

Querying and Ranking XML Documents

Torsten Schlieder*

Freie Universität Berlin

`schlied@inf.fu-berlin.de`

Holger Meuss

Ludwig-Maximilians-Universität München

`meuss@cis.uni-muenchen.de`

Abstract

XML allows to represent both content and structure of documents. Taking advantage of the document structure promises to greatly improve the retrieval precision. In this paper, we present a retrieval technique that adopts the similarity measure of the vector space model, incorporates the document structure, and supports structured queries. Our query model is based on tree matching as a simple and elegant means to formulate queries without knowing the exact structure of the data. Using this query model we propose a logical document concept by deciding on the document boundaries at query time. We combine structured queries and term-based ranking by extending the term concept to structural terms which include substructures of queries and documents. The notions of term frequency and inverse document frequency are adapted to logical documents and structural terms. We introduce an efficient technique to calculate all necessary term frequencies and inverse document frequencies at query time. By adjusting parameters of the retrieval process we are able to model two contrary approaches: the classical vector space model, and the original tree matching approach.

1 Introduction

XML has become a widely accepted standard for the representation and exchange of data, attracting growing attention in electronic commerce, information systems, and database research. But what are the benefits of XML from an information retrieval (IR) point of view? We believe that taking advantage of the XML document structure will greatly improve the retrieval *precision*. However, structure-aware retrieval should certainly not lower the *recall* compared to traditional IR techniques.

Neither traditional text retrieval techniques nor structured query languages stemming from the database community face both challenges as we will now discuss: Text retrieval models are based on flat text representations. They impose serious restrictions if they are used for XML, since they

- ignore the document structure,
- support flat queries only, and
- rely on a static document concept.

Ignoring the document structure means ignoring its semantics. The user cannot specify that she prefers the search terms to appear in certain contexts. The static document concept, which is often bound to physical files, cannot cope with different result granularities requested by the users. To solve these problems, a dynamic document concept is necessary – but a dynamic document concept does not allow a precalculation of basic values needed for ranking, like term frequencies and the inverse document frequencies.

XML query languages – though very successful in many applications – are also of limited use in an IR context, since they

*This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

- require a thorough knowledge of the document structure,
- retrieve exact matches of the query only, and
- do not support result ranking.

All these properties are in contrast to the notion of vagueness used in information retrieval. Note, that language constructs like regular path expressions do not solve these problems since the user must still know *that* such an expression is necessary and *how* it must look like. Moreover, regular path expressions do not support partial matches.

In this paper, we present a retrieval formalism, informally introduced in (Schlieder and Meuss, 2000), which seamlessly combines structured queries against XML documents, and relevance ranking. We adopt the similarity measure of the vector space model (VSM, Salton, 1971), incorporate the document structure, and add structure to the queries. Our query formalism is based on tree matching (Kilpeläinen, 1992) as a simple and elegant technique to query XML documents. Tree matching allows to formulate structured queries with only *partial* knowledge of the document structure. The scope of our queries are *logical* documents whose term weights are dynamically adapted. Tree matching is also the basis for the definition of *structural terms*. Structural terms extend the classical term concept to substructures of queries and documents. By treating substructures as terms we are able to retrieve documents that have only partial structural matches. Structural terms have, like ordinary terms, a distribution in and among documents. Based on this distribution we assign weights to structural terms. Furthermore, we show how the classical VSM can be extended to dynamically defined documents and structural terms. Our approach generalizes both the classical VSM and the original tree matching formalism: We can simulate both models by simply adjusting the weights of the query vector. The combination of tree matching and VSM requires new implementation techniques. We introduce efficient algorithms that simultaneously compute all full and partial query matches, and all term weights necessary to measure the similarity between the query and the documents. We also describe first experiences with the prototypical implementation of our approach.

2 Similarity between XML Documents and Structured Queries

In this section, we introduce and formalize our approach to rank XML documents with respect to a structured query. We first describe the general idea, and develop the details of our model in the following subsections.

The basis of our approach is the VSM as a well-known and widely used retrieval model. We adopt this model by extending the concepts *query*, *document*, and *term* to a structured interpretation. A query in the classical VSM is a list of keywords. We add structure to the keywords in such a way that the query can be interpreted as *labeled tree*. XML documents are interpreted as labeled trees, too: We model a document collection as a single tree, and consider each subtree as *logical document*. The root of the query tree determines our notion of admissible documents: Every logical document rooted at a node with the same label as the query root is a potential candidate to be returned as result. We compare it to the query, and assign a similarity score which determines its position in the ranked result list. The similarity scores are computed using the distribution of *structural terms*. Structural terms are essentially subtrees of the query and the documents. The set of structural terms includes, as special cases, all terms used by the standard VSM. We count the number of occurrences of a structural term within a logical document, count the number of logical documents containing the structural term, normalize these values, and use them to compute a *term weight*. The weights are used to construct document vectors. In contrast, the weights of the query vector are defined by the user. The query and document vectors are compared using the standard dot product.

2.1 Trees and their Properties

A *tree* is a structure $T = (N, E, \text{root}(T))$ that consists of a finite set of nodes N , a finite set of edges E , and a node $\text{root}(T) \in N$ that forms the *root node* of T . With $|N|$ we denote the *number of nodes* in T . The set of edges is a binary relation on N where each pair $(u, v) \in E$ establishes a relationship between two nodes of N . If $(u, v) \in E$, we say that u is the *parent* of v , and v is a *child* of u . E must satisfy the following conditions:

1. The root has no parent.
2. Every node of the tree except the root has exactly one parent.

The set of children of a node $u \in N$ is denoted by

$$\text{children}(u) = \{v \in N \mid (u, v) \in E\}.$$

We apply a unique numbering $1 \dots k$ to the k children of u and use the notation $\text{child}_i(u)$ to refer to the i th child of u . A node without children is a *leaf node*. Nodes that are not leaf nodes are called *inner nodes*.

A *path* in a tree is a sequence of nodes u_1, u_2, \dots, u_n such that for every pair u_i, u_{i+1} of consecutive nodes there is an edge $(u_i, u_{i+1}) \in E$. A node u is called an *ancestor* of v (and v a *descendant* of u) iff there exists a path $u = u_1, \dots, u_n = v$, where $n > 1$.

Let $T = (N, E, \text{root}(T))$ be a tree and $u \in N$ be a node. A *subtree* $T' = (N', E', u)$ of T is a tree, where

$$\begin{aligned} N' &= \{u\} \cup \{v \mid v \in N \wedge v \text{ is a descendant of } u \text{ in } T\}, \text{ and} \\ E' &= E \cap (N' \times N'). \end{aligned}$$

We write $T' \triangleleft T$ iff T' is a subtree of T .

A *labeled tree* is a tree $T = (N, E, \text{root}(T))$ together with a function $\text{label} : N \rightarrow \Sigma$, where Σ is a finite set of labels. A tree that has two equally-labeled nodes connected by a path is called *recursive tree*.

Two trees $T = (N, E, \text{root}(T))$ and $T' = (N', E', \text{root}(T'))$ are *isomorphic* iff there exists a bijective mapping f between N and N' such that f preserves the labels, the parent-child relationships, and the numbers assigned to the children of the nodes of T .

2.2 XML Documents and Queries as Labeled Trees

Using our notion of trees, we map each physical XML document to a labeled tree in a standard way (neglecting the semantics of links). To simplify our model, we only use a single node type for XML elements, attributes, and text data. Elements are represented by a node that has the element name as label. Text sequences are decomposed into words. The words are stemmed; stopwords are removed. Each word is mapped to a leaf node labeled with the respective word. Attributes are mapped to two nodes which are parent and child of each other: The attribute name is the label of the parent node, and the attribute value is the label of the child. Attribute values consisting of a sequence of words are decomposed as described for text data. Figure 1 shows the mapping of an example XML document to a normalized tree.

We add a single root node with a unique label to the normalized trees in order to model a document collection (see Figure2(b)):

Definition 2.1 (Document collection) *A document collection C is a labeled tree with a uniquely labeled root.*

Traditional IR models rely on a static document concept which is mostly bound to physical files. XML documents may vastly differ in their granularity: Some documents may contain information about a single book; others may collect data about an entire library consisting of thousands of books. Obviously, it is not appropriate to retrieve an entire library when the user is interested in

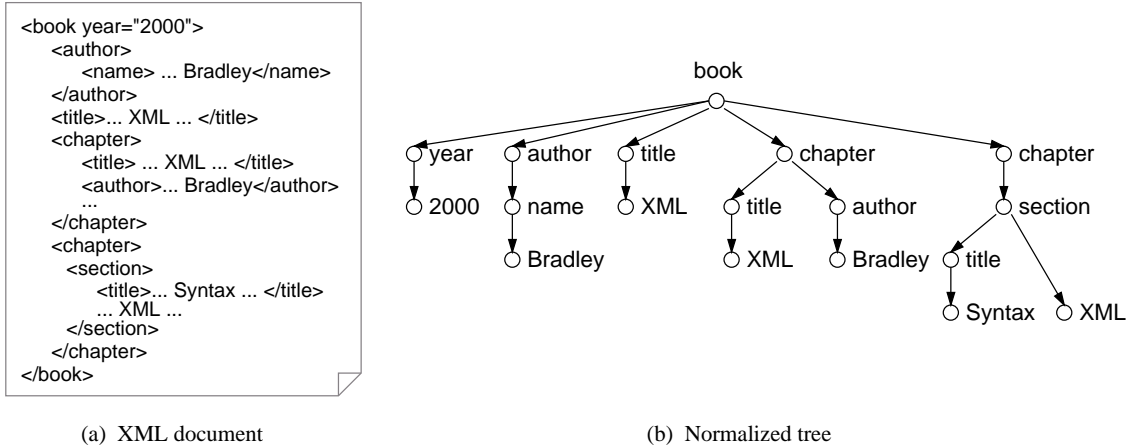


Figure 1: Mapping of an XML document to a normalized tree.

books. The problem remains even if each document contains only a single book – which may be forced by the database administrator – since the user may be interested in chapters or sections instead. We therefore replace the physical notion of a document by a more flexible, logical notion:

Definition 2.2 (Logical document) *A (logical) document D in a document collection C is a subtree of C .*

Example 2.1 In the collection part depicted in Figure 2(b) not only the subtree rooted at the book node (corresponding to the representation of a physical XML document) is a logical document, but also all its subtrees, e.g., the subtree rooted at the section node or the tree consisting only of the node labeled 2000.

The definition of a query in our approach is very simple since we do not distinguish between XML attributes and elements, and do not have logical operators:

Definition 2.3 (Query) *A query Q is a labeled tree.*

We assign types to documents and queries:

Definition 2.4 (Type) *Let $T = (N, E, root(T))$ be a query or a document, respectively. The type of T is the label of its root: $type(T) = label(root(T))$.*

We assume that the user defines with a query the kind of logical documents to be retrieved. If, for example, the root of the query has the label chapter, only logical documents of type chapter are retrieved. This is captured in the following definition of admissible documents:

Definition 2.5 (Admissible documents) *The set of admissible documents for a query Q with respect to a document collection C is $\mathcal{D}^{type(Q)} = \{D \mid D \triangleleft C \wedge type(D) = type(Q)\}$.*

2.3 Tree Matching

We have shown how to interpret both the data and the query as trees. With this interpretation, the problem of answering a query can be mapped to the problem of embedding a query tree in the data tree.

The *exact ordered* tree embedding problem has been extensively studied (see Ramesh and Ramakrishnan, 1992, for an overview) and is solvable in polynomial time. However, we are not interested in an *ordered* embedding, because ordering in the data may be inconsistent. Even a consistent order might not be known to the user. We are also not interested in an *exact* embedding

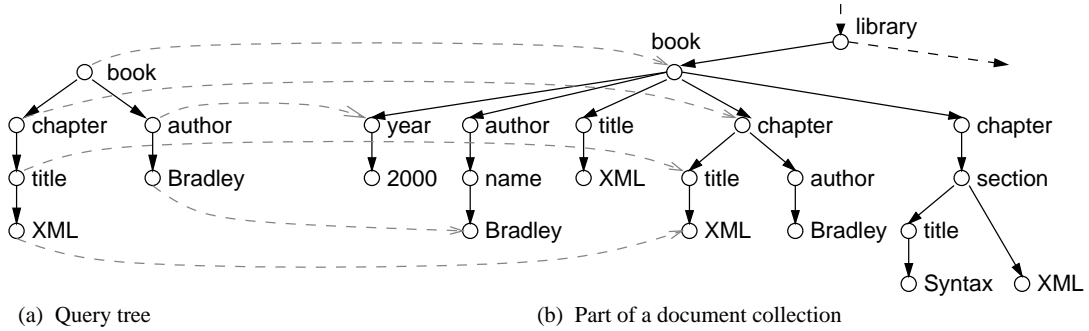


Figure 2: Unordered inclusion of a query tree in a data tree.

that preserves the parent-child relationship of the query. The user should specify her information need – but not the exact structure of the data.

Kilpeläinen (1992) introduced ten variants of tree embedding problems. We adopt the unordered tree inclusion variant, since this variant exactly meets our requirements. Unfortunately, the unordered tree inclusion problem is NP-complete. We therefore use a modified version, described by Meuss (2000), that omits some requirements of the original definition. This variant has a polynomial – typically sublinear – time complexity with respect to the number of nodes in the data tree.

Definition 2.6 (Embedding) Let $Q = (M, E, root(Q))$ be a query and $C = (N, F, root(C))$ be a document collection. A function f from M to N is called an embedding from Q into C iff for all $u, v \in M$ holds

1. $label(u) = label(f(u))$,
2. u is the ancestor of $v \Rightarrow f(u)$ is an ancestor of $f(v)$

The original definition additionally requires the embedding to be injective, and the implication in the second condition to be an equivalence.

Definition 2.7 (Match) Let Q be a query, C a document collection and f an embedding from Q into C . Then $f(root(Q))$ is a match of Q in C .

The logical document rooted at a match of Q in C is an *answer* to the query.

Example 2.2 Figure 2 shows an embedding of a query tree (left) into a data tree (right). There is only one admissible document depicted for the query: the subtree rooted at the **book** node. The dashed arrows show an embedding f that defines a match of the query in the document. The match is the **book** node. The corresponding answer is the subtree rooted at the **book** node (containing more nodes than actually participating in the respective embedding).

Note, that a match can be a result of different embeddings f . In our example, the **title** branch of the query can be mapped to two branches in the data tree which both belong to the same logical document.

2.4 Structural Terms and their Occurrences

In information retrieval, the notion of terms is restricted to unstructured terms (in most cases simply words). We extend the definition of terms to substructures of the query tree and the document collection in order to achieve three objectives: First, we want to relax the restrictions of the tree matching formalism that allows only full matches of the query. In our model, a query will have a partial match if at least one query subtree occurs in a logical document. Second, we

want to measure how good a query fits to the data by considering how many query subtrees have matches in a logical document. Third, we want to reflect the observation that not only flat terms but also substructures have a distribution in and among logical documents.

Definition 2.8 (Structural term) *Every labeled tree over the given set Σ of labels is a structural term.*

We use the notion of structural terms to draw the connection to the classical concept of terms as the smallest constituents of texts. A structural term has quite similar properties as a term in unstructured texts; from a technical point of view, however, the name “structural term” is just a synonym for “labeled tree”.

The definition of an occurrence in the traditional IR models is simple and intuitive: A term occurs in a query or document if it is contained in the query or document. Since our tree matching formalism maps a query tree to a data subtree that may have additional inner nodes, we cannot simply adopt this formalization. Instead, we use two different definitions for query and document occurrences of a structural term. In the following definition we extend the notion of matches, introduced for queries and documents, to terms and documents.

Definition 2.9 (Term occurrence) *Let T be a structural term.*

- *If there exists a structural term T' isomorphic to T such that $T' \triangleleft Q$ for a query Q , we say that T occurs in Q . The root of T' is called an occurrence of T in Q .*
- *If T matches a document D , we say that T occurs in D . We call the match of T in D an occurrence.*

In addition to our set-oriented notation of trees, we use a simple textual representation of structural terms: A node is represented by its label; the children of a node are enclosed in brackets and separated by commas. The query in Figure 2, for example, is represented by the expression `book[chapter[title[XML]],author[Bradley]]`.

Example 2.3 Six structural terms occur in the query in Figure 2: The two “atomic” subtrees consisting only of the nodes labeled XML and Bradley, respectively. Next, the three subtrees rooted at the inner nodes with labels title, chapter, and author, respectively. Finally, the query tree itself is a structural term occurring in the query. The XML document in Figure 2 has many more structural term occurrences, e.g., the four structural terms `book[author]`, `Bradley`, `author[Bradley]`, `book[Bradley,title[XML]]`.

Obviously, there are many other possible definitions of term occurrences in the query. For example, we could consider every single query node to be an occurrence of a term. Alternatively, we could define that every combination of branches in a query subtree is an occurrence of a structural term. However, the former definition does not allow to express a containment relation; the latter definition leads to a combinatorial explosion in the number of query terms. In the rest of this section, we will show that our definition of structural terms is a good compromise between expressiveness and efficiency.

2.5 Quantifying the Weights

Structural terms behave like ordinary terms: They have a number of occurrences in a logical document and a distribution across all logical documents of a given type. We adopt the standard definitions of *term frequency (tf)* and *inverse document frequency (idf)* to quantify the distribution of a structural term.

Definition 2.10 (Term frequency) *Let T be a structural term, and D be a logical document. Let $freq_T(D)$ be the number of occurrences of T in D , and $maxfreq(D)$ be the maximal number of occurrences of any term in D . The term frequency $tf_{T,D}$ of T in D is defined by*

$$tf_{T,D} = \frac{freq_T(D)}{maxfreq(D)}$$

Note, that we cannot have more occurrences of a structural term than nodes in D (since our definition of occurrences is bound to matches, not to embeddings). Therefore, the maximal number of occurrences of a structural term in D is equal to the maximal number of nodes in D that have the same label.

Example 2.4 Figure 2(b) shows a logical document of type `book` as part of a document collection. The maximal number of occurrences of a term in this document is three since we have three XML nodes, and three title nodes. The structural term `author[Bradley]` has therefore a term frequency of $2/3$ in the book document.

In our model, the inverse document frequency is bound to documents of a certain type, since a query only selects documents that match its type. The inverse document frequency idf_T^t of a structural term T represents the ratio of all documents of a type t to the documents of type t that have a match of T :

Definition 2.11 (Inverse document frequency) *Let T be a structural term, and t be a type. Let $|\mathcal{D}^t|$ be the number of documents of type t , and n_T be the number of documents in \mathcal{D}^t matched by T . The inverse document frequency idf_T^t of T is defined by*

$$idf_T^t = \log \frac{|\mathcal{D}^t|}{n_T} + 1,$$

Example 2.5 The collection detail depicted in Figure 2(b) shows two logical documents of type `chapter`, hence $|\mathcal{D}^{\text{chapter}}|$ is 2 in this part of the collection. The structural term $T = \text{author[Bradley]}$ occurs in one of these logical documents. It follows, that $n_T = 1$, and

$$idf_T^{\text{chapter}} = \log 0.5 + 1 = 1.30.$$

The term frequency reflects how good a certain term describes a document; the inverse document frequency reflects how good this term separates the document from all other documents in the collection. There are several methods known to combine these two values in order to calculate a term weight (see, e.g., Salton and McGill, 1983). We choose the product of tf and idf as a simple, and frequently used method:

Definition 2.12 (Document Weight) *Let $D \in \mathcal{D}^t$ be a document of type t . The weight $w_{T,D}^t$ of a structural term T in D is defined as*

$$w_{T,D}^t = tf_{T,D} \cdot idf_T^t$$

2.6 Determining the Query-Document Similarity

In analogy to the VSM we represent the query and documents as weight vectors. Let $\mathcal{T} = T_1, \dots, T_m$ be the list of all pairwise non-isomorphic structural terms that have occurrences in the document collection. It is possible to construct \mathcal{T} out of a given collection. However, the actual construction of \mathcal{T} is not necessary, and \mathcal{T} is used only as a formalism to present our model.

Definition 2.13 (Document vector) *The weight vector v_D for a document D of type t is defined by*

$$v_D = (w_{T_1,D}^t, \dots, w_{T_m,D}^t).$$

Definition 2.14 (Query vector) *The weight vector v_Q of a query Q is defined by*

$$v_Q = (w_{T_1,Q}, \dots, w_{T_m,Q}).$$

where $w_{T_i,Q} \geq 0$ iff T_i occurs in Q , and $w_{T_i,Q} = 0$ else.

With a weighted query vector the user has the possibility to give different substructures (structural terms) of the query different weights. This can be done, for example, by assigning every query node a weight, which is interpreted as the weight of the structural term rooted at the respective query node. Another possibility is to use a library of predefined weight vectors, some of which may prefer the structural parts of the query, others the textual parts.

We will now define the notion of similarity between a document and a query, which reflects the relevance of the document to the query. Again, it is a direct application of the standard VSM definition. Only admissible documents (i.e., documents of the same type as the query) are assigned positive similarity scores; other documents are not considered relevant. This reflects the fact that the root of the query defines a notion of logical documents considered for the retrieval. These logical documents are exactly the trees with the same type as the query.

Definition 2.15 (Similarity) *Let Q be a query with query vector v_Q , and $D \in \mathcal{D}$ be a document with document vector v_D . The similarity $sim(Q, D)$ between v_Q and v_D is defined by*

$$sim(Q, D) = \begin{cases} v_Q \cdot v_D, & \text{if } D \in \mathcal{D}^{type(Q)} \\ 0, & \text{else} \end{cases}$$

where \cdot denotes the standard dot product on vectors.

A common technique in the classical VSM normalizes document and query vectors to the unit sphere, i.e. to length 1, in order to avoid the effect of long documents (containing many terms and having thus longer document vectors) having a higher similarity to a query than shorter documents, and to have normalized similarity scores between 0 and 1. This technique can be applied in our model in the same way by dividing each document and query vector by its length. We omit a formal definition for this simple technique.

Obviously, there is a difference to the classical VSM. Our terms are *structurally dependent*, i.e., some terms are subtrees of others both in the query and the documents. Whenever a term has an occurrence in a document, all its subterms do also have occurrences. These dependencies are, in contrast to the term dependencies in the VSM, necessary for our model: Assigning weights to structural terms that contain each other allows the user to *prefer* terms in a certain context. However, no results are lost if the term does not appear in the desired context – except the user *wants* to neglect them. The following examples show how the user can prefer a term in a specified context (2.6), how she can require it to be in the context (2.7), and how she can prefer a certain query term with respect to another (2.8).

Example 2.6 Consider the query in Figure 2(a). If a user is interested in documents containing the term XML, but prefers documents with a title containing XML, she should assign the same weights to the structural query terms XML and title[XML].

Example 2.7 If a user is interested in documents that contain the term XML – but *only* inside a title, she can set the weight representing the term XML to 0, and the weight of title[XML] to a value larger than 0. Then, the term XML can only have a positive contribution to the similarity score if it is contained in a title element.

Example 2.8 If a user prefers documents containing the term author[Bradley] to documents containing title[XML], she can assign a larger weight to the former structural term, and a smaller weight to the latter.

3 Simulation of Classical Models

Our approach generalizes the classical vector space model and the original tree matching model. We simulate the classical VSM by masking all “complex” structural terms, i.e., we assign 0-values to all positions of the query vector that do not correspond to the leaf nodes of the query. Therefore,

only “atomic” terms of the documents are incorporated in the computation of the similarity score. Consequently, the scores computed by the classical VSM and by our model are exactly the same.

We simulate the tree matching approach in a similar way: Only the structural term representing the whole query gets the weight 1; all other components of the query vector are set to 0. With this technique, only full matches of the query achieve a score larger than 0. Non-matching documents as well as documents with partial matches all get the score 0. In the following, we will apply these techniques in a formal way.

3.1 Simulation of the Standard Vector Space Model

This section shows that the standard vector space model as proposed, e.g., by Salton and McGill (1983) can be modeled with our approach. For reasons of technical simplicity different weighting and distance measures are not considered here. We assume simple weight measures computed with $tf \cdot idf$, and distance measures based on the dot product between query and document vectors. Note, that all derivatives of these measures, like the Cosine, Dice or Jaccard function can be modeled with our model as well by simply adjusting our weighting and distance measures.

We will first show how we model flat documents in our approach. Then we prove the proposition that the weights attached to a term coincide in our model and the vector space model. As a corollary it follows that the similarity values are equal. We conclude this section with the observation that the similarity values do also coincide for both models if we take an XML document collection and treat it as flat text for the vector space model.

Definition 3.1 (Structured representation of a flat document) *Let \bar{D} be a flat document containing the term occurrences o_1, \dots, o_l . The structured representation of \bar{D} is a tree with a root node labeled *doc*, which has l children labeled with o_1, \dots, o_l .*

Definition 3.2 (Structured representation of a flat collection) *Let $\bar{C} = \{\bar{D}_1, \dots, \bar{D}_m\}$ be a collection of flat documents. The structured representation of \bar{C} is a tree with a root node labeled *col* and children D_1, \dots, D_m , where D_i is the structured representation of \bar{D}_i .*

In the following we will assume a term basis (t_1, \dots, t_k) for the standard vector space model. Each flat term t corresponds to an *atomic* structural term T that consists of one node with the label t . Without loss of generality, we can order the structural term basis $\mathcal{T} = (T_1, \dots, T_n)$ in a way that T_i is the corresponding atomic structural term to the flat term t_i for $1 \leq i \leq k$. The structural terms T_i with $i > k$ will be called *complex terms*.

Theorem 3.1 *Let t be a term in a flat document \bar{D} of a collection \bar{C} . Let $w_{t,\bar{D}}$ be the weight of t in \bar{D} according to the VSM. Let \bar{C} be the structured representation of \bar{C} , D be the structured representation of \bar{D} , and T be the atomic structural term corresponding to t . Let $w_{T,D}^{doc}$ be the weight of T in D . Then*

$$w_{T,D}^{doc} = w_{t,\bar{D}}.$$

Proof: We will show the assertion by proving that the parameters for the computation of the term weights coincide in our model and in the VSM: Definition 3.1 implies that $freq_{T_i}(D)$ in our model is equal to $freq_{t_i}(\bar{D})$ for all $1 \leq i \leq k$. Since the maximal number of structural term occurrences in any document D is determined by the maximal number of term occurrences in the respective flat document \bar{D} , it follows that $maxfreq(D) = maxfreq(\bar{D})$ for each pair D, \bar{D} of documents. From the Definitions 2.5 and 3.2 follows that $|\mathcal{D}^{doc}|$ is equal to the number of flat documents in \bar{C} . Due to Definition 3.1 the number n_T of logical documents of type *doc* containing the structural term T is equal to the number of flat documents in \bar{C} containing t . Since the formulas to compute the term frequencies, the inverse document frequencies, and the term weights coincide in our model and in the VSM; and since all parameters of the formulas are the same for all terms and documents, the assertion holds. \square

Definition 3.3 (Leaf weight vector) Let $v_{\bar{Q}} = (w_{t_1, \bar{Q}}, \dots, w_{t_k, \bar{Q}})$ be a flat weighted query vector of the VSM. The leaf weight vector $v_Q = (w_{T_1, Q}, \dots, w_{T_n, Q})$ for $v_{\bar{Q}}$ is a transformation from the term basis (t_1, \dots, t_k) to the structural term basis $\mathcal{T} = (T_1, \dots, T_n)$ such that $w_{i, Q} = w_{t_i, \bar{Q}}$ for all $1 \leq i \leq k$ and $w_{T_i, Q} = 0$ for $k < i \leq n$.

Corollary 3.1 Let \bar{Q} be a flat query and \bar{D} a flat document in a collection \bar{C} . Let $sim_{vsm}(\bar{Q}, \bar{D})$ denote the similarity of \bar{Q} and \bar{D} according to the VSM. Let v_D be the document vector of the structured representation D of \bar{D} and let v_Q be the leaf weight vector for the weighted query vector $v_{\bar{Q}}$ of \bar{Q} . Then the following holds:

$$sim_{vsm}(\bar{Q}, \bar{D}) = sim(Q, D).$$

Proof: The similarity $sim_{vsm}(\bar{Q}, \bar{D})$ is computed with

$$sim_{vsm}(\bar{Q}, \bar{D}) = \sum_{1 \leq j \leq k} w_{t_j, \bar{Q}} w_{t_j, \bar{D}},$$

where $w_{t_j, \bar{D}}$ is the weight of t_j in \bar{D} for the collection \bar{C} according to the VSM. The filtering of the query vector eliminates the contribution of complex terms in $sim(Q, D)$; only atomic terms T_i ($1 \leq i \leq k$) contribute to the similarity value. But according to Theorem 3.1 the weights $w_{t_j, \bar{D}}$ for the occurrences of the flat terms t_j ($1 \leq j \leq k$) are equal for our model and the vector space model. Therefore $sim(Q, D) = \sum_j w_{t_j, \bar{Q}} w_{t_j, \bar{D}}$ and also $sim_{vsm}(\bar{Q}, \bar{D}) = sim(Q, D)$ holds. \square

An XML document can be indexed by a traditional IR engine based on the VSM if either the markup is ignored or if element and attribute names are treated as normal words. With our model, we can achieve the same query results as with this ad-hoc technique without having to preprocess the collection: If markup is treated as normal words, then we can construct for every flat query a simple two-level structured query, such that the vector space model and our model produce the same similarity scores for the documents. If markup is ignored, the similarity scores may differ in pathological cases because of highly frequent tags that change the normalization value $maxfreq(D)$. Note, that this simulation of the classical vector space model only works for collection parts in which all documents have the same type (see Definition 2.15).

3.2 Simulation of Tree Matching

We simulate the original tree matching formalism using a weighted query vector v_Q that assigns the weight 1 to the structural term corresponding to the whole query tree, and the weight 0 to all other structural terms:¹

Definition 3.4 (Root weight vector) Let Q be a query. Let $v_Q = (w_{T_1, Q}, \dots, w_{T_m, Q})$ be the query vector defined in the following way: $w_{T_i, Q} = 1$ iff $T_i = Q$ and $w_{T_i, Q} = 0$ else. We call v_Q the root weight vector for Q .

The following theorem shows that the similarity value of a document vector and the root weight vector for a query is larger than 0 if, and only if the respective document is a tree matching match for the query.

Theorem 3.2 Let Q be a query with a root weight vector v_Q , and D a document with document vector v_D . Then $sim(Q, D) > 0$ iff D is a match of Q .

Proof: If D and Q have different types, the assertion is trivially true. If Q matches no term in \mathcal{T} , i.e. $Q \neq T_i$ for all i then $sim(Q, D) = 0$, since $v_Q = (0, \dots, 0)$. Because \mathcal{T} contains all terms that have occurrences in the document collection, Q has no tree matching match in D . Now let $Q = T_k \in \mathcal{T}$. The above definition implies $v_Q = (w_{T_1, Q}, \dots, w_{T_m, Q})$ with $w_{T_k, Q} = 1$, and $w_{T_i, Q} = 0$ for all $i \neq k$. Then $sim(Q, D) = w_{T_k, D}^t$. Since the inverse document frequency is always larger than 0, it follows that $w_{T_k, D}^t > 0$ iff $tf_{T_k, D} > 0$. From the definition of the term frequency follows the assertion, since $tf_{T_k, D} > 0$ iff T_k has a match in D . \square

¹If the query itself is no term in the term alphabet \mathcal{T} , the root weight vector v_Q consists of 0s only.

4 Implementation

The model proposed in the last sections provides a powerful formalism to combine structured queries with ranking. The implementation of this model is challenging:

1. The *matches* of all query subtrees (which represent occurrences of structural terms) into the document collection must be found.
2. The *weights* of all structural terms that have occurrences in both the query and the document collection must be calculated.

In this section, we describe how our formalism can be implemented efficiently. The following points outline the main building blocks of our approach:

- We apply a *preorder-bound encoding* to the document collection, which allows to verify whether two given nodes are ancestor and descendant of each other.
- An *inverted index* maps each label to a *posting* that stores references to all nodes carrying the label.
- The postings are used by a function `join_path`, which finds all ancestor-descendant pairs for a given pair of query labels.
- Our tree-matching algorithm processes the query bottom-up. It finds the matches of a query subtree by combining the matches of its included subtrees using the function `join_path`.
- Using an array of intermediate results filled by the tree-matching algorithm, the term-count algorithm calculates the *term frequencies* and the *inverse document frequencies*.

Every topic is explained in more detail in the following subsections. At the end of this section, we introduce our prototypical implementation.

4.1 Encoding and Indexing the Document Collection

We use an indexing technique that is inspired by the *partial index* introduced by Navarro (1995). Every node u in the document collection is represented by a triple of integers:

- $pre(u)$ is the preorder number of u ,
- $bound(u)$ is the preorder number of the rightmost leaf of the subtree rooted at u , and
- $maxf(u)$ is the maximal number of occurrences of any term in the subtree rooted at u .

An example of an encoded document collection is shown in Figure 3(a). Given two nodes u and v we can test if u is an ancestor of v by ensuring the invariant

$$pre(u) < pre(v) \wedge bound(u) \geq pre(v).$$

The algorithms that we will introduce in the following subsections operate on *postings*. A posting is a list of *entries*, which represent data nodes. An entry is a triple $(pre, bound, maxf)$. The entries of a posting are sorted by their preorder numbers in ascending order. We use the notations P_i to refer to the i th entry of posting P ; $pre(P_i)$, $bound(P_i)$, and $maxf(P_i)$ to access the preorder number, the bound value, and the maximal term frequency, respectively, of the node represented by P_i . The number of entries of a posting is denoted by $|P|$.

We distinguish between index postings and transformed postings. An *index posting* contains references to all nodes of the document collection that have the same label. The index I maps labels to postings. The expression $I(l)$ refers to the index posting that belongs to label l . We use the name *transformed posting* to denote a posting that is derived from an index posting $I(l)$, and that contains a part of the entries of $I(l)$.

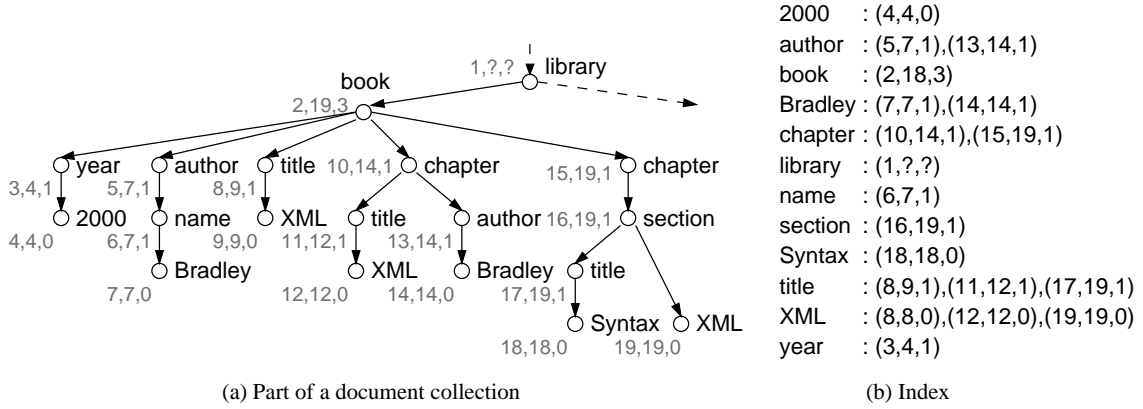


Figure 3: Index of a document collection

4.2 Joining the Ancestor-Descendant Relationship

The function `join_path` takes a posting P of potential ancestors and a posting R of potential descendants. It constructs the transformed posting S , which consists of all entries in P that are ancestors of entries in R .

Recall that the postings are sorted by the preorder node numbers in ascending order. If the document collection is not recursive (i.e., no node has an ancestor or descendant with the same label), then the ancestor-descendant relationship can be established by simultaneously iterating through the postings P and R . In particular, all descendants of a node P_i reside in a contiguous interval of R .

Algorithm 1 shows the function `join_path`. It works as follows: First, the posting S that will contain the results of the join is initialized by the empty list (Line 1). The outer loop (Lines 3-10) iterates through the posting of potential ancestors. Let P_i be the current entry of P . At Line 4 the algorithm searches the position of the first descendant of P_i in R . The algorithm marks this position (Line 5) and searches the end of the interval of descendants (Line 6). If the interval size is larger than zero, then P_i is appended to S (Lines 7-9). Note, that Algorithm 1 does not work correctly if the document collection is recursive. The interested reader may find an algorithm for recursive document collections in (Schlieder, 2001a).

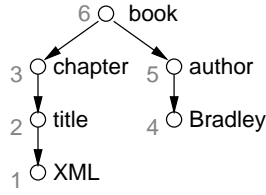
Algorithm 1 joins the ancestor-descendant relationship.

function `join_path`(P, R)

```

1:  $S := []$ 
2:  $j := 1$ 
3: for  $i := 1$  to  $|P|$  do
4:   while  $j \leq |R|$  and  $pre(R_j) \leq pre(P_i)$  do  $j := j + 1$ 
5:    $j' := j$ 
6:   while  $j \leq |R|$  and  $pre(R_j) \leq bound(P_i)$  do  $j := j + 1$ 
7:   if  $j' < j$  then
8:     Append a copy of  $P_i$  to  $S$ 
9:   end if
10: end for
11: return  $S$ 

```



- 1 : (8,8,0),(12,12,0),(19,19,0)
- 2 : (8,9,1),(11,12,1)
- 3 : (10,14,1)
- 4 : (7,7,0),(14,14,0)
- 5 : (5,7,1),(13,14,1)
- 6 : (2,19,3)

Figure 4: Query with postorder enumeration

Figure 5: Array A of transformed postings computed by Algorithm 2 for the query in Figure 4 and the collection in Figure 3.

4.3 Implementing Tree Matching

Our tree-matching algorithm processes the query tree bottom-up, fetches the occurrences of the leaves, and computes the matches of the inner nodes using the matches of its children. Let $Q = (M, E, root(Q))$ be a query. To access the nodes of Q in the correct order, we use a *postorder enumeration*. The postorder number of a query node $u \in M$ is accessed using the function $post(u)$. Figure 4 depicts a query tree and the postorder numbers of its nodes.

Algorithm 2 shows the tree-matching procedure. As a precondition, it requires that an array A is initialized. For each node $u \in M$, the array stores a transformed posting computed for u . We use the notation $A_{post(u)}$ to access the posting of node u .

The algorithm processes the query tree bottom-up, accessing the nodes by their postorder numbers in ascending order (Line 3). If u is a leaf, then the algorithm fetches the index posting belonging to the label of u (Line 4), and stores it in A (Line 14). If u is an inner node, then the matches belonging to the subtrees rooted at the children of u have already been computed. More precisely, if v is a child of u , then the matches computed for v are stored in the posting $R = A_{post(v)}$. The algorithm uses this information (Lines 6 and 9) and computes the matches of the subtree rooted at u by combining the matches of its children: For each child v of u , it calls the function `join_path` passing as parameters the index posting P belonging to $label(u)$ and the posting R containing the matches of v (Line 10). The result of this call is a transformed posting P in which each entry P_i refers to the root of a data subtree that contains a match of the query subtree rooted at v . By intersecting the postings returned by the function `join_path`, the algorithm selects only those matches of u that contain matches of *all* children of u (Line 11). The matches belonging to the subtree rooted at p are stored in A (Line 14). In particular, $A_{post(root(Q))}$ stores the matches of the entire query tree.

Figure 5 shows the array of transformed postings that have been computed while executing the query depicted in Figure 4 against the document collection depicted in Figure 3. The transformed posting that contains the matches of the entire query is stored at A_6 .

4.4 Computing the Term Weights

The typical implementation of the VSM consists of a lexically ordered list of all terms occurring in the document collection. Each term is annotated with its *idf* and a list of occurrences and term frequencies. The weights of a term in all documents can be computed by iterating through the list of term frequencies and calculating the *idf - tf* products. We cannot use this simple technique for two reasons: First, the set of structural terms that have matches in the document collection is exponentially larger than the set of document terms in the flat text model. Second, the scope of the term frequencies and inverse document frequencies are *logical* documents which are defined at query time. The maximal number logical documents is equal to the number of inner nodes of the document collection. If we computed all term frequencies at indexing time, the posting of a term would consist of references to all subtrees of the document collection that contain the term. Notice also, that a query selects only documents of a certain type. Hence, each term must refer to a list of *idfs* – one *idf* for each document type. Since the precalculation of all term weights would lead to an intractable index size, we compute the *tf* and *idf* values while executing the query.

Algorithm 2 finds all matches of a query tree.

input: A query $Q = (M, E, \text{root}(Q))$ and an index I

output: An array A of $|M|$ transformed postings

```

1: Initialize  $A$  such that it has  $|M|$  components
2: for  $i := 1$  to  $|M|$  do
3:   Get  $u \in M$  such that  $\text{post}(u) = i$ 
4:    $P := I(\text{label}(u))$ 
5:   if  $|\text{children}(u)| > 0$  then
6:      $R := A_k$ , where  $k = \text{post}(\text{child}_1(u))$ 
7:      $P := \text{join\_path}(P, R)$ 
8:     for  $j := 2$  to  $|\text{children}(u)|$  do
9:        $R := A_k$ , where  $k = \text{post}(\text{child}_j(u))$ 
10:       $S := \text{join\_path}(P, R)$ 
11:       $P := \text{intersection of } P \text{ and } S$ 
12:   end for
13: end if
14:  $A_i := R$ 
15: end for

```

Fortunately, we can restrict our computations (1) to structural terms occurring in the query and (2) to the document type determined by the label of the query root.

We use the intermediate results computed by Algorithm 2 together with the information provided by the index to compute all term frequencies and the inverse document frequency of a certain query term with little additional effort. To calculate the weight $w_{T,D}^t$ of term T in the logical document D of type t we need (see Definition 2.12):

- $|\mathcal{D}^t|$, the number of logical documents of type t in the collection,
- $\text{maxfreq}(D)$, the maximal number of occurrences of any structural term in D ,
- n_T , the number of logical documents of type t in which T occurs, and
- $\text{freq}_T(D)$, the number of occurrences of the structural term T in D .

The the number of logical documents of type t in a collection can be derived immediately from the information provided by the index: Let $Q = (M, E, \text{root}(Q))$ be a query, and

$$t = \text{type}(Q) = \text{label}(\text{root}(Q))$$

be the type of Q . Then the number of documents of type t in the document collection is

$$|\mathcal{D}^t| = |I(\text{label}(\text{root}(Q)))|.$$

Now assume that Q has been evaluated by Algorithm 2. Then the array A stores the matches of all query subtrees and thus, the matches of all structural terms that have occurrences in both the query and the document collection. In particular, the transformed posting $A_{\text{post}(\text{root}(Q))}$ contains the matches of the entire query. Let i be the index of the logical document D in the transformed posting computed for the query root. Then the maximal term frequency in D is

$$\text{maxfreq}(D) = \text{maxf}(A_{\text{post}(\text{root}(Q)),i}).$$

To compute (1) the frequencies of all structural terms occurring in the query with respect to all documents of type t and (2) the number of documents that contain a certain query term, we need an additional algorithm that postprocesses the postings stored in array A . Algorithm 3 realizes the postprocessing and creates the arrays TF and DF . We first describe the arrays TF and DF

and show how the values $freq_T(D)$ and n_T can be derived can be derived from these arrays. Then we explain the algorithm in detail.

TF is an array of *frequency vectors*: Each entry of TF refers to a logical document D and contains the frequencies of all query terms in D . We use the notation $F_{i,j}$ to refer to the j th component of the frequency vector that belongs to the i th entry of F . Let i be the index of D in TF , and let T be a structural term that has an occurrence u in Q . Then the frequency of T in D is

$$freq_T(D) = F_{i,post(u)}.$$

DF is an array that contains, for each term T in the query, the number of logical documents in which T occurs. Let u an occurrence of T in Q . Then the number of logical documents of type $type(Q)$ in which T occurs is

$$n_T = |DF_{post(u)}|.$$

Algorithm 3 works as follows: It fetches the posting P from the array of transformed postings (Line 1). P contains the roots of all matching documents. Now, the algorithm computes the frequency of each structural term occurring in the query with respect to the logical documents represented by P . More precisely, it visits each query node $u \in M$ (except the root) and gets the posting R from the array of intermediate results (Line 4). R contains the matches of the query subtree rooted at u . For each document root P_i the algorithm counts the number of matches from R that are descendants of P_i (Lines 8-14). If there is at least one match, then the document frequency of query term rooted at u is incremented (Line 13).

Algorithm 3 computes an array of frequency vectors and an array of document frequencies.

input: A query $Q = (M, E, root(Q))$ and an array A of transformed postings

output: An array TF frequency vectors and an array DF of document frequencies

```

1:  $P := A_{post(root(Q))}$ 
2: Initialize  $TF$  such that it has  $|P|$  frequency vectors, each having  $|M|$  components
3: Initialize  $DF$  such that it has  $|M|$  components
4: for  $k := 1$  to  $|M| - 1$  do
5:    $R := A_k$ 
6:    $DF_k := 0$ 
7:    $j := 1$ 
8:   for  $i := 1$  to  $|P|$  do
9:     while  $j \leq |R|$  and  $pre(R_j) \leq pre(P_i)$  do  $j := j + 1$ 
10:     $j' := j$ 
11:    while  $j \leq |R|$  and  $pre(R_j) \leq bound(P_i)$  do  $j := j + 1$ 
12:     $TF_{i,k} := j - j'$ 
13:    if  $j \neq j'$  then  $DF_k := DF_k + 1$ 
14:   end for
15: end for

```

Now we are ready to put the pieces together: Let i be the index of document D in the transformed posting computed for the query root, t be the type of the query, and u be an occurrence of term T in the query. Then the weight of term T with respect to document D is

$$w_{T,D}^t = tf_{T,D} \cdot idf_T^t = \frac{TF_{i,post(u)}}{\max(A_{post(root(Q)),i})} \cdot \left(\log \frac{|I(label(root(Q)))|}{|DF_{post(u)}|} + 1 \right).$$

Example 4.1 Assume that the query shown in Figure 4 has been executed against the document collection shown in Figure 3(a). The results of the evaluation have been stored in array A , which is depicted in Figure 5. Algorithm 3 uses the query Q and the array A of transformed postings as input and computes the following list F , which consists of a single frequency vector:

$$F = ((3, 2, 1, 2, 2, 1)).$$

The value of first component indicates that there are three occurrences of the query node 1 (labeled XML) in the logical document rooted at node 2. Similarly, the value of the second component states that the structural term title[XML] has two occurrences in the logical document, and so on. The maximal term frequency in the document is 3. The weight of the query term title[XML] in the first (and only) matching document is calculated as follows:

$$\frac{F_{1,2}}{\max f(A_{6,1})} \cdot \left(\log \frac{|I(\text{book})|}{|A_6|} + 1 \right) = \frac{2}{3} \cdot \left(\log \frac{1}{1} + 1 \right) = \frac{2}{3}.$$

If we calculate the weights for all six structural terms occurring in the query, then we get the document vector

$$v_D = \left(1, \frac{2}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3} \right),$$

which will be normalized and multiplied with the query vector to get the similarity between Q and D .

4.5 Computational Complexity

In the case of non-recursive trees the time complexity of the `join_path` operator is bound by $O(s)$. The parameter s denotes the selectivity, i.e. the maximal number of nodes with the same label. Note, that s is typically very small with respect to the number of data nodes. If the tree contains recursive labels, parts of the inner posting may be processed several times. Since the number of nodes carrying the same label along a path cannot exceed the height h of the tree, the complexity is bound by $O(s \cdot h)$. The time needed to intersect two postings is bound by $O(s)$. To find all logical documents matching the query tree, and to compute the *tf* and *idf* for these documents, we need at most $2(|M| - 1)$ calls to the function `join_path`. $|M|$ denotes the number of nodes of a query. The following table shows the overall time complexities of our algorithm:

trees without recursive labeling	trees with recursive labeling
$O(M \cdot s)$	$O(M \cdot s \cdot h)$

Note, that the structure of the algorithms, and in particular the division into a tree-matching part and a postprocessing part simplifies the explanation of the algorithm. However, the algorithms are not very optimal in time and space: First, the intersection of postings can be integrated into the function `join_path`. Second, the calculation of the term frequencies can be integrated into the tree-matching algorithm, since the function `join_path` can be easily extended to compute the frequencies as well. Third, in an integrated algorithm, the postings stored in the array of intermediate results can be deleted as soon as the matches belonging to the next level of the query tree have been found.

4.6 Prototype

We completely implemented the approach introduced in this paper. Our prototype consists of the five main components kernel, loader, indexer, query processor, and web interface. The kernel is implemented on top of the Berkeley Database toolkit (Sleepycat Software Inc., 2001), which provides basic access structures as well as several subsystems. The caching subsystem allowed us to implement a hybrid memory management strategy. We only hold the inverted list of labels in memory permanently; the postings are fetched on request. The LRU (least-recently-used) strategy provided by the cache ensures that frequently used postings will not be removed from the cache. On top of the kernel module reside the loader and the query processor. The loader uses the Xerces XML parser (The Apache Software Foundation, 2001) to import XML files. It sends the content of the parsed files to the indexer that creates the inverted list of labels. The query processor implements the concepts introduced in the preceding sections. It provides a command line interface as well as an interface to the web module. All components of our system

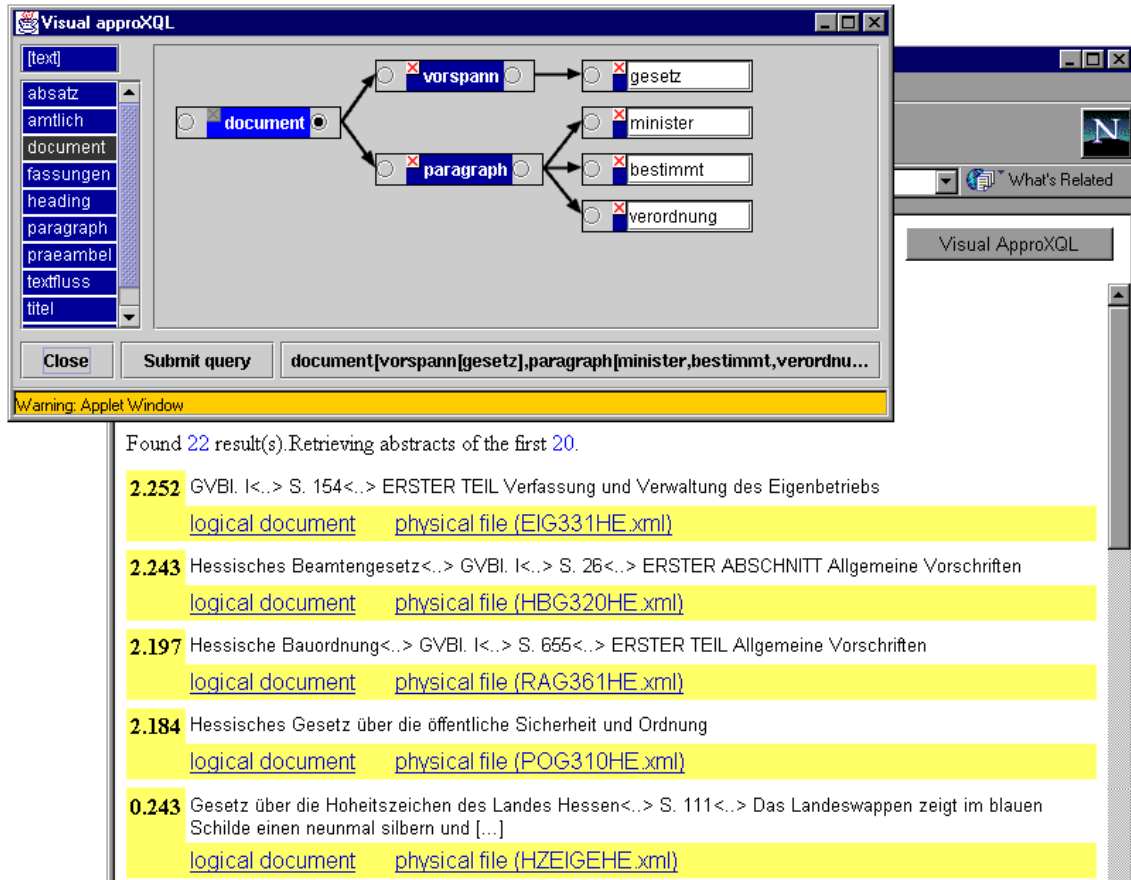


Figure 6: Web interface of the prototype

are implemented in C++; the web module uses FastCGI (Brown, 1996) to communicate with the HTTP-server.

The user can choose between a text interface and graphical editor to formulate her queries. The text interface expects queries in a syntax similar to the notation of structural terms (see Section 2.4). Our query language allows to assign a weight to every structural term of the query; all other terms implicitly get the weight 1. The following query assigns a zero weight to the entire query, the weight 2 to the “flat” term XML, and the weight 1.5 to the term author[Bradley]:

```
book:0[title[XML:2],author:1.5[Bradley]]
```

Since the formal specification of a query is often not practical for an occasional user, we also provide a graphical interface implemented as Java applet. This interface serves for two purposes: First, it displays all element and attribute names found in the indexed documents. Second, it supports the user in formulating queries. The list of labels is dynamically adapted to the context of the current query node: The user can only select labels that may be ancestors/descendants of the active query node. Figure 6 shows the graphical query editor and a list of ranked results for the given query.

5 Practical Experiences

In this section we present first experiences with the implementation of our model. Since there is currently no large test collection of XML files, we decided to use a small set of legal documents provided by the juris GmbH, Saarbrücken. This collection consists of 22 documents containing

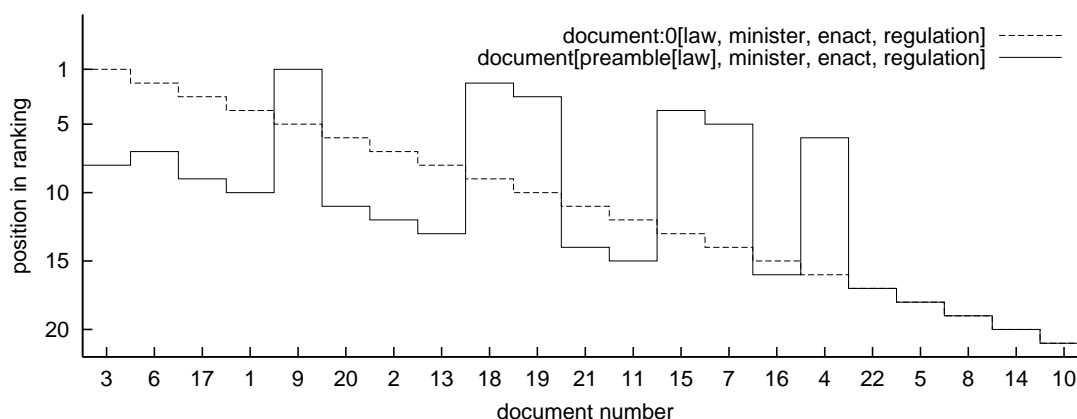


Figure 7: Ranking of the example collection with a flat query (dashed line) and a metadata query (solid line).

laws, regulations, and directives of the German federal state Hessen. We assumed several sample information needs, formulated different queries for each, varied the query weights, and studied the rankings generated by the system. We learned that structured queries do not necessarily lead to a better ranking if the keywords are semantically related. For example, the information need

”Get all documents that describe how the **police prevents offenses**”

produces similar rankings for a query that contains only the keywords *police*, *prevent*, and *offense*; and for the structured query `document[paragraph[police,prevent,offense]]`. This is due to the effect that the three keywords tend to occur in the same paragraph anyway, so that the structure imposes no additional restrictions. However, we observed a strong improvement of the ranking (1) if the query specifies that certain terms should be in the metadata of the logical documents (e.g., title, preamble), or (2) if the query terms are not closely semantically related. We illustrate the improvement of the retrieval precision using the sample information need

”Get all **laws** that describe how **ministers enact regulations**.”

The bold-faced keywords of this sentence sometimes occur together, but often in different semantic contexts. Our first query is unstructured:

```
document:0[law,minister,enact,regulation].
```

We applied the weight 0 to the whole query in order to simulate the flat VSM model (see Section 3.1). The ranking generated by the query is depicted in Figure 7 with a dashed line. The x-axis indicates the logical document numbers; the y-axis shows the relative position of the document in the ranking. We sorted the documents according to the ranking generated for the flat query. The second query

```
document[preamble[law],minister,enact,regulation]
```

additionally requires that *law* appears in the preamble of the document. The query also prefers full matches over partial matches by using the implicit weight 1 for the structural term representing the full query. The ranking generated for this query is depicted by the solid line in Figure 7.

The figure points out that adding structure to a query does not lower the recall (regarding the entire collection) but strongly improves the query precision. The shift from the flat query to the structured one prefers documents that are in fact laws – which is specified in the document preamble. That is, the documents 9,18,19,15,7,4 now take the first positions in the ranking. Note, that we are also able to weaken this effect by applying a smaller weight to the query node *preamble*. With this kind of query extensions we are able to involve *metadata* in the text retrieval process – but our query does not fail if the metadata does not contain the specified terms. The third query

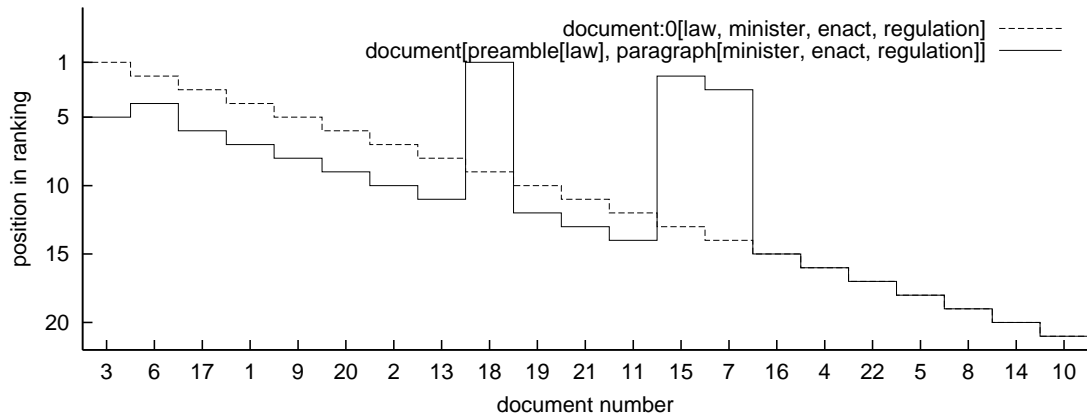


Figure 8: Ranking of the example collection with a flat query (dashed) and a fully structured query (solid line)

```
document[preamble[law], paragraph[minister, enact, regulation]]
```

extends the structured query and expects the keywords *minister*, *enact*, and *regulation* to occur in the same paragraph. This new requirement changes the ranking order once more. Now, documents of type *law* that contain all three keywords in a *single* paragraph take the first positions in the ranking (the documents 18,15,7 in Figure 8). And indeed, the documents 18, 15, and 7 contain the information satisfying the original information need – while other documents may also contain all keywords but not in the same context. In fact, the inner query node *paragraph* works similar to the near operator used in some retrieval engines but does not rely on positional distances but on *semantic closeness* specified by the document creator.

Another important observation is the *stability* of the rankings. Documents that have many matches of query subtrees are preferred with respect to the ranking generated by the flat query – but documents that have only partial matches preserve their relative positions to each other. This can be seen in the diagrams by the sequences of documents (6, 17, 1, 9, 20, 2, 13) or (22, 5, 8, 14, 10), that mainly keep their descending ranking order for all three queries.

6 Related Work

Many languages have been proposed for querying XML documents (Fernandez et al., 1999; Bonifati and Ceri, 2000, provide comparisons). But only few approaches exist that consider the problem of relevance ranking for structured queries. Even structured query models developed by the information retrieval community (see Navarro and Baeza-Yates, 1996; Baeza-Yates and Ribeiro-Neto, 1999, for surveys) do typically not face this challenge.

An early work that suggests to incorporate document subtrees in the computation of the scores is (Fuller et al., 1993). The queries are still unstructured, and no method is given to compute the exact term weights for arbitrary subtrees. The authors propose to propagate the weights up the document tree, or to generate a text abstract for each node, which serves to compute the weights. Lalmas (1997) describes a model based on the Dempster-Shafer theory of evidence. In this model, a generalized belief function is used to find the part of a structured document that fits best a given unstructured query.

Navarro et al. (1998) proposed to use result ranking for structured queries. The authors introduced a simple query language and a generic ranking model. Queries are logical formulas consisting of presence and inclusion operators. The score computed for each document is the sum of all occurrences of the content terms specified by the query. However, the distribution of substructures is not considered, and the document concept is static.

At the same time, a logic-based retrieval model for multimedia objects has been proposed (Fuhr et al., 1998). Aspects of this model have been adopted by the XML query language XIRQL (Fuhr and Großjohann, 2000). The language incorporates the notion of term weights and vague predicates into XQL; the content and structure of XML documents are mapped to facts and rules of an intensional probabilistic logic (Rölleke, 1999). In contrast to our approach, XIRQL cannot treat partial structural matches. In addition, Fuhr and Großjohann (2000) do not discuss how to derive the probabilities of facts and rules from the data. The probability of every document term is computed for a single context (i.e., subtree) only, and is augmented to more general contexts using application-dependent rules like “multiply weight by factor 0.7”.

Another approach based on a probabilistic interpretation of the query and document structure has been proposed in (Wolff et al., 1999). Some details of this work are similar to ours, e.g., weighting of keywords and substructures, partial matches, retrieval of subdocuments. Other aspects differ: Each query in the proposal of Wolff et al. is a set of pairs consisting of a term (a keyword or a phrase) and a path through the data. The path attached to each term specifies the subtree in which the term must appear. With this language one cannot express, for example, that one term should appear in the title, and another should appear in the body of the *same* chapter of a book. However, the most important difference is the ranking model: Wolff et al. suggest to measure the descriptiveness of a term *only* within the subtree specified by a path. The paths also have a weight that is incorporated in the ranking function. In contrast, our model measures the descriptiveness of a term within a logical document but rewards documents that *also* have matches of complex structural terms. We believe that measuring the distribution of terms only inside “micro”-trees like title is of little use. Moreover, terms that appear in the document but not in the specified context will get a zero score.

To the best of our knowledge, XXL (Theobald and Weikum, 2000) and ELIXIR (Chinenyanga and Kushmerick, 2001) are the only XML query languages stemming from the database community that incorporate the notion of relevance². Both XXL and ELIXIR add a similarity operator to a subset of XML-QL (Deutsch et al., 1998). In XXL, the similarity operator can be applied to element names as well as to text sequences. The query processor searches for matches similar to the name or text specified and assigns probabilities to the matches. The probabilities are combined to obtain a single score. ELIXIR is comparable with XXL but additionally supports similarity joins. Both languages do not allow partial structural matches. Moreover, the question of how the term weights in the documents can be adjusted to the dynamic document notion is not discussed in both papers. The static assignment of weights to terms or phrases at indexing time cannot reflect the descriptiveness of a term/phrase in the varying notions of a logical document determined at query time.

7 Conclusion and Future Work

We introduced a retrieval technique that allows to assign similarity scores to XML documents with respect to a structured query. Our approach allows partial matches of the query. It improves the precision of the query result compared to flat retrieval models, without losing results. We proved that our model generalizes the two main concepts it is based on: the tree matching formalism for structured queries without ranking, as well as the vector space model for ranking documents with respect to a flat query. We described the implementation and first experiences with the prototype.

Future work will include further development of the user interface, since the usability of our model depends on the user’s knowledge of the element names and the ancestor-descendant relationships between the elements. We also plan to do an extensive evaluation of our model under realistic conditions. In the long term we plan to combine our frequency-based model with cost-based models such as the approach presented in (Schlieder and Naumann, 2000; Schlieder, 2001b). Another cost-based model, which we plan to integrate, relaxes the condition that labels have to

²Here, we do not discuss query languages that add special information retrieval operators to the query algebra in order to enable similarity search in flat text.

be preserved in a match. A cost-function shall define the penalty for labels of the query that do not match XML element names. This feature will further enhance the flexibility of our model.

In addition, we plan to investigate structured ranking models based on the extended Boolean model (Salton et al., 1983), which generalizes the vector space model. With extended Boolean logic, we can use the logical formulation of a query tree as a basis for relevance assignments. Among other advantages, this provides the possibility to enrich the query language with vaguely interpreted Boolean operators.

Acknowledgements

This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

References

- Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison Wesley Longman.
- Bonifati, A. and Ceri, S. (2000). Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1).
- Brown, M. R. (1996). FastCGI: A high-performance web server interface. <http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm>.
- Chinenyanga, T. and Kushmerick, N. (2001). Expressive and efficient ranked queries for XML data. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, pages 1–6, Santa Barbara, USA.
- Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1998). XML-QL: A query language for XML. W3C Note. <http://www.w3.org/TR/NOTE-xml-ql>.
- Fernandez, M., Siméon, J., and Wadler, P. (1999). XML and query languages: Experiences and examples. <http://www-db.research.bell-labs.com/user/simeon/xquery.ps>.
- Fuhr, N., Gövert, N., and Rölleke, T. (1998). DOLORES: A system for logic-based retrieval of multimedia objects. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 257–265, Melbourne.
- Fuhr, N. and Großjohann, K. (2000). XIRQL: An extension of XQL for information retrieval. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece.
- Fuller, M., Mackie, E., Sacks-Davis, R., and Wilkinson, R. (1993). Structured answers for a large structured document collection. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 204–213, Pittsburgh.
- Kilpeläinen, P. (1992). *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Finland.
- Lalmas, M. (1997). Dempster-shafer's theory of evidence applied to structured documents: Modelling uncertainty. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 110–118, Philadelphia, USA.
- Meuss, H. (2000). *Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents*. PhD thesis, University of Munich.
- Navarro, G. (1995). A language for queries on structure and contents of textual databases. Master's thesis, Department of Computer Science, University of Chile.

- Navarro, G. and Baeza-Yates, R. (1996). Integrating content and structure in text retrieval. *SIGMOD Record*, 25(1):67–79.
- Navarro, G., Baeza-Yates, R., Vegas, J., and Fuente, P. (1998). A model and a visual query language for structured text. In *5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, Sta. Cruz de la Sierra, Bolivia.
- Ramesh, R. and Ramakrishnan, L. (1992). Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295–316.
- Rölleke, T. (1999). *POOL: Probabilistic Object-Oriented Logical Representation and Retrieval of Complex Objects — A Model for Hypermedia IR*. PhD thesis, University of Dortmund.
- Salton, G. (1971). *The SMART Retrieval System- Experiments in Automatic Document Processing*. Prentice Hall Inc., Englewood Cliffs, NJ.
- Salton, G., Fox, E., and Wu, H. (1983). Extended boolean information retrieval. *Communications of the ACM*, 26:1022–1036.
- Salton, G. and McGill, M. J. (1983). *Introduction to Modern Information Retrieval*. McGraw-Hill, Tokio.
- Schlieder, T. (2001a). ApproXQL: Design and implementation of an approximate pattern matching language for XML. Technical Report B 01-02, Freie Universität Berlin.
- Schlieder, T. (2001b). Similarity search in XML data using cost-based query transformations. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, pages 19–24, Santa Barbara, USA.
- Schlieder, T. and Meuss, H. (2000). Result ranking for structured queries against XML documents. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, Zurich, Switzerland.
- Schlieder, T. and Naumann, F. (2000). Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece.
- Sleepycat Software Inc. (2001). The Berkeley Database. <http://www.sleepycat.com/>.
- The Apache Software Foundation (2001). Xerces XML parser for C++. <http://xml.apache.org/xerces-c/>.
- Theobald, A. and Weikum, G. (2000). Adding relevance to XML. In *Proceedings of 3rd International Workshop on the Web and Databases (WebDB'00)*, Dallas, USA.
- Wolff, J., Flörke, H., and Cremers, A. (1999). XPRES: a ranking approach to retrieval on structured documents. Technical Report IAI-TR-99-12, University of Bonn.