A Signature-Based Approach for Efficient Relationship Search on XML Data Collections*

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

[1] ISTI-CNR, Pisa, Italy
firstname.lastname@isti.cnr.it
[2] Masaryk University, Brno, Czech Republic
zezula@fi.muni.cz

Abstract. We study the problem of finding relevant relationships among user defined nodes of XML documents. We define a language that determines the nodes as results of XPath expressions. The expressions are structured in a conjunctive normal form and the relationships among nodes qualifying in different conjuncts are determined as tree twigs of the searched XML documents. The query execution is supported by an auxiliary index structure called the tree signature. We have implemented a prototype system that supports this kind of searching and we have conducted numerous experiments on XML data collections. We have found the query execution very efficient, thus suitable for on-line processing. We also demonstrate the superiority of our system with respect to a previous, rather restricted, approach of finding the lowest common ancestor of pairs of XML nodes.

1 Introduction

A typical characteristic of XML objects is that they combine in a single unit data values and their structure. Such encapsulation of data and structure is very convenient for exchanging data, because the separation of schema and data instances in traditional databases might cause problems in keeping proper relationships between these two parts.

XML is becoming the preferable format for the representation of heterogenous information in many and diverse application sectors, such as electronic data interchange, multimedia information systems, publishing and broadcasting, public administration, health care and medical applications, and information outside the corporate database. This widespread use of XML has posed a significant number of technical requirements for storage and content-based retrieval of XML data – many of them are still waiting for effective solutions. In particular, retrieval of XML data, based on content and structure, has been widely studied and the problem has been formalized by the definition of query languages such as the XPath and XQuery. Consequently, most of the implementation effort have concentrated on the development of systems able to execute queries expressed

* This work was partially supported by the ECD project (Extended Content Delivery), funded by the Italian government, by the VICE project (Virtual Communities for Education), also funded by the Italian government, and by DELOS NoE, funded by the European Commission under FP6 (Sixth Framework Programme).

in these languages. To tackle the problem of structural relationships, the main stream of research on XML query processing has concentrated on developing indexing algorithms that respect the structure. Though other alternatives exist, the *structural or containment join* algorithms, such as [ZND+01], [LM01], [SAJ+02], [CVZ+02], and [BKS02], are most popular. They all take advantage of the *interval based* tree numbering scheme.

However, many other research issues are still open. A new significant problem is that of searching for relationships among XML components, that is either nodes and/or values. Indeed, there are many cases where the user may have a vague idea of the XML structure, either because it is unknown, or because it is too complex, or because many different, semantically close or equivalent, structure forms are used for XML coding. In these cases, what the user may need to search for are the relationships that exist among the specified components. For instance, in an XML encoded bibliography dataset, one may want to search for relationships between two specific persons to discover whether they were co-authors, editors, editor and co-author, cited in the bibliography, etc. In all these cases, the user may obviously have problems with languages that require to specify (as precisely as possible) the search paths. Any vagueness may result in a significant imprecision of search results and certainly in an undesirable increase of the computational costs. For example, provided that the schema is known, a very complex XQuery – taking into account all possible combinations of person roles – or several queries should be expressed in order to obtain the same result, with an obvious performance drawback. In Section 4.3 we give a real example on a car insurance policy application. This will show the suitability of our approach both in terms of the expressiveness of the query language and the performance efficiency.

In fact, there have been attempts to base search strategies on explicitly unknown structure of data collections. Algorithms for the proximity search in graph structured data are presented in [GS98]. The objective is to *rank* retrieved objects in one set, called the *Find* set, according to their proximity to objects in another given set, called the *Near* set. Specifically, applications generate the *Find* and *Near* queries on the underlying database. The database (or information retrieval) engine evaluates the queries and passes the *Find* and *Near* object result sets to the *proximity engine*. The proximity engine then ranks the *Find* set using available distance function represented as the length of the shortest path between a pair of objects from the *Find* and *Near* sets. In [GS98], the formal framework is presented and several implementation strategies are experimentally evaluated. However, the performance remains the main problem.

The Nearest Concept Queries from [SKW01] are defined for the XML data collections, and the queries use advantage of the *meet operator*. For two nodes o_1 and o_2 in given XML tree, the meet operator, *meet*(o_1, o_2), simply returns the *lowest common ancestor*, l.c.a., of nodes o_1 and o_2. Such node is called the *nearest concept* of nodes o_1 and o_2 to indicate that the type, i.e. the node's tag, of the result is not specified by the user. Though extensions of this operator to work on a set of nodes are also specified, reported experiments only consider pairs of nodes. Further more, the response time dramatically depends on the actual distance between the nodes.

Our approach can be seen as an extension or a generalization of [SKW01] and [GS98], assuming the XML structured object collections. In principle, the result of a relationship (or structure) search query can be represented as a set of twigs of the

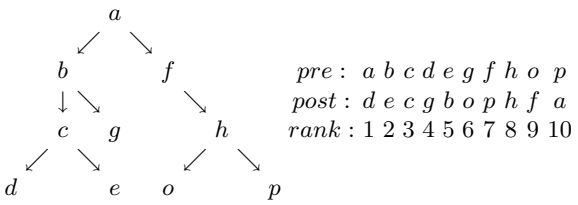


Fig. 1. Preorder and postorder sequences of a tree

searched data. In this way, all structural relationships, not only the l.c.a., are encapsulated and made available to the user for additional elaboration or ranking. To achieve high performance, we use the *tree signature* concept [ZAR03], which is a compressed XML tree representation supporting efficient search and navigation operations. The signatures have already been applied to the traditional XML searching with good success – see [ZAD03] for the *ordered* and [ZMM04] for the *unordered* inclusion of query trees in the data trees.

In the following, we summarize the concept of tree signatures in Section 2 and define their analytic properties. In Section 3, we specify the structure search queries and elaborate on procedures for their efficient evaluation. We describe our prototype implementation in Section 4 where we also report results from experimentation. Final discussion and our future research plans are in Section 5.

2 Tree Signatures

The idea of the tree signature [ZAD03,ZAR03,ZMM04] is to maintain a space effective representation of the XML tree structures. Formally, the XML data tree T is seen as an ordered, rooted, labelled tree. To linearize the trees, the *preorder* and *postorder ranks* from [Die82] are applied as the coding scheme. For illustration, see the preorder and postorder sequences of a sample tree in Figure 1 – the node's position in the sequence is its preorder/postorder rank, respectively.

The basic (short) signature of tree T with $m = |T|$ nodes has the following format

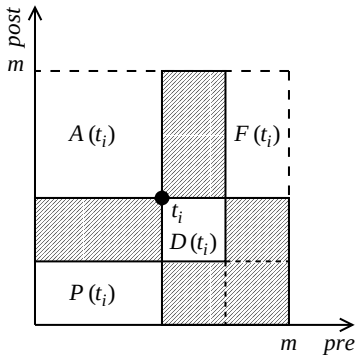
$$sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots; t_m, post(t_m)\rangle,$$

where t_i represents the name of node with preorder i and postorder $post(t_i)$. A more rich version of the tree signature, called the *extended signature*, is a sequence

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

where $ff(t_i)$ is the pointer to (preorder value of) the *first following* node, and $fa(t_i)$ refers to the *first ancestor*, that is the parent node of t_i. If there are no following nodes, $ff(t_i) = m + 1$. Since the root t_1 has no ancestors, $fa(t_1) = 0$. For illustration, the extended signature of the tree from Figure 1 is

$$\langle a, 10, 11, 0; b, 5, 7, 1; c, 3, 6, 2; d, 1, 5, 3; e, 2, 6, 3; \ldots; h, 8, 11, 7; o, 6, 10, 8; p, 7, 11, 8\rangle$$



descendants $D(t_i) = \{t_j | i < j < ff(t_i)\}$,
with $|D(t_i)| = size(t_i) = ff(t_i) - i - 1$;
following $F(t_i) = \{t_j | ff(t_i) \leq j \leq m\}$,
with $|F(t_i)| = m + 1 - ff(t_i)$;
ancestors $A(t_i) = \{t_j | j < i \wedge post(t_j) > post(t_i)\}$,
with $|A(t_i)| = level(t_i) = ff(t_i) - post(t_i) - 1$;
preceding $P(t_i) = \{t_j | j < i \wedge post(t_j) < post(t_i)\}$,
with $|P(t_i)| = i + post(t_i) - ff(t_i)$.

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

Given a node t_i with $pre(t_i) = i$, all the other nodes can be divided into four disjoined subsets of computable cardinalities, as illustrated in Figure 2. By definition of the nodes' ranks, the *descendant* D nodes (if they exist) form a continuous preorder as well as a postorder sequences. Further more, the *following* F nodes end the preorder sequence and the *preceding* P nodes start the postorder sequences. So there is actually some empty space in the $pre/post$ plain as highlighted by the dark area in Figure 2 (left). In the preorder sequence, the *ancestor* A nodes interleave with the preceding nodes, in the postorder sequence, the ancestor nodes interleave with the following nodes.

As demonstrated in [ZAD03], the signatures also efficiently support execution of tree operations such as the leaf detection, path slicing, (sub-)tree inclusion tests, and many others. The *lowest common ancestor*, l.c.a., of nodes t_i and t_j is the node with highest preorder of $\{A(t_i) \cap A(t_j)\}$. Assuming $i < j$, an efficient algorithm to find l.c.a. recursively follows the fa pointer to find the first node t_k, such that $ff(t_k) > j$. Without any increase of complexity, this strategy can easily be generalized to finding the l.c.a. of n nodes.

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, but it is not necessarily forming a tree. 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 $sig(T)$ with $1 \leq s_i \leq m$ for all i.

3 Structure Search Queries

The key construct of most XML query models and languages is the *tree pattern*, TP. Accordingly, research on evaluation techniques for XML queries has concentrated on tree pattern matching defined by the pair $Query = (Q, C)$, where $Q = (B, E)$ is the *query tree* in which each node from B has a name (label), not necessarily different,

each edge from E represents the *parent-child* or the *ancestor-descendant* relationship between pairs of nodes from B. The *constraint* C is a formula specifying restrictions on the nodes and their properties, including in general their tags, attributes, and contents. In order to improve query efficiency, this concept was recently extended in [CJL$^+$03] into the *generalized tree pattern* query, GTP, where edges of the query tree Q can be *optional* or *mandatory* and each node carries a real number valued *score*. The tree signature concept has already proved to be highly competitive with respect to numerous other alternatives to accelerate execution of TP queries both considering the query trees as ordered [ZAD03,ZAR03] and also unordered [ZMM04].

In this paper, we consider a different concept of query. It does not explicitly declare relationships among qualifying data tree nodes. Discovering relationships is actually the objective of querying to find hierarchically dependent subsets forming a tree. In the following, we first define the query model, then define its evaluation principles, and finally specify an efficient algorithm for the query execution.

3.1 Structure search query model

We define the structure search query SS_Q as a conjunctive normal form of node specification expressions E_i^j as

$$SS\_Q = (E_1^1 \vee \ldots \vee E_{n_1}^1) \wedge \ldots \wedge (E_1^k \vee \ldots \vee E_{n_k}^k),$$

where each E_i^j is an XPath expression determining candidate nodes to be used as a starting point for relationships discovery. The result of such a query is the set of twigs, of the searched data tree, where structural relationships existing among nodes qualifying E_i^j of different conjuncts are made explicit. Individual nodes can be constrained by full or partial name-path specifications as well as by content predicates. Observe that an expressions E_i^j might also search for all elements having a specific content, no matter what is the name of the element, or it might search for all elements of a certain name, independently of their content and structural relationships in the data tree.

Provided the XML schema is known, traditional XML processing tools, e.g. XQuery, can also determine instances of all such structural relationships. However, this would require execution of multiple TP queries, each of which considering a specific node relationships constrain. Our approach has a higher expressive power, because it does not require a priori knowledge of the structure of the XML documents and, as we will see later, offers higher processing performance.

More formally, the answer to a structure search query SS_Q is a set of sub-trees (or *twigs*) of the data tree, obtained as follows. Let T be the data tree and SS_Q a structure search query consisting of k conjuncts. Let R^Q be a *pattern of* k *nodes* of T such that:

- each node qualifies in a different conjunct, and
- all the nodes share a common ancestor.

Then, a qualifying twig T^Q is the sub-tree of T induced by R^Q. T^Q consists of all nodes of R^Q, but can also include additional, *induced*, nodes, which are the ancestors of nodes from R^Q up to their l.c.a., that is the root of the twig. Since T^Q is a sub-tree of T, the relationships of nodes in T^Q are the same as in T. In the following, we denote

the pattern of induced nodes as I^Q. For formal manipulations, we see the patterns R^Q and I^Q as well as the twig T^Q as sub-signatures of $sig(T)$, designated, respectively, as $sub\_sig_{S^R}(T)$ and $sub\_sig_{S^I}(T)$. For brevity, we use $sub\_sig_S(T)$ whenever explicit distinctions between the patterns and twigs are not necessary.

For illustration, consider the following query.

`(//person[name=John]` $\vee$ `//person[name=Jack])`
$\wedge$ `(//person[name=Ted])`.

The qualifying patterns are pairs, because the query consists of two conjuncts. Furthermore, one node of the pair is always a person with name Ted and the other is a person with name John or Jack. However, the resulting twigs can be quite different even for a specific pair of nodes. For example, Jack and Ted can appear below the element `<author>`, which implies that they are coauthors of the same article. But they can have the `<journal>` element as their l.c.a. where several alternatives for twigs can occur. Two persons can be authors of different journal papers, but they can also be editors of this journal, or one of them can be the editor and the other the author.

Level constrained structure search In some cases, it might be desirable to find twigs with the root at level of at least certain value – we assume the document's root to be at level 0. For example, consider the DBLP XML data set [DBLP]. It consists of just one large XML document (file), having the `<dblp>` element as its root. In this case, structure search queries would typically produce a large set of results, because all possible query patterns have at least the `<dblp>` element as the root. Further more, the fact that two names with the l.c.a. `<dblp>` appear in the DBLP bibliography is probably of low significance. It would be more useful to search for structural relationships excluding the `<dblp>` root element, that is searching for twigs with the root element on levels greater than 0. We denote such *level constrained* structure search queries as $SS\_Q^{lev_{min}}$, indicating that the level of the roots of the qualifying twigs should be greater than or equal to lev_{min}.

3.2 Query evaluation principles

Assume a collection of XML documents and an $SS\_Q^{lev_{min}}$ query. Suppose each conjunct produces a non-redundant list L_i, $i = 1, 2, \ldots, k$ of preorder values of qualifying nodes in T. The central problem of the query evaluation is to determine the query patterns R^Q as the sub-signature $sub\_sig_{S^R}(T) \mid S^R = \{s_1, s_2, \ldots, s_k\}$, which satisfies the following properties:

1. each s_j is from exactly one list L_i and no two instances are from the same list;
2. the constraint $s_1 < s_2 < \ldots < s_k$ is satisfied;
3. the l.c.a. of $\{s_1, s_2, \ldots, s_k\}$ exists on level $\geq lev_{min}$.

For example, if the query is `(//g)` $\wedge$ `(//f)` $\wedge$ `(//c)`, then the qualifying pattern in tree T from Figure 1 is $sub\_sig_{S^R}(T) = \langle c, 3; g, 4; f, 9\rangle$, because $S^R = \{3, 6, 7\}$. The l.c.a. of nodes c, g, and f is a, that is a node at level 0.

3.3 Structure search query evaluation strategies

The query evaluation proceeds in the following four phases: 1) evaluation of the expressions E_i^j; 2) evaluation of the conjuncts; 3) generation of the node patterns R^Q 4) generation of resulting twigs T^Q. This sub-section discusses efficient execution of phases 1), 2), and 3). Section 3.5 concerns the evaluation of the phase 4).

The evaluation of each expression E_i^j returns a list LE_i^j of nodes qualifying for the corresponding XPath expression. We suppose that these lists are ordered according to the preorder ranks. The efficient evaluation of expressions E_i^j and ordering of the lists LE_i^j can be obtained by using XML path indexes and/or the XML tree signatures, as discussed in [ADR+03,ZAD03].

The result of the j-th conjunct is the union of the lists LE_i^j for all i and a specific j. Since the lists LE_i^j are ordered with respect to the preorder ranks, multiple merge of corresponding lists ensures efficiency of this procedure. This will produce k lists L_i of nodes, one for each conjunct, still ordered by the preorder ranks.

A naive way to obtain patterns R^Q is to first produce the Cartesian product of the k lists L_i, and then eliminate those k-tuples that do not satisfy the conditions defined in Section 3.2. However, the cost of such process can be very high, because the Cartesian product can produce many tuples among which only few are finally qualifying. Such approach also assumes complete k lists to be available. In the following, we propose a new algorithm, called the *structure join*, which is able to perform this step of query execution efficiently.

1: **procedure** STRUCTUREJOIN($L_1, \ldots, L_k, lev_{min}$)
▷ $L_1, \ldots, L_k$ are the lists with elements sorted with respect to preorder numbering;
▷ lev_{min} is the minimum level accepted for roots of qualifying twigs;
▷ $L_i(j)$ is the j-th element in list L_i;
2: $result := \oslash$;
3: **if** $(\exists i : L_i = \oslash)$ **then return** $result$;
4: **else**
5: $c_i := 1[\forall i, i = 1, \ldots, k]$; ▷ Cursors pointing to current tops of the lists
6: $P := max\{\ pre(L_i(c_i))|\ i = 1, \ldots, k\}$;
7: $M \in \{\ j|\ pre(L_j(c_j)) = P\ \wedge\ \ 1 \leq j \leq k\}$;
8: $a_M \in \{a|\ a \in ancestors(L_M(c_M)) \wedge level(a) = lev_{min}\}$;
9: **if**$(a_M = null)$ **then** $c_M := c_M + 1$; **goto** step 6; **end if**
10: **if** $(\exists i :\ pre(L_i(c_i)) \leq pre(a_M))$ **then**
11: $c_i := min\{c|\ pre(L_i(c)) > pre(a_M)\}[\forall i : i = 1, \ldots, k]$;
12: **if**$(\exists i :\ c_i > length(L_i))$ **then return** $result$; **else goto** step 6;
13: **else**
14: $pl_M := ff(a_M) - 1$; ▷ pl_M is the preorder of the last node of the sub-tree rooted at a_M
15: $SL_i := \{L_i(c)|c_i \leq c \wedge pre(L_i(c)) \leq pl_M\}[\forall i : i = 1, \ldots, k]$;
16: <Generate sub-signatures of length k with each element from a different SL_i; put them in $result$ >
17: $c_i := min\{c|pre(L_i(c)) > pl_M\}[\forall i : i = 1, \ldots, k]$;
18: **if**$(\exists i : c_i > length(L_i))$ **then return** $result$; **else goto** step 6;
19: **end if**
20: **end if**
21: **end procedure**

Table 1. Structure Join Algorithm

The **StructureJoin** algorithm specified in Table 1 takes as input k lists of qualifying nodes, with elements sorted with respect to preorder numbering, and the level constraint

lev_{min}. The algorithm avoids the unnecessary generation of not valid sub-signatures by restricting the computation of the Cartesian product to portions of the lists L_i that contain elements which belong to the same potentially qualifying sub-tree. Step 3 checks for an empty list, which terminates the algorithm. Then cursors c_i are set to refer the first element of the lists in Step 5. Step 6 and 7 choose M as the index of the list that has the element $L_M(c_M)$ with maximum preorder. Provided that a pattern of nodes with structural relationships with $L_M(c_M)$ exists, the corresponding sub-signatures will have $L_M(c_M)$ as the last node. Step 8 determines the ancestor a_M of $L_M(c_M)$ which has level lev_{min}. The ancestor can be efficiently obtained by using the tree signature. This ancestor is the root of the sub-tree containing all possible nodes that can be joined with $L_M(c_M)$ (i.e. having valid structural relationships with $L_M(c_M)$). If a valid ancestor cannot be found, i.e. $level(L_M(c_M)) < lev_{min}$, the element is discarded by moving the cursor forward (Step 9). Step 10 checks that the top element of all lists belongs to the sub-tree rooted at a_M, that is, checks that the preorder of the top elements is greater than or equal to $pre(a_M)$. If the top element of at least one list is smaller than $pre(a_M)$, then the cursor of that list should be moved to the first element with preorder greater than $pre(a_M)$ (Step 11) and if the end of the list is reached, the algorithm ends (Step 12). If the top element of all lists is greater than $pre(a_M)$ then Step 14 uses the first following pointer, contained in the signatures, to compute pl_M, the preorder of the last node in the sub-tree rooted at a_M. Note that at this point no list can have a top element with preorder greater than pl_M, because that list should have been selected at Steps 6 and 7. All nodes in the lists that have preorder included between $pre(a_M)$ and pl_M are used to generate qualifying sub-signatures, that is, the Cartesian product will be computed only using the consecutive portions of the lists corresponding to elements having preorder between $pre(a_M)$ and pl_M. In fact, Step 15 determines the set of elements in each list that should be used for the Cartesian product – sub-lists SL_i always form a continues sequence in corresponding lists L_i. Step 16 computes the Cartesian product by generating all possible combination of nodes belonging to the sub-tree rooted at a_M present in the lists, and arranges the obtained tuples to form valid sub-signatures. Step 17 moves the cursor to point to the new top element corresponding to the next sub-tree in each list. If the end of at least one list is reached, the algorithm ends (Step 18).

Example: We illustrate the behavior of the algorithm with an example. Figure 3 shows a data tree template (on the left) and the manipulation process of the joined lists (on the right). Suppose a structure search query with three conjuncts, where the phases 1) and 2) have produced the ordered lists L_1, L_2, and L_3. Sub-trees involved in the algorithm execution are labelled from 1 to 4. The nodes of these sub-trees contained in the lists are highlighted with rectangles also labelled as the corresponding sub-trees. Finally, suppose to be at the beginning of a generic iteration of the algorithm with cursors pointing to positions c_1^1, c_2^1, and c_3^1. Steps 6 and 7 choose M^1 such that the preorder of $L_{M^1}(c_{M^1}^1)$ is the maximum (i.e. the rightmost) among the elements pointed by the list cursors. Step 8 determines a_{M^1} as the highest possible ancestor of such a node. Elements with admissible structural relationships with $L_{M^1}(c_{M^1}^1)$ should be in the sub-tree rooted at a_{M^1}. Step 10 detects that there are lists where the first element is not in the currently considered sub-tree. In fact, the first element of list L_1 is from sub-tree 1. Note that in order to have valid structure relationships involving elements

of sub-tree 1, all other lists must have elements of that sub-tree. However, the fact that cursors of the other lists refer elements of sub-list 3 means that no elements from sub-tree 1 were found in previous iterations. Step 11 moves cursors of the lists to point the first element with preorder greater than $pre(a_{M^1})$. Note that also elements of sub-tree 2 in list L_1 are skipped at this step. Now, Step 6 and 7 chose the new rightmost element $L_{M^2}(c^2_{M^2})$, which now belongs to sub-tree 4. Step 8 determines a_{M^2}, which now is the root of sub-tree 4. Step 10 determines that there are lists whose first element does not belong to sub-tree 4 (both lists L_2 and L_3 have the first element from sub-tree 3). As before, Step 11 moves the cursors forward to refer elements after a_{M^2}. Current situation is that now all lists have elements from sub-tree 4. Therefore the algorithm arrives at Step 14 were the preorder pl_{M^3} of the last element of sub-tree 4 is determined and the Cartesian product of sub-lists with elements of sub-tree 4 is performed at Step 16. Step 17 moves the cursor to the next sub-tree in the lists and goes for a new iteration.

3.4 Algorithmic complexity considerations

It is important to point out that this algorithm, in case that the width of the sub-trees is small compared to all the elements contained in the lists, would compute the Cartesian product of small (continues) portions of the lists. In fact, just elements belonging to the same sub-tree are joined.

From this observation we can show that the complexity of our algorithm, in the average case, can be considered linear in the dimension of the input. In fact, the complexity of a single Cartesian product is equal to the product of the sizes of the sub-lists SL_i. This can be realistically bounded by the average size of the sub-trees. Let's call s_{avg} that size. Then, the complexity of the Cartesian product is $O((s_{avg})^k)$. The number of Cartesian products computed is bounded by the number of sub-trees, since the algorithm computes at most one Cartesian product for every sub-tree. The number of sub-trees can be estimated as n/s_{avg}, where n is the total number of elements in the dataset. Therefore, the complexity of the algorithm is

$$O((s_{avg})^k \times (n/s_{avg})) = O((s_{avg})^{k-1} \times n).$$

In case that the average size of the sub-trees is much smaller than the size of the dataset ($s_{avg} \ll n$), which is generally true in practice, the complexity is linear with the size n of the input. Note also that in practice we have noticed that the sizes of the lists L_i are smaller than those of the corresponding sub-trees. Of course, in the worst case when the lev_{min} is set to 0 and the data set is composed of one large XML file the result is that $s_{avg} = n$ and the complexity is $O(n^k)$. However, this case does not occur in practice, since all combinations of occurring elements in the XML file will always be found and a structure search query would not make sense.

3.5 Data Twig Derivation

In order to derive a qualifying twig T^Q for query SS_Q in tree T, we start with the sub-signature $sub\_sig_{S^R}(T)$ representing the query's qualifying pattern. Then for each ancestor set $A(t_{s_i})$, $i = 1, 2, \ldots, k$, we determine a sub-signature $sub\_sig_{S_i}(T)$ of the

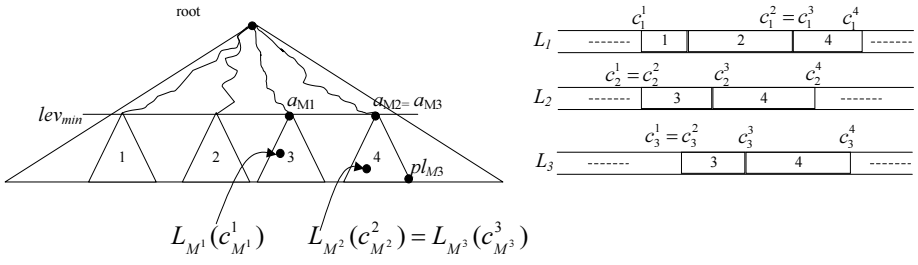


Fig. 3. Structure Join execution example

path from the node t_{s_i} to the l.c.a. of all nodes in S^R. In this way, we obtain sets S_i, and the qualifying twig is obtained as a tree induced from the pattern as

$$sig(T^Q) = sub\_sig_{S^T}(T) | S^T = \cup_{i=1}^{n} S_i \cup S^R,$$

considering the sets S_i as ordered – the set S^R is also ordered.

Provided $S^T = \{s_1, s_2, \ldots s_h\}$, the sub-signature $sub\_sig_{S^T}(T)$ defines a tree twig T^Q with root t_{s_1} of preorder value s_1 in T. Since the preorder/postorder values in sub-signatures are those of the original tree, leaves in sub-signatures are not necessarily leaves of T, so they can only be recognized by checking consecutive entries of the sub-signature. Specifically, if $post(t_{s_i}) < post(t_{s_{i+1}})$ then the i-th entry in the sub-signature is a leaf of the twig T^Q. Naturally, the last element, t_{s_h} in our case, is always a leaf. The element t_{s_1} needs not be checked, because it is the root of T^Q.

If we continue with our previous example of the structure search query `(//g)` $\wedge$ `(//f)` $\wedge$ `(//c)`, resulting in the query pattern sub-signature $sub\_sig_{S^R}(T) = \langle c, 3; g, 4; f, 9\rangle \; | S^R = \{3, 6, 7\}$, the sub-signatures of the ancestors define the sets $S_1 = \{1, 2\}$ (for the node c) , $S_2 = \{1, 2\}$ (for the node g), and $S_3 = \{1\}$ (for the node f). The union of the ordered sets of S^R, S_1, S_2, and S_3 is $S^T = \{1, 2, 3, 6, 7\}$, which is a sub-tree of nodes a, b, c, g, f from T, rooted at node a. If we change our query set to $B = \{g, c\}$, we get $sub\_sig_{S^R}(T) = \langle c, 3; g, 4; \rangle \; | S^R = \{3, 6\}$, so $S_1 = \{2\}$ and $S_2 = \{2\}$, because the lowest common ancestor of c and g is b. When we make the ordered union of all these sets, we get $S^T = \{2, 3, 6\}$, which defines a sub-tree rooted at node b.

4 Experimental Evaluation

In this section we validate the **Structure Join** algorithm from the performance point of view. Our algorithm, as demonstrated in Table 1, offers very high performance and scales very well with the increasing size of the joined lists. The structure join algorithm was implemented in Java, JDK 1.4.0 and the experiments 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 have conducted most of our experiments on the XML DBLP dataset, consisting of about 200 MB [DBLP] of data. We have verified the performance of the structure

join algorithm in three tests. First, we have measured its performance using different queries, which have different number of conjuncts and different sizes of the input sets (see Section 4.1). Second, we have compared the efficiency of our structural join with the *meet* operator proposed in [SKW01] (see Section 4.2) . Finally, we have run experiments in a real applicative scenario, with a more complex dataset containing insurance records, as reported in Section 4.3.

4.1 Performance measurements

The queries that we have used to run the first group of experiments are listed in the first column of Table 2. Each query is coded as "QD_n", where D indicate the size of the input set (which can be Small S, Medium M, or Large L) and n can be 2 or 3 to indicate the number of conjuncts. For all our queries, the level constrain lev_{min} was set to 1 (just below the root element that is on level 0). For each query, Table 2 reports the size of the corresponding input set, the size of the output set, the number of Cartesian products computed (that is the number of qualifying sub-trees containing elements in the input set), the average number of iterations executed in each Cartesian product, and the elapsed time to complete the structure join in milliseconds.

Queries		**#input set**	**#output set**	**#CP (#ICP)**	**Time (ms)**
QS_2	//phdthesis/title ∧ //phdthesis/author	72 72	72	72 (1)	<1
QM_2	//incollection/title ∧ //incollection/author	1410 2931	2931	1400 (2)	30
QL_2	//inproceedings/title ∧ //inproceedings/author	22004 53243	53243	21977 (2)	392
QS_3	QS_2 ∧ //phdthesis/year	72-72 72	72	72 (1)	<1
QM_3	QM_2 ∧ //incollection/year	1410-2931 1410	2931	1400 (2)	37
QL_3	QL_2 ∧ //inproceedings/year	22004-53243 22004	53243	21977 (2)	512

Table 2. Performance of Structure Join algorithm. In the last but one column **#CP** indicate the number of Cartesian products and **#ICP** the average number of iterations for each Cartesian product.

The reported processing time is obtained as the average over one hundred independent executions of the algorithm. It only includes the processing time required for the structure join algorithm and it does not include the time needed to obtain the input sets. The experiments demonstrate the linear trend with respect to the cardinality of largest input set, confirming our expectation of linear complexity.

As explained previously, a trivial technique to execute the structure search is to compute a Cartesian product of the input sets and to eliminate the non qualifying tu-

ples. The complexity of such a strategy would have been polynomial. In particular the number of iterations required would have been equal to the product of the cardinalities of the input sets. Our algorithm, on the other hand, computes several Cartesian products and the number of iterations in each product is small. Therefore, the overall cost of our algorithm is linear.

Note that the number of computed Cartesian products increases linearly with sizes of the input sets (given that the number of qualifying sub-trees increases linearly with the sizes of the input sets). On the other hand, the number of iterations computed in each Cartesian product is independent of the sizes of the input sets, it is very small in practical cases (and it can be considered constant). This is explicated considering the actual number of Cartesian products computed, the average number of iterations in each product, and the elapsed time reported in Table 2.

For example, the trivial Cartesian product technique would have required 72*72*72 iterations to process query QS_3, while our technique needs only 72*1 iterations. The trivial Cartesian product technique would have required 22004*53243*22004 iterations to process query QL_3, while our technique requires just 21977*2 iterations. Note that the number of computed Cartesian products (21977) is smaller than the size of the smaller input set (22004). This is due to the fact that sometimes elements were discarded, because no elements from the same sub-trees were found in the other input sets (inproceedings with title and without authors or years).

4.2 Comparison with other techniques

To compare our algorithm with the meet operator, we have repeated the experiments from [SKW01]. Specifically, we have searched DBLP for the string ”ICDE” and for the year records, incrementally including years from 1999 to 1984. We have performed the structure join on this two sets and we have computed the elapsed time of our algorithm. Figure 4 compares the elapsed time of the original meet operator and our algorithm varying the size of the input set (obtained in correspondence of the number of years included in one input set). The graph on the left side shows the performance of our algorithm, while the graph on the right shows the performance of the meet operator. Our technique is about two orders of magnitude faster than the meet operator – the time scale of the graph on the left is 100 times smaller than that of the graph on the right. For instance, with the maximum input set sizes, our technique is 96 times more efficient than the meet operator technique. Specifically our algorithm processes the query in about 30 ms, while the meet operator needs about 3 seconds.

4.3 On-the-field experiment

In addition to the previous performance tests, we have also conducted experiments with a dataset related to a more realistic and complex scenario. Figure 5 sketches the XML structure of information used by a car insurance company to keep track of the customers and their record of accidents, that is an information about the type of accident, its status, those involved in the accident as witnesses, those injured, etc. In order to detect possible frauds, the insurance company may be interested to discover all relationships between two (or more) of their customers. Examples of requests are as follows: have they been

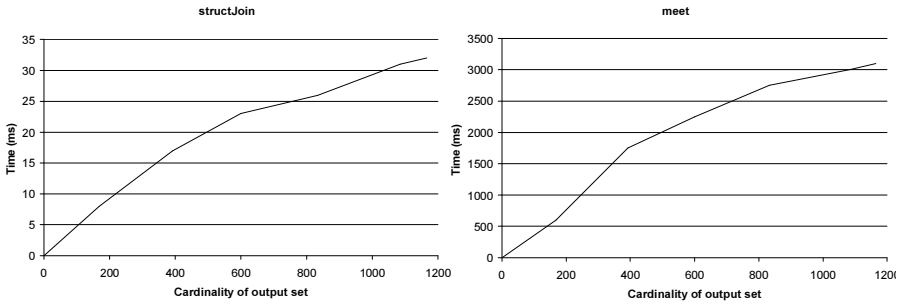


Fig. 4. Comparison between our structJoin algorithm and the meet operator

involved in many accident, possibly with different roles (i.e. causing the accident, witness, injured person, etc.)? Who were those involved as witness or as injured persons or as the insurance owners?

The size of this dataset is 45 Mb and it contains 10000 insurance policies. In the experiment, we have searched for co-occurrences of four specific persons, identified by their names, using as level thresholds (lev_{min}) 0, 1 and 2, corresponding, respectively, to elements `<Insurance Policy>`, `<Risk>`, and `<Accident>`. In the first case we implicitly searched for co-occurrences of the four persons either as an owner, witness, or subject involved in accidents of the same insurance policy. In the second case we implicitly searched for co-occurrences as a witness or subject involved in accidents of the same policy. In the last case, we searched for co-occurrences in the same accident. The number of twigs that we have found, which is satisfy the search criteria, are 6 for the level 0, 2 for the level 1, and 1 for the level 2. It is interesting to notice that we have discovered that the specified persons were somehow involved together in 6 different insurance policies and the related accidents with different roles. This could suggest that further investigation should be performed by the insurance company on these subjects. The time required for processing such queries was, respectively, 7, 5, and 4 milliseconds, confirming the high performance of the technique in real application scenarios.

5 Conclusions

Contemporary XML search engines reason about the meaning of documents by considering the structure of documents and content-based predicates, such as the set of words that are contained within them. However, the structure of the documents is not always known, so it can become the subject of searching.

In this paper, we have introduced a new type of queries, called the structure search, that lets users to query XML databases with value predicates and node names, but without specification of their actual relationships. The retrieved entities are sub-trees of the searched trees (twigs), which obviate the structural relationships among determined tree nodes, if they exist. The twigs can also contain additional tree nodes, not explicitly required by the query, so the transitive relationships are also discovered. Our prototype

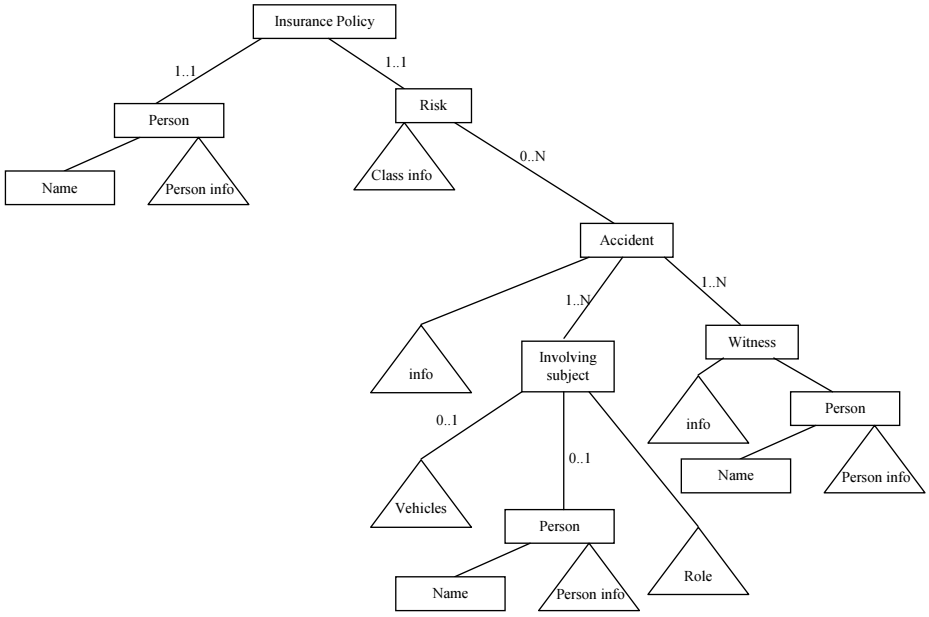


Fig. 5. Schema of the insurance records dataset

implementation demonstrates that the proposed algorithms yield useful results on real world data and scale well, enabling interactive querying. In this way, our approach can be seen as a considerable extensions of previous approaches to proximity searching in structured data. With the help of tree signatures, it also has a very efficient implementation as needed for processing of large XML data collections available on the web. Results are confirmed by systematic experiments on the DBLP dataset. A possible application is outlined by the structure search on insurance records including performance evaluation.

Our future plans concern introducing even more flexibility or vagueness to the specification of expressions determining nodes. For example by considering alternative names, such as Author or Writer, which can be automatically chosen from proper dictionaries or lexicons. We are also working on developing of *ranking* mechanisms that would order or group the retrieved twigs according to their relevance with respect to the query. In this place, the analytic properties of tree signatures will play an indispensable role.

References

[ADR+03] G. Amato, F. Debole, F. Rabitti, and P. Zezula. YAPI: Yet another path index for XML searching. In *ECDL 2003, 7th European Conference on Research and Advanced Technology for Digital Libraries, Trondheim, Norway, August 17-22, 2003*, 2003.

[BKS02] N. Bruno, N. Koudas, and D. 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.

[CVZ+02] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, September, 2002, Morgan Kaufmann, 2002, pp. 263-274.

[CJL+03] Z. Chen, H.V. Jagadish, V.S. Lakshmanan, and S. Paparizos. From Tree patterns to Generalized Tree Patterns: On Efficient Evaluation of XWQuery. In *Proceedings of the 29th VLDB Conference*, September 2003, Berlin, Germany, pp. 237-248.

[Die82] P.F. Dietz. Maintaining Order in a Linked List, In *Proceedings of STOC, 14th Annual ACM Symposium on Theory of Computing*, May 1982, San Francisco, CA, 1982, pp. 122-127.

[DBLP] M.Ley. DBLP Bibliography http://dblp.uni-trier.de/xml/.

[GS98] R. Goldman and N. Shivakumar. Proximity Search in Databases. In *Proceedings 1998 VLDB Conference*, pp. 26-37, New York, USA.

[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.

[LM01] Q. Li and B. Moon, Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th International Conference on Very Large Data Bases, VLDB01* September 11-14, 2001, Roma, Italy, Morgan Kaufmann, 2001, pp. 361-370.

[SKW01] A. Schmidt, M. Kersten, and M. Windhouwer Querying XML Documents Made Easy: Nearest Concept Queries. Proceedings of the *17th International Conference on Data Engineering*, April 02 - 06, 2001 Heidelberg, Germany, IEEE, pp. 321-329.

[SAJ+02] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu, Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engeneering, ICDE 2002*, March 2002, San Jose, California, 2002, pp. 161-171.

[ZAD03] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree Signatures for XML Querying and Navigation. In *Proceedings of the XML Database Symposium, XSym 2003*, Berlin, September 2003, LNCS 2824, Springer, pp. 149-163.

[ZAR03] P. Zezula, G. Amato, and F. Rabitti. Processing XML Queries with Tree Signatures. In *Intelligent Search on XML Data, Applications, Languages, Models, Implementations, and Benchmarks.*, 2003, LNCS 2818, Springer, pp. 247-258.

[ZMM04] P. Zezula, F. Mandreoli, R. Martoglia. Tree Signatures and Unordered XML Pattern Matching. In *Proceedings of the 30th SOFSEM Conference on Current Trends in Theory and Practice of Informatics.*, Czech Republic, January 2003, LNCS 2932, Springer, pp. 122-139.

[ZND+01] C., Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. M. Lohman, On Supporting Containment Queries in Relational Database Management Systems. In *ACM SIGMOD Conference 2001*: Santa Barbara, CA, USA, ACM-Press, 2001.