Relevance measures for XML information retrieval

Olli Luoma

Department of Information Technology
University of Turku, FIN-20014, Finland
E-mail: olli.luoma@it.utu.fi

Abstract: In recent years, a lot of work has been carried out to develop efficient methods for storing and querying XML data. Most of the proposals have approached the subject from the database point of view, *i.e.*, they have primarily aimed at providing exact matching capabilities. The problem can, however, also be addressed as an information-retrieval problem, which obviously introduces some challenges, such as the need for relevance ranking. The vast majority of the previous proposals have based the ranking primarily on content and, furthermore, if structural properties were taken into account, only containment relationships have been considered. In this paper, we focus on ranking the results based on their structural properties and aim at supporting a wide range of structural operations, such as operations based on preceding/following relationships. Our method is based on a fuzzy interpretation of the XPath query language which is also discussed in this paper. Finally, we discuss a relational implementation of our model and present the results of our experiments.

Keywords: information retrieval; semistructured documents; XML; relevance ranking.

Reference to this paper should be made as follows: Luoma, O. (2007) 'Relevance measures for XML information retrieval', *Int. J. Web and Grid Services*, Vol. 3, No. 2, pp.170–193.

Biographical notes: Olli Luoma is currently working as a Researcher at the University of Turku. His research interests include XML databases, XML information retrieval, XML compression and data mining.

1 Introduction

XML (W3C, 2006), a markup language recommended by the World Wide Web Consortium (W3C), plays a major role in many modern applications, and thus several methods for managing XML data have been proposed. The previous proposals, however, have primarily approached the issue as a database problem, and thus they have aimed at providing exact matching capabilities. As a consequence, they are based on the assumption that the user is fully aware of the structure of the documents. In an information retrieval setting, however, this is often not true. For example, consider the XML excerpts in Figure 1, which contain information about customers, orders and products. The user is trying to find information on customers who ordered the product of the name 'xyz', so (s)he writes a query to find all customer elements that contain

an order element that contains a product element that contains a name element that contains the word 'xyz'. In terms of XPath (W3C, 2007), a query language developed for querying XML documents, this is expressed as `//customer[//order//product//name='xyz']`.

Figure 1 Examples of different documents describing similar information

```
<customer>                              <product id="1">
 <name>Orimatti Teuras</name>            <name>xyz</name>
 <address>Teuraantie 100</address>       <price>100</price>
 <orders>                                <orders>
  <order  id="2" date="01012006">         <order  id="2" date="01012006">
   <product id="1">                        <customer>
    <name>xyz</name>                        <name>Mikki Hiiri</name>
    <price>100</price>                      <address>Ankkalinna 313</address>
   </product>                              </customer>
  </order>                                 </order>
 </orders>                                </orders>
</customer>                             </product>
```

In a database setting, the answer to this query is clear. Only the left excerpt matches the query; the right excerpt is rejected because the `product` element, for example, does not appear between `<customer>` and `</customer>` tags. However, one could argue that both excerpts should be returned but the right one should just be given a lower relevance rank. In many previous XML information-retrieval proposals, this would have been impossible since they based their ranking on the content of the documents and often interpreted the structural part of the query strictly. In this paper, we therefore aim at defining relevance measures for structural ranking. In summary, this paper contributes to the XML information-retrieval research as follows:

- We formulate a fuzzy interpretation of XPath queries. More accurately, we propose a method for calculating the relevance of any XML element with respect to an XPath query.
- In order to support structural ranking, we propose relevance measures for all 12 XPath axes, such as `ancestor`, `descendant`, `preceding`, `following`, `parent` and `child`. This sets our approach apart from previous proposals, which have considered only containment relationships among the elements, *i.e.*, the `descendant` axis. Furthermore, our proposal differs from the previous work in the sense that our relevance measures are based on simple geometric properties of the preorder and postorder ranks of the nodes in a tree rather than, for example, computationally expensive tree edit distances.
- We describe how our structural relevance measures can also be used to rank the results based on their content.
- Our proposal can be implemented efficiently on different platforms. To exemplify this, we describe an implementation based on a relational database.

The paper proceeds in the following manner. In Section 2, we review the related work and in Section 3, we briefly discuss the basics of the XPath query language. Section 4 discusses a fuzzy interpretation of XPath queries and Section 5 presents the similarity measures for XPath operations, such as axes, node tests and content tests. Section 6 discusses a relational implementation of our ideas and Section 7 presents the results of our experimental evaluation. Section 8 concludes this article and discusses our future work.

2 Related work

In recent years, XML has gained tremendous popularity and many methods for managing XML documents have therefore been proposed (Lee *et al*., 1996; Shanmugasundaram *et al*., 1999; Luoma, 2005a; 2005b). Early examples, such as BitCube (Poon *et al*., 2001) and Signature File Hierarchy (Chen and Aberer, 1998), typically aim at retrieving whole documents rather than parts of them. The main intuition behind BitCube, for example, is to generalise the idea of (*document*, *keyword*) bitmap into three dimensions to index (*document*, *keyword*, *path*) triplets. Since such an index can obviously be very large and sparse, the paper also describes a method for clustering the bitmap into smaller and denser cubes. This proposal is suitable only for finding all documents which contain certain paths, certain keywords, or certain keywords at the end of certain paths. Furthermore, whole documents must be returned as query results, and thus the practicability of this method is somewhat limited, especially if we have to deal with large XML documents.

It is interesting to notice, however, that both BitCube and Signature File Hierarchy can be used to support coordinate queries, and thus they can be used to rank query results according to their relevances. When retrieving all `record` elements which contain the words 'John' and 'Scofield', for example, we could first return all documents containing a record element which actually contains the words, then documents containing elements with other tags which contain the words or `record` elements which contain either 'John' or 'Scofield', *etc.* Signature File Hierarchy, on the other hand, provides access to individual elements. However, since both BitCube and Signature File Hierarchy only index (*element*, *keyword*) pairs, queries like 'find all record elements which contain a name element' cannot be supported without considerable extensions.

Recent XML information-retrieval proposals have typically aimed at more sophisticated relevance-ranking capabilities and greater granularity by providing access to parts of the documents (Kotsakis, 2002; Hatano *et al.*, 2002; Fuhr *et al.*, 2002; Schlieder and Meuss, 2002; List *et al.*, 2003; Weigel *et al.*, 2005). In many of these, an element is considered an individual document, which is then indexed using traditional information-retrieval methods for text documents, such as the $tf \cdot idf$ model (Salton, 1971). In our view, however, they more or less ignore the problem of matching the structural part of the queries; and even if it is considered, only containment relationships are taken into account. Since XML documents can be represented as trees, we can view these proposals as attempts to solve a fuzzy version of the tree inclusion problem (Kilpeläinen, 1992). A good example of this is the proposal of Weigel *et al*. (2005), in which the subtrees were considered *structured terms*, *i.e.*, flat (nonhierarchical) representations of trees, which were then indexed using traditional information-retrieval methods. Amer-Yahia *et al.* (2005) aimed at supporting both containment and direct

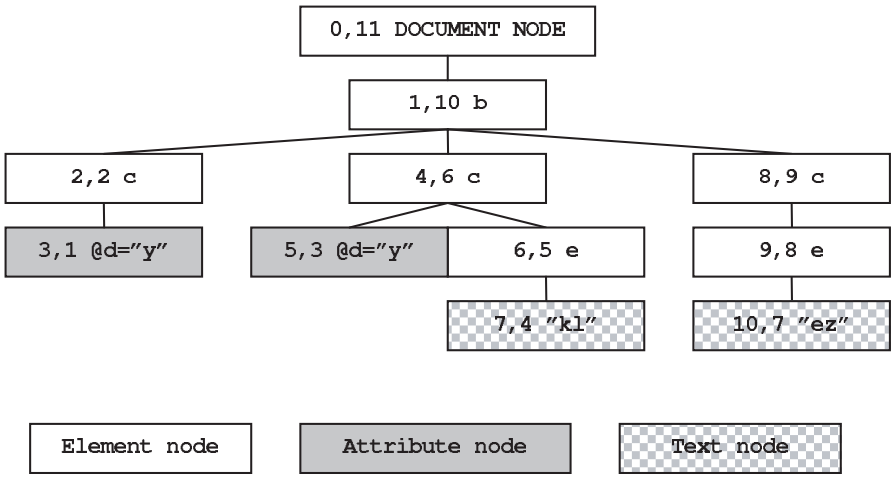
containment by proposing a method for relaxing the structural conditions. For example, if we are searching for `record` elements which directly contain the word 'Scofield', we get `record` elements which directly or indirectly contain 'Scofield'. In the latter case, the results are obviously assigned a lower relevance.

Restriction of tree inclusion is also evident in the field of XML information retrieval in general. INEX (Initiative for XML Information Retrieval),[1] for example, lists two tasks: the Content Only task (CO) and the Content and Structure task (CAS). In CO, only content is queried, whereas in CAS, some simple restrictions can be set to the structure of the returned elements. For example, if one is to retrieve all `record` elements which contain the words 'John' and 'Scofield', only elements with tag `record` are returned; the ranking of the results is performed based on content. In other words, structural conditions are interpreted strictly, which can hardly be regarded as structural ranking. In this paper, we aim at providing more sophisticated structural ranking capabilities and supporting a wide range of structural operations. Our proposal is able to support queries like 'find all bids by person X that were made after a bid by person Y' or 'find all acts in a play which precede an act in which words "Hamlet" and "danger" appear'. Rather than being based on tree edit distances (Bille, 2005; Campi *et al*., 2006), our method is based on simple geometric properties of the nodes, which makes it both efficient and easy to implement.

3 XPath basics

According to XPath (W3C, 2007), every XML document can be represented as a partially ordered, labelled *XML tree* in which each element, attribute and text node corresponds to an element, attribute and piece of text in the document, respectively. The ancestor/descendant relationships of the nodes correspond to the nested relationships between elements, attributes and pieces of text. A simple example of an XML tree is presented in Figure 2. The nodes are numbered in both pre- and postorder, which encodes a lot of information on the structural relationships among the nodes (Grust, 2002).

Figure 2 An XML tree corresponding to the XML document `<b><c d="y"/><c d="y"><e>kl</e></c><c><e>ez</e></c></b>`



In addition to the tree representation of XML documents, the XPath recommendation also lists 12 *axes*, *i.e.*, operators for tree traversals, which are listed in Table 1. These axes are used in *location steps*, which start from a *context node* and result in a set of nodes that satisfy the conditions of the step; the document node is used as the initial context node. More specifically, a location step is of the form `/axis::test[predicate]`, where `axis` is one of the XPath axes and `test` is either a *node type test* or a *name test.* Optional predicates can be used to further restrict the result. An XPath query, then, is simply a sequence of location steps. For example, we might issue query `/descendant::*[descendant::e]` to select all descendants of the document node that have one or more descendant nodes with tag 'e'. In the case of the tree presented in Figure 2, for example, nodes (1,10), (4,6) and (8,9) satisfy the query.

Table 1 XPath axes and their semantics

Axis	*Semantics of n/axis*
`parent`	Parent of *n*
`child`	Children of *n*, no attribute nodes
`ancestor`	Transitive closure of `parent`
`descendant`	Transitive closure of `child`, no attribute nodes
`ancestor-or-self`	Like ancestor, plus *n*
`descendant-or-self`	Like descendant, plus *n*, no attribute nodes
`preceding`	Nodes preceding *n*, no ancestors or attribute nodes
`following`	Nodes following *n*, no descendants or attribute nodes
`preceding-sibling`	Preceding sibling nodes of *n*, no attribute nodes
`following-sibling`	Following sibling nodes of *n*, no attribute nodes
`attribute`	Attribute nodes of *n*
`self`	Node *n*

XPath also provides means for querying content using *string value tests.* For example, the XPath query `/descendant::*[descendant::e='ez']` selects all descendants of the document node which have a descendant node with tag 'e', such that the content of the descendant is 'ez'. In our example case, nodes (1,10) and (8,9) satisfy these conditions. More accurately, the XPath recommendation defines the string value of an element node as the concatenation of the string values of all text node descendants of the element node in document order.

4 A fuzzy interpretation of XPath

As discussed in the previous section, three kinds of conditions can be set using XPath: axis conditions, node tests and string value tests, *i.e.*, content tests. It should therefore be obvious that the relevance-ranking algorithm of an XML information-retrieval engine should be able to assign ranks for nodes based on these three factors. Before defining the concept of step relevance, let us first define *axis relevance, node test relevance* and *string value relevance.*

Definition 1 *Let N denote the set of nodes in an XML tree and let A denote the set of XPath axes. Any function* $r_{axis}(n_0, n_1, a) : N \times N \times A \rightarrow [0,1]$ *is an* axis relevance *for* $n_1 \in N$ *with respect to* $n_0 \in N$ *and* $a \in A$.

In simple terms, this function answers the following question: "To what extent is node n_1 an ancestor, descendant, preceding sibling, following sibling, parent, child, *etc.* of node n_0, *i.e.*, the context node?" Notice that our definition is very general and any function that fills the requirements qualifies as an axis relevance function. To obtain relevant results, however, the function obviously has to be selected carefully. One possibility of defining relevance functions for the axes is discussed in Section 5.

Definition 2 *Let N denote the set of nodes in an XML tree and let T denote the set of node tests. Any function* $r_{test}(n,t) : N \times T \rightarrow [0,1]$ *is a* node test relevance *for* $n \in N$ *with respect to* $t \in T$.

The definition is again very broad, but it is rather easy to select a function to measure node test relevances. We could, for example, assign value 1 for cases in which a node satisfies the node test and 0.1 for other cases.

Definition 3 *Let N denote the set of nodes in an XML tree and let C denote the set of string value tests. Any function* $r_{content}(n, c) : N \times C \rightarrow [0,1]$ *is a* string value relevance *for* $n \in N$ *with respect to* $c \in C$.

This function could be defined by indexing the content of each element using traditional information-retrieval concepts, such as the $tf \cdot idf$ model (Salton, 1971). However, since the string value of a node is a concatenation of the text node descendants of the node (W3C, 2007), string value tests can be regarded as descendant tests, *i.e.*, axis tests. The next section therefore discusses an alternative relevance measure for string values which utilises axis relevances. Finally, we use the concepts of axis, node test and string value relevances to define the step relevance as follows:

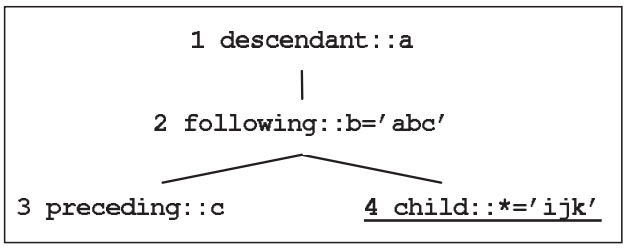
Definition 4 *Let N, A, T and C denote the same sets as before. Function* $r_{step}(n_0, n_1, s) : N \times N \times (A \times T \times C) \rightarrow [0,1]$ *is the* step relevance *for* $n_1 \in N$ *with respect to* $n_0 \in N$ *and location step s, consisting of* $a \in A$, $t \in T$ *and* $c \in C$ *starting from* $n_0 \in N$, *defined as* $r_{step}(n_0, n_1, s) = r_{axis}(n_0, n_1, a) \cdot r_{test}(n_1, t) \cdot r_{content}(n_1, c)$.

In simple terms, the step relevance is defined as a product of axis, node test and string value relevances. This function could also have been defined as a minimum of relevances or some other t-norm. Notice that our model is flexible enough to support many different approaches. The INEX CAS task, for example, can be regarded as a case in which the node test relevance is defined as a binary function, *i.e.*, nothing is retrieved if any of the node tests in a query does not match with any node in an XML tree. In the INEX CO family of methods, on the other hand, the node test relevance is always 1.

Armed with these concepts, we are ready to define the relevance for a node with respect to a query. An XPath query can always be represented as a *query tree*, in which each node corresponds to a location step in the query and branches correspond to predicates. Furthermore, every query tree has exactly one *active step*, *i.e.*, a step which

corresponds to the node set that is finally returned. An example of a query tree is presented in Figure 3; the active node is underlined. We assign a unique identifier to all nodes in the query tree; the nodes are numbered in preorder.

Figure 3 A query tree corresponding to `/descendant::a/following::b='abc' [preceding::c]/child::*='ijk'`



It is now important to notice that each step in a query is performed starting from the nodes resulting from the step corresponding to the parent of the step in the query tree. To denote the identifier of the parent of step s in the XPath query tree, we use $par(s)$; if s is the root, $par(s) = 0$. The following definition, then, defines the relevance of a multiset of XML tree nodes with respect to an XPath query using n_0 as an initial context node.

Definition 5 *Let $N = \{n_1, n_2, ..., n_l\}$ denote an ordered multiset of nodes in an XML tree and $S = \{s_1, s_2, ..., s_l\}$ an ordered set of location steps in an XPath query tree. Function $r_{query}(N, S)$ is the* query relevance *for N with respect to S, defined as* $r_{query}(N,S) = \prod_{i=1}^{l} r_{step}(n_{par(s_i)}, n_i, s_i)$.

The relevance of an individual node n with respect to a query, then, is simply defined as $r_{query}(N, S)$, such that n is the node in N which corresponds to the active location step. In other words, we are interested in finding a multiset of nodes such that performing the steps in a query via these nodes produces the maximum relevance with respect to the query; notice that one single node can appear in this multiset multiple times. However, given context node n_0, finding the most relevant node with respect to n_0 and a query consisting of location steps $s_1, s_2, ..., s_l$ can obviously involve checking $|N|^l$ node sequences, and thus we need to apply some heuristics in order to perform queries efficiently. For example, if the user is interested in finding the ten most relevant nodes, we could approximate the process and only take the 100 or 1000 most relevant nodes into account when performing the steps. After the final step, we would return the ten most relevant nodes.

5 Relevance measures for XPath

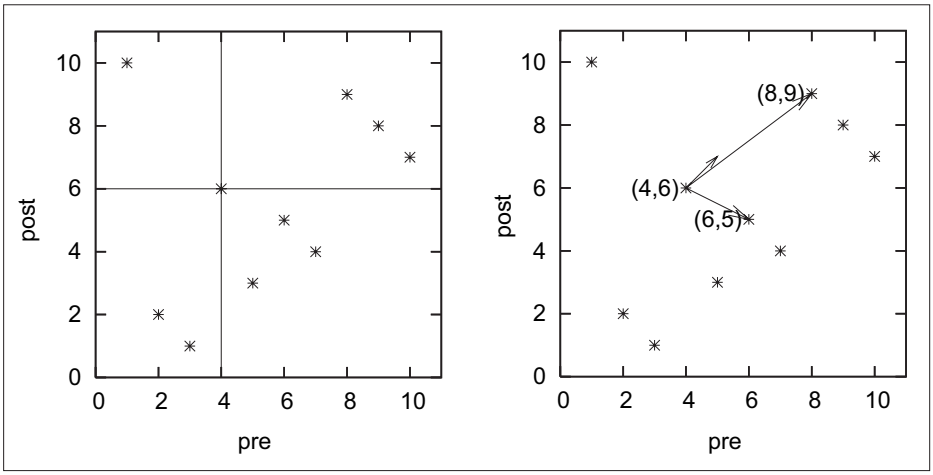
5.1 Relevance measures for the XPath axes

Four of the XPath axes, namely `ancestor`, `descendant`, `preceding` and `following`, can be regarded as the *major axes*. This is because of the fact that these axes partition the nodes of an XML tree into four disjoint partitions according to the following observation (Grust, 2002):

Proposition 1 *Let N denote the set of nodes in an XML tree and let pre(n_i) and post(n_i) denote pre- and postorder numbers of node n_i, respectively. For any two nodes n_0, $n_1 \in N$, $n_1 \in n_0$/`ancestor::*` iff pre(n_1) < pre(n_0) and post(n_1) > post(n_0), $n_1 \in n_0$/`descendant::*` iff pre(n_1) > pre(n_0) and post(n_1) < post(n_0), $n_1 \in n_0$/`preceding::*` iff pre(n_1) < pre(n_0) and post(n_1) < post(n_0), and $n_1 \in n_0$/`following::*` iff pre(n_1) > pre(n_0) and post (n_1) > post(n_0).*

This partitioning is illustrated on the left side of Figure 4, which shows the partitioning created by the axes among the nodes of the XML tree in Figure 2 using the node (4,6) as the context node.[2]

Figure 4 Examples of partitioning (left) and measuring the following relevance (right) with respect to node (4,6)



The same observation also gives us an idea for defining the relevances for the major axes. One can rather easily see that the descendants of any context node tend to be located in direction $\overrightarrow{(1,-1)}$. Similarly, the ancestors, predecessors and followers can, in general, be found in directions $\overrightarrow{(-1,1)}$, $\overrightarrow{(-1,-1)}$ and $\overrightarrow{(1,1)}$, respectively. We can therefore determine the `following` relevance, for example, for node n_1 with respect to n_0 by measuring the angle between $\overrightarrow{(1,1)}$ and the vector from ($pre(n_0)$, $post(n_0)$) to ($pre(n_1)$, $post(n_1)$). This is illustrated on the right side of Figure 4, which shows the direction $\overrightarrow{(1,1)}$ and the vector from node (4,6) to node (8,9) as well as the vector from node (4,6) to node (6,5). The angle between $\overrightarrow{(1,1)}$ and the vector from (4,6) to (8,9) is smaller than the angle between (1,1) and the vector from (4,6) to (6,5), and thus we can say that node (8,9) is a follower of node (4,6) to a greater extent than node (6,5).

More formally, the relevance for node n_1 with respect to node n_0 and the `following` axis is calculated using the formula shown:

$$r_{axis}(n_0, n_1, \texttt{following}) = \begin{cases} \dfrac{1 + cos(\overrightarrow{\alpha(n_0 n_1)}, \overrightarrow{(1,1)})}{2} & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1 \end{cases}$$

where $\overrightarrow{\alpha(n_0 n_1)}$ denotes the vector from ($pre(n_0)$, $post(n_0)$) to ($pre(n_1)$, $post(n_1)$), $cos(\overrightarrow{v_1}, \overrightarrow{v_2})$ denotes the cosine of the angle between vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$, and $\varepsilon_a \in [0,1]$ is a constant used to define the extent to which a node is considered to be a follower of itself. The cosine can be calculated using the dot product as follows:

$$cos(\overrightarrow{\alpha(n_0 n_1)}, \overrightarrow{(1,1)}) = \frac{\overrightarrow{\alpha(n_0 n_1)} \cdot \overrightarrow{(1,1)}}{\left|\overrightarrow{\alpha(n_0 n_1)}\right| \cdot \left|\overrightarrow{(1,1)}\right|}$$

The similarities for the other major axes can also be defined using their general directions. Notice that the `ancestor` and `descendant` axes, as well as the `preceding` and `following` axes, can be regarded as opposites, and thus the `preceding` similarity, for example, can alternatively be defined as:

$$r_{axis}(n_0, n_1, \texttt{preceding}) = \begin{cases} r_{axis}(n_1, n_0, \texttt{following}) & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1. \end{cases}$$

The `ancestor-or-self` and `descendant-or-self` axes are similar to their major axis counterparts with the exception that a node has much greater relevance with respect to itself; in this case, the similarity is defined as 1. Thus, the `ancestor-or-self` relevance, for example, is defined as follows:

$$r_{axis}(n_0, n_1, \texttt{ancestor-or-self}) = \begin{cases} r_{axis}(n_0, n_1, \texttt{ancestor}) & \text{if } n_0 \neq n_1 \\ 1 & \text{if } n_0 = n_1. \end{cases}$$

Relevances for other minor axes, *i.e.*, `parent`, `child`, `preceding-sibling`, `following-sibling`, `attribute` and `self`, have to be defined using a somewhat different approach. In the case of `parent` and `child` relevances, we extend the pre-/postorder plane with a third dimension, namely the level or the height of the nodes. We define the level of root of the tree as 1, the level of the children of the root as 2, *etc.* Again, the relevance is defined using the cosine of the angle between two vectors but now the vectors are three-dimensional. As the vector for the general direction of the `parent` axis, $\overrightarrow{(-1,1,-1)}$ is used. In other words, the parent of a node is an ancestor of the node and is located one level lower in the tree. The `parent` relevance is therefore calculated using the following formula:

$$r_{axis}(n_0, n_1, \texttt{parent}) = \begin{cases} \dfrac{1 + cos(\overrightarrow{\beta(n_1 n_0)}, \overrightarrow{(-1,1,-1)})}{2} & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1 \end{cases}$$

where $\overrightarrow{\beta(n_0 n_1)}$ serves as a denotation for the vector from ($pre(n_0)$, $post(n_0)$, $level(n_0)$) to ($pre(n_1)$, $post(n_1)$, $level(n_1)$). The relevance for the `child` axis, then, can be defined as:

$$r_{axis}(n_0, n_1, \texttt{child}) = \begin{cases} r_{axis}(n_0, n_1, \texttt{parent}) & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1. \end{cases}$$

We have to use a slightly different representation of the nodes to evaluate the relevances for the `preceding-sibling` and `following-sibling` axes. In this case, node n is uniquely identified as a triplet ($pre(par(n))$, $post(par(n))$, $ord(n)$) where $par(n)$ denotes the parent of node n and $ord(n)$ the order number of node n among its siblings. As the parent of the root node, the document node of the XPath recommendation can be used. It is obvious that all siblings share the same parent, and thus $\overrightarrow{(0,0,-1)}$ can be used as a general direction for the `preceding-sibling` axis and $\overrightarrow{(0,0,1)}$ for the `following-sibling` axis. The following formula is used as a relevance measure for the `preceding-sibling` axis:

$$r_{axis}(n_0, n_1, \texttt{preceding-sibling}) = \begin{cases} \dfrac{1 + cos(\overrightarrow{\gamma(n_0 n_1)}, \overrightarrow{(0,0,-1)})}{2} & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1 \end{cases}$$

where $\overrightarrow{\gamma(n_0 n_1)}$ denotes the vector from ($pre(par(n_0))$, $post(par(n_0))$, $ord(n_0)$) to ($pre(par(n_1))$, $post(par(n_1))$, $ord(n_1)$). Furthermore, since these axes can be regarded as opposites, we can define the `following-sibling` relevance as:

$$r_{axis}(n_0, n_1, \texttt{following-sibl.}) = \begin{cases} r_{axis}(n_1, n_0, \texttt{preceding-sibl.}) & \text{if } n_0 \neq n_1 \\ \varepsilon_a & \text{if } n_0 = n_1. \end{cases}$$

In order to support all XPath axes, we still need to define relevance measures for the `self` and `attribute` axes. We could use Euclidean distance to define the `self` relevance, but we can also simply remove the location steps involving the `self` axis. The `attribute` axis, on the other hand, can be treated as a combination of `child` axis, node type test and name test; the node test relevance measures are presented in the following section. Table 2 exemplifies our relevance measures using the XML tree presented in Figure 2; node (4,6) was used as the context node and value 0.1 for ε_a.

Table 2 Examples of axis relevances with respect to node (4,6)

	Node									
Axis	*(1,10)*	*(2,2)*	*(3,1)*	*(4,6)*	*(5,3)*	*(6,5)*	*(7,4)*	*(8,9)*	*(9,8)*	*(10,7)*
parent	0.95	0.37	0.22	0.10	0.06	0.03	0.04	0.44	0.29	0.18
child	0.05	0.63	0.78	0.10	0.94	0.97	0.96	0.56	0.71	0.82
ancestor	0.99	0.34	0.22	0.10	0.05	0.03	0.01	0.43	0.30	0.21
descendant	0.01	0.66	0.78	0.10	0.95	0.97	0.99	0.57	0.70	0.79
preceding	0.42	0.97	0.91	0.10	0.72	0.34	0.40	0.01	0.04	0.09
following	0.58	0.03	0.09	0.10	0.28	0.66	0.60	0.99	0.96	0.91
prec-sibl	0.79	1.00	0.56	1.10	0.60	0.50	0.57	0.00	0.57	0.56
foll-sibl	0.21	0.00	0.44	0.10	0.40	0.50	0.43	1.00	0.43	0.44
attribute	0.00	0.06	0.78	0.05	0.94	0.10	0.0	0.06	0.07	0.08

Although the axis relevances were defined using simple cosine functions, they actually work rather well. As an interesting detail in Table 2, one might notice that the `child` relevance of node (4,6) with respect to node (7,4) is rather large, 0.96, although node (7,4) is actually a grandchild of node (4,6). In general, our method tends to exaggerate the `parent` and `child` relevances especially in the case of small documents, *i.e.*, when the nodes have a small number of descendants compared to the number of ancestors.

Although our relevance measures do not have metric properties, which rules out the use of M-tree indexes (Ciaccia *et al*., 1997), it should be rather easy to design an index to avoid unnecessary axis relevance computations. One possibility is to assign each node n to a partition $q(n) = \left(\left\lfloor \frac{pre(n)}{p} \right\rfloor, \left\lfloor \frac{post(n)}{p} \right\rfloor, level(n) \right)$ using a relatively large p. We can then calculate the axis relevances between these disjoint partitions similarly as we did with individual nodes, store these values, and use them as estimates for axis relevances. For example, the estimated relevance for n_1 with respect to n_0 and the `ancestor` axis would therefore be defined as follows:

$$r_{estimate}(n_0, n_1, \texttt{ancestor}) = \begin{cases} \dfrac{1 + cos(\overrightarrow{\alpha(q(n_0)q(n_1))}, \overrightarrow{(-1,1)})}{2} & \text{if } q(n_0) \neq q(n_1) \\ r_{axis}(n_0, n_1, \texttt{ancestor}) & \text{if } q(n_0) = q(n_1). \end{cases}$$

Using this approach, the axis relevances are computed only for nodes residing in the same partition with the context node; for nodes in other partitions, the precomputed estimates are used. By using the preorder and postorder numbers of their parents and the order number of the nodes as partitioning criteria, the relevances for `preceding-sibling` and `following-sibling` axes can be indexed similarly.

One could also take advantage of some well-known multidimensional XML indexing approaches based on R-trees (Grust, 2002) or UB-trees (Krátký *et al*., 2004). In this case, the relevance with respect to `descendant` axis, for instance, could be defined simply as 1 for the nodes which are actual descendants of the context node, 0.5 for the followers and predecessors, and 0 for the nodes which are ancestors of the context node. However, one could argue that node (10,7) of the tree in Figure 2, for example, is a descendant of node (4,6) to a greater extent than node (8,9), since (10,7) is a descendant of the immediately following sibling of (4,6) whereas (8,9) is the immediately following sibling. Nevertheless, if we were to rely on the simple definition of `descendant` axis relevance, both nodes would have a relevance of 0.5. Our method based on the cosine measure, on the other hand, gives a relevance of 0.79 for (10,7) and 0.57 for (8,9).

5.2 Relevance measures for node tests

As discussed in the previous section, defining relevance functions for the node tests is rather easy. We define the relevance for node n with respect to node test t as follows ($\varepsilon_t \in [0,1]$ can be used to tune the importance of node tests):

$$s_{test}(n, t) = \begin{cases} 1 & \text{if } n \text{ matches } t \\ \varepsilon_t & \text{if } n \text{ does not match } t. \end{cases}$$

We could actually formulate a more accurate relevance measure for node type tests. For example, element and attribute nodes could be regarded as more similar than element and comment nodes. We could also take the semantics of different tags into account, use edit distances to allow spelling errors, *etc.* In this paper, however, we rely on the simple definition.

5.3 Relevance measures for content conditions

It is interesting to notice that our structural relevance measures can also be used to support queries involving content conditions. The basis for this is again the XPath recommendation (W3C, 2007), which defines the string value of an element node as follows: "The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order".

To evaluate string value tests as descendant queries, we split text nodes containing many words into multiple nodes that all contain only one word. After this, we can evaluate our example query `/descendant::*='kl ez'` by checking which descendants of the document node have descendant text nodes 'kl' and 'ez'. More precisely, the query is transformed into `/descendant::*[descendant-or-self::text()='kl'] [descendant-or-self::text()='ez']`. We assign value 1 for the cases in which the value matches the string value test and value $\varepsilon_c \in [0,1]$ for other cases. Obviously, some standard information-retrieval methods, such as stemming, can be applied before indexing the text nodes.

Using the axis, node test and string value test relevances discussed in this section, we now take a look at how some example queries match with nodes of the XML tree in Figure 2. Our query set is presented in Table 3 and the relevances of the nodes with respect to these queries in Table 4. In the case of Q1, for example, nodes (6,5) and (9,8) actually belong to the result of the query, and thus they are assigned large relevances, 0.92 and 0.96, respectively. Nodes (4,6), (7,4), (8,9) and (10,7) are also somewhat relevant since they follow a node with label 'c'. Value 0.1 was used for ε_a and value 0.5 for ε_t and ε_c.[3]

Table 3 Query examples

#	*Query*
Q1	`/descendant-or-self::c/following::e`
Q2	`/descendant-or-self::c[attribute::d="y"]`
Q3	`/descendant-or-self::c[attribute::d="x"]`
Q4	`/descendant-or-self::*="ez"`
Q5	`/descendant-or-self::*="kl ez"`

Table 4 Relevances with respect to query examples

	Node									
#	*(1,10)*	*(2,2)*	*(3,1)*	*(4,6)*	*(5,3)*	*(6,5)*	*(7,4)*	*(8,9)*	*(9,8)*	*(10,7)*
Q1	0.37	0.09	0.12	0.45	0.22	0.92	0.44	0.50	0.96	0.46
Q2	0.48	0.92	0.11	0.93	0.12	0.31	0.19	0.62	0.26	0.21
Q3	0.23	0.46	0.06	0.47	0.06	0.16	0.10	0.31	0.13	0.11
Q4	0.87	0.56	0.25	0.74	0.27	0.63	0.50	0.84	0.86	0.24
Q5	0.79	0.36	0.14	0.66	0.17	0.57	0.13	0.61	0.54	0.12

6 Relational implementation

6.1 Relational schema

To put our ideas to the test, we implemented a simple XML information-retrieval engine on top of a relational database. In many proposals which aim at providing exact matching capabilities (Grust, 2002; Luoma, 2005a; 2005b; 2006), a single relation was used to store both structure and content, but in order to support efficient string value matching, we store the nodes using relation `Node` and the keywords, *i.e.*, individual words appearing in the text nodes, using the relation `Word`.

```
Node(Pre, Post, Level, Order, Par, Type, Name)
Word(Pre, Order, Word)
```

In the relation Node, the database attributes `Pre`, `Post` and `Level` correspond to the preorder number, the postorder number and the level of the node, respectively. The database attribute `Order` corresponds to the order number of the node among its siblings, and `Par` to the preorder number of the parent of the node. Finally, the database attributes `Type` and `Name` correspond to the type and name of the node, respectively. In the relation `Word`, the database attributes `Pre`, `Order` and `Word` correspond to the preorder number of the text node, the number of the word among the words in the node, and the keyword itself, respectively. For simplicity, we identify attribute nodes as element nodes; the values of the attributes are treated as text nodes. The underlined database attributes serve as the primary key in both relations.

In order to perform XPath queries one step at a time, we also need a relation to store the intermediate results. To do this, we employ the relation `Result`:

```
Result(Step, Context, Pre, Relevance)
```

In this relation, the database attribute `Step` corresponds to the preorder number of the step in the query tree, and `Context` to the preorder number of the context node from which the step was performed. `Pre` corresponds to the preorder number of the resulting node, and the database attribute `Relevance` to the relevance of node `Pre` with respect to node `Context` and step `Step`.

6.2 SQL queries for performing the steps

The XPath-to-SQL query translation of our prototype system is based on the algorithms described in this section. Algorithm **evaluate**(*S*) issues the SQL queries corresponding to query tree *S*. As mentioned earlier, the document node is used as the initial context node, and thus the algorithm starts by issuing an SQL query in order to insert the document node into the `Result` table with relevance 1. After this, the algorithm traverses the query tree in preorder and issues two SQL queries for each location step in the tree. The first of these queries inserts the context nodes, *i.e.*, the nodes stored in the table `Result`, and their relevances with respect to themselves into the `Result` table. Since the document node is not in the result of any of the axes, the first query is not performed during the first iteration. The second query calculates and stores the relevances of other nodes with respect to the context nodes. After all of the nodes in the query tree have been processed, a query generated by algorithm **genRankSQL**(*S*) is issued to actually rank the nodes. Algorithm **id**(*s*) returns the id of step *s* in a query tree:

evaluate(S)
IN: Query tree S
OUT: Set of nodes sorted according to their relevance with respect to S
  issue("`INSERT INTO Result VALUES (0, -1, 0, 1)`")
  for each $s \in S$ do
    if (id(s)> 1) then
      issue(genEvalSQL(s, true))
    issue(genEvalSQL(s, false))
  issue(genRankSQL(S))

Algorithm **genEvalSQL**(s, f) simply generates the SELECT, FROM and WHERE parts of the SQL query corresponding to location step s; the boolean value f is used to express whether we are inserting the context nodes themselves or other nodes:

genEvalSQL(s, f)
IN: Location step s, boolean f
OUT: SQL query corresponding to s
  return "`INSERT INTO Res`" +
  genEvalSELECT(s, f) +
  genEvalFROM(s, f) +
  genEvalWHERE(s, f)

A very straightforward algorithm, **genEvalSELECT**(s, f), is used to generate the SELECT parts of the queries:

genEvalSELECT(s, f)
IN: Location step s, boolean f
OUT: SELECT part of SQL query corresponding to s
  if (f) then
    return "`SELECT`" + id(s) + "`n.Pre, n.Pre,`" + genRel(s, f)
  else
    return "`SELECT`" + id(s) + "`c.Pre, n.Pre,`" + genRel(s, f)

Algorithm **genEvalFROM**(s, f) generates the right tuple variables into the FROM part. In general, this is rather simple; but if the axis of the step is either `preceding-sibling` or `following-sibling`, we need extra tuple variables `cp` and `np`, which correspond to the parents of the context nodes and resulting nodes, respectively. Furthermore, if the step involves a string value test, we need an extra tuple variable `w` to check the values of the nodes. Algorithm **axis**(s) simply returns the axis of step s and **valueTest**(s) returns the string value test involved in s; if there is no test the algorithm returns null. Algorithm **par**(s) returns the id of the parent of step s in a query tree; if s is the root, the algorithm returns as 0:

genEvalFROM(s, f)
IN: Location step s, boolean f
OUT: FROM part of SQL query corresponding to s
  result ← "`FROM Res r, Node n`"
  if (not f) then
    result ← result + "`, Node c`"
    if (axis(s) = preceding-sibling or axis(s) = following-sibling) then

result ← result + ", Node cp, Node np"
if (not valueTest(*s*) = null) then
result ← result + ", Word w"
return result

The WHERE parts of the SQL queries are generated using algorithm **genEvalWHERE**(*s*, *f*). Again, if the axis of the step is `preceding-sibling` or `following-sibling`, or the step involves a string value test, we need to generate extra join conditions to match tuple variables `cp`, `np` and `w`:

genEvalWHERE(*s*, *f*)
IN: Location step *s*, boolean *f*
OUT: WHERE part of SQL query corresponding to *s*
result ← "WHERE r.Step=" + par(*s*)
if (*f*) then
result ← result + "AND n.Pre=r.Pre"
else
result ← result + "AND c.Pre=r.Pre AND NOT n.Pre=c.Pre"
if (axis(*s*) = preceding-sibling or axis(*s*) = following-sibling)
then
result ← result + "AND cp.Pre=c.Par AND np.Pre=n.Par"
if (not valueTest(*s*) = null) then
result ← result + "AND w.Pre=n.Pre"
return result + ";"

Finally, the algorithm **genEvalRel**(*s*, *f*) generates the expression to calculate the relevances. In general, the axis relevance measures discussed earlier are calculated using tuple variables `c` and `n`, and thus we have omitted the code generating the axis relevances for the sake of brevity. For the `preceding-sibling` and `following-sibling` axes, however, we need to use `cp.Pre`, `cp.Post`, `np.Pre` and `np.Post` to determine the preorder and postorder numbers of the parents.

genEvalRel(*s*, *f*)
IN: Location step *s*, boolean *f*
OUT: Expression for the relevance of SQL query corresponding to *s*
if (not nodeTest(*s*) = null and not nodeTest(*s*) = "*") then
result ← result + "((l-ε_t)*(n.Name=\'" + nodeTest(*s*) + "\')+ε_t)*"
if (not valueTest(*s*) = null) then
result ← result + "((l-ε_c)*(w. Value=\'" + valueTest(*s*) + "\')
+ ε_c)*"
if (*f*) then
if (axis(*s*) = ancestor-or-self or axis(*s*) = descendant-or-self)
then
result ← result + "1"
else
result ← result + ε_a
else
Calculate the axis relevances as discussed earlier.
return result

Using these algorithms, the step `preceding::c` in the query tree presented in Figure 3 is transformed into the following two SQL queries:

```
INSERT INTO Result
SELECT 3, n.Pre, n.Pre, (0.5 * (n.Name='c') + 0.5) * 0.1
FROM Result r, Node n
WHERE r.Step=2
AND n.Pre=r.Pre;

INSERT INTO Result
SELECT 3, c.Pre, n.Pre, (0.5 * (n.Name='c') + 0.5) *
((1 + (n.Pre - c.Pre) * (-1) + (n.Post - c.Post)*(-1))/
(SQRT(2) * SQRT(POWER(n.Pre - c.Pre, 2) *
POWER(n.Post - c.Post, 2)))) / 2
FROM Result r, Node c, Node n
WHERE r.Step=2
AND c.Pre=r.Pre
AND NOT n.Pre=c.Pre;
```

6.3 SQL queries for ranking the nodes

After processing all the steps in the query, algorithm **genRankSQL**(*S*) is used to generate the query which ranks the nodes according to their relevances. Different algorithms are used to generate the SELECT, FROM and WHERE parts of the query. In simple terms, the algorithm constructs the XML tree node multisets by matching the database attribute `Step` with the identifier of the step that the tuple variable in question corresponds to and matching the preorder numbers of the context nodes with the preorder numbers of the nodes resulting from the parent step of the corresponding location step. Algorithm **active**(*S*) returns the identifier of the active step in *S*.

genRankSQL(*S*)
IN: Query tree *S*
OUT: SQL query to rank the nodes with respect to *S*
  genRankSELECT(*S*) +
  genRankFROM(*S*) +
  genRankWHERE(*S*) +
  "`GROUP BY r`" + active(*S*) "`.Pre`" +
  "`ORDER BY Relevance DESC;`"

genRankSELECT(*S*)
IN: Query tree *S*
OUT: SELECT part of SQL query to rank the nodes with respect to *S*
  result ← "`SELECT r`" + active(*S*) + "`.Pre, MAX(r1.Relevance`"
  for each $s \in S$ such that id(s)> 1 do
    result ← result + "`* r`" + id(s) + "`.Relevance`"
  return result + "`) AS Relevance`"

genRankFROM(*S*)

IN: Query tree *S*
OUT: FROM part of SQL query to rank the nodes with respect to *S*
  result ← "`FROM r1`"
  for each $s \in S$ such that id(*s*)> 1 do
    result ← result + "`, r`" + id(*s*)
  return result

genRankWHERE(*S*)
IN: Query tree *S*
OUT: WHERE part of SQL query to rank the nodes with respect to *S*
  result ← "`WHERE r1.Step=1 AND`"
  for each $s \in S$ do
    result ← result + "`r.Step`" + id(*s*)
    result ← result + "`AND r`" + id(*s*) + "`.Context=r`" + par(*s*) + "`.Pre`"
  return result

Using the SQL query generated by these algorithms, it is possible to obtain the final node ranks. For example, the final results of the query tree presented in Figure 3 can be determined using the following query:

```
SELECT r4.Pre, MAX(r1.Relevance * r2.Relevance *
r3.Relevance * r4.Relevance) AS Relevance
FROM Result r1, Result r2, Result r3, Result r4
WHERE r1.Step=1 AND r2.Step=2 AND r3.Step=3 AND r4.Step=4
AND r2.Context=r1.Pre AND r3.Context=r2.Pre AND
r4.Context=r2.Pre
GROUP BY r4.Pre
ORDER BY Relevance DESC;
```

6.4 Practical issues

With the exception of perhaps very small XML documents, it is obviously not practical to perform an exhaustive search, *i.e.*, to check all possible multisets of nodes in an XML tree. Thus, we need a way to prune the intermediate results. Most relational database management systems provide means for retrieving only the top of the result relation. In MySQL, for example, this is done by adding the limiting condition `LIMIT n`, which returns only the top *n* rows. This can be used to store, for example, only the best matches of each step to the relation `Result`. Obviously, we also need to add ORDER BY clauses to our queries to sort the nodes in descending order according to their relevances in order to obtain the most relevant nodes.

This, however, is not enough, since in order to locate the most relevant nodes with respect to a given context node and a step, the database management system still has to visit all nodes in the database. For this reason, we need to move the conditions used to calculate the node test and string value relevances from the fourth expression of the SELECT part to the WHERE part of the query. The expression (`0.5 * (n.Name='c') + 0.5`), for example, is thus removed from the SELECT part

and `n.Name='c'` is added to the WHERE part. Similarly, we can remove conditions corresponding to string value tests from the SELECT part and add them to the WHERE part. This greatly improves the query performance since the database management system only has to visit nodes which satisfy the node test or the string value test.

However, if no nodes satisfy the node test, we need means to quickly obtain some rather relevant nodes to act as context nodes for the next step. For this purpose, we can use the query conditions intended for exact matching (Grust, 2002; Luoma, 2005a; 2005b). Since we can now basically obtain any nodes with high axis relevance – after all, there were no nodes that satisfied the node test – we select some subsets of the nodes which belong to the result of the axis. More accurately, we perform `parent` instead of `ancestor` and `ancestor-or-self` axes, and `preceding-sibling` instead of `preceding` axis. Similarly, we perform `child` instead of `descendant` and `descendant-or-self`, and `following-sibling` instead of `following`. For example, if we cannot obtain any node satisfying a node test and we are dealing with a step involving the `following` axis, we simply add `AND n.Pre>c.Pre AND n.Par=c.Par` to the WHERE part of our query and multiply the relevance by 0.5, *i.e.*, by ε_n. This query can also be executed efficiently since the relevances are only computed for the following siblings rather than for all following nodes, let alone the whole node set. The other alternative, of course, is to define ε_t as 0. In this case, nothing is returned if one of the name tests in a query does not match any node in the database.

Analogously, we need to ensure that if there are no keywords which match the string value test, we still add some relevant nodes into the relation `Result`. Before performing a step which retrieves nodes matching a string value test, we thus always issue an SQL query to insert all context nodes into the relation `Result` with themselves with relevance ε_c. In a sense, this is indeed correct since every node has a string value and any string value matches any string value test at least with relevance ε_c. After this, we perform the SQL query, which retrieves the nodes which actually satisfy the string value tests.

7 Experimental results

We tested our relevance measures by implementing a simple XML information retrieval engine on top of a relational database, as described in Section 6. In simple terms, our prototype system traverses query trees in preorder and issues a series of SQL queries to retrieve and store the resulting nodes and their relevances for each step. After all steps have been processed, our system issues an SQL query to reconstruct the node multisets and computes the final relevances for the nodes. We conducted the experimental evaluation using a 2.00 GHz Pentium PC running Windows XP and MySQL Server 5.0. The PC was equipped with 512 MB of RAM and standard IDE disks. In order to support querying multiple documents within one database, we extended all relations in our database with document identifier Doc. The algorithms for generating the SQL queries were obviously extended accordingly, *i.e.*, we added conditions to match the document identifiers. Values 0.1, 0 and 0.5 were used for ε_a, ε_t and ε_c, respectively.

One should keep in mind, however, that our prototype system was implemented as a proof of concept rather than as a practicable XML information-retrieval engine. In order to support retrieval from massive data sets, less precise and more efficient methods should be applied. To this end, we feel that processing partitions instead of individual nodes as described earlier could greatly enhance query speed and yet provide sufficient search quality. Furthermore, our relevance measures should obviously be validated using some commonly agreed-on benchmark. However, all current benchmarks are focused on content-based retrieval, whereas our method is aimed at supporting retrieval based on structure, and thus there were no suitable benchmarks for testing our model.

7.1 Test set-up

As our test data, we used the complete works of Shakespeare marked up in XML.[4] This 7.5 MB collection consisted of 37 documents, 327 131 nodes and 887 660 keywords in total. Thus, an average document contained 8841 nodes and 23 991 keywords; the average depth of a node was 5.3. It is worth noticing that the documents were rather large compared to, for example, normal XHTML documents both in terms of kilobytes and number of nodes. In our evaluation, we used the queries presented in Table 5. This table also lists the number of nodes which were considered relevant with respect to each query.

Table 5 Query set

#	*Query*	*Nodes*
Q1	`/child::ACT/descendant::SCENE='Puck'/ preceding-sibling::SCENE`	2
Q2	`/descendant::ACT='Hamlet danger'`	47
Q3	`/descendant::ACT='Hamlet danger'/following::ACT`	77
Q4	`/descendant::SPEECH='murder Caesar'`	204
Q5	`/descendant::ACT/descendant::SPEECH`	155 140

In the case of Q1, no nodes satisfied the XPath query, since the root nodes in all documents are titled 'PLAY' rather than 'ACT'; nodes with label 'ACT' are actually children of 'PLAY' nodes. Nevertheless, we considered relevant the nodes with the name 'SCENE' which were preceding siblings of nodes with the name 'SCENE' and string value 'Puck'. All of these nodes resided in the play 'A Midsummer Night's Dream'. In the case of Q2, a node was considered relevant if its name was 'ACT' and one of its text node descendants contained either 'Hamlet' or 'danger'; relevant nodes for Q4 were defined similarly. A node was relevant with respect to Q3 if its name was 'ACT' and it followed a node which was relevant with respect to Q2. Finally, we considered a node relevant with respect to Q5 if it (in a strict sense) satisfied the XPath query `/descendant::ACT/descendant::SPEECH`. These sets of relevant nodes were used to study the precision and recall of our method.

7.2 Query times

We started by studying the effect of pruning with respect to query times. According to our initial experiments, only modest pruning can be applied to the context node sets without lowering search precision. As a rule of thumb, one can use node sets 50 or 100

times larger than the number of relevant nodes. However, node sets resulting from steps checking string values do not act as context nodes for any step, and thus they can be pruned rather aggressively. We studied three alternatives, retrieving and storing the 100 and the 1000 most relevant text nodes and retrieving all matching text nodes. One should notice that the methods discussed in Section 6.4 were used in order to make the string value and node test matching more efficient. These results are presented in Table 6. Obviously, the pruning affects not only the time consumed by the queries which compute and store the relevances (Query), *i.e.*, the SQL queries generated using algorithms presented in Section 6.2, but also the time needed to evaluate the query which finally ranks the nodes (Rank), *i.e.*, the SQL query generated using algorithms presented in Section 6.3.

Table 6 Query times in seconds

	100		*1000*		*All*	
#	*Query*	*Rank*	*Query*	*Rank*	*Query*	*Rank*
Q1	6.5	0.3	7.8	0.5	6.9	0.5
Q2	0.3	0.1	0.3	0.1	0.4	0.1
Q3	0.3	0.0	0.3	0.1	0.4	0.2
Q4	39.5	1.4	40.5	1.45	525.1	17.4
Q5	21.5	1.5	21.5	1.5	21.5	1.5

Overall, the queries were evaluated rather efficiently, and thus the pruning had quite a small effect on query times. An exception, of course, is Q4, which consumed an extremely long time, especially when pruning was not applied. This was actually expected since there were over 150 000 nodes with the name 'SPEECH', which all had to be retrieved and stored. In the case of Q4, pruning had a great effect, since instead of considering 370 000 text nodes, the system only had to check 200 or 2000 text nodes, *i.e.*, 100 or 1000 for 'murder' and 100 or 1000 for 'Caesar'. Similarly, the time needed to rank the nodes was considerably shorter with these smaller node sets. One should also notice that in the case of Q5, no text nodes were retrieved, and thus cases 100, 1000 and all share similar query times.

7.3 Search performance

We were also interested in finding out how the pruning affects search precision and recall. For each query Q1, Q2, Q3 and Q4, we selected $r/2$, r, $2r$ and $4r$ most relevant nodes, where r is the number of relevant nodes with respect to the query as presented in Table 5. Query Q5 was left out since it did not contain any string value tests, and thus no pruning took place. As one can notice in Tables 7–10, the effect of the pruning clearly depends on the number of relevant nodes or, more accurately, the number of text nodes which are relevant with respect to the string value tests in the query. In the case of Q1, Q2 and Q3, there was only a small number of relevant text nodes, and thus the results were very good even when the greediest pruning was applied. In the case of Q4, in which the number of relevant text nodes as well as the number of relevant nodes in general was larger, the pruning played a bigger role. Nevertheless, the results were quite good even

when only the 100 most relevant text nodes were taken into account. Furthermore, in the case of the 1000 most relevant text nodes, the search precision and recall were very close to the case in which no pruning was applied. The total query times with no pruning were ten times longer, and thus pruning less relevant text nodes is a worthwhile idea.

Table 7 Search performance with different degrees of pruning for Q1

	100		*1000*		*All*	
Nodes	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
r/2	1.00	0.50	1.00	0.50	1.00	0.50
r	1.00	0.50	1.00	0.50	1.00	0.50
2*r*	1.00	0.50	1.00	0.50	1.00	0.50
4*r*	1.00	0.50	1.00	0.50	1.00	0.50

Table 8 Search performance with different degrees of pruning for Q2

	100		*1000*		*All*	
Nodes	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
r/2	0.92	0.47	0.96	0.49	0.96	0.49
r	0.83	0.83	0.83	0.83	0.83	0.83
2*r*	0.50	1.00	0.50	1.00	0.50	1.00
4*r*	0.25	1.00	0.25	1.00	0.25	1.00

Table 9 Search performance with different degrees of pruning for Q3

	100		*1000*		*All*	
Nodes	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
r/2	1.00	0.50	1.00	0.50	1.00	0.50
r	0.97	0.97	0.97	0.97	0.97	0.97
2*r*	0.50	1.00	0.50	1.00	0.50	1.00
4*r*	0.25	1.00	0.25	1.00	0.25	1.00

Table 10 Search performance with different degrees of pruning for Q4

	100		*1000*		*All*	
Nodes	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*
r/2	0.65	0.33	0.60	0.30	0.62	0.31
r	0.46	0.46	0.63	0.63	0.64	0.64
2*r*	0.23	0.47	0.48	0.96	0.48	0.96
4*r*	0.13	0.51	0.25	1.00	0.25	1.00

Overall, we saw our relevance measures behave as expected and assign intuitively correct ranks for the nodes. In the case Q1, for example, our prototype system assigned a relevance of 0.73 for nodes that were actually considered relevant. A node which was in the result of the `preceding` axis rather than `preceding-sibling` axis was

ranked third with relevance 0.57. After this node our system ranked the 'SCENE' nodes, which were preceding siblings of some 'SCENE' node; these nodes had relevances smaller than 0.45. We also replaced the first axis in this query with `descendant` axis to see how the relevances grow if some nodes actually match the XPath query in the strict sense. In this case, the relevances of the first two nodes were 0.80 and the relevance of the third node, 0.62; the nodes were obviously in the same order as in the previous case. After this, the relevance of the next nodes dropped to 0.50.

8 Conclusion

This paper discussed relevance measures for XML information retrieval. In order to support XPath, an XML query language recommended by W3C, we first described a fuzzy interpretation of XPath and then moved on to discussing the actual relevance measures. Our main focus was on axis relevances, which were defined using simple cosine measures, but our relevance functions can also be used to support querying string values, *i.e.*, textual content. One of the strong points of our relevance measures is their simplicity. To exemplify this, we implemented a prototype system based on relational databases and described how an XPath query can be transformed into a series of SQL queries. A more efficient implementation could be achieved by partitioning the nodes of an XML tree into disjoint partitions and storing the axis relevances between these partitions. These stored relevances can then be used as approximations for axis relevances. Thus, the relevances actually have to be computed only for nodes in the same partition with the context node. Investigating this possibility will be an important part of our future work.

It is also easy to tune our model by changing the values of different parameters. Although our relevance measures are rather intuitive and easy to tune, we still feel that finding suitable values for ε_a, ε_t and ε_c requires relevance tests conducted with a large group of users (Lehtonen, 2006). There are also other interesting open issues concerning structural relevance measures. For example, is it correct to regard the `ancestor` and `descendant` or `parent` and `child` axes as opposites, or should we somehow group these axes together to form a symmetric relevance measure for 'nestedness'? For example, consider finding all products from brand 'X' in documents …`<product brand = "X">…</product>`… and …`<brand name="X"> <product>…</product>…</brand>`… In these documents, attribute 'brand' appears either as a parent or as a child of a 'product' node, and thus the result of the search depends on which axis the user chooses to use in his or her query. This problem could be avoided if we only consider the fact that in both cases, 'product' nests inside 'brand' or vice versa.

All in all, we feel that there is still room for many interesting discoveries in the field of XML information retrieval. For example, the very fundamental problem of measuring the search performance is still under discussion (Hiemstra and Mihajlović, 2005; Piwowarski and Dupret, 2006). Furthermore, since the result of a query can be thought of as a ranked set of subtrees rather than a ranked set of individual nodes, we may have to deal with overlapping results (Clarke, 2005). For example, if we have retrieved two subtrees s_1 and s_2 with relevances 1.0 and 0.5, respectively, such that s_2 is a part of s_1, should we return s_2 at all? After all, s_2 is returned as a part of s_1 with much higher relevance. It would also be interesting to study how taking the length or size of the

elements (Kamps *et al*., 2004) into account would affect our relevance measures, especially the string value relevance. Most importantly, however, we feel that there is a need for an XML information-retrieval benchmark, in which the possibility of querying based on structural relationships or axes is taken into account.

Acknowledgement

This work was supported by the Academy of Finland.

References

Amer-Yahia, S., Koudas, N., Marian, A., Srivastava, D. and Toman, D. (2005) 'Structure and content scoring for XML', in K. Böhm *et al*. (Eds.) *Proceedings of the 31st International Conference on Very Large Databases*, pp.361–372.

Bille, P. (2005) 'A survey of tree edit distance and related problems', *Theoretical Computer Science*, Vol. 337, Nos. 1–3, pp.217–239.

Campi, A., Guinea, S. and Spoletini, P. (2006) 'Fuzzy querying of semi-structured data', in N. Guimarães *et al*. (Eds.) *Proceedings of the 3rd IADIS International Conference on Applied Computing*, pp.241–248.

Chen, Y. and Aberer, K. (1998) 'Layered index structures in document database systems', in D.J. DeWitt *et al*. (Eds.) *Proceedings of the 7th International ACM Conference on Information and Knowledge Management*, pp.406–413.

Ciaccia, P., Patella, M. and Zezula, P. (1997) 'M-tree: an efficient access method for similarity search in metric spaces', in M. Jarke *et al*. (Eds.) *Proceedings of the 23rd International Conference on Very Large Databases*, pp.426–435.

Clarke, C.L.A. (2005) 'Controlling overlap in content-oriented XML retrieval', in R.A. Baeza-Yates *et al*. (Eds.) *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp.314–321.

Fuhr, N., Gövert, N. and Großjohann, K. (2002) 'HyREX: Hyper-media retrieval engine for XML', in M. Beaulieu *et al*. (Eds.) *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, p.449.

Grust, T. (2002) 'Accelerating XPath location steps', in M.J. Franklin *et al*. (Eds.) *Proceedings of the 2002 ACM SIGMOD Conference on Management of Data*, pp.109–120.

Hatano, K., Kinutani, H., Yoshikawa, M. and Uemura, S. (2002) 'Information retrieval system for XML documents', in A. Hameurlain *et al*. (Eds.) *Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pp.758–767.

Hiemstra, D. and Mihajlović, V. (2005) 'The simplest evaluation measures for XML information retrieval that could possibly work', in A. Trotman *et al*. (Eds.) *Proceedings of the INEX 2005 Workshop on Element Retrieval Methodology*, pp.6–13.

Kamps, J., de Rijke, M. and Sigurbjörnsson, B. (2004) 'Length normalization in XML retrieval', in M. Sanderson *et al*. (Eds.) *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp.80–87.

Kilpeläinen, P. (1992) 'Tree matching problems with applications to structured text databases', PhD Thesis, University of Helsinki.

Kotsakis, E. (2002) 'Structured information retrieval in XML documents', in G.B. Lamont *et al*. (Eds.) *Proceedings of the 2002 ACM Symposium on Applied Computing*, pp.663–667.

Krátký, M., Pokorný, J. and Snášel, V. (2004) 'Implementation of XPath axes in the multi-dimensional approach to indexing XML data', *Proceedings of Current Trends in Database Technology*, pp.219–229.

Lee, Y.K., Yoo, S., Yoon, K. and Berra, B. (1996) 'Index structures for structured documents', in E.A. Fox and G. Marchionini (Eds.) *Proceedings of the First ACM International Conference on Digital Libraries*, pp.91–99.

Lehtonen, M. (2006) 'Designing user studies for XML retrieval', in A. Trotman *et al*. (Eds.) *Proceedings of the SIGIR 2006 Workshop on XML Element Retrieval Methodology*, pp.28–34.

List, J., Mihajlovic, V., de Vries, A.P., Ramirez, G. and Hiemstra, D. (2003) 'The TIJAH XML-IR system at INEX 2003', in N. Fuhr *et al*. (Eds.) *Proceedings of the 2nd International Workshop on the Initiative for the Evaluation of XML Retrieval*, pp.102–109.

Luoma, O. (2005a) 'Modeling nested relationships in XML documents using relational databases', in P. Vojtáš *et al*. (Eds.) *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science*, pp.259–268.

Luoma, O. (2005b) 'Supporting XPath axes with relational databases using a proxy index', in S. Bressan *et al*. (Eds.) *Proceedings of the 3rd International XML Database Symposium*, pp.99–113.

Luoma, O. (2006) 'Xeek: an efficient method for supporting XPath evaluation with relational databases', in Y. Manolopoulos *et al*. (Eds.) *Communications of the Tenth East-European Conference on Advances in Databases and Information Systems*, pp.30–45.

Piwowarski, B. and Dupret, G. (2006) 'Evaluation in (XML) information retrieval: expected precision-recall with user modelling (EPRUM)', in E.N. Efthimiadis *et al*. (Eds.) *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp.260–267.

Poon, J.P., Raghavan, V., Chakilam, V. and Kerschberg, L. (2001) 'BitCube: a three-dimensional bitmap indexing for XML documents', *Journal of Intelligent Information Systems*, Vol. 17, Nos. 2–3, pp.241–254.

Salton, F. (1971) *The SMART Retrieval System – Experiments in Automatic Document Processing*, New Jersey: Prentice-Hall.

Schlieder, T. and Meuss, H. (2002) 'Querying and ranking XML documents', *Journal of the American Society for Information Science and Technology*, Vol. 53, No. 6, pp.489–503.

Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D.J. and Naughton, J.F. (1999) 'Relational databases for querying XML documents: limitations and opportunities', in M.P. Atkinson *et al*. (Eds.) *Proceedings of the 25th International Conference on Very Large Databases*, pp.302–314.

Weigel, F., Schulz, K.U. and Meuss, H. (2005) 'Ranked retrieval of structured documents with the s-term vector space model', in N. Fuhr *et al*. (Eds.) *Proceedings of the 4th International Workshop on the Initiative for the Evaluation of XML Retrieval*, pp.238–252.

World Wide Web Consortium (W3C) (2006) *Extensible Markup Language (XML) 1.0*, www.w3c.org/TR/REC-xml.

World Wide Web Consortium (W3C) (2007) *XML path language (XPath) 2.0*, www.w3c.org/TR/xpath20.

Notes

1 inex.is.informatik.uni-duisburg.de

2 The document node was left out of Figure 4 since it does not belong to the result of any XPath axis.

3 A text node cannot be an ancestor of any node, and thus the minimum relevance for a text node matching a string value test with respect to any node is $(1 + cos(135°))/2 = 0.15$. However, since our focus is mainly on structural ranking, we use a larger value, 0.5, in order to emphasise the structural part of the queries.

4 Available at www.ibiblio.org/xml/examples.