Tree Signatures for XML Querying and Navigation

Pavel Zezula[1], Giuseppe Amato[2], Franca Debole[2], and Fausto Rabitti[2]

[1] Masaryk University, Brno, Czech Republic,
zezula@fi.muni.cz
http://www.fi.muni.cz

[2] ISTI-CNR, Pisa, Italy,
{Giuseppe.Amato,Franca.Debole,Fausto.Rabitti}@isti.cnr.it
http://www.isti.cnr.it

Abstract. In order to accelerate execution of various matching and navigation operations on collections of XML documents, new indexing structure, based on tree signatures, is proposed. We show that XML tree structures can be efficiently represented as ordered sequences of preorder and postorder ranks, on which extended string matching techniques can easily solve the tree matching problem. 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 over the containment join strategy. Other alternatives of using the tree signatures in intelligent XML searching are outlined in the conclusions.

1 Introduction

With the rapidly increasing popularity of XML, 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, most of them offering various features of a pattern language and construction expressions. Since the data objects are typically trees, the tree pattern matching and navigation are the central issues of the query execution.

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

Available approaches to the construction of structural indexes for XML query processing are either based on mapping pathnames to their occurrences or on mapping element names to their occurrences. In the first case, entire pathnames occurring in XML documents are associated with sets of element instances that can be reached through these paths. However, query specifications can be more complex than simple path expressions. In fact, general queries are represented as pattern trees, rather than paths. Besides, individual path specifications are typically *vague* (containing for example *wildcards*), which complicates the matching. In the second case, element names are associated with *structured references* to the occurrences of names in XML documents. In this way, the indexed information is *scattered*, giving more freedom to ignore unimportant relationships. However, a document structure reconstruction requires expensive merging of lengthy reference lists through *containment joins*.

Contrary to the approaches that accelerate retrieval through the application of joins [10,1,2], we apply the *signature file* approach. In general, signatures are compact (small) representations of important features extracted from actual documents, created with the objective to execute queries on the signatures instead of the documents. In the past, see e.g. [9] 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, image retrieval, and data mining.

We define the *tree signature* as a sequence of tree-node entries, containing node names and their structural relationships. In this way, incomplete tree inclusions can be quickly evaluated through extended string matching algorithms. We also show how the signature can efficiently support navigation operations on trees. Finally, we apply the tree signature approach to a complex query processing and experimentally compare such evaluation process with the structural join.

The rest of the paper is organized as follows. In Section 2, the necessary background is surveyed. The tree signatures are specified in Section 3. In Section 4, we show the advantages of tree signatures for XPath navigation, and in Section 5 we elaborate on the XML query processing application. Performance evaluation is described and discussed in Section 6. Conclusions and a discussion on alternative search strategies are available in Section 7.

2 Preliminaries

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

2.1 Labelled Ordered Trees

Let Σ be an alphabet of size $|\Sigma|$. 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 length n, we say that $level(i) = n$. Finally, $size(i)$ denotes the number of descendants of node i – the size of any leaf node is zero. In the following, we consider ordered labelled trees.

2.2 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 [5]. The *preorder* and *postorder* sequences are ordered 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 sequences of our sample tree in Fig. 1 – the node's position in the sequence is its preorder/postorder rank.

$$\begin{tikzcd}
& & a \arrow[dl] \arrow[dr] & & & \\
& b \arrow[d] \arrow[dr] & & f \arrow[dr] & & \\
& c \arrow[dl] \arrow[dr] & g & & h \arrow[dl] \arrow[dr] & \\
d & & e & o & & p
\end{tikzcd}$$

$$\begin{array}{r l}
pre: & a\ b\ c\ d\ e\ g\ f\ h\ o\ p \\
post: & d\ e\ c\ g\ b\ o\ p\ h\ f\ a \\
rank: & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10
\end{array}$$

Fig. 1. Preorder and postorder sequences of a tree

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

As proposed in [5], such properties can be summarized in a two dimensional diagram, as illustrated in Fig. 2, where the *ancestor* (A), *descendant* (D), *preceding* (P), and *following* (F) nodes of v are separated in their proper regions.

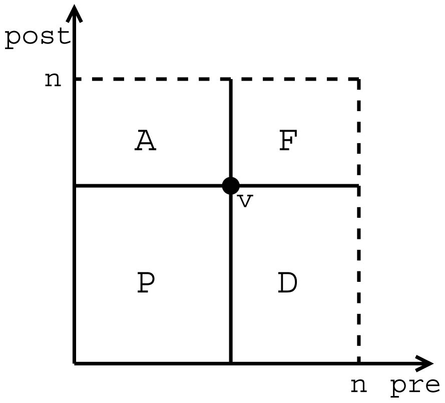

Fig. 2. Properties of the preorder and postorder ranks.

2.3 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 of the edit distance is defined by an $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$ that is filled so that for every $0 < i \leq n$ and $0 < j \leq m, M[i, j]$ 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* (l.c.s.). 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**lgo**r**i**t**hm. In the 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**lgo**r**i**t**hm and p**ar**achu**t**e.

By analogy to edit distance, the computation uses an $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$ such that for every $0 < i \leq n$ and $0 < j \leq m$, $M[i, j]$ 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 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.

Obviously, the matrix can be filled in $O(n \cdot m)$ time. But algorithms such as [7] can find l.c.s. much faster.

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 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 for the sequence-inclusion. If string x is string-included in y, it is also sequence-included in y, but not the other way around.

For example, the matrix for searching the l.c.s. of "art" and "parachute" is:

	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 $M[n, m]$-$= min\{m, n\}$. Because we do not have to compute all elements of the matrix, the complexity is $O(p) \mid p = max\{m, n\}$.

3 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. We use the preorder and postorder ranks to linearize the tree structures, which allows to apply the sequence inclusion algorithms for strings.

3.1 The Signature

The tree signature is an ordered list (sequence) 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 1. *Let T be an ordered labelled tree. The signature of T is a sequence, $sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots; t_m, post(t_m)\rangle$, of $m = |T|$ 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.*

Observe that the index in the signature sequence is the node's preorder, so the value serves actually two purposes. In the following, we use the term preorder if we mean the rank of the node, when we consider the position of the node's entry in the signature sequence, we use the term index. 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 Fig. 1. 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 from Fig. 3.

A sub-signature $sub\_sig_S(T)$ is a specialized (restricted) view of T through signatures, which retains the original hierarchical relationships of nodes in T.

Considering $sig(T)$ as a sequence of individual entries representing nodes of T, $sub\_sig_S(T) = \langle t_{s_1}, post(t_{s_1}); t_{s_2}, post(t_{s_2}); \ldots; t_{s_k}, post(t_{s_k})\rangle$ is a sub-sequence of $sig(T)$, defined by the ordered set $S = \{s_1, s_2, \ldots, s_k\}$ of indexes (preorder values) in $sig(T)$, such that $1 \leq s_1 < s_2 < \ldots < s_k \leq m$. For example, the set $S = \{2, 3, 4, 5, 6\}$ defines a sub-signature representing the subtree rooted at the node b of our sample tree.

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 $sub\_sig_S(T)$ be the sub-signature of $sig(T)$ induced by a sequence-inclusion of $sig(Q)$ in $sig(T)$, just considering the equality of node names. The following lemma specifies the tree inclusion problem precisely.

Lemma 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 $sub\_sig_S(T)$ through the ordered set of indexes $S = \{s_1, s_2, \ldots, s_n\} | q_1 = t_{s_1}, q_2 = t_{s_2}, \ldots, q_n = t_{s_n}$, (2) for all pairs of entries i and $i+j$ in $sig(Q)$ and $sub\_sig_S(T)$ $(i, j = 1, 2, \ldots |Q| - 1$ and $i + j \leq |Q|)$, $post(q_{i+j}) > post(q_i)$ implies $post(t_{s_{i+j}}) > post(t_{s_i})$ and $post(q_{i+j}) < post(q_i)$ implies $post(t_{s_{i+j}}) < post(t_{s_i})$.*

Proof. Because the index i increases in (sub-)signatures according to the preorder rank, node $i + j$ must be either the descendent or the following node of i. If $post(q_{i+j}) < post(q_i)$, the node $i+j$ in the query is a descendent of the node i, thus also $post(t_{s_{i+j}}) < post(t_{s_i})$ is required. By analogy, if $post(q_{i+j}) > post(q_i)$, the node $i+j$ in the query is a following node of i, thus also $post(t_{s_{i+j}}) > post(t_{s_i})$ must hold.

A specific query signature can determine zero or more data sub-signatures. Regarding the node names, any $sub\_sig_S(T) \equiv siq(Q)$, because $q_i = t_{s_i}$ for all i, see point (1) in Lemma 1. But the corresponding entries can have different postorder values, and not all such sub-signatures necessarily represent qualifying patterns, see point (2) in Lemma 1.

The complexity of tree inclusion algorithm according to Lemma 1 is $\sum_{i=1}^{n-1} i$ comparisons. Though the number of the query tree nodes is usually not high, such approach is computationally feasible. Observe that Lemma 1 defines the *weak inclusion* of the query tree in the data tree, in the sense that the parent-child relationships of the query are implicitly reflected in the data tree as only the ancestor-descendant. However, due to the properties of preorder and postorder ranks, such constraints can easily be strengthened, if required.

For example, consider the data tree T in Fig. 1 and the query tree Q in Fig. 3. Such query qualifies in T, i.e. $sig(Q) = \langle h, 3; o, 1; p, 2\rangle$ determines a

$$\begin{tikzcd} & h \arrow[dl] \arrow[dr] & \\ o & & p \end{tikzcd}$$

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

Fig. 3. Sample query tree Q

compatible $sub\_sig_S(T) = \langle h, 8; o, 6; p, 7\rangle$ through the ordered set $S = \{8, 9, 10\}$, because (1) $q_1 = t_8$, $q_2 = t_9$, and $q_3 = t_{10}$, (2) the postorder of node h is higher than the postorder of nodes o and p, and the postorder of node o is smaller than the postorder of node p (both in $sig(Q)$ and $sub\_sig_S(T)$). If we change in our query tree Q the lable 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 1 does not insist on the strict parent-child relationships, and implicitly consider 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 the query signature is also sequence-included (on the level of names) determining the sub-signature $sub\_sig_S(T) = \langle g, 4; o, 6; p, 7\rangle | S = \{6, 9, 10\}$. The reason for the false qualification is that the query requires the postorder to go down from node 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, which can be verified in Fig. 1.

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 Fig. 4, where A, P, D, F represent

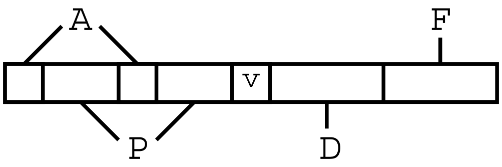

Fig. 4. Signature structure

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 preceding nodes. This suggests to extend 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

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

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 $m+1$. For illustration, the extended signature of the tree from Fig. 1 is

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

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

Given a node with index i, the cardinality of the descendant node set is $size(i) = ff_i - i - 1$, and the level of the node with index i is $level(i) = i - post(i) + ff_i - i - 1 = ff_i - post(i) - 1$. Further more, the tree inclusion problem can be solved in linear time, as the following lemma obviates.

Lemma 2. *Using the extended signatures, 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 $sub\_sig_S(T)$ through the ordered set of indexes $S = \{s_1, s_2, \ldots s_n\} | q_1 = t_{s_1}, q_2 = t_{s_2}, \ldots, q_n = t_{s_n}$, (2) for $i = 1, 2, \ldots |Q| - 1$, if $post(q_i) < post(q_{i+1})$ (leaf node, no descendants, the next is the following) then $ff(t_{s_i}) \leq s_{i+1}$, otherwise (there are descendants) $ff(t_{s_i}) > s_{ff(q_i)-1}$.*

4 Evaluation of XPath Expressions

XPath [3] 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.

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 through the extended signatures, using the tree from Fig. 1 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 of 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 at position $i+1$, and the last descendant object is at position $ff_i - 1$. If we consider node b (with $i = 2$), we immediately decide that the descendants are at positions starting from $i + 1 = 3$ to $ff_2 - 1 = 6$, i.e. nodes c, d, e, and g.

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.

Observe that the postorder values, $post(t_i)$, are not used for navigation, so the size of a signature for this kind of operations can even be reduced.

5 Query Processing

A query processor can also exploit tree signatures to evaluate *set-oriented* primitives similar to the XPath axes. Given a set of elements R, the evaluation of $Parent(R,$ `article`$)$ gives back the set of elements named `article`, which are parents of elements contained in R. By analogy, we define the $Child(R,$ `article`$)$ set-oriented primitive, returning the set of elements named `article`, which are children 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 first use these indexes to obtain elements satisfying the predicates, and then verify the structural relationships using signatures. Consider the following XQuery [4] 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 = "Smith");
2. let $R_2$ = $ContentIndexSearch$(first = "John");
3. let $R_3$ = $Parent$($R_1$,name);
4. let $R_4$ = $Parent$($R_2$,name);
5. let $R_5$ = $Intersect$($R_3$,$R_4$);
6. let $R_6$ = $Parent$($R_5$,people);
7. let $R_7$ = $Child$($R_6$,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 = "Smith");
2. let $R_2$ = $Parent$($R_1$,name);
3. let $R_3$ = $Child$($R_2$,first);
4. let $R_4$ = $FilterContent$($R_3$,John);
5. let $R_5$ = $Parent$($R_4$,name);
6. let $R_6$ = $Parent$($R_5$,people);
7. let $R_7$ = $Child$($R_6$,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.

Obviously, the outlined execution plans are not necessarily optimal. For example, they do not take into consideration the selectivity of predicates. But the query optimization with tree signatures is beyond the scope of this paper.

6 Experimental Evaluation

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

We compared the extended signatures with the Multi Predicate MerGe JoiN (MPMGJN) proposed in [10] – we expect to obtain similar results comparing with other join techniques as for instance [1]. As suggested in [10], the *Element Index* was used to associate each element of XML documents with its start and end positions, where the 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 as a B^+-tree. Retrieval of the inverted list associated with a key (the element name) was implemented with the bulk retrieval functionality, provided by the BerkeleyDB.

In our experiments, we have used queries of the following template:

```
for $a in //<e_name>
where <pred($a)>
return
      <result> $a/<e_1> ...$a/<e_n> </result>
```

Table 1. Selectivity of element names

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

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). 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 of expressions n 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 1 shows the number of occurrences of the element names that we used in our experiments, while Table 2 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 following execution plan was used to process our queries with the signatures:

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>`$)$.

The content predicate is evaluated by using a content index. The remaining steps are executed by navigating in the extended signatures.

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

Table 2. Selectivity of predicates

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

```
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.

Analysis. Results of performance comparison are summarized in Table 3, 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 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 of elements satisfying the low selective predicate. Since such queries retrieve large portions of the database, they are not supposed to be 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

Table 3. Performance comparison between extended signatures and containment join. Processing time is expressed in milliseconds.

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

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 reference lists.

7 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. 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. 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.

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 (or bag) 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 [8] is a big challenge for XML searching. Due to the extensive literature on string processing, see e.g. [6], 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

1. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 310–321, Madison Wisconsin, USA, June 2002*. ACM, 2002.
2. S. Chien, Z. Vagena, D.Zhang, V.J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 28rd VLDB Conference, Honk Kong, China*, pages 263–274, 2002.
3. World Wide Web Consortium. XML path language (XPath), version 1.0, W3C. Recommendation, November 1999.
4. World Wide Web Consortium. XQuery 1.0: An XML query language. W3C Working Draft, November 2002. http://www.w3.org/TR/xquery.
5. Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002, Madison, Wisconsin*, pages 109–120. ACM Press, New York, NY USA, 2002.
6. D. Gusfield. *Algorithms on Strings, trees, and Sequences.* Cambridge University Press, 1997.
7. J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350, 353 1977.
8. Anja Theobald and Gerhard Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25–27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 477–495. Springer, 2002.
9. Paolo Tiberio and Pavel Zezula. Storage and retrieval: Signature file access. In A. Kent and J.G. Williams, editors, *Encyclopedia of Microcomputers*, volume 16, pages 377–403. Marcel Dekker Inc., New York, 1995.
10. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In Walid G. Aref, editor, *ACM SIGMOD Conference 2001: Santa Barbara, CA, USA, Proceedings.* ACM, 2001.