Searching XML Documents – Preliminary Work

Marcus Hassler and Abdelhamid Bouchachia

Dept. of Informatics, Alps-Adria University, Klagenfurt, Austria
marcus.hassler@uni-klu.ac.at, hamid@isys.uni-klu.ac.at

Abstract. Structured document retrieval aims at exploiting the structure together with the content of documents to improve retrieval results. Several aspects of traditional information retrieval applied on flat documents have to be reconsidered. These include in particular, document representation, storage, indexing, retrieval, and ranking. This paper outlines the architecture of our system and the adaptation of the standard vector space model to achieve focussed retrieval.

1 Introduction and Motivation

Traditionally, content-based retrieval systems rely either on the Boolean model or the vector space model (VSM) [1–3] to represent the flat structure of documents as a bag of words. Extensions of these models have been proposed, e.g., the fuzzy Boolean model and knowledge-aware models. However, all of these indexing models do ignore the organization of text and the structure of documents until recently with the advent of "queriable digital libraries".

XML documents have a standard structure defined by a DTD or XML schema. While this structure provides documents with hierarchical levels of granularity, and hence more precision can be achieved by means of focussed retrieval, it does, however, imply more requirements on the representation and retrieval mechanisms. With the new generation of retrieval systems, the two aspects, the structure and the content, have to be taken into account. To minimally achieve that in presence of nested structure like chapter-section-subsection-paragraph, the traditional information retrieval techniques, e.g., the VSM, have to be adapted to fit the context of structure-aware retrieval. To design such systems, four basic aspects are of high importance:

(a) Representation: Textual content of the hierarchically structured documents is generally restricted to the leave nodes. Hence, representation mechanisms of the inner nodes content have to be defined.
(b) Retrieval granularity: A basic question is whether the indexing/retrieval unit must be known ahead of time or is completely dynamically decided by the user or eventually by the system itself.
(c) Ranking: Related to the first two aspects, a strategy for ranking the retrieved results has to be decided.
(d) Result presentation: The way results are presented is a key issue [4–6] and has to be considered early in the design of the system as part of requirements

engineering. Once ranked, the results are displayed showing their context of appearance. Further functionality enabling browsing is required.

Taking these aspects into account, we developed a retrieval system. It is fully implemented in Java and consists of three subsystems: indexing, retrieval and RMI (Remote Method Invocation) communication server as depicted in Fig. 1.

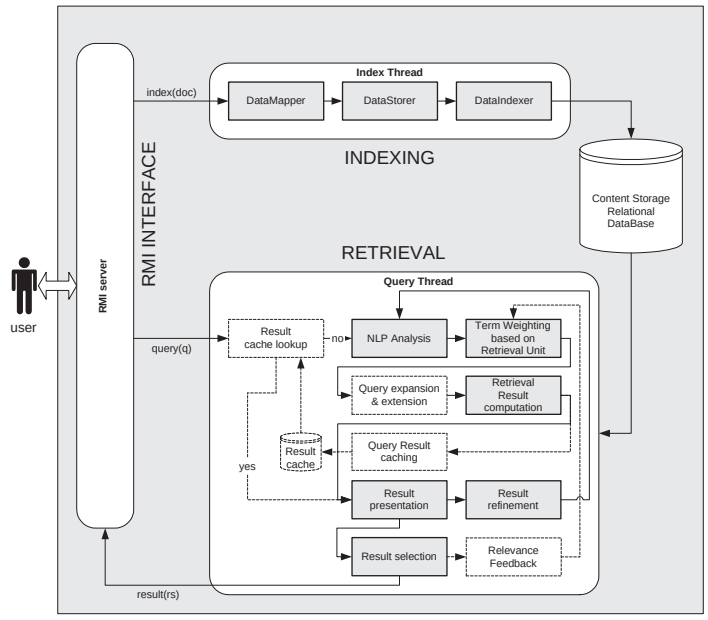


Fig. 1: Architecture of the system

The RMI server takes incoming requests for indexing and querying the system and initiates a new thread for each call. The basic motivation behind this is to achieve some degree of parallelism. The maximum number of parallel threads depends on the performance of the hardware. From the software architecture point of view, both index and query subsystem, use a pipelined pattern of processing units (Fig. 1). Dashed components describe planned extensions. For portability and tuning purposes, all subsystems are independently configurable via configuration files. During indexing, documents are transformed into our XML schema (DataMapper), stored in the database (DataStorer), and indexed for retrieval (DataIndexer). As soon as a query is sent to the system it is analyzed by a query thread. Documents in the database are matched against the query and relevant elements[1] are ranked in decreasing order.

In this paper, we will discuss the aspects (a)–(d), but with more focus is more on the representation and the indexing/retrieval problem. First, in Sec. 2,

[1] we use element and node interchangeable

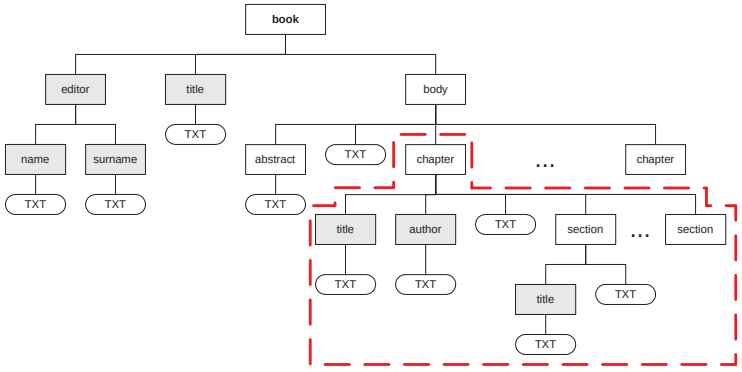


Fig. 2: Example XML document

a generic schema for document representation is presented, onto which the XML documents are mapped. Section 3 describes the underlying database model used for storing the content and the corresponding representation. The most interesting issues namely indexing and retrieval are discussed in Sec. 4 and Sec. 5 respectively. Section 7 concludes the paper.

2 Document Structure

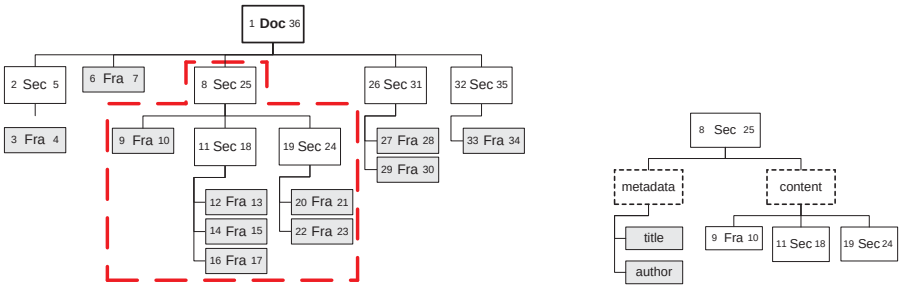
The hierarchical structure for the content of documents is usually described by means of a set of tags (e.g. chapter, section, subsection, etc.), as shown in Fig. 2[2].

In order to represent a collection of documents having different structure, we apply an XSLT transformation to derive a common document format (schema). This step eliminates structural ambiguities and resolves semantic relativism [7].

As illustrated in Fig. 3a, we introduce a general document format (defined through XML schema) that consists of only three main elements: `DOC` (document), `SEC` (section) and `FRA` (fragment). The `DOC` element defines the root of the document. `SEC` is the basic structural element of a document. By recursively defining `SEC` (e.g., section) as either containing raw content `FRA`s (e.g., paragraphs) and/or made up of other `SEC`s (e.g., subsections), the depth of nested structures is unlimited. To define smallest retrievable units for indexing and retrieval, we rely on fragments (`FRA`s). They stand for the leaf nodes in our document schema (see Fig. 3a). Note that if a query refers to another tag not in the set {*DOC, SEC, FRAGMENT*}, this latter is interpreted as *SEGMENT*.

A node in an XML document is viewed as a tuple (*metadata*,*content*), where *metadata* refers to descriptive information of the node itself, while *content* refers

[2] This example will be used throughout the paper. It is important to point out that the approach presented here is general, but we used the IEEE collection just to illustrate the processing steps of the system. In other words, the system is collection-independent and therefore portable.



(a) Transformed example

(b) Metadata and Content blocks

Fig. 3: XML document representation

to the segment's content, properly said (see Fig. 3b). Generally, the first type of nodes requires database-supported (exact) match during retrieval, while the second type is subject to partial matching (VSM).

2.1 Metadata

In addition to the content block, the metadata block of a node contains information describing that node. Examples of metadata are `author`, `year` and `keywords` for a `DOC` or the `title` for a `SEC` element. The fragment metadata block is used to describe its actual content by means of `content_type`, `language`, and possibly `title` (figures, tables).

To allow a semantic interpretation of the content of an element, a type hierarchy is proposed by Gövert [8]. An extension of the proposed type hierarchy for metadata is depicted in Fig. 4. There, types are derived from a common base element. The first level in the hierarchy (bold) corresponds to database supported data types. Further types in subsequent levels in the hierarchy have one of the basic database types as supertype (e.g., `PersonName` is a String). In addition, data types predicates for comparison are defined. This allows to process section titles, phone numbers, and author names for instance. This type hierarchy is used during parsing to optimize the storage efficiently. In addition, it helps characterize the type of match required during retrieval.

2.2 Content

Generally, the content block of `DOC`s and `SEC`s are defined as ordered lists of further (sub)`SEC`s and `FRA`s. The content block of `FRA`s is defined as bytecode or empty. For indexing and retrieval purposes, content is interpreted based on its type (metadata). Defining a fragment's content as text block (paragraph) only might be too restrictive. Therefore, a fragment in our sense refers to paragraphs, enumerations, lists, figures, tables, formulas, images, sounds, videos, definitions,

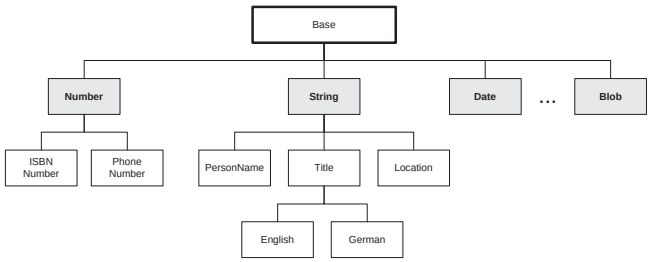


Fig. 4: Hierarchical metadata types

theorems, etc. On the other hand, a fragment (`FRA`) defines the smallest retrievable unit of a document. It can be understood as building block (elementary content container). However, the granular unit is application specific and can be set at wish to fit sentences as well as the whole text of a chapter.

From the structuring point of view, additional markup within a `FRA`'s content might be needed. Our schema supports mathematical environments (using `MATH`) and two types of links (using `LINK`), internal and external links. Internal links are links within the same document, e.g., citations, figure/table references within the text and the table of contents. External links refer to external resources, including reference entries in the references section, references to email/internet addresses and file references.

While the content block in `DOC`s and `SEC`s is mandatory, in `FRA`s it is not. This allows to include external content by its metadata only. An external source attribute within the metadata block can be used to refer to the content somewhere else (e.g., a picture file). In contrast to `SEC` elements, which define their own context based on their path, e.g., `/DOC/SEC/SEC`, fragments define a separate context. As to indexing and retrieval, a fragment in a section lies within the same context as a fragment in a chapter or subsubsection. This difference is important in the context of a dynamic term space, discussed in Sec. 5.3.

3 Storage

For efficiency purposes, we use a relational database to store the XML documents. The goal is to accelerate the access to various structural neighbors of each node in the document (descendants, ancestors, and siblings). Being a tree, an XML document can easily and unambiguously be traversed. Therefore, each node is represented by its document ID and preorder/postorder. We depart from the idea of *preorder* and *postorder* introduced in [9, 10], supporting non-recursive ancestor/descendant detection and access. Table 1 shows an excerpt of the structural information of a document representation. Likewise, we designed another for the corresponding content.

A structural entry is described by the tuple ($docID$, pre, $post$, $parentID$, $tagID$, $pathID$). The root element has $pre = 1$ and $parentID = 0$ (no parent)

per definition. The attribute $tagID$ is included for fast name lookup and access. For the sake of performance, we added the elements path (XPath without positional information) $pathID$ to circumvent recursive path generations by using the $parentID$ relation.

Table 1: Structural entries

$docID$	pre	$post$	$parentID$	tag	$path$
d_1	1	36	0	`Doc`	`/Doc`
d_1	2	5	1	`Sec`	`/Doc/Sec`
d_1	3	4	2	`Fra`	`/Doc/Sec/Fra`
d_1	6	7	1	`Fra`	`/Doc/Fra`
d_1	8	25	1	`Sec`	`/Doc/Sec`
d_1	9	10	8	`Fra`	`/Doc/Sec/Fra`
d_1	11	18	8	`Fra`	`/Doc/Sec/Sec`
d_1	12	13	11	`Fra`	`/Doc/Sec/Sec/Fra`

The content of nodes (in particular leaf nodes) is stored in a separate table, as suggested in [11]. However, the content of inner nodes can always be recovered from their descendants as will be discussed in Sec. 4. Note that some content entries do not have a corresponding representation entry (e.g. figures, tables).

To improve retrieval performance, metadata handling is completely shifted to the database. This is achieved by grouping all metadata according to its element. Instead of having multiple structural and content entries, a single row ($docID$, pre, $meta_1$, ..., $meta_n$) is used to store all metadata together. Metadata of nodes (`DOC`, `SEC`, `FRA`) are stored in separated but very similar tables as shown in Tab. 2 for the case of sections. The reason of supporting only a single set of `SEC` metadata is that all `SEC` elements (chapters, sections, subsections, etc.) are assumed to have quite homogenous metadata (e.g., `title`). Although this may lead to some '$NULL$' values (unavailable metadata for some elements) in the database, the whole set can be accessed by a single select statement. This simplifies and speeds up querying of metadata considerably. A global view is

Table 2: Metadata entries for `SEC`)

$docID$	pre	$title$	$author$	...
d_1	2	`Introduction`	`R. Smith`	
d_1	8	`XMl Retrieval`	`J. Alf`	
d_1	11	`Granularity`	$NULL$	

depicted in Fig. 5. Both, metadata and content entries, are optional. Additional types of representations (e.g. semantic concepts, figure representations, etc.) can easily be integrated.

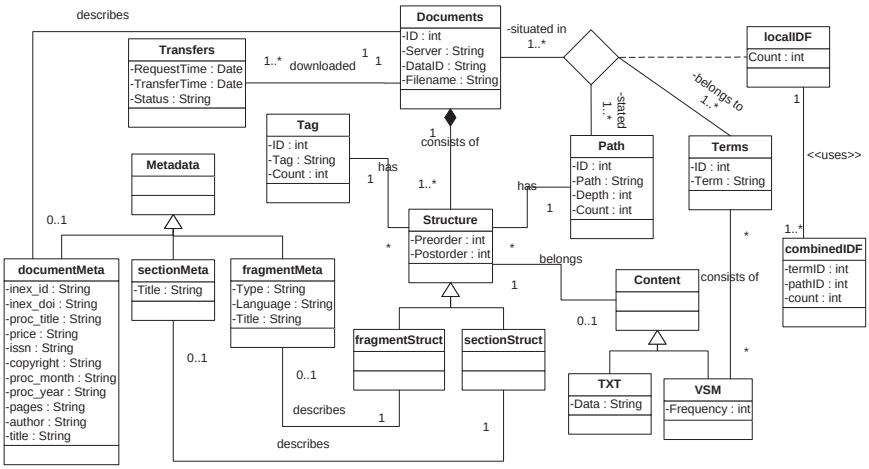


Fig. 5: Conceptual database schema

4 Indexing

To represent texts as a vector of terms and their term frequencies, our natural language processing (NLP) involves several subtasks containing tokenization, tagging, term extraction, stemming, filtering and term frequency calculation. Our implementation is based on abstract components. Taking advantage of the the modularity aspect, different implementations of the same component can be instantiated and selected during runtime. Hence, our system can easily be adapted to process documents in other languages. Our prototype also involves ready made-components like the tagger, and the stemmer.

During the indexing process, only the content of leaf nodes need to be parsed. Their representation, a term frequency vector, is stored in the database (`VSM` table). Consequent updates of the `localIDF`, `combinedIDF` table, and `Terms` table are immediately done. These update operations are also carried out during re-indexing or removal of documents.

The index of inner nodes is obtained by simply merging the index of its descendants. This is done by summing up their term frequencies. This operation is equivalent to process the concatenated contents of the descendant nodes. It is also possible to store the result of the merge operation so that no index computation is required later during the retrieval process. This reduces search time, but increases the size of the database. It is important to stress that the weight vectors are computed during retrieval using the available term frequency vectors.

We define the *context of a node* as the set of all elements having the same path (all chapters, all sections, etc.). In order to dynamically characterize both, the granularity during indexing and retrieval, we applied a propagation of term statistics (e.g. tf), in contrast to the weight propagation methodology [12]. In

addition, the inverse document frequency (*idf*) for each node is calculated dynamically based on the node's context. Term weights are computed based on the term frequencies and the *idf* in this context. This allows to perform focussed retrieval on any level in the document tree. To achieve that in a given context, tf of all nodes lying at this level will require tfs of their descendants. Using term statistic propagation, the descendants' tf are simply summed up. We avoid recursive data accesses by exploiting preorder and postorder of document elements (only one SQL select statement).

As to term weighing, we use different $idf_{j,c}$s of the same term j in different contexts c. This strategy weighs the same term with the same term frequency differently depending on c (e.g. chapter vs. subsection). Clearly our approach puts more attention on the actual context during retrieval. If the unit of retrieval is defined explicitly, elements in this context are focussed and compared only among them. Representations of elements in other contexts do not influence the result.

To implement this idea, we use two tables (see Fig. 5): a table `localIDF` stores tuples of the form ($docID$, $pathID$, $termID$, n_j), where n_j refers to the number of elements containing term $termID$ in the path $pathID$ within a document $docID$. Consider the example given in Tab. 3, the first Tab. 3a indicates that the term "car" occurs twice in `/DOC/SEC` nodes of document d_1. To calculate the $idf_{j,c}$ of a term j in a context c, we have to define N_c and n_j. N_c is the number of nodes with $pathID = c$. N_c can simply be derived via the table holding the structural entries (see Tab. 1). n_j is given by counting the rows containing $pathID = c$ and $termID = j$. In the above example, this results in an inverse document frequency for the term "car" in the node `/DOC/SEC` of $idf_{car,/DOC/SEC} = \log \frac{3}{2}$. This definition of $idf_{j,c}$ leads to different *idf*s in different contexts.

Table 3: *idf* calculation

(a) Table `localIDF`

docID	*path*	*term*	n_j
d_1	`/DOC/SEC`	car	2
d_1	`/DOC/SEC/SEC`	mouse	1
d_2	`/DOC/SEC`	car	1
d_2	`/DOC/SEC/SEC`	dog	1
d_2	`/DOC/SEC/SEC`	mouse	3
d_3	`/DOC/SEC`	water	1
d_3	`/DOC/SEC/SEC`	dog	2

(b) Table `combinedIDF`

path	*term*	*n*
`/DOC/SEC`	car	3
`/DOC/SEC`	water	1
`/DOC/SEC/SEC`	mouse	4
`/DOC/SEC/SEC`	dog	3

Since Tab. 3a is quite large, we introduced a summarized shortcut-table `combinedIDF` Tab. 3b with the overall goal to reduce the time access to *idf* values. Same paths associated with the same terms are precalculated (e.g. term "car"). For the sake of dynamic document environments (adding, removing and re-indexing), we still need the information provided by Tab. 3a to adjust the n

values correctly. In addition, all N_c values, the numbers of elements with the same path, are stored in the `Path` table (see Fig. 5).

Given a particular context (e.g. `/DOC/SEC`), our indexing strategy allows on-the-fly computation of the representations associated with these nodes (considered as documents). Hence, our indexing method stores only term frequency vectors in the database; weight computation is totally executed on the fly during the retrieval process. The advantages of this methodology are:

- It behaves exactly like the traditional models at the document level.
- There is no need for empirical parameters as augmentation weights.
- Elements of smaller granularity (of lower level) do not automatically have sparser feature vectors (leading to smaller similarity), hence they define their own context. Since the number of terms is at max the total number of terms in the whole collection.
- Documents can dynamically be added, removed, and re-indexed, without impacting the weights of other representations.

5 Retrieval

This section explains the retrieval process. In particular, it describes how and which information is required by the system to answer a user query appropriately. This includes formulation of the query, setting of specific parameters, matching, filtering, and presentation of the result.

5.1 Query formulation

The actual query input is done via an input interface which allows to enter different types of queries: `KWD` (keyword) and `NLQ` (natural language query, free text), which are translated into `INEX` queries. The `INEX` query supports NEXI-like inputs. Hence, we distinguish between metadata and content, we adapted our query parser to support both kinds of information. Similar to the `about(path,terms)` syntax, we added a construct: `meta(path,condition)`. This allows us, for example, to efficiently deal with queries like: "return all documents written by Einstein" using the command `//DOC[meta(.,author like '%Einstein%')]`.

In order to avoid long and confusing single-line queries, we use chains of INEX queries. In our opinion, this concept is also closer to the natural way of questioning, by successively refining the list of results. Each subquery result works as a strict filter, allowing only elements of the same or smaller granularity to be retrieved. This improves the performance without skipping searched elements. Furthermore, we use these chains for reweighing elements regarding to a user-defined generality factor (gf), described below. In addition to the INEX-query chains, several query parameters can be specified by the user (see Fig. 6):

- **Maximum results ($maxRes$):** Defines the maximum number of returned results ranging from 1 to $MAXINT$.

– **Minimum similarity ($minSim$):** Defines the minimum similarity of returned results ranging from 0 to 1, truncating the list of results below a given similarity threshold.
– **Content importance (ci):** Defines the importance of the content similarity to calculate the retrieval status value (rsv). This parameter ranges from 0 (only meta similarity) to 1 (only content similarity). The final similarity is computed as $rsv = simCont * ci + simMeta * (1 - ci)$.
– **Generality factor (gf):** This parameter ($\in [0, 1]$) influences the retrieval granularity. The higher the parameter, the more importance of first subqueries, computed as $sim_{new} = sim_{old} * gf + sim_{new} * (1 - gf)$.
– **Result type (rt):** Defines which kind of results we wish to obtain: thorough or focussed (see Sec. 5.6).

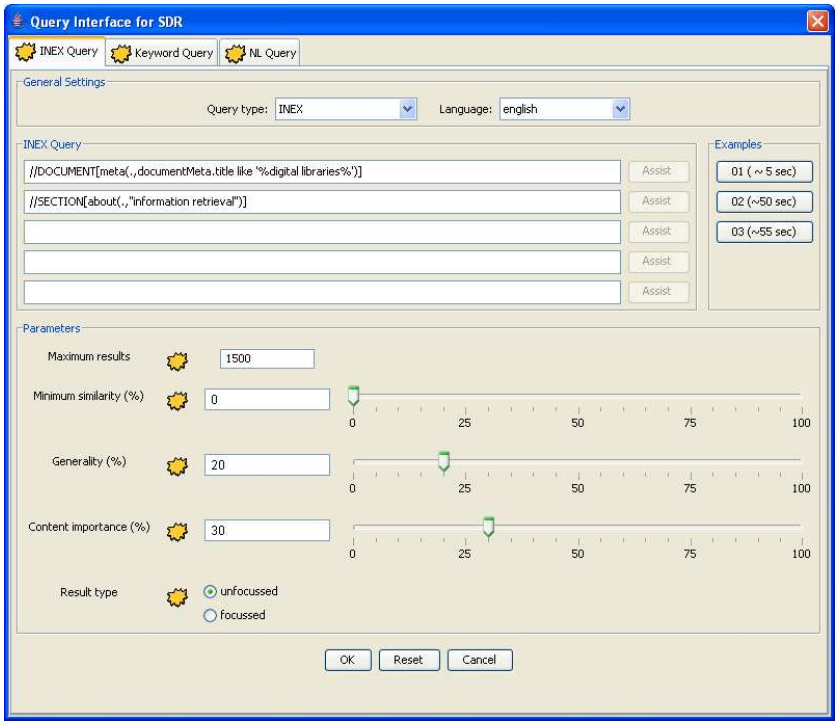


Fig. 6: Query Interface

5.2 Search and Retrieval paths

The search path specifies which elements are to be investigated and matched against the current query. In contrast, the retrieval path specifies which elements are to be returned to the user. Generally these two path are equal, e.g. `//SEC[about(.,wine)]`. This means that the retrieval path is always the same

or more general as the search path. So first relevant documents, then relevant sections within those documents, and at a last stage relevant fragments within those sections are identified. Difficulties arise when relevant ancestor elements contain smaller elements that are further specified. For instance, a query that retrieves sections containing paragraphs about a certain topic is not easy given the recursive structure that a section can have.

Our parser for NEXI-like queries implements the following strategy: if the searched element satisfies the retrieval path, only the element itself is returned. Otherwise, the closest parent satisfying the retrieval path condition is returned. In all cases, at most one element is retrieved. So a query like `//SEC[about( ./FRA,global warming)]` searches all `SEC` elements at any level (retrieval paths) containing `FRA` paragraphs about "global warming" and returns the most relevant element. A more complex query is `//(DOC|SEC)[about(./SEC,anything)]`. Here only document or section elements containing sections about "anything" are to be retrieved, not the sections themselves that are about "anything".

5.3 Dynamic term space

In the context of structured documents, the idea of representing elements at different structural levels within the same term space has to be reconsidered. Assume a number of document sections $\mathbb{S} = \{s_1 \dots s_n\}$ containing a set of unique terms $\mathbb{T}_s$ and a set of chapters $\mathbb{C} = \{c_1 \dots c_m\}$ containing a set of unique terms $\mathbb{T}_c$. Note the implicit relation between term space $\mathbb{T}_s$ and term space $\mathbb{T}_c$: $\mathbb{T}_s \subseteq \mathbb{T}_c$. Let q be a query containing terms $\mathbb{T}_q$ addressing sections $\mathbb{S}$ and chapters $\mathbb{C}$. To calculate the similarity $sim(s_i, q)$ between a section and a query, both feature vectors have to be within the same term space. The same thing holds for comparing chapters and the query $sim(c_i, q)$.

Neglecting the context, sections and chapters are represented in the same (global) term space. As a consequence, the feature vectors of low level nodes become sparser and their similarities compared to nodes of higher levels drop. To overcome this problem, we adopted the concept of a "dynamic term space". In contrast to the global term space, and following the concept of context, nodes in the same context generate a term space. Using a static term space improves performance, but unfortunately decreases the similarity of low-level nodes compared with higher ones. Reducing zero weighted elements in the feature vectors leads to higher precision during the match of low-level nodes. The number of different indexing representations (different contexts) is expected to be quite limited. For instance, the mapped INEX collection does not exceed six structural levels (`/DOC/SEC/SEC/SEC/SEC/FRA`). During retrieval the term space for each context is constructed once, so retrieval performance drops insignificantly.

5.4 Result computation

INEX queries are stated using keywords in the `about(path,`kwd_1 kwd_2 `...`kwd_n`)` syntax. This syntax allows to express several different semantics of keywords that have to be considered:

- `information retrieval techniques`
- `+information +retrieval techniques`
- `information retrieval -techniques`
- `"information retrieval" techniques`
- `+"information retrieval" techniques`

'`+`' (MUST) and '`-`' (CANNOT) indicate whether a term has to be or should not be present in an element. Based on this, a fast preselection is systematically done on candidate elements. Hence, index terms are stemmed, also these terms have to be for comparison.

More complex is the treatment of quoted keywords. Are the keywords `books` and `"books"` equivalent? This depends on whether `"books"` should occur as it is (noun in plural form), or should it be stemmed and treated so. It is obvious that quoted expressions are particulary difficult to process. Consider `"red cars"`. The term `red` is an adjective, it is not included in the index. Furthermore, it is possible that in another context (e.g. "`Red Cross`"), it is (part of) a proper noun and, therefore, exists in the index. In our approach, we treat quoted keywords in two steps: First, we treat them as unquoted, calculating the similarity as given. Then, we apply a string matching strategy on the original text associated with the element to sort the results.

Combinations of MUST/CANNOT and quoted expressions are treated as if all terms within quotes are separately marked as MUST/CANNOT and an initial result set is computed. This result is reduced to those node containing exactly the quoted expression.

Note that the computed result consists of tuples of the form ($docID$, $preorder$, $postorder$, $simMeta$, $simCont$). $docID$ (document ID), $preorder$ and $postorder$ come directly from the database. $simMeta$ and $simCont$ are the calculated meta similarity and content similarity.

5.5 Ranking and result presentation

Ranking is the task by which retrieved elements are decreasingly ordered by their relevance. Therefore, we use a combination of metadata and content similarity to compute a retrieval status value rsv (see Sec. 5.1). The ranking process itself is impacted strongly by the desired granularity. Note that this granularity is either pre-specified or stated explicitly in the user query. For example, if the user specifies the document level (context), say section, the system should return only relevant sections. The similarity can be calculated using two strategies. The first strategy is to match the query against sections with content aggregated from its descendants. The second strategy, which we have considered, is to match the query against the most representative fragment of each section.

After all desired elements are matched against the user query, the combined similarity values $metaSim$ and $contSim$ are used for ranking. The results are presented to the user as a sorted list in decreasing order (see Fig. 7). The user is then able to select a particular result, enabling a display of whole document in an explorer-like view (see Fig. 8). The document structure is presented as an

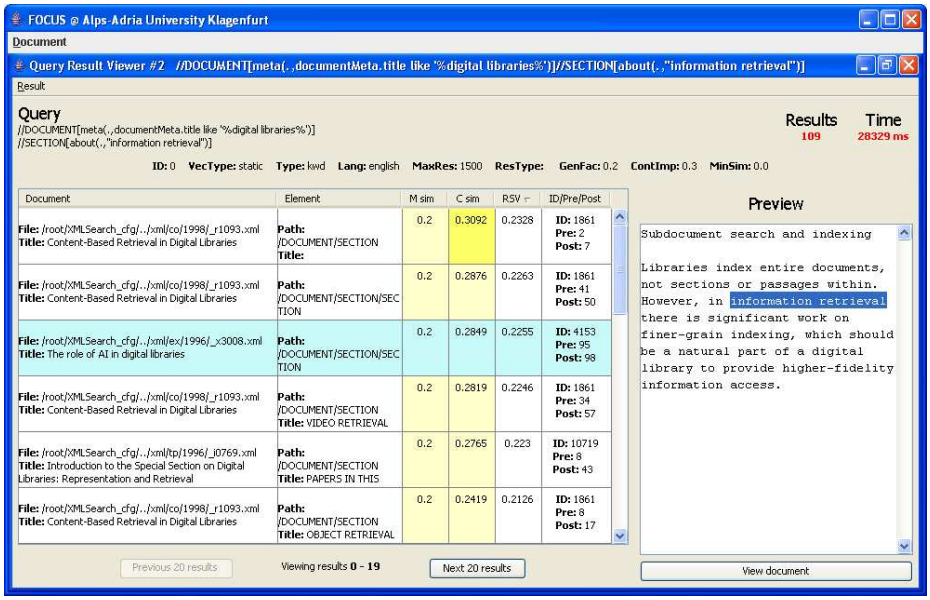


Fig. 7: Result Set

expandable tree, where the selected element is expanded and focused. Having similarity values available on the screen, the document can be efficiently browsed. Colors are used to reflect the degree of similarity of the matched elements.

5.6 Result filtering

In INEX 2004, two kinds of retrieval strategies, thorough and focussed, were defined. *Thorough retrieval* returns all relevant elements of a document. Hence, all ancestors of a relevant element are relevant to a certain degree. This may lead to multiple result elements along the same path (e.g., a section and its contained paragraphs).

Focussed retrieval, on the other hand, aims at returning only the most relevant element along a path. Basically, it relies on two principles [13]: (a) if an element is relevant to a certain degree, so must be its parent; (b) only one node along a path of relevant elements is returned. Overlapping elements in the result set are discarded. This strategy is implemented as post filtering process to refine the result set. We rely on preorder and postorder to do this efficiently. In other words, all low-ranked ancestors and descendants are discarded. This strategy reduces the number of returned elements drastically.

5.7 Query Refinement

In most cases a final search result is achieved through iterative refinement of the query. The number of results is reduced step by step by adding new information

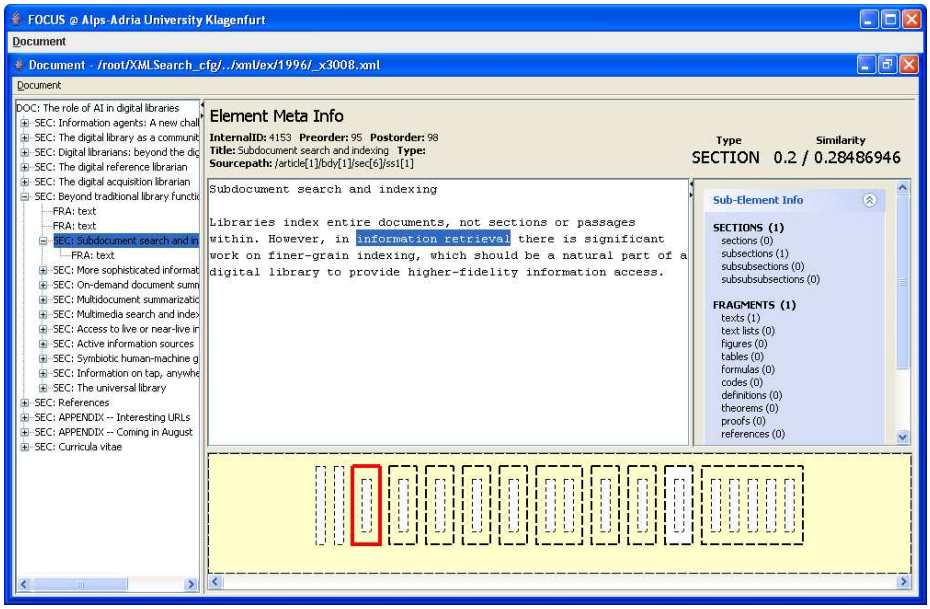


Fig. 8: Result Browser

to the query. To enable such a feature, we allow the user to include a list of preliminary results together with a query. If such a result is set within a query it acts as a strict filter during query computation.

6 Initial Experiments

In the current evaluation, we will show only some initial experiments. Indeed, only three retrieval runs were evaluated (`CO` and `COS` -both `Thorough`- and `SSCAS`). The results are shown in Tab. 4 ($nxCG$) and Tab. 5 (ep/gr). The number between parentheses in each cell indicates the rank of our system compared with the other participating systems. The results illustrate that our approach is less competitive in the case of `CO` and `COS` tasks. In contrast to that, it is ranked among the first 10 systems in the case of `CAS`.

Table 4: Metric: nxCG, Quantization: strict, Overlap=off

nxCG at	CO	COS	SSCAS
10	0.0115(47)	0.0000(28)	0.3250(4)
25	0.0221(42)	0.0094(24)	0.3200(7)
50	0.0416(41)	0.0118(26)	0.3489(9)

To overcome the limitations observed in the case of `CO` and `COS` tasks, further work is underway. It concerns various aspects related to document processing

Table 5: Metric: ep/gr, Quantization: strict, Overlap=off

CO	COS	SSCAS
0.0051(45)	0.0016(30)	0.1001(5)

(e.g. stop words filtering of metadata) and adjustment of the system parameters (sec. 5.1). Additional experimental work with regard to `Focused` tasks is to be done.

7 Conclusion

The paper described the basic tasks of an XML retrieval system. Details on the methodology are provided. An initial experimental evaluation is already, but only partly, conducted showing promising results. However, a thorough empirical work is still needed along with some additional features of the system.

References

1. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley, ACM Press, New York, Essex, England (1999)
2. Salton, G., Lesk, M.E.: Computer evaluation of indexing and text processing. Journal of the ACM **15** (1968) 8–36
3. Salton, G.: The SMART Retrieval System - Experiments in Automatic Document Processing. Prentice Hall Inc., Englewood Cliffs, NJ (1971)
4. Grosjohann, K., Fuhr, N., Effing, D., Kriewel, S.: A user interface for XML document retrieval. In: 32. GI-Jahrestagung. Springer (2002)
5. Grosjohann, K., Fuhr, N., Effing, D., Kriewel, S.: Query formulation and result visualization for XML retrieval. In: Proceedings ACM SIGIR 2002 Workshop on XML and Information Retrieval, ACM (2002)
6. Fuhr, N., Grosjohann, K., Kriewel, S. In: A Query Language and User Interface for XML Information Retrieval. Volume 2818 of LNCS. Springer (2003) 59–75
7. Fuhr, N., Grosjohann, K.: XIRQL: A query language for information retrieval in XML documents. In: Proc. of the 24th ACM SIGIR, ACM Press (2001) 172–180
8. Gövert, N.: Bilingual information retrieval with HyREX and Internet translation services. In: Cross-Language Information Retrieval and Evaluation. Volume 2069 of LNCS. (2001) 237–244
9. Grust, T.: Accelerating XPath location steps. In: Proc. of the 2002 ACM SIGMOD, ACM Press (2002) 109–120
10. Hiemstra, D.: A database approach to content-based xml retrieval. In: INitiative for the Evaluation of XML Retrieval (INEX, Workshop), ERCIM (2003) 111–118
11. Florescu, D., Kossmann, D.: A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report (1999)
12. Abolhassani, M., Fuhr, N.: Applying the divergence from randomness approach for content-only search in XML documents. In: 26th European Conf. on Information Retrieval Research (ECIR), Springer Verlag (2004)
13. Kazai, G., Lalmas, M., Rölleke, T.: Focussed structured document retrieval. In: Proceedings of the 9 Retrieval (SPIRE 2002), Springer (2002) 241–247