect to the short signature, and D-E is the average speedup of the extended signature with respect to the DOM implementation.

The experiments confirm that implementations on extended signatures are always faster than on the short signatures, and the larger the signature, the better. The biggest advantage can be observed in the ancestor axis, mainly when processing large low trees. For all the other axes, the performance improvements are less significant and actual values depend on the shape of the tree and the position of the reference.

axis	small		large-high		large-low	
	S-E	D-E	S-E	D-E	S-E	D-E
child	1.43	0.81	2.43	0.9	1.6	0.91
descendant	1.09	3.40	1.14	2.82	1.18	4.39
ancestor	14.1	0.71	178	0.55	662	0.84
following	1.11	1.62	1.43	3.16	1.45	3.3
preceding	1.08	2.59	1.4	4.15	1.42	4.14
foll-sibling	1.44	0.63	2.62	0.83	1.46	0.50
pre-sibling	1.48	0.75	1.79	1.03	1.34	0.5

Table 2: Performance comparison

As expected, the DOM structure is very suitable for navigation. Except when processing the descendant, following, and preceding axes, the DOM structure was slightly better than the extended signatures. Observe that the performance of the DOM algorithms was maximally 2 times better (speedup 0.5), compared to the more than 4 times better performance (speedup 4.39) of the extended signatures in some other situations. However, the space requirements of the DOM only structural part was 3 times the extended signature size and 6 times more of the short signature. In real applications, the DOM structure also contains the data, which can result in very large data objects, unable to be processed in internal memory.

7.3 Query evaluation

We compared the extended signatures with the *containment join* according to the implementation from [ZND+01]. An *Element Index* was used to associate each element of XML documents with its start and end positions – start and end positions are, respectively, the positions of the start and the end tags of elements in XML documents. This information is maintained in an inverted index, where each element name is mapped to the list of its occurrences in each XML file. The inverted index was implemented by using the BerkeleyDB. Specifically, we used B^+-trees with multiple keys. Retrieval of the inverted list associated with a key (the element name) was implemented with the bulk retrieval functionality, provided by the BerkeleyDB. The containment join was implemented as the Multi Predicate MerGe JoiN (MPMGJN), as described in [ZND+01].

We have compared signatures and containment joins using XQuery queries of the following template:

```
for $a in //<e_name>
where <pred($a)>
return
      <result>
```

element name	# elements
phdthesis	71
book	827
inproceedings	198960
author	679696
year	313531
title	313559
pages	304044

Table 3: Selectivity of element names

predicate	# elements
$a/author="Michael J. Franklin"	73
$a/year="1980"	2595
$a/year="1997"	21492

Table 4: Selectivity of predicates

```
        $a/<e_1>
        ...
        $a/<e_n>
</result>
```

In this way, we are able to generate queries that have different *element name selectivity* (i.e. the number of elements having a given element name), *element content selectivity* (i.e. the number of elements having a given content), and the number of navigation steps to follow in the pattern tree (twig), corresponding to the query. Specifically, by varying the element name <e_name> we can control the element name selectivity, by varying the predicate <pred($a)> we can control the content selectivity, and by varying the number n of the expression in the return clause, we can control the number of navigation steps.

We run our experiments by using the XML DBLP data set containing 3,181,399 elements and occupying 120 Mb of memory. We chose three degrees of the element name selectivity by setting <e_name> to phdthesis for high selectivity, to book for medium selectivity, and to inproceedings for low selectivity. The degree of content selectivity was controlled by setting the predicate <pred($a)> to $a/author="Michael J. Franklin" for high selectivity, $a/year="1980" for medium selectivity, and $a/year="1997" for low selectivity. In the return clause, we have used title as <e_1> and pages as <e_2>. Table 3 shows the number of occurrences of the element names that we used in our experiments, while Table 4 shows the number of elements satisfying the predicates used.

Each query generated from the previously described query template is coded as "QNCn", where N and C indicate, respectively, the element name and the content selectivity, and can be H(igh), M(edium), or L(ow). The parameter n can be 1 or 2 to indicate the number of steps in the return clause.

The query execution plan to process our queries with the signatures is the following:

1. let $R_1 = ContentIndexSearch($<pred>$)$;
2. let $R_2 = Parent(R_1,$<e_name>$)$;
3. let $R_3 = Child(R_2,$<e_1>$)$;
4. let $R_4 = Child(R_2,$<e_2>$)$.

We process content predicates by using a content index. The remaining processing steps are executed using the support for the XPath axes offered by the extended signatures.

The query execution plan to process the queries through containment joins is the following:

1. let $R_1 = ContentIndexSearch($<pred>$)$;
2. let $R_2 = ElementIndexSearch($<e_name>$)$;
3. let $R_3 = ContainingParent(R_2, R_1)$;
4. let $R_4 = ElementIndexSearch($<e_1>$)$;
5. let $R_5 = ContainedChild(R_4, R_3)$;
6. let $R_6 = ElementIndexSearch($<e_2>$)$;
7. let $R_7 = ContainedChild(R_6, R_3)$.

By analogy, we first process the content predicate by using a content index. Containment joins are used to check containment relationships: first the list of occurrences of necessary elements is retrieved by using an element index (*ElementIndexSearch*); then, structural relationships are verified by using the containment join (*ContainingParent* and *ContainedChild*).

For queries with $n = 1$, step 4, for the signature based query plan, and steps 6 and 7, for the containment join based query plan, do not apply.

7.3.1 Analysis

Results of performance comparison are summarized in Table 5, where the processing time in milliseconds and the number of elements retrieved by each query are reported. As intuition suggests, performance of extended tree signatures is better when the selectivity is high. In such case, improvements of one order of magnitude are obtained.

The containment join strategy seems to be affected by the selectivity of the element name more than by the tree signature approach. In fact, using high content selective predicates, performance of signature files is always high, independently of the element name selectivity. This can be explained by the fact that, using the signature technique, only these signature records corresponding to elements that have parent relationships with the few elements satisfying the predicate are accessed. On the other hand, the containment join strategy has to process a large list of elements associated with the low selective element names.

In case of low selectivity of the content predicate, we have a better response than containment join with the exception of the case where low selectivity of both content and names of elements are tested. In this case, structural relationships are verified for a large number

Query	Ext. sign	Cont. join	#Retr. el
QHH1	80	466	1
QHM1	320	738	1
QHL1	538	742	1
QMH1	88	724	1
QMM1	334	832	9
QML1	550	882	60
QLH1	95	740	38
QLM1	410	1421	1065
QLL1	1389	1282	13805
QHH2	90	763	1
QHM2	352	942	1
QHL2	582	966	1
QMH2	130	822	1
QMM2	376	1327	9
QML2	602	1220	60
QLH2	142	1159	38
QLM2	450	1664	1065
QLL2	2041	1589	13805

Table 5: Performance comparison between extended signatures and containment join. Processing time is expressed in milliseconds.

of elements satisfying the low selective predicate. We believe that such queries are not frequent in practice.

The difference in performance of the signature and the containment join approaches is even more evident for queries with two steps. While the signature strategy has to follow only one additional step for each qualifying element, that is to access one more record in the signature, containment joins have to merge potentially large lists.

8 Concluding remarks

Inspired by the success of signature files in several application areas, we propose tree signatures as an auxiliary data structure for XML databases. The proposed signatures are based on the preorder and postorder ranks and support tree inclusion evaluation, respecting the sibling and ancestor-descendant relationships. Extended signatures are not only faster than the short signatures, but can also compute node levels and sizes of subtrees from only the partial information pertinent to specific nodes. Navigation operations, such as those required by the XPath axes, are computed very efficiently – extended signatures typically outperform even the much more space demanding DOM structure. We demonstrate that query processing can also benefit from the application of the tree signature indexes. For highly selective queries, i.e. typical user queries, query processing with the tree signature is about 10 times more efficient, compared to the strategy with containment joins.

The proposed signature file approach also creates good bases for dealing with dynamic XML collections. Even though the preorder and postorder numbering scheme is affected by document updates – node ranks change when inserting or deleting a node – the effects are always *local* within specific signatures. So it is up to the database designer to choose a suitable signature granularity, which should be rather small for very dynamic collections, while relatively stable or static collections can use much larger signatures. This locality property cannot be easily exploited with approaches based on containment join or approaches like [Gr02], where updates (as well as insertions and deletions) usually require extensive reorganization of the index.

In this paper, we have discussed the tree signatures from the traditional XML query processing perspective, that is for navigating within the tree structured documents and retrieving document trees containing user defined query twigs. However the tree signatures can also be used for solving queries such as:

> Given a set of tree node names, what is the most frequent structural arrangement of these nodes.

or, alternatively

> What set of nodes is most frequently arranged in a given hierarchical structure.

Another alternative is to search through tree signatures by using a query sample tree as a paradigm with the objective to rank the data signatures with respect to the query according to a convenient proximity (similarity or distance) measure. Such an approach results in the implementation of the *similarity range* queries, the *nearest neighbor* queries, or the *similarity joins*.

In general, ranking of search results [TW00, TW02] is a big challenge for XML searching. Due to the extensive literature on string processing, see e.g. [Gu97], the string form of tree signatures offers a lot of flexibility in obtaining different and more sophisticated forms of comparing and searching. We are planning to investigate these alternatives in the near future.

References

[BKS02] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 310–321, Madison, Wisconsin, USA, June 2002.

[CVZ02] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. Proceedings of the *28rd VLDB Conference*, Hong Kong, China. 2002, pp. 263-274.

[Gu97] D. Gusfield. *Algorithms on Strings, trees, and Sequences.* Cambridge University Press, 1997.

[Gr02] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002, pp. 109-120.

[HS77] J.W. Hunt and T.G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Comm. ACM*, 20:350-53, 1977.

[MW99] Jason McHugh and Jennifer Widom Query optimization for xml, *Proceedings of 25th VLDB Conference*, September 7-10, 1999, Edinburgh, Scotland, UK, Morgan Kaufmann, 1999, pp. 315-326.

[NM02] A. Nanopoulos, Y. Manolopoulos. Efficient similarity search for market basket data. *VLDB Journal*, Springer, Vol. 11. No. 2., 2002, pp. 138-152.

[NTC02] M.A. Nascimento, E. Tousidou, V. Chitkara, and Y. Manolopoulos. Image indexing and retrieval using signature trees. *Data and Knowledge Engeneering*, Elsevier, Vol. 43, No. 1, 2002, pp. 57-77

[TW00] T. Theobald and G. Weikum. Adding Relevance to XML. *3-rd International Workshop on the Web and Databases*, Dallas, Texas, 2000, LNCS 1997, Springer, pp. 105-124.

[TW02] T. Theobald and G. Weikum. Search Engine for Querying XML Data with Relevance Ranking. *Proceedings of EDBT 2002*, Prague, 2002, LNCS 2287, Springer, pp. 477-495.

[TZ95] P. Tiberio and P. Zezula. Storage and Retrieval: Signature file access. *Encyclopedia of Microcomputers*, Vol. 16, edited by A. Kent and J.G. Williams. Marcel Dekker, Inc., New York, 1995, pp, 377-403.

[WWW99] World Wide Web Consortium. XML Path Language (XPath), Version 1.0, W3C Recommendation, November 1999.

[XQ02] World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft, Nov. 2002. http://www.w3.org/TR/xquery.

[WSB98] R. Weber, H.J. Schek, S. Blott A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. Proceedings of the *24rd VLDB Conference*, New York City, USA. 1998, pp. 194-205.

[Xer] Xerces Java Parser, http://xml.apache.org/xerces-j/

[ZS97] K. Zhang and D. Shasha. Tree Pattern Matching. *Pattern Matching Algorithms*, Apostolico and Galil, Editors, Oxford University Press, 1997.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. *Proceedings of ACM SIGMOD Conference 2001*: Santa Barbara, CA, USA, ACM, 2001.