XIRQL: A Query Language for Information Retrieval in XML Documents[*]

Norbert Fuhr
University of Dortmund, Germany
fuhr@ls6.cs.uni-dortmund.de

Kai Großjohann
University of Dortmund, Germany
grossjoh@ls6.cs.uni-dortmund.de

ABSTRACT

Based on the document-centric view of XML, we present the query language XIRQL. Current proposals for XML query languages lack most IR-related features, which are weighting and ranking, relevance-oriented search, datatypes with vague predicates, and semantic relativism. XIRQL integrates these features by using ideas from logic-based probabilistic IR models, in combination with concepts from the database area. For processing XIRQL queries, a path algebra is presented, that also serves as a starting point for query optimization.

1. INTRODUCTION

With the steady growth of the WWW, retrieval of Web documents becomes more and more important. HTML, however, provides for visual rather than semantic markup. Information systems need information about the logical structure as a prerequisite for interoperability of Web-based information systems.

In order to overcome these difficulties, the WWW consortium (W3C) developed the XML (extended markup language) standard. Given such a standard, the next step is the definition of a query language that allows for formulation of queries with respect to the logical structure. However, there are two different views on XML which both should be supported:

- The *document-centric* view focuses on XML applications exchanging (structured) documents in the traditional sense, i.e. markup mainly serves for exposing the logical structure of a document.
- The *data-centric* view uses XML for exchanging data in a structured form, like classical EDI applications (for orders, bills and the like), spreadsheets or even whole databases.

[*]This work was funded in part by the German BMBF project "Carmen" (grant no. 08SFC01).

Comparing these two views, it becomes clear that a query language for the data-centric view should be very much in the line of database query languages (see e.g. [8], [5]), whereas the document-centric view should be supported by a language that builds on concepts developed in the area of information retrieval (IR).[1]

Roughly speaking, there are two kinds of IR approaches that deal with the retrieval of structured documents:

- The *structural* approach enriches text search by conditions relating to the document structure, e.g. that words should occur in certain parts of a document, or that a condition should be fulfilled in a document part preceding the part satisfying another condition. The paper [18] gives a good survey on these approaches. However, all these approaches are restricted to Boolean retrieval, so no weighting of index terms and no ranking is considered.
- *Content-based* approaches aim at the retrieval of the most relevant part of a document with respect to a given query. In the absence of explicit structural information, passage retrieval has been investigated by several researches (see for instance [12]). Here the system determines a sequence of sentences from the original document that best fit the query.

Only a few researchers have dealt with the combination of explicit structural information and content-based retrieval. The paper [17] uses belief networks for determining the most relevant part of structural documents, but allows only for plain text queries, without structural conditions. The FERMI multimedia model [6] presents a general framework for relevance-based retrieval of documents. This model formulates the *structured document retrieval principle*: A system should always retrieve the most specific part of a document answering a query. [14] and [10] describe refinements of this approach based on different logical models.

Comparing all these approaches, it turns out that they address different facets of the the XML retrieval problem, but there is no approach that combines the important issues: The data-centric view as well as the structural approach in IR only deal with the structural aspects, but do not support any kind of weighting or ranking. On the other hand, the content-based IR approaches address the weighting issue, but do not allow for structural conditions.

[1]The W3C list of requirements for an XML query language [4] lists the ability for processing simple text conditions as the only IR-related feature besides 18 features that follow from the data-centric view.

In this paper, we present a new query language that combines the structural and the content based approach. We illustrate the problems raised by such a combination, and we describe the new concepts for solving these issues.

In the following, we first briefly describe XML and the query language XQL. Then we discuss the problem of IR on XML documents, and elaborate the features missing from XQL. Based on this discussion, we introduce our new query language XIRQL, and we describe an algebra for processing XIRQL queries. Finally, we give an outlook on future work.

2. XML RETRIEVAL

XML is a text-based markup language similar to SGML. Text is enclosed in *start tags* and *end tags* for markup, and the *tag name* provides information on the kind of *content* enclosed. As an exception to this rule, #PCDATA elements (plain text) have no tags. Elements can be nested, as in the following example:

```
<author><first>John</first>
<last>Smith</last></author>
```

Elements also can be assigned attributes, which are given in the start tag, e.g. `<date format="ISO">2000-05-01</date>`; here the *attribute name* is `format`, and the *attribute value* is `ISO`.

Following is an example XML document, which also illustrates the tree structure resulting from the nesting of elements. Figure 1 shows the corresponding document tree (the dashed boxes are explained later).

```
<book class="H.3.3">
  <author>John Smith</author>
  <title>XML Retrieval</title>
  <chapter>
    <heading>Introduction</heading>
    This text explains all about XML and IR.
  </chapter>
  <chapter>
    <heading>
      XML Query Language XQL
    </heading>
    <section>
        <heading>Examples</heading>
    </section>
    <section>
        <heading>Syntax</heading>
        Now we describe the XQL syntax.
    </section>
  </chapter>
</book>
```

All XML documents have to be *well-formed*, that is, the nesting of elements must be correct (e.g. `<a><b></a></b>` is forbidden). In addition, a *document type definition (DTD)* may be given, which specifies the syntax of set of XML documents. An XML document is *valid*, if it conforms to the corresponding DTD.

As starting point for developing XIRQL, we have chosen XQL, which we describe briefly in the following (for the details see [19]). XQL retrieves elements (i.e. subtrees) of the XML document fulfilling the specified condition. The query `heading` retrieves the four different heading elements from our example document. Attributes are specified with a preceding '`@`' (e.g. `@class`). Context can be considered by means of the child operator '`/`' between two element names, so `section/heading` retrieves only headings occurring as children of sections, whereas '`//`' denotes descendants (`book//heading`). Wildcards can be used for element names, as in `chapter/*/heading`. A '`/`' at the beginning of a query refers to the root node of documents (`/book/title`). The filter operator filters the set of nodes to its left. For example, `//chapter[heading]` retrieves all chapters which have a heading. (In contrast, `//chapter/heading` retrieves only the heading elements of these chapters). Explicit reference to the context node is possible by means of the dot (`.`), e.g. `//chapter[.//heading]` searches for a chapter containing a heading element as descendant. Brackets are also used for subscripts, which indicate position of children within an element, as in `//chapter/section[2]`.

In order to pose restrictions on the content of elements and the value of attributes, comparisons can be formulated. For example, `/book[author="John Smith"]` refers to the value of the element author, whereas `/book[@class="H.3.3"]` compares an attribute value with the specified string. Besides strings, XQL also supports numbers and dates as data types, along with additional comparison operators like `$gt$` and `$lt$` (for > and <).

Subqueries can be combined by means of Boolean operators `$and$` and `$or$` or be negated by means of `$not$`.

These features of XQL allow for flexible formulation of conditions wrt. to structure and content of XML documents. The result is always a set of elements from the original document(s). So XQL follows the document-centric view. Other XML query languages, such as XML-QL [8], focus on the data-centric view, offering a wide variety of operators for restructuring the result as well as aggregation operators, similar to standard database query languages like SQL or OQL. XQuery [1], the current working draft of the XML Query working group at the W3 Consortium, combines the features of XQL with those of XML-QL and thus supports both the document-centric and the data-centric view. Extending XQL with IR features is a first step towards extending XQuery in that direction.

3. XIRQL CONCEPTS

3.1 Requirements

The discussion from above has shown that XQL seems to be a good starting point for IR on XML documents. However, from an IR point of view, the following features are missing in XQL:

Weighting. IR research has shown that document term weighting as well as query term weighting are necessary tools for effective retrieval in textual documents. So comparisons in XQL referring to the text of elements should consider index term weights. Furthermore, query term weighting also should be possible, by introducing a weighted sum operator (e.g. $0.6\cdot$"XML"$+$ $0.4\cdot$"retrieval"). These weights should be used for computing an overall retrieval status value for the elements retrieved, thus resulting in a ranked list of elements.

Relevance-oriented search. The query language also should support traditional IR queries, where only the requested content is specified, but not the type of elements to be retrieved. In this case, the IR system

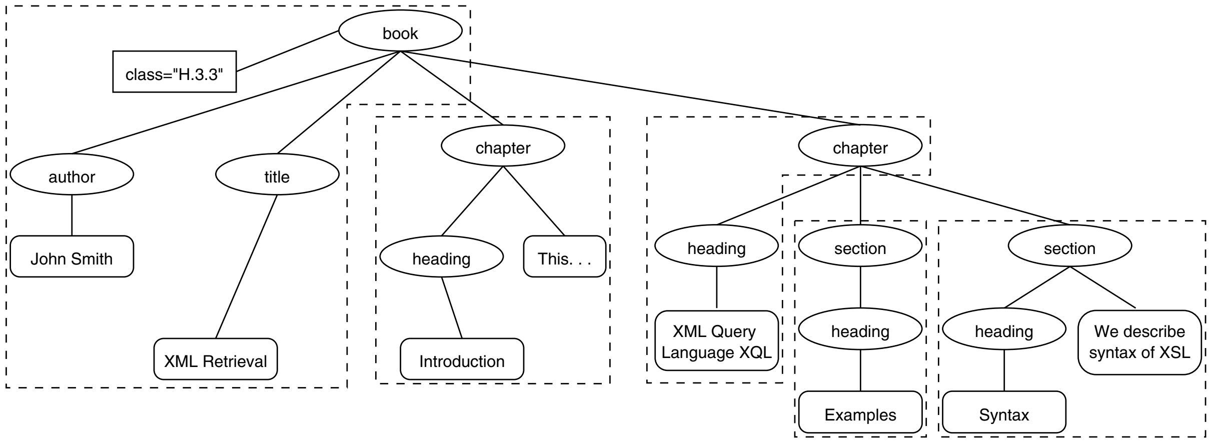

Figure 1: Example XML document tree

should be able to retrieve the most relevant elements; following the FERMI multimedia model cited above, this should be the most specific element(s) that fulfill the query. In the presence of weighted index terms, the tradeoff between these weights and the specificialness of an answer has to be considered, for example by an appropriate weighting scheme.

Data types and vague predicates. Whereas the standard IR approach only deals with one data type, name plain text, XML provides a way to explicitly mark up data items of various data types. Hence, there should be a way to express vague searches for these data types, too. For example, similarity search for proper names should be supported for XML elements containing person names. Numerical 'less' and 'greater' predicates should be provided for elements containing technical measurements, say. Each supported data type should come with a specific set of search predicates, most of which should be vague.

Semantic relativism. XQL is closely tied to the XML syntax, but it is possible to use syntactically different XML variants to express the same kind of meaning. For example, a particular information item could be encoded as an XML attribute or as an XML element. In some cases, a user may wish to search for a value of a specific datatype in a document (e.g. a person name), without bothering about the element names. Thus, appropriate generalizations should be included in the query language.

In the remainder of this section, we describe concepts for integrating the features listed above in XIRQL. Due to space limitations, we do not specify the complete syntax of XIRQL; instead, we focus on the extensions to XQL.

3.2 Weighting

At first glance, extending XQL by a weighting mechanism seems to be straightforward. Assuming probabilistic independence, the combination of weights according to the different Boolean operators is obvious, thus leading to an overall weight for any answer; such an approach has been described in [20]. However, there are two major problems that have to be solved first: 1) How should terms in structured documents be weighted? 2) What are the probabilistic events, i.e. which term occurrences are identical, and which are independent? Obviously, the answer to the second question depends partly on the answer to the first one.

As we said before, classical IR models have treated documents as atomic units, whereas XML suggests a tree-like view of documents. One possibility for term weighting in structured documents would be the development of a completely new weighting mechanism. Given the long experience with weighting formulas for unstructured documents, such an approach would probably take a big effort in order to achieve good performance; furthermore, we would have to cope with the problem of partial dependence of events (see below). As an alternative, we suggest to generalize the classical weighting formulas. Thus, we have to define the "atomic" units in XML documents that are to be treated like atomic documents. The benefit of such a definition is twofold:

1. Given these units, we can apply some kind of tf·idf formula, say, for term weighting.

2. For relevance-oriented search, where no type of result element is specified, only these units can be returned as answers, whereas other elements are not considered as meaningful results.

We start from the observation that text is contained in the leaf nodes of the XML tree only. So these leaves would be an obvious choice as atomic units. However, this structure may be too fine-grained (e.g. markup of each item in an enumeration list, or markup of a single word in order to emphasize it). A more appropriate solution is based on the concept of *index objects* from the FERMI multimedia model: Given a hierarchic document structure, only nodes of specific types form the roots of index objects. In the case of XML, this means that we have to specify the names of the elements that are to be treated as index nodes. This definition can be part of an extended XML DTD.

From the weighting point of view, index objects should be disjoint, such that each term occurrence is considered only once. On the other hand, we should allow for retrieval results of different granularity: For very specific queries, a single paragraph may contain the right answer, whereas more general questions could be answered best by returning a whole chapter of a book. Thus, nesting of index objects should be possible. In order to combine these two views, we first start with the most specific index nodes. For the higher-level index objects comprising other index objects, only the text that is not contained within the other index objects is indexed. As an example, assume that we have defined section, chapter and book elements as index nodes in our example document; the corresponding disjoint text units are marked as dashed boxes in figure 1.

So we have a method for computing term weights, and we can do relevance based search. Now we have to solve the problem of combining weights and structural conditions. For the following examples, let us assume that there is a comparison predicate `$cw$` (contains word) which tests for word occurrence in an element. Now consider the query
`//section[heading $cw$ "syntax"]`
and assume that this word does not only occur in the heading, but also multiple times within the same index node (i.e. section). Here we first have to decide about the interpretation of such a query: Is it a content-related condition, or does the user search for the occurrence of a specific string? In the latter case, it would be reasonable to view the filter part as a Boolean condition, for which only binary weights are possible. We offer this possibility by means of data types with vague predicates (see below).

In the content-related interpretation, there are two possibilities for computing the term weight: We could either compute a weight for this specific structural condition only, or we could use the weight from the corresponding index node. In the first case, there would be the problem of computing the weight on the fly. Furthermore, in case we have a query with multiple structural conditions referring to the same term, it would be very difficult to make sure that the weighting mechanism considers each term occurrence at most once. For example, when applying the query
`/book[.//heading $cw$ "XML" $or$ .//section//*`
`$cw$ "XML"]`
to our example document, one can see that there are several elements which fulfill both structural conditions. In this simple case, one could just count the total number of occurrences fulfilling at least one of the two conditions before applying a weighting function; in general, we would have to compute weights for each of the conditions. Using a probabilistic interpretation, however, the (possible) partial overlapping of the underlying occurrences would imply a partial dependence of the probabilistic events associated with the different query conditions; thus, it would not be possible to specify a correct combination function that leads to a point probability for the result.[2] Besides these technical problems, we think that the context should never be ignored in content-oriented searches, even when structural conditions are specified; these conditions should only work as additional filters. So we take the term weight from the index node. Thus the index node determines the significance of a term in the context given by the node.

With the term weights defined this way, we also have solved the problem of independence/identity of probabilistic events: Each term in each index node represents a unique probabilistic event, and all occurrences of a term within the same node refer to the same event (e.g. both occurrences of the word "syntax" in the last section of our example document represent the same event). Assuming unique node IDs, events can be identified by event keys that are pairs [node ID, term]. For retrieval, we assume that different events are independent. That is, different terms are independent of each other. Moreover, occurrences of the same term in different index nodes are also independent of each other. Following this idea, retrieval results correspond to Boolean combinations of probabilistic events which we call event expressions. For example, a search for sections dealing with the syntax of XQL could be specified as
`//section[.//* $cw$ "XQL" $and$ .//* $cw$ "syntax"]`
Here our example document would yield the conjunction [5, XQL] $\wedge$ [5, syntax]. In contrast, a query searching for this content in complete documents would have to consider the occurrence of the term "XQL" in two different index nodes, thus leading to the Boolean expression
([3, XQL] $\vee$ [5, XQL]) $\wedge$ [5, syntax].

For dealing with these Boolean expressions, we adopt the idea of event keys and event expressions described in [11]. Since the event expressions form a Boolean algebra, we can transform any event expression into disjunctive normal form (DNF), that is:

$$e = C_1 \vee \ldots \vee C_n,$$

where the C_i are event atoms or conjunctions of event atoms, and an event atom is either an event key or a negated event key (n is the number of conjuncts of the DNF). Then the inclusion-exclusion formula (e.g. [3, p. 20]) yields the probability for this event expression as follows:

$$\begin{aligned} P(e) &= P(C_1 \vee \ldots \vee C_n) \\ &= \sum_{i=1}^{n} (-1)^{i-1} \left(\sum_{\substack{1 \le j_1 < \\ \ldots < j_i \le n}} P(C_{j_1} \wedge \ldots \wedge C_{j_i}) \right) \end{aligned}$$

For example, the last example expression from above would be transformed into [3, XQL] $\wedge$ [5, syntax] $\vee$ [5, XQL] $\wedge$ [5, syntax]. Then the resulting probability would be computed as P([3, XQL] $\wedge$ [5, syntax]) + P([5, XQL] $\wedge$ [5, syntax]) $-$ P([3, XQL] $\wedge$ [5, syntax] $\wedge$ [5, XQL] $\wedge$ [5, syntax]). (Note the duplicate event in the last conjunction, which can be eliminated due to idempotency.) Since different events are independent, the probability of the conjunctions can be expressed as the product of the probabilities of the single events, thus resulting in P([3, XQL]) $\cdot$ P([5, syntax]) + P([5, XQL]) $\cdot$ P([5, syntax]) $-$ P([3, XQL]) $\cdot$ P([5, syntax]) $\cdot$ P([5, XQL]).

Following the ideas from [11], this approach can be easily extended in order to allow for query term weighting. Assume that the query for sections about XQL syntax would be reformulated as
`//section[0.6 · .//* $cw$ "XQL" + 0.4 · .//* $cw$`
`"syntax"]`.
For each of the conditions combined by the weighted sum operator, we introduce an additional event with a probability as specified in the query (the sum of these probabilities must not exceed 1). Let us assume that we identify these events as pairs of an ID referring to the weighted sum expression, and the corresponding term. Furthermore, the operator '·' is mapped onto the logical conjunction, and '+'

[2]The only other possible solution would be to define each term occurrence as probabilistic event — but then we would have to deal with the dependence of multiple occurrences of a term in the same XML element.

onto disjunction. For the last section of our example document, this would result in the event expression [q1, XQL] $\wedge$ [5, XQL] $\vee$ [q1, syntax] $\wedge$ [5, syntax]. In order to yield the scalar product, we have to assume that different query conditions belonging to the same weighted sum expression are disjoint events (e.g. $P([\text{q1, XQL}] \wedge [\text{q1, syntax}]) = 0$). For the last section of our example document, the final probability would be computed as
$P([\text{q1, XQL}] \wedge [\text{5, XQL}]) + P([\text{q1, syntax}] \wedge [\text{5, syntax}]) - P([\text{q1, XQL}] \wedge [\text{5, XQL}] \wedge [\text{q1, syntax}] \wedge [\text{5, syntax}])$.
Due to the disjointness of query conditions, the probability of the last conjunct equals 0, and thus we end up with the scalar product of query and document term weights:
$P([\text{q1, XQL}]) \cdot P([\text{5, XQL}]) + P([\text{q1, syntax}]) \cdot P([\text{5, syntax}])$.

3.3 Relevance-oriented search

Above, we have described a method for combining weights and structural conditions. In contrast, relevance-based search omits any structural conditions; instead, we must be able to retrieve index objects at all levels. The index weights of the most specific index nodes are given directly. For retrieval of the higher-level objects, we have to combine the weights of the different text units contained. For example, assume the following document structure, where we list the weighted terms instead of the original text:

```
<chapter> 0.3 XQL
<section> 0.5 example </section>
<section> 0.8 XQL 0.7 syntax </section>
</chapter>
```

A straightforward possibility would be the OR-combination of the different weights for a single term. However, searching for the term ‘XQL’ in this example would retrieve the whole chapter in the top rank, whereas the second section would be given a lower weight. This result contradicts the structured document retrieval principle mentioned before. Thus, we adopt the concept of augmentation from [10]. For this purpose, index term weights are downweighted (multiplied by an augmentation weight) when they are propagated upwards to the next index object.

In [10], augmentation weights (i.e. probabilistic events) are introduced by means of probabilistic rules. In our case, we can attach them to the root element of index nodes. Denoting these events as index node number, the last retrieval example would result in the event expression [1, XQL] $\vee$ [3] $\wedge$ [3, XQL]. Using an augmentation weight of 0.6 for the event [3], the corresponding probability is computed as $P([1, \text{XQL}]) + P([3]) \cdot P([3, \text{XQL}]) - P([1, \text{XQL}]) \cdot P([3]) \cdot P([3, \text{XQL}]) = 0.3 + 0.6 \cdot 0.8 - 0.3 \cdot 0.6 \cdot 0.8 = 0.636$, ranking the section ahead of the chapter.

In the following, paths leading to index nodes are denoted by ‘/\’. As an example, the query `/\[./\* $cw$ "XQL" $and$ ./\* $cw$ "syntax"]` searches for index nodes about ‘XQL’ and ‘syntax’, thus resulting in the event expression ([1, XQL]$\vee$[3]$\wedge$[3, XQL])$\wedge$[2]$\wedge$[2, syntax].

In principle, augmentation weights may be different for each index node. A good compromise between these specific weights and a single global weight may be the definition of type-specific weights, i.e. depending on the name of the index node root element. The optimum choice betweeen these possibilities will be subject to empirical evaluations.

3.4 Data types and vague predicates

Given the possibility of fine-grained markup in XML documents, we would like to exploit this information in order to perform more specific searches. For the content of certain elements, structural conditions are not sufficient, since the standard text search methods are inappropriate. For example, in an arts encyclopedia, it would be possible to mark e.g. artist’s names, locations or dates. Given this markup, one could imagine a query like “Give me information about an artist whose name is similar to Ulbrich and who worked around 1900 near Frankfort, Germany”, which should also retrieve an article mentioning Ernst Olbrich’s work in Darmstadt, Germany in 1899. Thus, we need *vague predicates* for different kinds of data types (e.g. person names, locations, dates). Besides similarity (vague equality), additional datatype-specific comparison operators should be provided (e.g. ‘near’, $<$, $>$, or ‘broader’, ‘narrower’ and ‘related’ for terms from a classification or thesaurus). In order to deal with vagueness, these predicates should return a weight as a result of the comparison between the query value and the value found in the document.

The XML standard itself only distinguishes between three datatypes, namely text, integer and date. The XML schema recommendation[3] extends these types towards atomic types and constructors (tuple, set) that are typical for database systems. For the document-oriented view, most of these data types are useless. In order to support IR in XML documents, there should be a core set of appropriate datatypes. Furthermore, a mechanism for introducing application-specific datatypes should be provided. Text, classification schemes, thesauri, and person names could be supported in the core set. XML, being based on Unicode, allows for coding almost any language in the world, so different subtypes of ‘text’ should be provided to support the various languages. Operations such as stemming and searching for noun phrases and compound words are language-specific. Vague predicates for classification schemes and thesauri should allow for automatic inclusion of related and similar terms. Person names should be searchable by phonetic similarity to compensate for spelling differences due to, for instance, transliteration (“Chebychef”). Different documents provide different detail, so searching for, say, “Jack Smith” should find “J. Smith” as well, with a reduced weight.

Application-specific datatypes must support the similarity of the datatypes that are common in this area. For example, in technical texts, measurement values often play an important role; thus, dealing with the different units, the linear ordering involved ($<$) as well as similarity (vague equality) should be supported (e.g. show me all measurements taken at room temperature). For texts describing chemical elements and compounds, it should be possible to search, say, for elements of compound, or to search for common generalizations (e.g. search for ‘aluminum salts’, without the need to enumerate them).

As a framework for dealing with these problems, we adopt the concept of datatypes in IR from [9], where a datatype T is a pair consisting of a domain $|T|$ and a set of (vague comparison) predicates $C_T = \{c_1, \dots, c_n\}$. It is useful to introduce an inheritance hierarchy (e.g. Text – Western_Language – English), where the subtype restricts the domain and/or provides additional predicates. Through this

[3]`http://www.w3.org/TR/xmlschema-0/`

mechanism, additional datatypes can be defined easily by refining the appropriate datatype (e.g. introduce French as refinement of Western_Language).[4]

In order to exploit these datatypes in retrieval, the datatypes of the XML elements have to be defined. This should happen at the DTD level, as part of an extended XML DTD.

Together with the specification of index nodes, we have two issues that require an extended DTD. This approach contrasts with the initial XQL proposal, where no DTD is required; thus, XQL also can handle well-formed XML, whereas XIRQL is restricted to valid XML documents. This is a natural consequence of the fact that we want to enhance the query semantics: Without additional information, it is impossible to provide functions like relevance-oriented search or vague predicates for specific datatypes.

Another good reason for requiring valid XML documents in order to perform IR is user guidance. For a set of only well-formed XML documents, it would be very hard to formulate meaningful XML queries. Without knowledge about document structure or even element names, most queries would retrieve no documents at all. On the other hand, based on a DTD, it is possible to guide the user in the query formulation process. However, we should mention that we view the role of XIRQL similar to the one that SQL plays in relational databases. Typical end users do not formulate queries in this language; usually, they are offered some form for entering query conditions, from which the user interface generates the correct query syntax.

3.5 Semantic Relativism

Since typical queries in IR are vague, the query language also should support vagueness in different forms. Besides relevance-based search as described above, relativism wrt. elements and attributes seems to be an important feature. The XQL distinction between attributes and elements may not be relevant for many users. In XIRQL, `author` searches an element, `@author` retrieves an attribute and `~author` is used for abstracting from this distinction.

Another possible form of relativism is induced by the introduction of datatypes. For example, we may want to search for persons in documents, without specifying their role (e.g. author, editor, referenced author subject of a biography) in these documents. Thus, we provide a mechanism for searching for certain data types, regardless of their position in the XML document tree. For example, `#persname` searches for all elements and attributes of the datatype persname.

4. PROCESSING XIRQL QUERIES

In this section, we describe a path algebra for processing XIRQL queries. Due to space limitation, we only describe the major concepts. We do not give a formal specification of the transformation of XIRQL queries into the algebra. Furthermore, some specific features of XIRQL are not addressed here (e.g. indexes and weighted sum).

The major purpose of the description below is the specification of the behavior of the different operators. First, we give some basic definitions concerning datatypes, the document base and event expressions.

[4]Please note that we make no additional assumptions about the internal structure of the text datatype (and its subtypes), like representing text as set or list of words.

As mentioned before, we use the notion of IR datatypes from [9], where a datatype T is a pair consisting of a domain $|T|$ and a set of (vague) predicates C_T; a subtype restricts the domain and/or extends the set of predicates.

DEFINITION 1. *A data type T is a pair $(|T|, C_T)$, where $|T|$ is the domain and $C_T = \{c_1, \ldots, c_n\}$ is the set of (vague comparison) predicates, where each predicate is a function $c_i{:}|T| \times |T| \to [0,1]$. Let $\mathcal{T}$ denote the set of all data types, and $\mathcal{D} = \cup_{t\in\mathcal{T}}|T|$ be the union of all domains.*

DEFINITION 2. *The subtype relationship $\preceq_{\mathcal{T}} \subset \mathcal{T}\times\mathcal{T}$ is a hierarchic relationship and a partial order on $\mathcal{T}$, which also fulfills the following condition:*

$$T \preceq_{\mathcal{T}} T' \Rightarrow |T| \subseteq |T'| \wedge C_T \supseteq C_{T'}.$$

Let $T_\top = (\mathcal{D}, \emptyset)$ denote the top element, of which all other types are subtypes.

For modeling an XML document base, we modify the FERMI multimedia model appropriately. In the following, we drop the distinction between XML elements and attributes and refer to both of them as elements.

Like in the database field, a document base consists of a schema and an instance. In our case, the schema is given by the extended DTD. That is, we have information about the structure of valid XML documents plus the datatype information. For value-based retrieval we are only interested in the leaf elements that have a nonempty content; thus we assume that all other elements are assigned the datatype $T_\top$. Since the structural constraints are irrelevant for the topic of this paper, we assume that they are given as a set of semantic constraints which are not explained any further (see R in Definition 4 below).

DEFINITION 3. *A document base is a pair $D = (S, I)$, where S is the schema and I is the instance.*

DEFINITION 4. *The schema of a document base is a tuple*

$$S = (N, \iota, X, \tau, R),$$

where: N is a set of element names occurring in the DTD, ι is the name of the root element, $X \subseteq N$ is the set of element names of index node roots, τ is a mapping $\tau : N \to \mathcal{T}$ that specifies the data type for each element name, and R is a set of semantic constraints that follows from the DTD.

For specifying the instance of a document base, we assume that it consists of a set of XML elements having a name and possibly data as content, with aggregative and sequential relationships in between. In order to keep our model simple, we do not explicitly specify the notion of a document here.[5]

DEFINITION 5. *The document base instance I is a tuple*

$$I = (E, \prec_{str}, \prec_{seq}, \nu, \tau, \delta)$$

where: E is a set of XML elements; $\prec_{str}$ is an aggregative relation on E that defines the hierarchical composition between elements; $\prec_{seq}$ is a partial order on E that describes the sequential order among elements; ν is a mapping $E \to N$ that gives the name of each element; δ is a partial mapping $E \to \mathcal{D}$ yielding the content of an element e if $\nu(e) \in L$, with the restriction $\delta(e) \in |\tau(\nu(e))|$.

[5]Documents should be modeled as disjoint subsets of E, and the relations $\prec_{str}$ and $\prec_{seq}$ only contain pairs of elements belonging to the same document.

Between the elements E of a document base instance, there is an aggregative relation $\prec_{str}$ that models the child-parent relationship: $e \prec_{str} e'$ if e is a child of e'. The sequential relationship $\prec_{seq}$ describes the order among all children of a parent node: $e \prec_{seq} e'$ if e and e' are children of the same parent element and e' follows e. The function $\nu(e)$ gives us the name of element e, and $\delta(e)$ gives the content of leaf elements.

The general idea for processing XIRQL queries is the manipulation of sets of paths. This approach is similar to the proximal nodes model [18].[6] However, we give a more formal specification of the semantics of the different operators. Furthermore, we extend this model by dealing with datatypes and weighting.

A path is a sequence of elements, where each pair of subsequent elements is in the aggregative relation $\prec_{str}$. In addition, each path starts with a dummy element o which is introduced in order to simplify subsequent definitions.

Definition 6. *For a document base instance* $I = (E, \prec_{str}, \prec_{seq}, \nu, \tau, \delta)$, *a path is a list* $p = (e_0, e_1, \ldots, e_n)$ *with* $n \geq 0$ *and* $e_0 = o$ *(the dummy element). Unless* $n = 0$, *the following also must hold:* $\nu(e_1) = \iota$ *and* $e_i \in E$ *for* $1 \leq i \leq n$; *in addition, for* $1 \leq j \leq n-1$, $e_{j+1} \prec_{str} e_j \wedge \neg\exists e' : e_{j+1} \prec_{str} e' \prec_{str} e_j$. *Let* P *denote the set of all paths that can be formed from* I.

Furthermore, let $lst(p) = e_n$ *and* $head(p) = (e_0, e_1, \ldots, e_{n-1})$.

In order to deal with weighting, we are using event keys and event expressions. The former identify the probabilistic events, whereas the latter describe Boolean combinations of events. In order to distinguish event expressions from ordinary Boolean expressions, we use underlined Boolean operators for the former.

Definition 7. *A set of* event keys $\mathbf{EK}$ *is a set of identifiers, that also contains the special elements* $\perp$ *(always false) and* $\top$ *(always true).*

The set of event expressions $\mathbf{EE}$ *is defined recursively:*

1. $w \in \mathbf{EK} \rightarrow w \in \mathbf{EE}$.
2. $w \in \mathbf{EE} \rightarrow \underline{\neg} w \in \mathbf{EE}$.
3. $w, w' \in \mathbf{EE} \rightarrow w \underline{\wedge} w' \in \mathbf{EE}$ *and* $w \underline{\vee} w' \in \mathbf{EE}$
4. *These are all event expressions.*

As shorthand for the disjunction $w_1 \underline{\vee} w_2 \underline{\vee} \ldots \underline{\vee} w_n$, *we also use the notation* $\underline{\bigvee}_i w_i$.

Based on the notion of paths and event expressions, we can now discuss the notion of XIRQL queries. Given a document base, a query should produce a result set consisting of pairs (path, event expression). The path points to the XML element to be retrieved. Below, we will show that we need a second path in order to handle intermediate results. In a subsequent step, the event expressions are used for computing the probabilistic weight for each answer, as described before. XIRQL operators take one or two result sets as input and produce another result set as output. This model is similar to query processing in standard text retrieval, where inverted list entries (consisting of document IDs and indexing weights) are combined in order to produce a result list of document IDs with weights. However, our path algebra approach is flexible enough to allow for other kinds of processing as well, e.g. using different kinds of access paths or processing parts of the query by scanning a set preselected of documents

First, we need a transformation operator from a set of paths into a query result:

Definition 8. *Let* R *denote a set of paths. Then the operator* ε *is defined as:*
$\varepsilon(R) = \{(p, p, \top) | p \in R\}$

By applying ε onto the set P of all paths, we can get a starting point for the other operators.

In classical text retrieval, the basic operator is single term retrieval: Given a term, it returns a set of document IDs with weights. In our case, a term corresponds to a triple (datatype, predicate, comparison value). Since we are dealing with structured documents, the document ID is extended to the path describing the element where the condition matched. Instead of a simple weight, we return an event key (with an associated weight), in order to compute the resulting probability in a correct way.

Definition 9. *Let* T *denote a datatype,* $V \in |T|$ *a comparison value and* $\tilde{c}$ *be the name of a predicate* $c \in C_T$. *Furthermore, let* $w = event(v, e, T, \tilde{c}, V)$ *denote a function that generates an event key with probability* v *for the result of applying the value selection condition* $[T\, \tilde{c}\, V]$ *on the element* e. *Then value selection on a query result* Q *is defined as* $\omega[T\, \tilde{c}\, V](Q) = \{(p, r, w) | (p, r, w') \in Q \wedge lst(r) = e \wedge \tau(e) \preceq_{\mathcal{T}} T \wedge c(V, \delta(e)) = v \wedge w = w' \underline{\wedge}\, event(v, e, T, \tilde{c}, V)\}$

Query results consist of triples (processing path, result path, event expressions). As an example, consider the simple query `/*/chapter/section[heading $cw$ "syntax"]`. For our example document, value selection would return two paths, namely `/book[1]/chapter[2]/section[2]/heading[1]/#PCDATA[1]` and `/book[1]/chapter[2]/section[2]/#PCDATA[1]` (in our examples, we use relative indexes for identifying the elements). In order to test the structural conditions, we check them in a bottom-up way. During this process, we have to distinguish between the path that leads to the result element (in our case `section` elements) and the position in the path where we test the next structural condition. For illustrating this procedure, let us enclose the processing path in parentheses, while the full path always represents the (current) result path. As output from value selection, the example paths from above are both processing and result paths. Testing for the `heading` condition in the filter, we get the result `(/book[1]/chapter[2]/section[2])`. Next, we have to test for the `/section` condition, without moving the result pointer, thus giving us `(/book[1]/chapter[2])/section[2]`. In the same way, we test the `/chapter` condition and the condition `/*`, thus yielding finally `()/book[1]/chapter[2]/section[2]`.

Now consider a variant of the query from above: `/*/chapter/section[./* $cw$ "syntax"]`. Here the value selection would yield the same paths as before, which would also both pass the filter. Thus, our query result contains twice the path `(/book[1]/chapter[2]/section[2])`. Now let us look at the event expressions, which would be the

[6]The close relationship between XQL and proximal nodes is discussed in [2].

event key [5, syntax] in both cases.[7] Logically, when the result paths are equal, we have to form the disjunction of the corresponding event keys, thus eliminating the duplicate element of the result in this case. As another example, consider the query `/*/chapter[.//* $cw$ "XQL"]`, where value selection would yield the path-event combinations (`/book[1]/chapter[2]/section[2]/#PCDATA[1]`, [5, XQL]) and (`/book[1]/chapter[2]/heading[1]`, [3, XQL]). Here the test on the structural condition `/chapter` would identify two equal paths, but with different event keys, thus yielding the result (`(/book[1])/chapter[2]`, [3, XQL] $\vee$ [5, XQL]).

The last problem concerning the evaluation of structural conditions is the notation `//`; in this case, we have to consider all possible subpaths of each argument path. In contrast to other operators or conditions, this condition increases the size of the result.

Based on these considerations, we can now give the definition of the structural projection operator Π and the structural selection operator σ (similar to relational algebra, where projection also modifies the structure of the result, whereas selection only filters elements from the input).

Definition 10. *Let S denote a query result and c a structural condition of the form '/', '//', '/*' or '/a' (where 'a' denotes an element name). For a path $p = (e_0, e_1, \dots, e_n)$, we define a function* $proj(c, p) =$

$$\begin{cases} \{(e_0)\} & \text{if } c = \text{'/'} \wedge n = 0 \\ \{(e_0, e_1, \dots, e_j) | 0 \leq j \leq n\} & \text{if } c = \text{'//'}, \\ \{(e_0, e_1, \dots, e_{n-1})\} & \text{if } c = \text{'/a'} \wedge n \geq 1 \wedge \\ & \nu(e_n) = a, \\ \{(e_0, e_1, \dots, e_{n-1})\} & \text{if } c = \text{'/*'} \wedge n \geq 1, \\ \emptyset & \text{otherwise.} \end{cases}$$

Then we define the following operations

$$\Pi[c](S) = \{(r, r, w) | T = \{(p', r', w') | r \in proj(c, r') \wedge (p', r', w') \in S\} \wedge T \neq \emptyset \wedge w = \underline{\bigvee}_{(p', r', w') \in T} w'\}$$

$$\sigma[c](S) = \{(p, r, w) | (p', r, w) \in S \wedge p \in proj(c, p')\}\}$$

The binary operators are fairly straightforward: we combine two elements, if they contain identical result and processing paths, and the event expressions are combined according to the semantics of the operator. As a variant of intersection, the subpath operator '/' only considers equality of processing paths and then takes the result path from its right argument. As an example, consider the query `/book [@class $clsim$ "H.3.3"] /chapter [./heading $cw$ "XQL"]` For our example document, the first filter condition would produce the path `(/book[1])`, whereas the second filter and the subsequent test on `/chapter` would yield `(/book[1])/chapter[2]`. The subpath operator would produce the second path as result (plus the conjunction of the corresponding event expressions).

Like in relational algebra, negation in XIRQL queries is mapped onto difference of intermediate results. If no other argument is given, we form the difference to the complete database; for example, the query `/book[$not$ title]` searching for all documents that have no title is transformed into $\sigma[\text{/book}](\varepsilon(P)) - \sigma[\text{/book}](\Pi[\text{/title}](\varepsilon(P)))$

Definition 11. *Let S and T denote two query results. Then we define the following operations:*

$$S \cap T = \{(p, r, w | \exists (p, r, w') \in S \wedge \exists (p, r, w'') \in T \wedge w = w' \underline{\wedge} w''\}$$

$$S / T = \{(p, r, w | \exists (p, r', w') \in S \wedge \exists (p, r, w'') \in T \wedge w = w' \underline{\wedge} w''\}$$

$$S \cup T = \{(p, r, w) | \exists (p, r, w') \in S \wedge \exists (p, r, w'') \in T \wedge w = w' \underline{\vee} w'' \vee \exists (p, r, w) \in S \wedge \neg \exists (p, r, w') \in T \; \exists (p, r, w) \in T \wedge \neg \exists (p, r, w') \in S\}$$

$$S - T = \{(p, r, w) | \exists (p, r, w') \in S \wedge \exists (p, r, w'') \in T \wedge w = w' \underline{\vee} \underline{\neg} w'' \vee \exists (p, r, w') \in S \wedge \neg \exists (p, r, w'') \in T \wedge w = w'$$

Finally, we specify the relevance selection operator for relevance-oriented search. Its definition is similar to projection for the descendant operator, but it yields only paths leading to index nodes, and it adds events corresponding to augmentation weights to the event expression.

Definition 12. *For $\nu(e) \in X$ (names of root elements of index nodes) let $rw(e)$ denote a function that gives the augmentation weight for e along with the corresponding event. Then relevance selection is defined as*

$$\varrho(Q) = \begin{cases} \tilde{\varrho}(Q) \cup \varrho(\Pi[/*](Q)) & \text{if } Q \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

where $\tilde{\varrho}$ is defined as
$\tilde{\varrho}(Q) = \{(r, r, w) | (p', r', w') \in Q \wedge lst(r') = e \wedge \nu(e) \in X \wedge r = head(r') \wedge w = rw(w') \underline{\wedge} w'\}$.

Based on these specifications, we can transform XIRQL queries into combinations of XIRQL operators. We give two examples illustrating this process:

`/book//section[title $cw$ "syntax" $and$ #PCDATA $cw$ "XQL"]` is mapped onto $\sigma[/](\sigma[\text{/book}](\sigma[//](\sigma[\text{/section}](\Pi[\text{/title}](\omega[\text{text \$cw\$ “syntax”}](\varepsilon(P)) \cap \Pi[\text{\#PCDATA}] (\omega[\text{text \$cw\$ “XQL”}](\varepsilon(P))))))))$

`/book [@class $clsim$ "H.3.3"] /chapter [./heading $cw$ "XQL"]` can be expressed as $\sigma[/](\sigma[\text{/book}](\Pi[\text{@class}](\omega[\text{class \$clsim\$ “H.3.3”}](\varepsilon(P))) / (\sigma[\text{/chapter}](\Pi[\text{/heading}](\omega[\texttt{text} \text{ \$cw\$ “XQL”}](\varepsilon(P)))))))$

It should be obvious that the transformation step is rather straightforward, only negation requires some special attention.

In terms of database systems, here we have described the logical algebra only. The actual implementation of query processing has to be based on a physical algebra, where the operators make additional assumptions, like about the availability of access paths and the sorting order of objects. A major task of the query optimization step is the mapping of logical operators onto appropriate physical operators; in addition, the logical algebraic expression can be optimized first. For this purpose, we have to identify transformation rules of the path algebra that keep the result unchanged (as in relational algebra, some of these rules only hold in one direction [15]), e.g.

$$\Pi[c](S \cap T) \longrightarrow \Pi[c](S) \cap \Pi[c](T)$$
$$\sigma[c](\omega[s](Q)) \longleftrightarrow \omega[s](\sigma[c](Q))$$

[7]For illustration purposes, we keep the notation of event keys more simple than required by the definition of the function event(.).

The first rule tells us that we can move a projection inside the arguments of an intersection operator, (e.g. for reducing the size of intermediate results), but not vice versa (e.g. when $c =$ '/*'). The second rule allows us to exchange the processing order of structural and value selections; this may be useful for exploiting the nature of the access paths available (e.g. value-oriented inverted lists vs. structure-oriented access paths). After developing the complete path algebra, we can apply standard query optimization techniques from the area of database systems (see e.g. [13]); however, since most users are interested in the top-ranking documents only, additional work may be necessary in order to modify the query optimization step accordingly.

5. CONCLUSIONS AND OUTLOOK

In this paper, we have described a new query language for information retrieval in XML documents. Current proposals for XML query languages lack most IR-related features, which are weighting and ranking, relevance-oriented search, datatypes with vague predicates, and semantic relativism. We have presented the new query language XIRQL which integrates all these features, and we have described the concepts that are necessary in order to arrive at a consistent model for XML retrieval. For processing XIRQL queries, we have described a path algebra, which also serves as a starting point for query optimization. In parallel, XIRQL can be extended to include the data-centric features from XQuery.

We have implemented a first prototype retrieval engine that accepts XIRQL queries, transforms them into a path algebra expression and then processes this expression. Currently, we are using a simple extension of inverted files as access method. In order to achieve efficiency, we will investigate different kinds of access methods and different processing strategies, which will form the basis for query optimization. This work is part of a project that will develop an open source retrieval engine for XML retrieval, especially for digital libraries. Further research is necessary to make the full functionality of XIRQL accessible to the end-user; form-based interfaces to execute queries with a predefined structure are easily built, however.

6. REFERENCES

[1] XQuery: A query language for XML, Feb. 2001. `http://www.w3.org/TR/xquery/`.

[2] R. Baeza-Yates and G. Navarro. XQL and proximal nodes. In *Proceedings ACM SIGIR 2000 Workshop on XML and Information Retrieval*, 2000. `http://www.haifa.il.ibm.com/sigir00-xml/final-papers/RBaetza/att1.htm`.

[3] P. Billingsley. *Probability and Measure*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Inc, New York, 1979.

[4] D. Chamberlin, F. Fankhauser, M. Marchiori, and J. Robie. XML query requirements, 2000. `http://www.w3.org/TR/xmlquery-req`.

[5] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000. `http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf`.

[6] Y. Chiaramella, P. Mulhem, and F. Fourel. A model for multimedia information retrieval. Technical report, FERMI ESPRIT BRA 8134, University of Glasgow, Apr. 1996. `http://www.dcs.gla.ac.uk/fermi/tech\_reports/reports/fermi96-4.ps.gz`.

[7] Croft et al., editor. *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, 1998. ACM.

[8] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. In Marchiori [16]. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[9] N. Fuhr. Towards data abstraction in networked information retrieval systems. *Information Processing and Management*, 35(2):101–119, 1999.

[10] N. Fuhr, N. Gövert, and T. Rölleke. Dolores: A system for logic-based retrieval of multimedia objects. In Croft et al. [7], pages 257–265.

[11] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 14(1):32–66, 1997.

[12] M. Hearst and C. Plaunt. Subtopic structuring for full-length document access. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 59–68, New York, 1993. ACM.

[13] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16:111–152, 1984.

[14] M. Lalmas. Dempster-Shafer's theory of evidence applied to structured documents: Modelling uncertainty. In N. J. Belkin, A. D. Narasimhalu, and P. Willet, editors, *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 110–118, New York, 1997. ACM.

[15] D. Maier. *Relational Database Theory*. Computer Science Press, Rockville, Md., 1983.

[16] M. Marchiori, editor. *QL'98 — The Query Languages Workshop*. W3C, Dec. 1998. `http://www.w3.org/TandS/QL/QL98/`.

[17] S. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo. A flexible model for retrieval of sgml documents. In Croft et al. [7], pages 138–145.

[18] G. Navarro and R. Baeza-Yates. Proximal nodes: a model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435, 1997.

[19] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In Marchiori [16]. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[20] A. Theobald and G. Weikum. Adding relevance to XML. In *3rd International Workshop on the Web and Databases (WebDB)*, 2000. `http://www-dbs.cs.uni-sb.de/papers/webdb2000.ps`.