XIRQL: An XML Query Language Based on Information Retrieval Concepts

Norbert Fuhr, Kai Großjohann
University of Dortmund, Germany

December 17, 2001

Abstract

Most proposals for XML query languages are based on the data-centric view on XML and do not support uncertainty and vagueness, thus being insuitable for information retrieval (IR) of XML documents. Based on the document-centric view, we present the query language XIRQL which implements IR-related features such as weighting and ranking, relevance-oriented search, datatypes with vague predicates, and structural 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 which also serves as a starting point for query optimization.

1 Introduction

More and more, XML is acknowledged as standard document format, especially for Web applications. In contrast to HTML which is mainly layout-oriented, XML follows the fundamental concept of separating the logical structure of a document from its layout. The major purpose of XML markup is the explicit representation of the logical structure of a document (whereas the layout of a document (type) is described in a separate style sheet).

Given the logical markup, different kinds of operations referring to the logical structure can be performed on XML documents: Multiple views on a document can be generated (e.g. for different audiences), specific elements of an XML document can be extracted, or documents fulfilling specific structural conditions can be retrieved from a document base. Overall, if information is represented in XML format, exchange of this information between different software systems (especially on the Web) is simplified, thus supporting interoperability.

From an IR point of view, the role of XML as the forthcoming standard format for fulltext documents offers new opportunities. Since XML supports logical markup of texts both at the macro level (e.g. chapter, section, paragraph) and the micro level (e.g. MathML for mathematical formulas, CML for chemical formulas), retrieval methods dealing with both kinds of markup should be developed. At the macro level, fulltext retrieval should allow for selection of appropriate parts of a document in response to a query, such as by returning a section or a paragraph instead of the complete document. At the micro level, specific similarity operators for different types of text or data should be provided (e.g. similarity of chemical structures, phonetic similarity for person names).

Although a large number of query languages for XML have been proposed in recent years, none of them fully addresses the IR issues related to XML; especially, the XQuery proposal of the W3C working group [Fernandez & Marsh 01] offers almost no support for IR-oriented querying of XML sources. There are only a few approaches that provide partial solutions to the IR problem, namely by taking into account the intrinsic imprecision and vagueness of IR; however, none of them is based on a consistent model of uncertainty (see section 6).

In this paper, we present a new query language XIRQL that combines the major concepts of XML querying with those from IR. XIRQL is based on a subset of XQuery, which we extend by IR concepts, along with giving a consistent model for dealing

with the uncertainty issue.

In the following, we first briefly describe XML and XML path expressions. Then we discuss the problem of IR on XML documents, and present the major concepts of our new query language XIRQL (section 3). Section 4 describes the underlying algebra for processing XIRQL queries, and section 5 deals with the transformation of XIRQL into algebra expressions. A survey on related work is given in section 6, followed by the final conclusions and outlook.

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 can also 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, in section 3.2).

```
<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 (`<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.

When our development of XIRQL started, we chose the query language XQL ([Robie et al. 98], [Robie et al. 99]) as starting point. The XQuery proposal [Fernandez & Marsh 01] has adopted a slight variant of XQL (XPath, see [Clark & DeRose 99]) for its path expression part (see section 6). Here we give a brief description of XQL.

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 e.g. `section/heading` retrieves only headings occurring as children of sections, whereas '`//`' denotes descendants (e.g. `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 (e.g. `/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 (`.`): `//chapter[.//heading]` searches for a chapter containing a heading element as descendant. Brackets are also used for subscripts indicating the position of children within an element, with separate counters for each element type; for example `//chapter/section[2]` refers to the second section in a chapter (which is the third child of the second chapter in our example document).

In order to pose restrictions on the content of elements and the value of attributes, comparisons

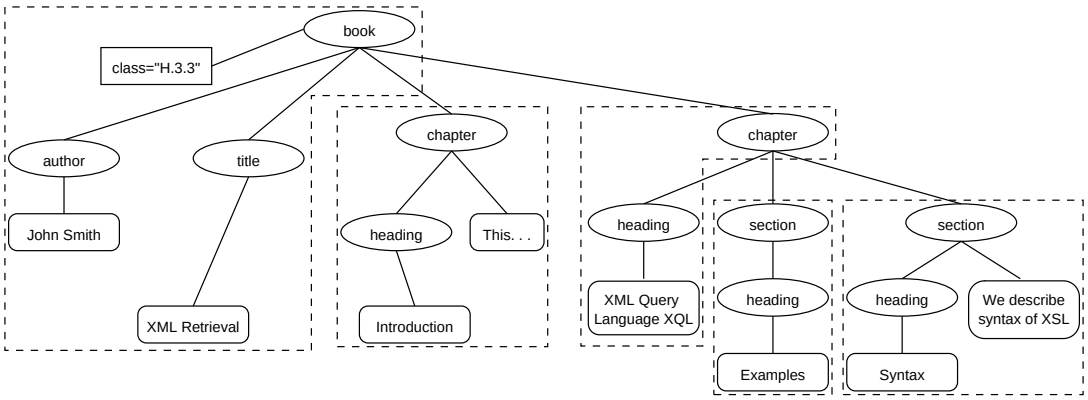

Figure 1: Example XML document tree

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`.

For considering the sequence of elements, the operators `before` and `after` can be used, as in `//chapter[section/heading = "Examples" before section/heading = "Syntax"]`.

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).

3 XIRQL concepts

3.1 Requirements

From an IR point of view, the combination of content with logical markup in XML offers the following opportunities for enhancing IR functionality in comparison to plain text:

- Queries referring to content only should retrieve relevant document parts according to the logical structure, thus overcoming the limitations of passage retrieval. The FERMI model [Chiaramella et al. 96] suggests the following strategy for the retrieval of structured (multimedia) documents: A system should always retrieve the most specific part of a document answering the query.

- Based on the markup of specific elements, high-precision searches can be performed that look for content occurring in specific elements (e.g. distinguishing between the sender and the addressee of a letter, finding the definition of a concept in a mathematics textbook).

- The concept of *mixed content* allows for the combination of high precision searches with plain text search. An element contains mixed content if both plain text (#PCDATA) as well as other elements may occur in it. Thus, it is possible to mark up specific items occurring in a text. For example, in an arts encyclopedia, names of artists, places they worked, and titles of pieces of art may be marked up (thus allowing for example, to search for Picasso's paintings of toreadors, avoiding passages mentioning Picasso's frequent visits to bull-fights).

With respect to these requirements, XQL seems to be a good starting point for IR on XML documents. However, the following features should be added to 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 should also 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 should also 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 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 specifity of an answer has to be considered, e.g. by an appropriate weighting scheme.

Data types and vague predicates. The standard IR approach for weighting supports vague searches on plain text only. XML allows for a fine grained markup of elements, and thus, there should be the possibility to use special search predicates for different types of elements. For example, for an element containing person names, similarity search for proper names should be offered; in technical documents, elements containing measurement values should be searchable by means of the comparison predicates $>$ and $<$ operating on floating point numbers. Thus, there should be the possibility to have elements of different data types, where each data type comes with a set of specific search predicates. In order to support the intrinsic vagueness of IR, most of these predicates should be vague (e.g. search for measurements that were taken at about 20 °C).

Structural relativism. XQL is closely tied to the XML syntax, but it is possible to use syntactically different XML variants to express the same meaning. For example, a particular information could be encoded as an XML attribute or as an XML element. As another example, 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.

3.2 Weighting

At first glance, extending XQL by a weighting mechanism seems to be a 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. 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 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 e.g. some kind of $\mathrm{tf} \cdot \mathrm{idf}$ formula 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 the XML Schema (see below).

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 of 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, in would be reasonable to view the filter part as a Boolean condition, for which only binary weights are possible. We offer this possibility by providing data types with a variety of predicates, where some of them are Boolean and others are vague (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

```
/document[.//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.[1] 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 have also 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

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

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 $[5, \text{XQL}] \wedge [5, \text{syntax}]$. 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 $([3, \text{XQL}] \vee [5, \text{XQL}]) \wedge [5, \text{syntax}]$.

For dealing with these Boolean expressions, we adopt the idea of event keys and event expressions described in [Fuhr & Rölleke 97]. 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. [Billingsley 79, 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 $([3, \text{XQL}] \wedge [5, \text{syntax}]) \vee ([5, \text{XQL}] \wedge [5, \text{syntax}])$. Then the resulting probability would be computed as $P([3, \text{XQL}] \wedge [5, \text{syntax}]) + P([5, \text{XQL}] \wedge [5, \text{syntax}]) - P([3, \text{XQL}] \wedge [5, \text{syntax}] \wedge [5, \text{XQL}] \wedge [5, \text{syntax}])$. (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 $P([3, \text{XQL}]) \cdot P([5, \text{syntax}]) + P([5, \text{XQL}]) \cdot P([5, \text{syntax}]) - -P([3, \text{XQL}]) \cdot P([5, \text{syntax}]) \cdot P([5, \text{XQL}])$.

Following the ideas from [Fuhr & Rölleke 97], 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 '$\cdot$' is mapped onto the logical conjunction, and '+' onto disjunction. For the last section of our example document, this would result in the event expression $[q1, \text{XQL}] \wedge [5, \text{XQL}] \vee [q1, \text{syntax}] \wedge [5, \text{syntax}]$. 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. $P([q1, \text{XQL}] \wedge [q1, \text{syntax}]) = 0$). For the last section of our example document, the final probability would be computed as $P([q1, \text{XQL}] \ \wedge \ [5, \text{XQL}]) + P([q1, \text{syntax}] \ \wedge \ [5, \text{syntax}]) - P([q1, \text{XQL}] \wedge [5, \text{XQL}] \wedge [q1, \text{syntax}] \wedge [5, \text{syntax}])$.
Due to the disjointness of query conditions, the probability of the last conjunct equals zero, and thus we end up with the scalar product of query and document term weights: $P([q1, \text{XQL}]) \cdot P([5, \text{XQL}]) + P([q1, \text{syntax}]) \cdot P([5, \text{syntax}])$.

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 orig-

inal 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. It can be easily shown that this strategy always assigns the highest weight to the most general element. This result contradicts the structured document retrieval principle mentioned before. Thus, we adopt the concept of augmentation from [Fuhr et al. 98]. For this purpose, index term weights are downweighted (multiplied by an augmentation weight) when they are propagated upwards to the next index object. In our example, using an augmentation weight of 0.6, the retrieval weight of the chapter wrt. to the query 'XQL' would be $0.3+0.6\cdot 0.8-0.3\cdot 0.6\cdot 0.8 = 0.596$, thus ranking the section ahead of the chapter.

For similar reasons as above, we use event keys and expressions in order to implement a consistent weighting process (e.g. equivalent query expressions should result in the same weights for any given document). In [Fuhr et al. 98], 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 $[1, \text{XQL}] \vee [3] \wedge [3, \text{XQL}]$.

In the following, paths leading to index nodes are denoted by 'inode()' and recursive search with downweighting is indicated via '...'. As an example, the query `/document//inode()[... cw "XQL" and ... cw "syntax"]` searches for index nodes about 'XQL' and 'syntax', thus resulting in the event expression $([1, \text{XQL}] \vee [3] \wedge [3, \text{XQL}]) \wedge [2] \wedge [2, \text{syntax}]$.

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 investigations.

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 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 [Fallside 01] extends these types towards atomic types and constructors (tuple, set) that are typical for database systems.

For the document-oriented view, this notion of data types is of limited use. This is due to the fact that most of the data types relevant for IR applications can hardly be specified at the syntactic level (consider for instance names of a geographic locations, or English vs. French text). In the context of XIRQL, data types are characterized by their sets of vague predicates (such as phonetic similarity of names, English vs. French stemming). Thus, for supporting IR in XML documents, there should be a core set of appropriate datatypes and there should be a mechanism for adding application-specific datatypes.

Candidates for the core set are texts in different languages, hierarchical classification schemes,

thesauri and person names. In order to perform text searches, some knowledge about the kind of text is necessary. Truncation and adjacency operators available in many IR systems are suitable for western languages only (whereas XML in combination with unicode allows for coding of most written languages). Therefore, language-specific predicates, e.g. for dealing with stemming, noun phrases and composite words should be provided. Since many documents may contain elements in multiple languages, the language problem should be handled at the datatype level.[2] Classification schemes and thesauri are very popular now in many digital library applications; thus, the relationships from these schemes should be supported, e.g. by including narrower or related terms in the search. Vague predicates for this datatype should allow for automatic inclusion of terms that are similar according to the classification scheme. Person names often pose problems in document search, as the first and middle names may sometimes be initials only (so, searching for "Jack Smith" should also retrieve "J. Smith", with a reduced weight). A major problem is the correct spelling of names, especially when transliteration is involved (e.g. "Chebychef"); thus, phonetic similarity or spelling-tolerant search should be provided.

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 e.g. for elements of compounds, 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 [Fuhr 99], where a datatype T is a pair consisting of a domain $|T|$ and a set of (vague comparison) predicates $P_T = \{c_1, \ldots, c_n\}$. Like in other type systems, IR data types should also be organized in a type hierarchy (e.g. Text – Western_Language – English), where the subtype restricts the domain and/or provides additional predicates (e.g. n-gram matching for general text, plus adjacency and truncation for western languages, plus stemming and noun phrase search for English). Through this mechanism, additional datatypes can be defined easily by refining the appropriate datatype (e.g. introduce French as refinement of Western_Language)[3].

In order to exploit these datatypes in retrieval, the datatypes of the XML elements have to be defined. Thus, in addition to the DTDs of the documents, we need some schema information. Although the XML Schema recommendation [Fallside 01] is targeted towards the data-centric view of XML, it can also be used for our purpose. Most of the data types discussed above are simple types in terms of XML Schema (i.e. have no internal structure), but do not belong to the builtin types of XML Schema. Thus, they have to be derived by means of restriction from the builtin types. However, in most cases, it is not possible to give necessary conditions for the restriction (e.g. English as a specialization of normalizedString). On the other hand, XML Schema does not deal with (vague) predicates of data types; they can be listed as application info only and are treated like comments by the schema processor.

By using XML Schema (although in a non-standard way), our approach contrasts with the initial XQL proposal, which requires neither a schema nor a DTD; thus, XQL can also handle well-formed XML, whereas XIRQL is restricted to XML documents satisfying a given schema declaration. 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. As a minimum requirement, XIRQL can also operate with valid XML documents only (where the DTD is given in the form of an XML Schema that specifies only the DTD structure, but no data types).

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 meaning-

[2] Cross-lingual retrieval should be implemented on top of the retrieval language.

[3] 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.

ful 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 schema, 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 Document classes and hyperlinks

In many applications, documents will belong to different schemas. For this reason, we assume that a document base may contain different *document classes*. All documents belonging to a single class conform to the same schema. When formulating a XIRQL query the name of the document class addressed has to be specified first, (e.g. `class(book)//chapter[heading cw "XML"]`). As syntactical sugar, the class declaration can be omitted in case it is identical with the name of the top-level element named in the query.

As a major extension over XQL, XIRQL also supports hyperlinks. Along with the possibility of different document classes in a document base, this feature allows for powerful querying of XML documents. For example, assume that we have a document class `article` where citation references are given as links to the cited article. Using the operator `=>` for dereferencing, we can search for all articles that are cited by at least one publication in the following way: `/article/citations/cite=>/article`. Selection criteria may occur on both sides of the link. For example, searching for articles by Jones cited by anybody can be accomplished by placing a restriction on the target side: `/article/citations/cite=>/article[au = "Jones"]`. The next example returns results from the source side of the link, where we search for papers by Smith citing Jones' articles: `/article[au = "Smith" and ./citations/cite=>/article[au = "Jones"]]`. Conceptually, this type of query can also be seen as a way of following links in the 'inverse' direction.

Since XIRQL cannot construct new documents, the result of a query is either an element from the source side or from the target side, but not a combination of both. If the link operator occurs within a filter, the results are elements of the source document class, otherwise from the target class. However, XIRQL allows for restrictions both on the source and target side of a link.

For document bases with several document classes, the classes of both the source and the target of a link must be specified in the query. As an example, assume that we have the document class `book` in addition to `article`. Then we would need a second query for retrieving Jones' books cited in Smith's articles: `/article[au = "Smith" and ./citations/cite=>/book[au = "Jones"]]`.

Hyperlinks may not only refer to complete documents, they may point to any element in any document. For example, assume that there are citation links pointing to chapters of books, and we want so see the headings of these chapters only: `/article/citations/cite=>class(book) chapter/heading`.

In general, a query cannot follow links between arbitrary document classes. At the schema level, both the source and the target of a hyperlink are XML elements. However, only the source element must be specified explicitly in the XML Schema definition, by using the builtin type URI.

3.6 Structural Relativism

Since typical queries in IR are vague, the query language should also 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.

Further abstraction from the concrete XML syntax is possible by introducing datatypes. For example, a date value can be represented in various forms in an XML document, as illustrated the following example:

```
<date year="2001" month="12" day="11"/>
<date>2001-12-11</date>
<date><year>2001</year>
      <month>12</month>
      <day>11</day></date>
```

With the 'date' datatype, users just specify the date in a standard format in their query and don't need to know how dates happen to be represented in the current document class.

3.7 XIRQL Syntax

In the previous sections, examples for XIRQL queries have been presented. Table 1 gives a specification of the complete syntax using EBNF, which is derived from the grammar for XQL. In addition to literal character strings, the grammar uses the following terminal symbols: ELEMENT matches XML element names, ATTRIBUTE matches XML attribute names, CLASS_NAME matches document class names, STRING matches single-quoted and double-quoted strings, and PREDICATE matches predicate names such as e.g. `cw` or `clsim`.

4 Processing XIRQL queries

In this section, we describe a path algebra for processing XIRQL queries.

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

4.1 Schemas and paths

As mentioned above, XIRQL can only process XML documents conforming to an XML Schema. However, since XIRQL offers no special operators for dealing with complex types, we can use a simplified view on data types here. We treat all data types as simple types, and only need to consider the subtype relationship between types. Following the notion of IR datatypes from [Fuhr 99], a datatype T is a pair consisting of a domain $|T|$ and a set of (vague) predicates P_T; a subtype restricts the domain and/or extends the set of predicates.

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

$$T \preceq_{\mathcal{T}} T' \implies |T| \subseteq |T'| \wedge P_T \supseteq P_{T'}.$$

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

From XIRQL's point of view, complex datatypes (e.g. a person name as a sequence of first name and last name) have no internal structure, their structure is hidden by the implementation of the data type. Thus, complex data types usually will be direct subtypes of the top element (except for specializations of complex data types).

Based on this interpretation of datatypes, XIRQL conditions referrring to the content of elements usually will address leaf nodes only. Internal nodes will have complex data types, for which appropriate predicates will not be available in most cases. A major exception to this statement is the invocation of functions for data type comparison (e.g. the text() function), thus mapping a complex data type into a simple one, for which XIRQL offers appropriate predicates.

For modeling an XML document base, we draw on ideas from the FERMI multimedia model as well as from the XQuery semantics. Like with the latter, we drop the distinction between XML elements and attributes and refer to both of them as elements.

As with databases, a document base consists of a schema and an instance. Similar to relational databases containing multiple relations (or object-oriented databases with multiple classes), we as-

Query	::=	Sequence
Sequence	::=	Disjunction
	|	Disjunction (`"before"` | `"after"`) Sequence
Disjunction	::=	Conjunction
	|	Conjunction `"or"` Disjunction
Conjunction	::=	Negation
	|	Negation `"and"` Conjunction
Negation	::=	Union
	|	`"not"` Negation
Union	::=	Intersection
	|	Intersection UnionOp Union
UnionOp	::=	`"union"` | `"|"`
Intersection	::=	Comparison
	|	Comparison `"intersect"` Intersection
Comparison	::=	Path | LValue CompOp RValue
CompOp	::=	PREDICATE
LValue	::=	Path
RValue	::=	Path | Number | Text
Path	::=	AbsolutePath | RelativePath
AbsolutePath	::=	Root
	|	`"/"` RelativePath
	|	`"//"` RelativePath
	|	Class AbsolutePath
Class	::=	`"class("` CLASS_NAME `")"`
RelativePath	::=	Filter
	|	Filter `"/"` RelativePath
	|	Filter `"//"` RelativePath
Filter	::=	Grouping
	|	Filter `"["` IndexList `"]"`
	|	Filter `"["` Subquery `"]"`
	|	Filter `"=>"` NameTest
NameTest	::=	Class Element | Class | Element
IndexList	::=	IndexArg
	|	IndexArg `","` IndexList
IndexArg	::=	Integer | Range
Range	::=	Integer `"-"` Integer
Subquery	::=	Sequence
Grouping	::=	RelativeTerm | `"("` Sequence `")"`
RelativeTerm	::=	`"."` | `"..."` | Element | Attribute
Element	::=	ELEMENT | `"*"`
Attribute	::=	`"@"` ATTRIBUTE
ParameterList	::=	Parameter
	|	Parameter `","` ParameterList
Parameter	::=	Sequence | Number | Text
Text	::=	STRING

Table 1: EBNF for XIRQL Syntax

sume that a document base contains multiple document classes, where the documents of each class conform to one document schema.

Since XIRQL only deals with the access to elements of existing XML documents (without constructing new documents), we do not describe a complete document model here. Thus we do not address the issue of the structural constraints of documents, we assume that they are given as a set of semantic constraints which are not explained any further.

Definition 3 *A document base is a pair* $\mathbf{D} = (\mathbf{S}, \mathbf{I})$*, where* $\mathbf{S}$ *is the schema and* $\mathbf{I}$ *is the instance.*

Definition 4 *The schema of a document base is a tuple*

$$\mathbf{S} = (S_1, \ldots, S_m)$$

with

$$S_i = (M_i, N_i, X_i, \tau_i, R_i) \quad \textit{for } i = 1 \ldots m$$

where

M_i *is the class name*

N_i *is a set of element names occurring in the DTD of class* S_i*, plus '/' (the name of the root element),*

$X_i \subseteq N_i$ *is the set of element names of index node roots,*

τ_i *is a mapping* $\tau : N_i \rightarrow \mathcal{T}$ *that specifies the data type for each element name,*

R_i *is a set of semantic constraints that follows from the XML Schema of class* S_i*.*

For specifying the instance of a document base, we assume that each document class consists of a set of XML elements having a name and content of the datatype specified, with aggregative and sequential relationships in between.

In order to model hyperlinks, we have an additional hyperlink relation on pairs of document elements. Hyperlinks may occur within the same document, between documents of the same class or even between elements from different document classes.

Definition 5 *For a document base* $\mathbf{D} = (\mathbf{S}, \mathbf{I})$ *with schema* $\mathbf{S} = (S_1, \ldots, S_m)$*, the document base instance* $\mathbf{I}$ *is a tuple*

$$\mathbf{I} = (\mathbf{C}_1, \ldots \mathbf{C}_m, H)$$

with

$$\mathbf{C}_i = (E_i, \prec_i, \kappa_i, \lambda_i, \nu_i, \tau_i, \delta_i) \quad \textit{for } i = 1 \ldots m$$

where

E_i *is a set of XML elements.*

$\prec_i \subseteq E_i \times E_i$ *is an aggregative relation on* E_i *that defines the hierarchical composition between elements.*

κ_i *is a mapping* $E_i \rightarrow \mathbb{N}$ *that describes the sequential order among elements that are children of the same parent element.*

λ_i *is a mapping* $E_i \rightarrow \mathbb{N}$ *that gives the relative index among children with the same name that belong to the same parent.*

ν_i *is a mapping* $E_i \rightarrow N_i$ *that gives the name of each element.*

δ_i *is a mapping* $E_i \rightarrow \mathcal{D}$ *yielding the content of an element* e *with the restriction* $\delta_i(e) \in |\tau_i(\nu_i(e))|$*.*

$H \subseteq \mathbf{E} \times \mathbf{E}$ *is the hyperlink relation, where*

$$\mathbf{E} = \bigcup_{i=1}^{m} E_i$$

Furthermore, let $\kappa = \cup_{i=1}^{m} \kappa_i$ *and* $\lambda = \cup_{i=1}^{m} \lambda_i$*.*

Between the elements E of a document class instance, there is an aggregative relation $\prec_i$ that models the parent-child relationship: $e \prec_i e'$ if e is the parent of e'. The function $\nu(e)$ gives us the name of element e, and $\delta(e)$ gives the content of leaf elements.

The sequential order among all children of a parent node is given by the index function κ_i, which also satisfies the condition $e \prec_i e' \wedge e \prec_i e'' \implies (\kappa_i(e') = \kappa_i(e'') \iff e = e'')$. In addition, the function λ_i gives the relative index for children of the same type, thus satisfying the condition $e \prec_i e' \wedge e \prec_i e'' \wedge \lambda_i(e') = \lambda_i(e'') \implies e' = e'' \vee \nu_i(e') \neq \nu_i(e'')$.

Due to the fact that XIRQL only allows for accessing document elements, the objects manipulated by XIRQL are mainly paths, not complete XML documents.

A path is a sequence of elements, where each pair of adjacent elements is in the aggregative relation $\prec_i$. Similar to the definition of the XQuery semantics, we assume that there is a root element for each document (with the special name '/'), which has exactly one child, namely the top-level element of the corresponding document class.

Definition 6 *For a document class instance* $\mathbf{C}_i = (E_i, \prec_i, \kappa_i, \lambda_i, \nu_i, \tau_i, \delta_i)$, *a path is a list* $p = (e_0, e_1, \ldots, e_n)$ *with* $n \geq 0$ *and* $\nu_i(e_0) =$'/' *and* $e_j \in E_i$ *for* $1 \leq j \leq n$; *in addition, for* $1 \leq k \leq n-1$, $e_k \prec_i e_{k+1} \wedge \nexists e' : e_k \prec_i e' \prec_i e_{k+1}$. *Let* C_i *denote the set of all paths that can be formed from* $\mathbf{C}_i$, *and let* $\mathcal{C} = \cup_{i=1}^{m} C_i$. *(In the following, we will identify the class name* M_i *with the set* C_i.*)*

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

For two paths $p = (e_0, e_1, \ldots, e_n)$ *and* $p' = (e'_0, e'_1, \ldots, e'_m)$ *, we define the following relations*

$p \subseteq p'$ *if* $n \leq m$ *and* $e_i = e'_i$ *for* $i = 0 \ldots n$,

$p < p'$ *if* $e_i = e'_i$ *for* $i = 0 \ldots k$ *for some* $k \geq 0$ *with* $k < \min(n, m)$ *and* $\kappa(e_{k+1}) < \kappa(e'_{k+1})$.

Here $p \subseteq p'$ denotes containment of paths, i.e. the element pointed to by p contains the element pointed to by p'. $p < p'$ refers to the sequence of elements, being true iff p points to an element that comes before the element pointed to by p'.

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

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

The set of event expressions **EE** *is defined recursively as the smallest set satisfying the following conditions:*

1. $w \in \mathbf{EK} \rightarrow w \in \mathbf{EE}$.
2. $w \in \mathbf{EE} \rightarrow \underline{\neg} w \in \mathbf{EE}$.
3. $w, w' \in \mathbf{EE} \rightarrow w \underline{\wedge} w' \in \mathbf{EE}, w \underline{\vee} w' \in \mathbf{EE}$

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

4.2 Path algebra

The general idea for processing XIRQL queries is the manipulation of sets of paths. 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:*
$\varepsilon(R) = \{(p, p, \top) | p \in R\}$. *As a shorthand notation for* $\varepsilon(R)$, *we will write* $\overline{R}$ *in the following.*

By applying ε onto the set of paths of a document class, we 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* e *denote an element,* T *a datatype,* $V \in |T|$ *a comparison value and* $\tilde{c}$

be the name of a predicate $c \in P_T$. *Then* $\text{event}(v, e, T, \tilde{c}, V)$ *is defined to be a function which generates an event key with probability* v *for the result of applying the value selection condition* $T\,\tilde{c}\,V$ *on the element* e.

Note that we expect $\text{event}(v, e, T, \tilde{c}, V) = \text{event}(v, e', T, \tilde{c}, V)$ if e and e' are in the same index node. See the discussion on relevance-oriented search in section 3.3 on page 7.

Definition 10 *Let* e, T, V, $\tilde{c}$, *and* v *be as in the previous definition. Furthermore, let* $w = \text{event}(v, e, T, \tilde{c}, V)$. *Then value selection on a query result* Q *is defined as* $\omega[T\,\tilde{c}\,V](Q) = \{(p, r, w) | \exists e\, \exists v\, \exists w'\ (p, r, w') \in Q \wedge \text{lst}(r) = e \wedge \tau(e) \preceq_{\mathcal{T}} T \wedge c(V, \delta(e)) = v \wedge w = w' \triangleq \text{event}(v, e, T, \tilde{c}, V)\}$

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]` and `/book[1]/chapter[2]/section[2]/#PCDATA[1]`.[4] 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 event key $[5, \text{syntax}]$ in both cases.[5] 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]`, $[5, \text{XQL}]$) and (`/book[1]/chapter[2]/heading[1]`, $[3, \text{XQL}]$). 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]`, $[3, \text{XQL}] \vee [5, \text{XQL}]$).

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 modifies the structure of the result, whereas selection only filters elements from the input).

Definition 11 *Let* S *denote a query result and* $c = s[i]$ *a condition, where* i *denotes a set of indexes (which may also be empty) and* s *is structural condition of the form '/', '//', '*' or 'a' (where 'a' denotes an element name). For a path* $p = (e_0, e_1, \ldots, e_n)$, *we define a function*

$$\text{proj}(s[i], p) = \begin{cases} \text{struc}(s, p), & \text{if } \lambda(e_n) \in i \vee i = \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

[4] In our examples, we denote paths $p = (e_1, \ldots, e_n)$ by writing sequences of $(\nu(e_i), \lambda(e_i))$ pairs, separated by slashes.

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

with $\text{struc}(s,p) =$

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

Then we define the following operations

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

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

As shorthands for complex structural conditions, we define

$$\begin{aligned} \Pi[c](S) &:= \Pi[.](\sigma[c](S)) \quad \textit{and} \\ \sigma[c/c'](S) &:= \sigma[c](\sigma[c'](S)) \quad \textit{and} \\ \Pi[c/c'](S) &:= \Pi[c](\Pi[c'](S)) \end{aligned}$$

For relevance-oriented search, we extend the definition of selection. We introduce the condition '/\' as a variant of the descendant operator '//'. The only difference between the two operators lies in the consideration of augmentation weights when paths are truncated through the function $\text{struct}(s,p)$: Whenever we chop off an element (from the processing path) which is an index node, then the corresponding augmentation weight of this element e should be considered. For this purpose, we assume that augmentation weights are given as part of the class instance.

Definition 12 *For each document class instance* $\mathbf{C}_i$*, there is a function* $\alpha_i : E_i \rightarrow \mathbf{EK}$ *that yields a probabilistic event representing the augmentation weight of elements, with the restriction* $\nu_i(e) \notin X_i \implies \alpha_i(e) = \top$*.*

For two paths $r \subseteq r' \in C_i$ *with* $r = (e_0, e_1, \ldots, e_n)$ *and* $r' = (e_0, e_1, \ldots, e_n, e_{n+1}, \ldots, e_m)$*, the function* $\text{rw}_i : C_i \times C_i \rightarrow \mathbf{EE}$ *is defined as follows:*

$$\text{rw}_i(r, r') = \top \mathbin{\underline{\wedge}} \bigwedge_{k=n+1}^{m} \alpha_i(e_k)$$

Let S *denote a query result with paths from class* $\mathbf{C}_i$*. Then relevance selection* $\sigma[/\backslash](S)$ *is defined as*

$\sigma[/\backslash](S) = \{(p,r,w) | T = \{(p',r',w') | p \in \text{struc}(//,p') \wedge (p',r',w') \in S\} \wedge T \neq \emptyset \wedge w = \underline{\bigvee}_{(p',r',w') \in T}(w' \mathbin{\underline{\wedge}} \text{rw}_i(p,p'))\}$

The definition of relevance selection handles the weighting part of relevance-oriented search. In order to retrieve only index nodes as answers, the XIRQL query is transformed internally by listing the names of index node elements as alternative types of answers. For example, the query `class(book)//inode() [... cw "XML"]` would be transformed into `class(book)//(document | chapter | section) [... cw "XML"]`.

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 `/document[not title]` searching for all documents that have no title is transformed into $\sigma[/\text{document}](\overline{R} - \Pi[\text{title}](\overline{R}))$

For the XIRQL operators `before` and `after`, the corresponding algebra operators $<$ and $>$ are processed by means of pairwise comparison of paths using the relation '$<$'.

Definition 13 *Let* S *and* T *denote two query results. Then we define the following operations:*

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

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

$S \cup T = \{(p,r,w) || \exists w' \exists w'' \quad (p,r,w') \in S \wedge \exists (p,r,w'') \in T \wedge w = w' \veebar w'' \vee \exists (p,r,w) \in S \wedge \nexists (p,r,w') \in T \exists (p,r,w) \in T \wedge \nexists (p,r,w') \in S\}$

$S - T = \{(p,r,w) | \exists w' \quad (p,r,w') \in S \wedge ((\exists w'' \quad (p,r,w'') \in T \wedge w = w' \veebar \sqsupset w'') \vee ((\nexists w'' (p,r,w'') \in T) \wedge w = w'))\}.$

$S < T = \{(p,r,w) | \exists w' \exists w'' \quad (p,r,w') \in S \wedge (p',r',w'') \in T \wedge w = w' \barwedge w'' \wedge r < r'\}.$

$S > T = \{(p,r,w) | \exists w' \exists w'' \quad (p,r,w') \in S \wedge (p',r',w'') \in T \wedge w = w' \barwedge w'' \wedge r' < r\}.$

$\alpha \cdot S + \beta \cdot T = \{(p,r,w) |) \exists w' \exists w'' \ (p,r,w') \in S \wedge (p,r,w'') \in T \wedge w = \tilde{\alpha} \barwedge w' \veebar \tilde{\beta} \barwedge w'') \vee (\exists w' \ (p,r,w') \in S \wedge (\nexists w'' (p,r,w'') \in T) \wedge w = \tilde{\alpha} \barwedge w') \vee (\exists w'' (p,r,w'') \in T \wedge (\nexists w' (p,r,w') \in S) \wedge w = \tilde{\beta} \barwedge w'')\}$

In the definition of the weighted sum operator, $\tilde{\alpha}$ and $\tilde{\beta}$ denote query-specific event keys with the corresponding probabilities α and β.

With the operators described so far, we can already transform most 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

σ[/book//section](
Π[title](ω[text cw "syntax"]($\overline{R}$)) $\cap$
Π[#PCDATA](ω[text cw "XQL"]($\overline{R}$)))

`/book[@class clsim "H.3.3"]/chapter[./heading cw "XQL"]` can be expressed as

σ[/book](Π[@class](ω[class clsim "H.3.3"]($\overline{R}$)) /
(σ[chapter](Π[heading](ω[text cw "XQL"]($\overline{R}$)))))

Details of the transformation process are described in the next section.

For following hyperlinks and for comparing the values of two XML elements, we need two additional operators. Since they are similar to joins in relational databases, we also call them join operators. In order to follow hypertext links, we define link join:

Definition 14 *For a document base instance* $\mathbf{I} = (\mathbf{C}_1, \ldots \mathbf{C}_m, H)$ *and two query results* R, S*, the following link join operations are defined:*

$R => S := \{(s,s',w'') | \exists r \exists r' \exists w \exists w' \ (r',s) \in H \wedge (r,r',w) \in R \wedge (s,s',w') \in S \wedge w'' = w \barwedge w'\}$

$R >= S := \{(r,r',w'') | \exists s \exists s' \exists w \exists w' \ (r',s') \in H \wedge (r,r',w) \in R \wedge (s,s',w') \in S \wedge w'' = w \barwedge w'\}$

Note that the => operator refers to the processing path of the target, whereas the >= operator uses the target's result path. In the former case, we want to be able to follow the link and navigate down to a more specific element, and this can be accomplished only by referring to the processing path. As an example for this problem, assume that we have a class `article` containing citation references are given as links; searching for authors of articles cited by anybody can be accomplished by the query: `/article/citations/cite=>/article/au`,
with the corresponding path algebra expression (σ[/article/citations/cite]($\overline{\text{article}}$)) => σ[/article/au]($\overline{\text{article}}$). For focusing on the source side of the link, assume a query searching for papers citing Jones' articles: `/article[citations/cite=>/article[au = "Jones"]]`, which yields in path algebra: σ[/article](Π[citations/cite]($\overline{\text{article}}$ >= (σ[/article/au](ω[persname = "Jones"]($\overline{\text{article}}$))))).

The *value join* operator is similar to value selection, but instead of comparing a constant value specified in the query with the value of an element, it compares the value of two elements.

Definition 15 *Let* T *denote a datatype,* $\tilde{c}$ *be the name of a predicate* $c \in P_T$ *and the function* event(.) *be defined as in definition 10. For two query results* R *and* S *the* value join *operator is defined as*

$R \bowtie [T\tilde{c}] \ S = \{(p,r,w) | \exists p'' \exists r'' \exists w' \exists w'' \exists e' \exists e'' \exists v \ (p,r,w') \in R \wedge (p'',r'',w'') \in S \wedge e' = \text{lst}(r) \wedge e'' = \text{lst}(r'') \wedge c(\delta(e'),\delta(e'')) = v \wedge w = w' \barwedge w'' \barwedge \text{event}(v,e',T,\tilde{c},\delta(e''))\}.$

We need the value join for the case where the comparison operator in a filter expression is not a literal, but another XML element. For example, the query `book[editor=./chapter/author]` searches for books where the editor is also the author of one of its chapters, which yields in our algebra:
σ[book](Π[.](σ[editor]($\overline{R}$) $\bowtie$[*persname*=] (σ[chapter](σ[author]($\overline{R}$)))).

4.3 Processing path algebra expressions

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 e.g. 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, e.g.

$$\begin{aligned} \sigma[c](S \cap T) &\longrightarrow \sigma[c](S) \cap \sigma[c](T) \\ \sigma[c](\omega[s](Q)) &\longleftrightarrow \omega[s](\sigma[c](Q)) \end{aligned}$$

The first rule tells us that we can move a structural selection inside the arguments of an intersection operator, (e.g. for reducing the size of intermediate results). 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. [Jarke & Koch 84]); 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 Conversion and implementation

5.1 Conversion to logical algebra

In most cases, the mapping from XIRQL queries to path algebra expressions is fairly obvious, but in some cases, complex transformations are needed. Some examples for the relationship between XIRQL and the path algebra are presented in the other sections of this paper; this section contains a set of rules for converting any XIRQL query to an equivalent path algebra expression. The XIRQL query should be parsed according to the EBNF given in table 1, then the top-level element of the parse tree should be matched against the rules shown in table 2, in turn, until one of them matches. From there, subtrees of the parse tree should be matched, and so on, until the whole query is converted.

For example, the query `/book/au = "Smith"` would be parsed as an instance of the rule "Comparison ::= LValue CompOp RValue", which can be found in the transformation rules as x `op` V, so that line 16 applies.

The transformation is specified as a function "cvt" of three arguments. The first argument is the query fragment to process, the second argument is a "source specification", and the last argument is a Boolean value which says whether we are inside a XIRQL 'filter' operator (square brackets `[ ]`, EBNF rule "Filter ::= Filter `"["` Subquery `"]"`"). We will use the variable *infilter* when talking about this Boolean value. When converting a full XIRQL query, i.e. an expression satisfying the "Query" nonterminal given in the EBNF, the value of *infilter* should be `false`.

The source specification can be an arbitrary path algebra expression, or it can be a class name M_i, which is identified with $\varepsilon(C_i)$, C_i being the set of paths in that document class. The implementation assures that some class name is always passed to the cvt function; the user can choose which class name that should be.

For notational convenience, we also introduce some shortcuts. It turns out that some rules come in pairs; the only difference is that one rule uses Π where the other uses σ, and the value of *infilter* is different. To avoid having to write (almost) the same rule twice, we will write θ which is understood to mean Π if the value of *infilter* is `true`, whereas it means σ if the value of *infilter* is `false`.

The same principle applies to dealing with links. We will write $\ominus$ which is understood to mean >= if *infilter* is `true`, whereas it means => if *infilter* is `false`.

And finally, some variables on the left hand side of a rule have a restricted set of values. We use a, b, l, r, and x, for arbitrary XIRQL (sub-)queries, but *elem* is restricted to element names (or attribute names or wildcards). So for example the rule for *elem* `/` r only applies if the left hand side of the `/` operator is indeed an element name. The vague predicates `op` from XIRQL (e.g. `cw` or `clsim`) are mapped onto the corresponding predicate $\odot$ of the algebra.

$$
\begin{aligned}
\mathrm{cvt}(\texttt{.}, \mathit{src}, \mathit{infilter}) &= \mathit{src} \\
\mathrm{cvt}(\texttt{...}, \mathit{src}, \mathit{infilter}) &= \Pi[.](\sigma[/\backslash](\mathit{src})) \\
\mathrm{cvt}(\mathit{elem}, \mathit{src}, \mathit{infilter}) &= \theta[\mathit{elem}](\mathit{src}) \\
\mathrm{cvt}(\texttt{/}\,\mathit{elem}, \mathit{src}, \mathit{infilter}) &= \theta[/](\theta[\mathit{elem}](\mathit{src})) \\
\mathrm{cvt}(\texttt{//}\,\mathit{elem}, \mathit{src}, \mathit{infilter}) &= \theta[\mathit{elem}](\mathit{src}) \\
\mathrm{cvt}(\texttt{/}\,x, \mathit{src}, \mathit{infilter}) &= \theta[/](\mathrm{cvt}(x, \mathit{src}, \mathit{infilter})) \\
\mathrm{cvt}(\texttt{//}\,x, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(x, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(\mathrm{class}(M_i)\ \texttt{/}\ x, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(\texttt{/}\,x, M_i, \mathit{infilter}) \\
\mathrm{cvt}(\mathrm{class}(M_i)\ \texttt{//}\ x, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(\texttt{//}\,x, M_i, \mathit{infilter}) \\
\mathrm{cvt}(l\ \texttt{and}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(l, \mathit{src}, \mathit{infilter}) \cap \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(l\ \texttt{or}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(l, \mathit{src}, \mathit{infilter}) \cup \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(l\ \texttt{and not}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(l, \mathit{src}, \mathit{infilter}) - \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(\texttt{not}\ x, \mathit{src}, \mathit{infilter}) &= \mathit{src} - \mathrm{cvt}(x, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(l\ \texttt{before}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(l, \mathit{src}, \mathit{infilter}) < \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(l\ \texttt{after}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(l, \mathit{src}, \mathit{infilter}) > \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(x\ \texttt{op}\ V, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(x, \omega[T \odot V](\mathit{src}), \mathit{infilter}) \\
\mathrm{cvt}(x\ \texttt{op}\ y, \mathit{src}, \texttt{false}) &= \mathrm{cvt}(x, \mathit{src}, \texttt{false}) \bowtie[T\odot] \mathrm{cvt}(y, \mathit{src}, \texttt{false}) \\
\mathrm{cvt}(x\ \texttt{op}\ y, \mathit{src}, \texttt{true}) &= \Pi[.](\mathrm{cvt}(x, \mathit{src}, \texttt{false}) \bowtie[T\odot] \mathrm{cvt}(y, \mathit{src}, \texttt{false})) \\
\mathrm{cvt}(\mathit{elem}[r], \mathit{src}, \mathit{infilter}) &= \theta[\mathit{elem}](\mathrm{cvt}(r, \mathit{src}, \texttt{true})) \\
\mathrm{cvt}(\mathit{elem}\ \texttt{/}\ r, \mathit{src}, \mathit{infilter}) &= \begin{cases} \theta[\mathit{elem}](\mathrm{cvt}(a, \mathit{src}, \mathit{infilter})) \ominus b \\ \qquad \text{if } \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) = (a \ominus b) \\ \theta[\mathit{elem}](\mathrm{cvt}(r, \mathit{src}, \mathit{infilter})) \qquad \text{otherwise} \end{cases} \\
\mathrm{cvt}(\mathit{elem}\ \texttt{//}\ r, \mathit{src}, \mathit{infilter}) &= \begin{cases} \theta[//](\theta[\mathit{elem}](\mathrm{cvt}(a, \mathit{src}, \mathit{infilter}))) \ominus b \\ \qquad \text{if } \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) = (a \ominus b) \\ \theta[//](\theta[\mathit{elem}](\mathrm{cvt}(r, \mathit{src}, \mathit{infilter}))) \qquad \text{otherwise} \end{cases} \\
\mathrm{cvt}(\texttt{.}\ \texttt{/}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) \\
\mathrm{cvt}(\texttt{.}\ \texttt{//}\ r, \mathit{src}, \mathit{infilter}) &= \begin{cases} \theta[//](\mathrm{cvt}(a, \mathit{src}, \mathit{infilter})) \ominus b & \text{if } \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) = (a \ominus b) \\ \theta[//](\mathrm{cvt}(r, \mathit{src}, \mathit{infilter})) & \text{otherwise} \end{cases} \\
\mathrm{cvt}(\mathit{fl}[\mathit{fr}]\ \texttt{/}\ r, \mathit{src}, \mathit{infilter}) &= \begin{cases} (\mathrm{cvt}(\mathit{fl}[\mathit{fr}], \mathit{src}, \mathit{infilter}) \ /\ \mathrm{cvt}(a, \mathit{src}, \mathit{infilter})) \ominus b \\ \qquad \text{if } \mathrm{cvt}(r, \mathit{src}, \mathit{infilter}) = (a \ominus b) \\ (\mathrm{cvt}(\mathit{fl}[\mathit{fr}], \mathit{src}, \mathit{infilter}) \ /\ \mathrm{cvt}(r, \mathit{src}, \mathit{infilter})) \\ \qquad \text{otherwise} \end{cases} \\
\mathrm{cvt}(\mathit{fl}[\mathit{fr}]\ \texttt{//}\ r, \mathit{src}, \mathit{infilter}) &= \begin{cases} (\mathrm{cvt}(\mathit{fl}[\mathit{fr}], \mathit{src}, \mathit{infilter}) \ /\ \theta[//](\mathrm{cvt}(a, \mathit{src}, \mathit{infilter}))) \ominus b \\ \qquad \text{if } \theta[//](\mathrm{cvt}(r, \mathit{src}, \mathit{infilter})) = (a \ominus b) \\ \mathrm{cvt}(\mathit{fl}[\mathit{fr}], \mathit{src}, \mathit{infilter}) \ /\ \theta[//](\mathrm{cvt}(r, \mathit{src}, \mathit{infilter})) \\ \qquad \text{otherwise} \end{cases} \\
\mathrm{cvt}(x\texttt{=>}\mathrm{class}(M_i)\mathit{elem}\ \texttt{/}\ r, \mathit{src}, \mathit{infilter}) &= \mathrm{cvt}(x, \mathit{src}, \texttt{true}) \ominus \theta[\mathit{elem}](\mathrm{cvt}(r, M_i, \mathit{infilter}))
\end{aligned}
$$

Table 2: Transformation rules

Most of these rules are fairly straight-forward, but there are two areas of dissimilarity between XIRQL and the path algebra which require special treatment. The first area concerns the `class(X)` syntactical element and the second one concerns the treatment of links.

From a high-level point of view, a XIRQL query can be viewed as a sequence of 'steps', separated by slashes (or double slashes, as the case may be). In the simple case, where each step is just an element name, the query is converted into a sequence of σ operators. For example, the query `/a/b/c` is parsed as `/(a/(b/(c)))`. However, if the XIRQL query specifies a document class via `class(X)` on the very left, then the class name needs to be used as the input of the right-most σ operator. The conversion rules are designed in such a way that a class name from the `class(X)` operator is passed down the chain of slashes until it reaches the right-most element name, which can then be directly converted to a σ operator. In the example, `class(X)/a/b/c` would be parsed as `class(X)/(a/b/c)`, and cvt(`class(X)/(a/b/c)`, *src*, `false`) is evaluated by evaluating cvt(`/(a/b/c)`, X, `false`). This is then evaluated as $\sigma[/](x)$ where x = cvt(`a/b/c`, X, `false`), and so on.

The XIRQL syntax for dealing with links was adopted from the XQuery specification. According to this specification, a link is an edge in the XML "tree", similar to the edges from a parent node to a child node. Just like the `"/"` operator can be used to follow an edge from a parent node to a child node, the `"=>"` operator can be used to follow a link from one node to the other. Thus, the `"=>"` operator is simply part of a 'step'. But in the path algebra, the operators Π and σ which are used for parent/child edges are unary operators whereas the operators >= and => for links are binary operators. Additionally, the `"=>"` operator in XIRQL binds rather tightly, whereas the >= and => path algebra operators have low precedence. This difference needs to be accomodated in the transformation rules.

Consider as an example a document base comprising two document classes, one class for documents and one for person descriptions. Suppose that the author is stored, in the 'document' class, as a link to one of the 'person' documents. Then the query `class(document)/article/author=>class(person)/name/family` will result in a list of all family names of article authors. This can be thought of navigating from the `article` root node to the `author` child, from there to follow the link to the 'person' class, to go to the `name` child node in that class, and from there to the `family` child node. But the corresponding path algebra expression would be σ[article](σ[author](document)) => σ[name](σ[family](person)).

The rules are designed in such a way that the algorithm first 'looks' to see if a `"=>"` operator appears. This is achieved by having all rules that deal with slash-like operators 'look ahead' at the right hand side to see if the conversion produces a Θ (=> or >=) expression. If that is the case, these operators 'pull' the link operator Θ 'up' in the result.

We think that the current syntax for dealing with links was not chosen thoughtfully. Instead of viewing links as being similar to path steps, it would be more appropriate to treat them like binary operators with low precedence (i.e. similar to the `before` and `after` operators). Since links can be followed in both directions, they are dissimilar to path steps. Due to the fact that links often connect different documents, a low precedence for the link operator would yield a better separation of the query parts addressing the individual documents. Our path algebra matches this view, whereas the current syntax of XIRQL (as well as that of XQuery) does not.

5.2 Implementation

Based on the concepts described in this paper, we have implemented a retrieval engine named HyREX (*Hy*permedia *R*etrieval *E*ngine for *X*ML). In order to set up a document base with HyREX, first the XML Schema descriptions (along with the HyREX-specific application information) for the documents must be specified. Given the document base schema, the system accepts XML documents, indexes them and creates its internal index structures. (Currently, we use B*-trees and variants of inverted lists for this purpose.) Following this step, the HyREX server accepts XIRQL queries and returns pointers to the elements retrieved.

In order to use HyREX as a standalone retrieval system, we have developed a simple (Web-based) user interface (HyGate) that accepts query formulations either in XIRQL or based on application-specific forms, sends the query to the server and

receives result lists as well as single result elements. For presenting the output in HyGate, the document base administrator has to specify appropriate XSLT stylesheets, both for the results survey page(s) and the display of single result elements.

HyREX is designed as an extensible IR architecture. The whole system is open source, written in Perl (with minor parts in C). For specific applications, new datatypes cam be added to the system, possibly together with new index structures.

6 Related work

Looking at the broad variety of XML applications and systems that are currently under development, one can see that there are in fact two different views on XML:

- The *document-centric view* focuses on structured documents in the traditional sense (based on concepts from electronic publishing, especially SGML). Here XML is used for logical markup of texts both at the macro level (e.g. chapter, section, paragraph) and the micro level (e.g. MathML for mathematical formulas, CML for chemical formulas).

- The *data-centric view* uses XML for exchanging formatted data in a generic, serialized form between different applications (e.g. spreadsheets, database records). This is especially important for e-business applications (e.g. for exchanging orders, bills).

In both views, there is a need for a query language for XML. However, the requirements for such a language are very much view-dependent:

- The document-centric view requires a query language that mainly supports selection based on conditions with respect to both structure and content, taking into account the intrinsic uncertainty and vagueness of content-based retrieval.

- The data-centric view asks for a query language that allows for selection as well as restructuring (of result documents) and aggregation operators (e.g. count, sum).

Unfortunately, the W3C working group on XML query languages has focused on the data-centric view only, thus ignoring most issues related to IR. Following earlier proposals for XML query languages like XML-QL [Deutsch et al. 98] or Quilt [Chamberlin et al. 00], the proposed query language XQuery [Fernandez & Marsh 01] draws heavily on concepts from query languages for object-oriented databases (e.g. OQL) or semistructured data (e.g. Lorel [Abiteboul et al. 97]). Due to this origin, XQuery has a much higher expressiveness than XQL. XQL (and XIRQL) offer only selection operators, thus results are always complete elements of the original documents. In contrast, XQuery also provides operators for restructuring results as well as for computing aggregations (count, sum, avg, max, min).

A typical XQuery expression has the following structure:

```
FOR PathExpression
WHERE AdditionalSelectionCriteria
RETURN ResultConstruction
```

Here PathExpression may contain one or more path expressions as in XQL, where each expression is bound to a variable. Thus, the FOR clause returns ordered lists of tuples of bound variables. The WHERE clause prunes these lists of tuples by testing additional criteria. Finally, the RETURN clause allows for the construction of arbitrary XML documents by combining constant text with the content of the variables.

As a simple example illustrating the expressiveness of XQuery, assume that we have a whole class of documents of type `book`, and we would like to have a kind of excerpt documents, containing only the titles and headings of a each book. This can be accomplished by means of the the following XQuery formulation

```
FOR $a in class("book")
RETURN
  {<excerpt> <title> $a/title </title>
  FOR $b in $a//heading
  RETURN {<heading> $b </heading>}
  </excerpt>}
```

In contrast, the XQL query `/book/(title|heading)` would return single `title` and `heading` elements only, without

the possibility of collecting them in `excerpt` documents.

In principle, XQL is a subset of XQuery (with minor syntactical differences), supporting only the FOR clause with a single path expression. Thus, an XQL query can be rewritten in XQuery as `FOR $a in XQLquery RETURN $a`. Since XIRQL is based on XQL, it can be seen as an extension of a subset of XQuery in order to support IR.

As the only feature supporting information retrieval in XML, XQuery supports querying for single words in texts. There is no possibility for weighting or ranking, no support for vague query conditions, and no operator for relevance-oriented search. XIRQL fills this gap for a subset of the XQuery language.

In information retrieval, previous work on structured documents has focused on two major issues:

- 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 [Navarro & Baeza-Yates 97] 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 are 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 e.g. [Hearst & Plaunt 93]). Here the system determines a sequence of sentences from the original document that fit the query best.

  Only a few researchers have dealt with the combination of explicit structural information and content-based retrieval. The paper [Myaeng et al. 98] 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 [Chiaramella et al. 96] mentioned before is a general framework for relevance-based retrieval of documents. [Lalmas 97] and [Fuhr et al. 98] describe refinements of this approach based on different logical models.

Comparing the different approaches described above, it turns out that they address different facets of the the XML retrieval problem, but there is no approach that solves all 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.

Only a few researchers have tried to combine structural conditions with weighting. The paper [Theobald & Weikum 00] extends XML-QL by weighted document indexing; however, this approach is not based on a consistent probabilistic model. As another approach based on XML-QL, [Chinenyanga & Kushmerik 01] introduces an operator for text similarity search on XML documents; so this extension supports only a very specific type of queries. A nice theoretical concept for vagueness with respect to both value conditions and structural conditions is proposed in [Schlieder & Meuss 00]; however, the underlying query language is rather restricted.

The path algebra approach for processing XIRQL is similar to the proximal nodes model described in [Navarro & Baeza-Yates 97]. (The close relationship between XQL and proximal nodes is discussed in [Baeza-Yates & Navarro 00].) However, we give a more formal specification of the semantics of the different operators and we also consider hyperlinks. Furthermore, we extend this model by dealing with datatypes and weighting.

7 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 structural 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 specified a path algebra, which also serves as a starting point for query optimization.

In order to use XIRQL for retrieval, there are a number of open issues. At the system level, there is the question of appropriate access methods and query processing strategies. For the user interface, it is not clear in which form end users should formulate their queries. Currently, we are investigating both menu-based strategies as well as methods based on the concept of query by example. Also, the presentation of results poses a number of problems. Since several result elements may belong to the same document (some results even may contain others), presentation as a simple ranked list may not be appropriate. For a single result element, there is the question if this element should be shown out of context, or within the context of the document it belongs to. In the latter case, there is the question how this context should be displayed (logical structure vs. layout structure).

A major goal of our work is the integration of XIRQL into the forthcoming standard XML query language. For this purpose, we are working on a probabilistic version of the full XQuery language.

References

Abiteboul, S.; Quass, D.; McHugh, J.; Widom, J.; Wiener, J. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries 1(1)*, pages 68–88.

Baeza-Yates, R.; Navarro, G. (2000). XQL and Proximal Nodes. In: *Proceedings ACM SIGIR 2000 Workshop on XML and Information Retrieval*. ACM. `http://www.haifa.il.ibm.com/sigir00-xml/final-papers/RBaetza/att1.htm`.

Billingsley, P. (1979). *Probability and Measure*. John Wiley & Sons, Inc, New York.

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

Chiaramella, Y.; Mulhem, P.; Fourel, F. (1996). *A Model for Multimedia Information Retrieval*. Technical report, FERMI ESPRIT BRA 8134, University of Glasgow. `http://www.dcs.gla.ac.uk/fermi/tech\_reports/reports/fermi96-4.ps.gz`.

Chinenyanga, T.; Kushmerik, N. (2001). Expressive Retrieval from XML documents. In: *Proceedings of the 24th Annual International Conference on Research and development in Information Retrieval*, pages 163–171. ACM, New York.

Clark, J.; DeRose, S. (1999). *XML Path Language (XPath) Version 1.0*. `http://www.w3.org/TR/xpath`.

Croft et al. (ed.) (1998). *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York. ACM.

Deutsch, A.; Fernandez, M.; Florescu, D.; Levy, A.; Suciu, D. (1998). XML-QL: A Query Language for XML. In [Marchiori 98]. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

Fallside, D. (2001). *XML Schema Part 0: Primer*. `http://www.w3.org/TR/xmlschema-0/`.

Fernandez, M.; Marsh, J. (2001). *XQuery 1.0 and XPath 2.0 Data Model*. `http://www.w3.org/TR/query-datamodel/`.

Fuhr, N.; Rölleke, T. (1997). A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. *ACM Transactions on Information Systems 14(1)*, pages 32–66.

Fuhr, N. (1999). Towards Data Abstraction in Networked Information Retrieval Systems. *Information Processing and Management 35(2)*, pages 101–119.

Fuhr, N.; Gövert, N.; Rölleke, T. (1998). DOLORES: A System for Logic-Based Retrieval of Multimedia Objects. In [Croft et al. 98], pages 257–265.

Hearst, M.; Plaunt, C. (1993). 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. ACM, New York.

Jarke, M.; Koch, J. (1984). Query Optimization in Database Systems. *ACM Computing Surveys 16*, pages 111–152.

Lalmas, M. (1997). Dempster-Shafer's Theory of Evidence Applied to Structured Documents: Modelling Uncertainty. In: Belkin, N. J.; Narasimhalu, A. D.; Willet, P. (eds.): *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 110–118. ACM, New York.

Marchiori, M. (ed.) (1998). *QL'98 — The Query Languages Workshop*. W3C. `http://www.w3.org/TandS/QL/QL98/`.

Myaeng, S.; Jang, D.-H.; Kim, M.-S.; Zhoo, Z.-C. (1998). A Flexible Model for Retrieval of SGML Documents. In [Croft et al. 98], pages 138–145.

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

Robie, J.; Lapp, J.; Schach, D. (1998). XML Query Language (XQL). In [Marchiori 98]. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

Robie, J.; Derksen, E.; Fankhauser, P.; Howland, E.; Huck, G.; Macherius, I.; Murata, M.; Resnick, M.; Schöning, H. (1999). *XQL (XML Query Language)*. `http://www.ibiblio.org/xql/xql-proposal.html`.

Schlieder, T.; Meuss, M. (2000). Result Ranking for Structured Queries against XML Documents. In: *First DELOS workshop on Information Seeking, Searching and Querying in Digital Libraries*.

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