XML Retrieval By HyREX
and
Some Improvements By Relevance Feedback

by
Tamoghna Ghosh

Dissertation
Submitted in partial fulfillment of the requirements for
M.Tech in Computer Science

Advisor Dr. Mandar Mitra
Computer Vision and Pattern Recognition Unit

Indian Statistical Institute

July 07

Certificate

This is to certify that this dissertation thesis titled “XML Retrieval By HyREX and Some Improvements by Relevance Feedback” submitted by Tamoghna Ghosh in partial fulfillment of the requirements for the degree of M-Tech (CS) at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

Dr. Mandar Mitra

Acknowledgments

I hereby take this opportunity to thank my advisor Prof Dr. Mandar Mitra for his support and insightful suggestions. It is an honour to work under his supervision. It is my pleasure to thank Sukomal Pal and Kuntal Chakraborty, both ISI fellows for their suggestions and help during this work.

Tamoghna Ghosh

Indian Statistical Institute
July 2007

Abstract

The e*X*tensible *M*arkup *L*anguage (XML) is the emerging standard for representing knowledge for many applications. XML retrieval is thus becoming increasingly important. A number of research groups from all over the world are actively working in this area.

HyREX is an XML retrieval engine designed by Norbert Govert, University of Dortmund, Germany. In this thesis we have made a detail study of the HyREX system and finally improved the search performance by implementing relevance feedback. Here the assessments were done manually using the queries and assessment files provided by INEX 2002.

Contents

Chapter 1

XML Retrieval

1.1 Introduction

There are two types of information retrieval problems that are intermediate between text retrieval and search over relational data. The second type, XML retrieval, is the subject of this chapter. We will view XML documents as trees that have *leaf nodes* containing text and *labeled internal nodes* that define the roles of the leaf nodes in the document. We call this type of text semistructured and retrieval over it *semistructured retrieval*. Semistructured retrieval has become increasingly important in recent years because of the growing use of *Extensible Markup Language* or *XML*. XML is used for web content, for documents produced by office productivity suites, for the import and export of text content in general, and many other applications. These days, most semistructured data are encoded in XML.

1.2 Basic XML concepts

An XML document is an ordered, labeled tree. The nodes of the tree are *XML elements* and are written with an opening and closing *tag*. An element can have one or more *XML attributes*. One of the elements in the example XML document in Figure 1.1 is *scene*, which is enclosed by the two tags <scene ...> and </scene>. The element has an attribute *number* with value *VII* and two child elements, *title* and *verse*.

```
<play>
  <author>Shakespeare</author>
  <title>Macbeth</title>
  <act number="I">
    <scene number="VII">
      <title>Macbeth's castle</title>
      <verse>Will I with wine and wassail ...</verse>
    </scene>
  </act>
</play>
```

Figure 1.1 An XML document.

There is a standard way of accessing and processing XML documents,viz the XML Document Object Model or *DOM*. DOM represents elements, attributes and text within elements as nodes in a tree. *XPath* is the standard for paths in XML. We also need the concept of XML *schema.* A schema puts constraints on the structure of allowable XML documents for a particular application. Two standards for schemas for XML documents are *XML DTD* (document type definition) and *XML Schema*. The purpose of a DTD is to define the legal building blocks of an XML document. Using DTDs , each XML file can carry a description of its own format with it. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. XML schema is a XML based alternative to DTD.

1.3 Why XML?

Relational database systems cannot meet all the demands of electronic business because they process data independently of its context. Traditional databases may be well suited for data that fits into rows and columns, but cannot adequately handle rich data such as audio, video, nested data structures or complex documents, which are characteristic of typical Web content. To deal with XML, traditional databases are typically retrofitted with external conversion layers that mimic XML storage by translating it between XML and some other data format. This conversion is error-prone and results in a great deal of overhead, particularly with increasing transaction rates and document complexity.

XML databases, on the other hand, store XML data natively in its structured, hierarchical form. Queries can be resolved much faster because there is no need to map the XML data tree structure to tables. This preserves the hierarchy of the data and increases performance. XML documents can contain any imaginable data type - from classical data like text and numbers, or multimedia objects such as sounds, to active formats like Java applets or ActiveX components. The look and feel of documents or even entire websites can be changed with XSL Style Sheets without manipulating the data itself. XML documents can consist of data from many different databases distributed over multiple servers. In other words: With XML the entire World Wide Web is being transformed into a single all-encompassing database.

Internationalization is of utmost importance for electronic business applications. XML supports multilingual documents and the Unicode standard. XML's one-of-a-kind open structure allows the addition of other state-of-the-art elements when needed. This means that a system can always be adapted to embrace industry-specific vocabulary.

In future Web development, it is most likely that XML will be used to describe the data, while HTML will be used to format and display the same data. Since XML data is stored in plain text format, XML provides a software and hardware independent way of sharing data. Users can write structural queries for an XML retrieval system if they have some minimal knowledge about the schema of the underlying collection.

1.4 Challenges in semi structured retrieval

We need to choose a document unit for indexing and retrieval. In unstructured retrieval, it is usually clear what the right document unit is. A traditional Unix (mbox-format) email file stores a sequence of email messages (a folder) in one file, but one might wish to regard each email message as a separate document. Many email messages now contain attached documents, and you might then want to regard the email message and each contained attachment as separate documents. Sometimes people index each paragraph of a document as a separate pseudo-document, because they believe it will be more helpful for retrieval to be returning small pieces of text so that the user can find the relevant sentences of a document more easily. The first challenge in semi structured retrieval is that we don't have such a natural document unit or *indexing unit*. There are at least three different approaches to defining the indexing unit in XML retrieval. One is to index all components that are eligible to be returned in a search result. All subtrees in Figure 1.1 meet this criterion. This scheme has the disadvantage that search results will contain overlapping units that have to be filtered out in a post processing step to reduce redundancy. Another approach is to group nodes into non-overlapping pseudo documents as shown in Figure 1.2. This avoids the overlap problem, but pseudo documents may not make intuitive sense to the user. And they have to be fixed at indexing time, leaving no flexibility to answer queries at a more specific or more general level. The third approach is to designate one XML element as the substitute for the document unit.

If we query Shakespeare's plays for Macbeth's castle, should we return the scene, the act or the whole play in Figure 1.1? In this case, the user is probably looking for the scene. On the other hand, an otherwise unspecified search for Macbeth should return the play of this name, not a subunit. One decision criterion that has been proposed for selecting the most appropriate part of a document is the *structured document retrieval principle*:

Structured document retrieval principle. A system should always retrieve the most specific part of a document answering the query. This principle motivates a retrieval strategy that returns the smallest unit that contains the information sought, but does not go below this level. However, it can be hard to implement this principle algorithmically.
The user interface should expose the tree structure of the collection and allow users to specify the nodes they are querying. As a consequence the query interface is more complex than a

search box for keyword queries in unstructured retrieval. This is one of the challenges currently being addressed by the research community.

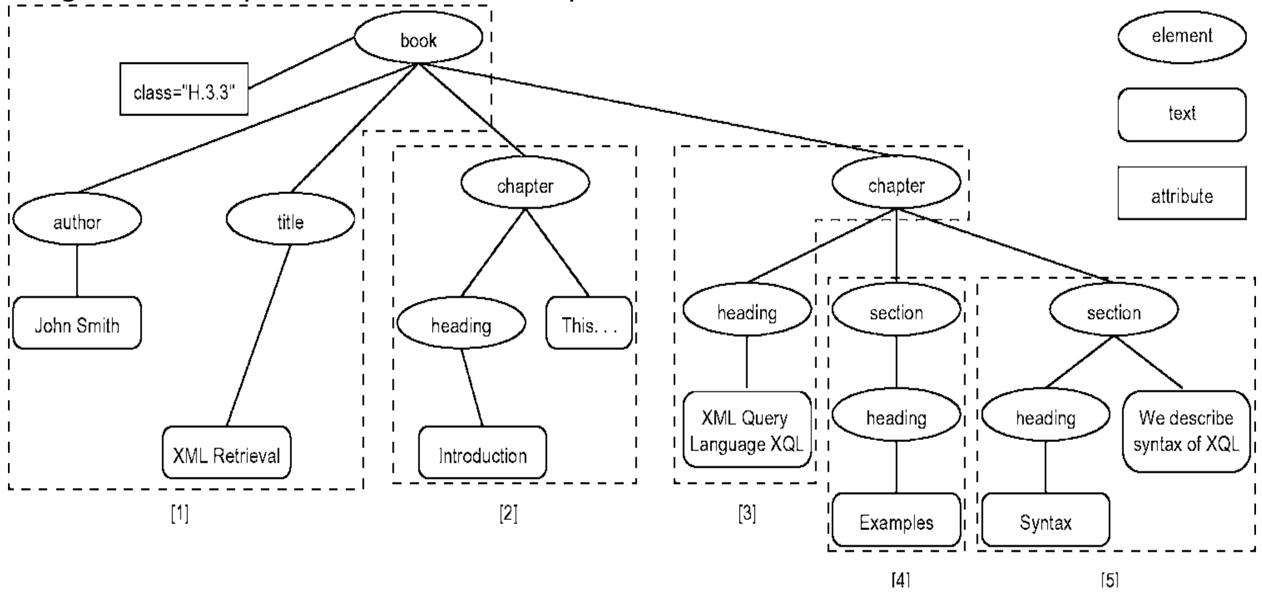

Figure 1.2 Indexing units in XML retrieval. Unlike conventional retrieval, XML retrieval does not have a natural indexing unit. In this example, books, chapters and sections have been designated to be indexing units, but without overlap. For example, the leftmost dashed indexing unit contains only those parts of the tree dominated by book that are not already part of other indexing units results.

Another challenge is how we store the structure information of the whole document collection in main memory at the retrieval time. One solution to this is the XML structure tree or ***XS-tree*** data structure. It is highly compressed so that the XS-tree of the whole document collection can be kept in main memory. Consider the XML document in figure 1.3

We can assign a path handle to each element, attribute and text part sequentially as follows: Attribute class is assigned handle number 1, the element title handle number 2, then the text part 'John Smith' a handle number 3, <title> a handle number 4 , 'XML Retrieval' a handle number 5 and so on. Given this position there is a resolution method that yields the corresponding path. For creating a linear representation of the XS-tree the following design is chosen.

Enumerate nodes top-down (in preorder): By choosing a top-down sequence, we can apply context-specific compression methods. Given the DTD, there is only a small set of elements that can occur as children of a specific element, so we only need a few bits for coding each of these alternatives.

Parent-child relationship via level numbers: Level numbers are compact. For compressing level numbers, we use run length encoding, thus only the relative differences between the level numbers are stored.

```
<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>
```

Figure 1.3

Positions as element numbers: Element numbers are compact, and therefore efficient encoding for use as path handles in the inverted lists is possible.
Element and sequence indexes implicit: Each element in a path has an element index and a sequence index that denote its relative position among the children of the parent node. These indexes could be encoded explicitly, but would require additional storage space; in contrast, by scanning the representation linearly, the indexes of each element can be computed on the fly.
Universal codes are used for compression of the level numbers. In order to encode the element names given for each node of an XML tree, we use Huffman coding.
A part of the XS tree of the above document is shown below:

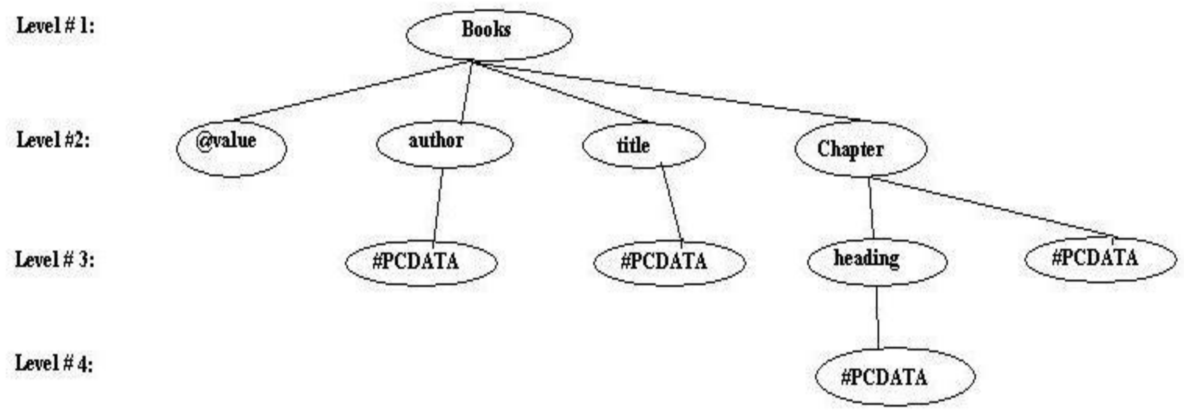

Figure 3.4

The xs tree of the above doc is basically two linear arrays,

Array1=(Books,@value,author,#PCDATA,title,#PCDATA,Chapter,heading,#PCDATA,#PCDATA,…)
Array2=(1,2,2,3,2,3,2,3,4,3,…).

1.5 An introduction to XPath and XIRQL

XPath is a language for addressing parts of an XML document. XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node. XPath retrieves elements (i.e., subtrees) of the XML document fulfilling the specified condition. The simplest kind of query specifies elements by giving their names, for instance, the query heading retrieves the four different heading elements from our example document 1.3. Attributes are specified with a preceding "@" (as in @class). Context can be considered by means of the child operator '/' between two element names, so section/heading retrieves only headings occurring as children of sections, or by the descendant operator ("//"),so that book//heading finds headings which are descendants of a book element. 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., the query /book/title specifies that the book element should be the root element of the document). The filter operator (denoted with square brackets) filters the set of nodes to its left. For example, //chapter[heading] retrieves all chapters having a heading. (In contrast, //chapter/heading retrieves 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. Square brackets also are 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). Disjunctive conditions can be specified via the | operator, for example //(chapter|section) will find all chapter elements in addition to all section elements. In order to pose restrictions on the content of elements and the value of attributes, comparisons can be formulated. For example, /book [author = "John Smith"] refers to the value of the element author, whereas /book[@class ="H.3.3"] compares an attribute value with the specified string. Besides strings, XPath also supports numbers and dates as datatypes, along with additional comparison operators like > and <.Sub queries 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 XPath allow for flexible formulation of conditions with respect to the structure and the content of XML documents. The result is always a set of elements from the original document(s).

XIRQL Concepts: From an IR point of view, 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 some limitations of passage retrieval. The FERMI model [Chiaramella et al. 1996] 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. As an example, consider a user searching for information about multimedia databases. If multimedia and databases are discussed in the same section, then surely that section should be returned as the query result. If the two concepts are covered in two distinct sections within the same chapter, then the chapter should be returned.
—Based on the markup of specific elements, high-precision searches can be performed that look for content occurring in specific elements. Possible scenarios include distinguishing between the sender and the addressee of a letter, and finding the definition of a concept in 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) and 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 bullfights).
—For query conditions referring to the structure of documents, it should be possible to perform vague comparisons with the actual document structure, such as semantically related element names.
With respect to these requirements, XPath seems to be a good starting point for IR on XML documents. However, the following features should be added to XPath and we get XIRQL:

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 XPath referring to the text of elements should consider index term weights. Furthermore, query term weighting also should be possible, by introducing a weighted sum operator [e.g., WSUM (0.6 · "XML" + 0.4 · "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.
The query is decomposed into basic query conditions (leaf nodes of the XPath expression parse tree) and combining operators (inner nodes). The basic query conditions return weighted sets of results, which then are processed by the combining operators. 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:
(1) How should terms in structured documents be weighted?
(2) What are the probabilistic events, that is, which term occurrences are identical, and which are independent?
At first, we need to define the "atomic" units (or hierarchic aggregations of those) in XML documents. Text is contained in the leaf nodes of the XML tree only. So these leaves would be an obvious choice as atomic units. But we can use the concept of index nodes as in figure 3.2 to construct atomic units.
Given these units, we can separate indexing and retrieval methods. Thus, for indexing, we can start with a standard formula such as a kind of *tf* · *idf* (for details refer to chapter 2).We interpret a weight as the probability that the corresponding condition is true. For example the XML retrieval engine HyREX uses the BM25 formula [Robertson et al. 1995] multiplied by a

normalization constant, thus yielding values from the interval [0, 1] which can be interpreted as probabilities.

Thus we have a method for computing term weights and we can do relevance-oriented search. For this, we must be able to retrieve index nodes at all levels. The indexing weights of terms within the most specific index nodes are given directly. For retrieval of the higher-level objects, we have to consider that their content is made up by the content of the index node under consideration plus the content of the descendent index nodes. Therefore, for a given index node its term weights have to be combined with the term weights of the descendant index nodes. For example, assume the following document structure, where we list the weighted terms instead of the original text:

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

A straightforward possibility would be the OR-combination of the different weights for a single term. However, searching for the term 'XQL' in this example would retrieve the whole chapter in the top rank, whereas the second section would be given a lower weight. It can easily be shown that this strategy always assigns the highest weight to the most general element. This result contradicts the structured document retrieval principle mentioned before. For this purpose, index term weights are down weighted (multiplied by an augmentation factor) when they are propagated upwards to the next index node. In above example, using an augmentation factor of 0.6, the retrieval weight of the chapter w. r. t. 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.

Datatypes and Vague Predicates: Since XML allows for a fine-grained markup of elements, there should be the possibility to use special search predicates for different elements of various data types (e. g. person names, dates, names of geographic regions). For each data type, the system must provide appropriate search predicates, most of which should be vague. 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 Ulbricht and who worked around 1900 near Frankfort, Germany", which also should retrieve an article mentioning Joseph Maria Olbrich's work in Darmstadt, Germany, in 1901. Thus, we need *vague predicates* for different kinds of datatypes (e.g., person names, locations, dates). 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(i.e, a precompiled list of important words in a given domain of knowledge and for each words in this list a set of related words is provided) 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, for dealing with stemming, noun phrases, and compound words and so on should be provided.

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. For retrieval, the only operations to be applied are vague predicates, where even a few type errors can be tolerated: vague predicates are hardly always correct (e.g., equality based on stemming), so single incorrect values also won’t have significant effects on retrieval quality.

As a framework for dealing with these problems, we adopt the concept of datatypes in IR from [Fuhr 1999], where a datatype T is a pair consisting of a domain $|T|$ and a set of (vague comparison) predicates $PT = \{c_1, . . . , c_n\}$.Like in other type systems, IR datatypes also should 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 added to the system implementation by refining the appropriate datatype (e.g., introduce French as refinement of Western Language).

Structural Vagueness: XIRQL supports four different types of structural vagueness, which are described in the following.

Elements vs. Attributes. The distinction between elements and attributes may not be relevant to many users. Thus, in XIRQL, author searches for elements,@author is for attributes, and =author is used for abstracting from this distinction.

Datatypes as Generalization of Elements and Attributes. 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 by 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 (e.g., /article [pub-date>"2001-12-11"] and do not need to know how dates happen to be represented in the current document type.

Besides abstracting from the concrete syntax, datatypes also can be used for generalizing from specific element or attribute names. 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 datatypes, regardless of their name and their position in the XML document tree. For example, #persname searches for all elements and attributes of the datatype persname (for person names).

Similarity of Element Names. The precise naming of elements (or attributes) may be a major problem when formulating structural conditions in a query. For this purpose, we provide a similarity operator for element names, which is expressed in XIRQL via the tilde as prefix of an element name: Whereas author searches for an element by specifying an element name,˜author searches for an element semantically similar to “author”; in the latter case, elements with different, but similar names will also match, but with a lower score than elements with the specified name.
Generalizing Parent/Child Relationships. In the case of complex DTDs, most users will have problems in specifying the precise path to an element. However, instead of replacing all child operators in the query by descendant operators, they might want to prefer those matches that are close to the path specification in the query. For this purpose, we provide the vague extension of the child operator, which is written as \\. So chapter/title specifies a parent child relationship, chapter//title specifies an ancestor-descendant relationship, and chapter\\title specifies that a parent-child relationship should get a higher weight than a grandparent-grandchild relationship and that the weight should decrease with the number of intervening levels of elements.

Document Classes: 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"]).

1.6 Evaluation of XML Retrieval

A large part of academic research on XML retrieval is conducted within the *INEX* (**IN***itiative for the* ***E****valuation of* **X***ML retrieval*) program, a collaborative effort that includes reference collections, sets of queries, relevance judgments and a yearly meeting to present and discuss research results. There are two types of queries, called *topics*, in INEX: content-only or CO topics and content-and-structure or CAS topics. *CO topics* are regular key word queries as in unstructured information retrieval. *CAS topics* have structural constraints in addition to keywords These two different components of CAS queries make relevance assessments more complicated than in unstructured retrieval. INEX defines *component coverage* and *topical relevance* as orthogonal dimensions of relevance. The component coverage dimension evaluates whether the element retrieved is “structurally” correct, i.e., neither too low nor too high in the tree.

Chapter 2

Term Weighting-BM25

2.1 Introduction

In case of large document collections, the number of matching documents for a query can be far in excess of the number a human user could possibly sift through. Accordingly, it is essential for search engines to rank-order the documents matching a query. To do this, an engine computes, for each matching document, a score with respect to the query at hand. In this chapter we discuss methods of assigning a score to a (query, document) pair.

2.2 Term frequency and weighting

We assign to each term in a document a *weight* for that term in that document that depends on the number of occurrences of the term in the document. The simplest approach is to assign the weight to be equal to the number of occurrences of the term *t* in document *d*. This weighting scheme is referred to as *term frequency* and is denoted $\mathrm{tf}_{t,d}$, with the subscripts denoting the term and the document in order.
For a document *d*, the set of weights (determined by the tf weighting function above, or indeed any weighting function that maps the number of occurrences of *t* in *d* to a positive real value) may be viewed as a vector, with one component for each distinct term. In this view of a document, known in the literature as the *bag of words model*, the exact ordering of the terms in a document is ignored. The vector view only retains information on the number of occurrences. Thus, the document “Mary is quicker than John” is, in this view, identical to the document “John is quicker than Mary”. Nevertheless, it seems intuitive that two documents with similar vector representations are similar in content.

2.3 Inverse document frequency

Raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevance for a query. Certain terms have little or no discriminating power in determining relevance. For instance, a collection of documents on the insurance industry is likely to have the term insurance in almost every document. To this end,

we introduce a mechanism for attenuating the effect of terms that occur too often in the collection to be meaningful for relevance determination. An immediate idea is to scale down the term weights of terms with high *collection frequency,* defined to be the total number of occurrences of a term in the corpus. The idea would be to reduce the tf weight of a term by a factor that grew with its collection frequency.
Instead, it is more commonplace to use for this purpose the *document frequency* df_t, defined to be the number of documents in the corpus that contain a term *t*.

Word	cf	Df
Ferrari	10422	17
Insurance	10440	3997

Figure 3.1 Collection frequency (cf) and document frequency (df) behave differently.

Term	df	idf
Calpurnia	1	6
Animal	100	4
Sunday	1000	3
Fly	10000	2
Under	100000	1
The	1000000	0

Figure 3.2 Example of idf values. Here we give the idf's of terms with various frequencies in a corpus of 1,000,000 documents.

The reason to prefer df to cf is illustrated in Figure 3.1, where a simple example shows that collection frequency (cf) and document frequency (df) can behave rather differently. In particular, the cf values for both Ferrari and insurance are roughly equal, but their df values differ significantly. This suggests that the few documents that do contain ferrari mention this term frequently, so that its cf is high but the df is not. Intuitively, we want such terms to be treated differently: the few documents that contain ferrari should get a significantly higher boost for a query on ferrari than the many documents containing insurance get from a query on insurance. Denoting as usual the total number of documents in a corpus by *N*, we define the *inverse document frequency* (idf) of a term *t* as follows:

$$\mathrm{idf} = \log \frac{N}{df}$$

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low. Figure 3.2 gives an example of idf's in a corpus of 1,000,000 documents; in this example logarithms are to the base 10.

2.4 Tf-idf weighting

We now combine the above expressions for term frequency and inverse document frequency, to produce a composite weight for each term in each document. The *tf-idf* weighting scheme assigns to term *t* a weight in document *d* given by:

$$\text{tf- idf}_{t,d} = \text{tf}_{t,d} \times \text{id}_{ft.}$$

In other words, tf-idf$_{t,d}$ assigns to term *t* a weight in document *d* that is
1. highest when *t* occurs many times within a small number of documents (thus lending high discriminating power to those documents);
2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
3. lowest when the term occurs in virtually all documents.
We may view each document as a *vector* with one component corresponding to each term, together with a weight for each component that is given by above equation.
The *score* of a document *d* is the sum, over all query terms, of the number of times each of the query terms occurs in *d*. We can refine this idea so that we add up not the number of occurrences of each query term *t* in *d*, but instead the tf-idf weight of each term in *d*.

$$\text{Score}\ (q, d) = \textstyle\sum_{\text{t in q}} \text{tf- idf}_{t,d}$$

2.5 Maximum tf normalization

One well-studied technique is to normalize the individual tf weights for a document by the maximum tf in that document. For each document *d*, let Then, we compute a normalized term frequency for each term *t* in document *d* by

$$\text{ntf}_{\text{t,d}} = \text{a} + (1\text{-a})\frac{\text{ntf t,d}}{\text{Max ntf t,d}}$$

where *a* is a value between 0 and 1 and is generally set to 0.5. The term *a* in is a *smoothing* term whose role is to damp the contribution of the second term – which may be viewed as a scaling down of tf by the largest tf value in the dictionary. The main idea of maximum tf normalization is to mitigate the following anomaly: we observe higher term frequencies in longer documents, merely because longer documents tend to repeat the same words over and over again. To appreciate this, consider the following extreme example: supposed we were to take a document *d* and create a new document d' by simply appending a copy of *d* to itself. While d' should be no more relevant to any query than *d* is, the use of would assign it twice as high a score as *d*. Replacing tf-idf$_{t,d}$ in by ntf-idf$_{t,d}$ eliminates the anomaly in this example. Maximum tf normalization does suffer from a couple of issues:

1. A document may contain an outlier term with an unusually large number of occurrences of that term, not representative of the content of that document.
2. More generally, a document in which the most frequent term appears roughly as often as many other terms should be treated differently from one with a more skewed distribution.

2.6 The effect of document length

The above discussion of weighting ignores the length of documents in computing term weights. However, document lengths are material to these weights, for several reasons. First, longer documents will – as a result of containing more terms – have higher tf values. Second, longer documents contain more distinct terms. These factors conspire to raise the scores of longer documents, which (at least for some information needs) is unnatural. Longer documents can broadly be lumped into two categories: (1) *verbose* documents that essentially repeat the same content – in these, the length of the document does not alter the relative weights of different terms; (2) documents covering multiple different topics, in which the search terms probably match small segments of the document but not all of it – in this case, the relative weights of terms are quite different from a single short document that matches the query terms.

2.7 Relevance weights

The relevance weighting model (Robertson and Sparck Jones, 1976), referred to as RSJ, shows that under some simple assumptions, the probability of relevance ordering can be achieved by assigning weights to query terms, and scoring each document by adding the weights of the query terms it contains. The term weight may be defined as follows. First we define two probabilities for term t_i:

$$p_i = P(\text{document contains } t_i | \text{document is relevant})$$
$$q_i = P(\text{document contains } t_i | \text{document is not relevant})$$

RSJ weight $w^{(1)}_i = \log \frac{pi(1-qi)}{(1-pi)qi}$ (3.)

If we have, as before, a total of N documents of which ni contain the terms, and further R out of the N are relevant and ri relevant documents contain the term, then the obvious estimates of pi and qi are:

$$pi \approx \frac{ri}{R} \qquad\qquad qi \approx \frac{ni-ri}{N-R} \qquad (3.)$$

However, because of the form of Equation of $w^{(1)}_I$ a modification to these estimates is appropriate. Here 0.5 added to each of the components can be seen as a smoothing correction and we have:

$$wi = log \frac{(ri+0.5)(\mathrm{N - R - ni + ri + 0.5})}{(\mathrm{R - ri + 0.5})(\mathrm{ni - ri + 0.5})}$$

2.8 BM25

The weighting function known as Okapi BM25 (it was developed as part of the Okapi experimental system (Robertson, 1997), and was one of a series of Best Match functions). The following is a brief account of the derivation of BM25:

First, we assume that all documents are the same length – this assumption will be dropped below. Next we assume (this is the central assumption of the model) that each term (word) represents a concept; and that a given document is either 'about' the concept or not. This property is described as eliteness: the term is elite in the document or not. This terminology, as well as the statistical model described below, is taken from Bookstein and Swanson (1974) and Harter (1975). Eliteness is a hidden variable – we cannot observe it directly. However, we then assume that the text of the document is generated by a simple unigram language model, where the probabilities of any term being in any given position depend on the eliteness of this term in that document.

If we take all the documents for which this particular term is elite, then we can infer the distribution of within-document frequencies we should observe. If all documents are the same length, then the distribution will be approximately Poisson. If we take instead all the documents for which this term is not elite, we will again see a Poisson distribution (presumably with a smaller mean). But of course we cannot know eliteness in advance; so if we consider the collection as a whole, we should observe a mixture of two Poisson distributions. This two-Poisson mixture is the basic Bookstein/Swanson/Harter model for within-document term frequencies.

Eliteness thus provides the bridge between the event-space of documents (or rather the cross-product of documents in the collection and terms in the vocabulary) and the event-space of term positions, which gives us term frequencies. For each term, eliteness is a property of the document, but determines the properties of the term positions in the document. Finally, we have to make the connection with relevance. Relevance is a property at the document level, so the connection is with eliteness rather than with term occurrences; but now we have to make the bridge between query and document terms. But we know how to do that, having done it in the original relevance weighting model. So now we re-use the relevance weighting model, only applying it to query-term eliteness rather than to query-term presence or absence.

From the above arguments, we can formulate a weighting scheme which involves the following five parameters for each query term:

- the mean of the Poisson distribution of within-document term frequencies for elite documents;
- ditto for non-elite documents;
- the mixing proportion of elite and non-elite documents in the collection;
- the probability of eliteness given relevance; and
- the probability of eliteness given non-relevance.

The BM25 weighting function is based on an analysis of the behavior of the full eliteness model under different values of the parameters. Essentially, each term would have a full eliteness weight, that is, a weight that a document would gain if we knew that the term was elite in that

document. If we do not know that, but have a TF value which provides probabilistic evidence of eliteness, we should give partial credit to the document. This credit rises monotonically from zero if tf = 0 and approaches the full eliteness weight asymptotically as tf increases. In general the first occurrence of the term gives most evidence; successive occurrences give successively smaller increases.
BM25 first estimates the full eliteness weight from the usual presence-only RSJ weight for the term, then approximates the TF behavior with a single global parameter k1 controlling the rate of approach. Finally it makes a correction for document length.

BM25 weight $wi = f(tf_i) \times w_i^{(1)}$
$w_i^{(1)}$ is the usual RSJ weight,
$f(tf\ i) = \frac{(k+1)tfi}{K+tfi}$
$K = k1((1 - b) + b \times \frac{dl}{avdl}$

dl and *avdl* are the document length and average document length respectively k1 and b are global parameters which are in general unknown, but may be tuned on the basis of evaluation data.

Chapter 3

Relevance Feedback and Query Expansion

3.1 Relevance feedback

The idea of *relevance feedback* is to involve the user in the retrieval process so as to improve the final result set. In particular, the user gives feedback on the relevance of documents in an initial set of results. The basic procedure is:
• The user issues a (short, simple) query.
• The system returns an initial set of retrieval results.
• The user marks some returned documents as relevant or not relevant.
• The system computes a better representation of the information need based
on the user feedback.
• The system displays a revised set of retrieval results.
Relevance feedback can go through one or more iterations of this sort. The process exploits the idea that it may be difficult to formulate a good query when you don't know the collection well, but it is easy to judge particular documents, and so it makes sense to engage in iterative query refinement of this sort. In such a scenario, relevance feedback can also be effective in tracking a user's evolving information need: seeing some documents may lead users to refine their understanding of the information they are seeking. Image search provides a good example of relevance feedback. Not only is it easy to see the results at work, but this is a domain where a user can easily have difficulty formulating what they want in words, but can easily indicate relevant or non-relevant images.

3.2 The Rocchio Algorithm for relevance feedback

The Rocchio Algorithm is the classic algorithm for implementing relevance feedback. It models a way of incorporating relevance feedback information into the vector space model.
Underlying Theory: We want to find a query vector that maximizes similarity with relevant documents while minimizing similarity with non relevant documents. If C_r is the set of relevant documents and C_{nr} is the set of non-relevant documents.

Consider first the unrealistic situation in which the complete set C_r of relevant documents to a given query is known in advance. In such a situation the best query vector for distinguishing the relevant document from the non-relevant documents is given by,

$$\mathbf{q}_{\mathbf{opt}} = (1/|C_r|) \sum_{\forall \mathbf{d}_j \in \mathbf{C}_r} \mathbf{d}_j \; - \; (1/|C_{nr}|) \sum_{\forall \mathbf{d}_j \in \mathbf{C}_{nr}} \mathbf{d}_j$$

i.e, optimal query is the vector difference between the centroids of the relevant and non-relevant documents.

The Rocchio (1971) algorithm. This was the relevance feedback mechanism introduced in and popularized by Salton's SMART system around 1970. In a real IR query context, we have a user query and partial knowledge of known relevant and irrelevant documents. The algorithm proposes using the modified query $\mathbf{q}_m$:

$$\mathbf{q}_m = \alpha \mathbf{q}_0 + \beta(1/|C_r|) \sum_{\forall \mathbf{d}_j \in \mathbf{C}_r} \mathbf{d}_j \; - \; \gamma(1/|C_{nr}|) \sum_{\forall \mathbf{d}_j \in \mathbf{C}_{nr}} \mathbf{d}_j.$$

where $\mathbf{q}_0$ is the original query vector and α, β, and γ are weights attached to each term. These control the balance between trusting the judged document set versus the query. Starting from $\mathbf{q}_0$, the new query moves some distance toward the centroid of the relevant documents and some distance away from the centroid of the non-relevant documents.

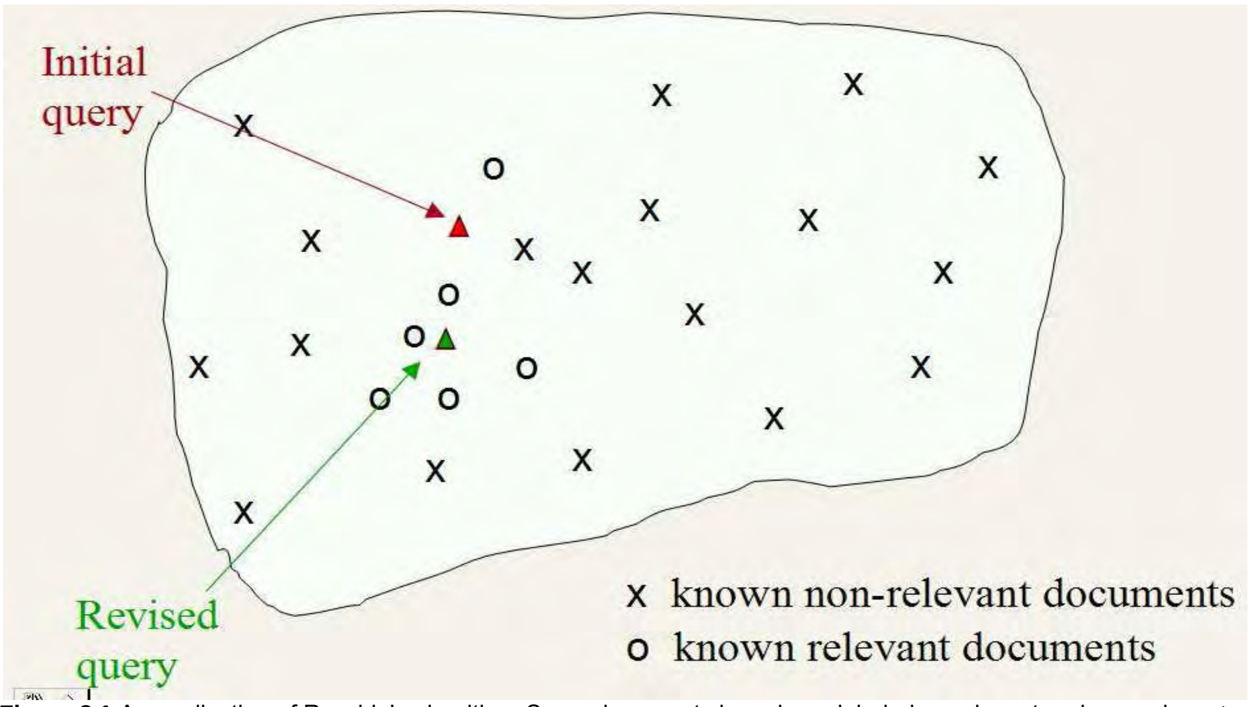

Figure 3.1 An application of Rocchio's algorithm. Some documents have been labeled as relevant and non-relevant and the initial query vector is moved in response to this feedback.

This new query can be used for retrieval in the standard vector space model. We can easily leave the positive quadrant of the vector space by subtracting off a non-relevant document's vector. In the Rocchio algorithm, negative term weights are ignored. That is, the term weight is set to 0. Figure 3.1 shows the effect of applying relevance feedback. Relevance feedback can improve both recall and precision. But, in practice, it has been shown to be most useful for increasing recall in situations where recall is important. This is partly because the technique expands the query, but it is also partly an effect of the use case: when they want high recall, users can be expected to take time to review results and to iterate on the search. Positive feedback also turns out to be much more valuable than negative feedback, and so most IR systems set $\gamma < \beta$. Reasonable values might be $\alpha = 1$, $\beta = 0.75$, and $\gamma = 0.15$. In fact, many systems, such as the image search system, allow only positive feedback, which is equivalent to setting $\gamma = 0$. Another alternative is to use only the marked non-relevant document which received the highest ranking from the IR system as negative feedback

3.3 Pseudo-relevance feedback

Pseudo-relevance feedback, also known as *blind relevance feedback*, provides a method for automatic local analysis. It automates the manual part of relevance feedback, so that the user gets improved retrieval performance without an extended interaction. The method is to do normal retrieval to find an initial set of most relevant documents, to then *assume* that the top *k* ranked documents are relevant, and finally to do relevance feedback as before under this assumption. This automatic technique mostly works. Evidence suggests that it tends to work better than global analysis. It has been found to improve performance in the TREC ad-hoc task. But it is not without the dangers of an automatic process. For example, if the query is about copper mines and the top several documents are all about mines in Chile, then there may be query drift in the direction of documents on Chile.

3.4 Query expansion

In relevance feedback, users give additional input on documents (by marking documents in the results set as relevant or not), and this input is used to reweight the terms in the query for documents. In *query expansion* on the other hand, users give additional input on query words or phrases, possibly suggesting additional query terms. Some search engines (especially on the web) suggest related queries in response to a query; the users then opt to use one of these alternative query suggestions. The central question in this form of query expansion is how to generate alternative or expanded queries for the user. The most common form of query expansion is global analysis, using some form of thesaurus. For each term, *t*, in a query, the query can be automatically expanded with synonyms and related words of *t* from the thesaurus. Use of a thesaurus can be combined with ideas of term weighting: for instance, one might weight added terms less than original query terms.

Chapter 4

HyREX – A XML Retrieval Engine

4.1 Introduction

HyREX is the Hyper-media Retrieval Engine for XML. Hyper because it offers explicit and implicit hyperlinks to the user. Media because it offers search facilities for text but also for other media than text, at least conceptually. Retrieval engine because it allows users to explore all kinds of information structures available through XML, not only plain document retrieval. XML because it allows retrieval under consideration of content and structure inherent in XML documents. HyREX is Open Source software. The current version allows for efficient retrieval of XML collections up to the gigabyte range.

4.2 HyREX Architecture

Figure 4.1 displays HyREX's architecture. On the top-most level the user contacts HyREX by means of an arbitrary Web browser.
Information needs issued through the Web browser are accepted by HyGate. It converts the user's request into a XIRQL query and delegates the processing to the lower levels of HyREX; the results are properly presented to the user.
On the conceptual level, XIRQL queries are accepted and processing. Whenever access paths are needed in order to further process a query, this request is handed to the physical level, which is named HyPath. On the physical level, there are a number of access paths for each datatype and predicate given in the XML documents.
The task of the document base administrator can be described by means of HyREX's different levels:
HyGate Describe the layout for search results and documents. This is done by specifying XSL stylesheets (see also Section 4.6).
XIRQL Specify data types of the various parts of documents by means of the DTD. This is done within a so-called document definition language (DDL) which is to be prepared for each document class. Section 4.2 describes how to do that.
HyPath Specify access structures for predicates and the structure of documents. This is also done within a DDL instance. See Section 4.2.

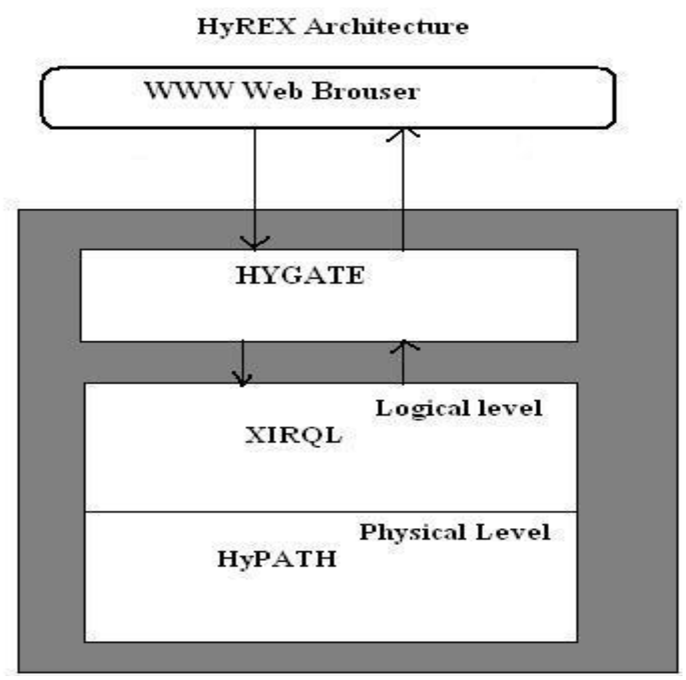

Figure 4.1

4.3. Index Structure Overview

We need to tell the system how to index the documents. For this, we have the so-called ‘data definition language’ (DDL).
A document collection is called a “base”. Inside it, there may be several “classes”; a class corresponds to a set of documents all conforming to the same DTD. In a class, there are several “datatypes”, each datatype provides several search “predicates”.

Document Definition Language: For indexing a given set of documents, we need the documents themselves, their respective DTDs (one for each of your document classes) and a DDL (data definition language) file for each of our documents classes. This DDL file tells the HyREX indexer how to index the documents. HyREX provides a DTD which describes the format of the DDL files (which are XML files).
A DDL file looks like this:

```
<? xml version="1.0" encoding="iso-8859-1”?>
<!DOCTYPE hyrex SYSTEM ".../doc/hyrex.dtd">
<hyrex attributes >
<access attributes > ... </access>
<convert attributes > ... </convert>
<summary attributes > ... </summary>
<datatype attributes > ... </datatype>
```

<inodes> ... </inodes>
<structure> ... </structure>
</hyrex>

The attributes of the <hyrex> element are:
directory This gives the directory where the index lives. In this directory, HyREX creates a directory named after the document base.
base This string gives the name of the document base. It is also used as a name for the directory where the index files for your various document classes live.
class The name of the document class to create within your document base.
dtd All documents of a given class to be indexed must comply with a DTD. Its file name is given here.
All attributes are required.

Example:
```
<hyrex directory="/tmp/hyrex"    base="example"    class="books"     dtd="/tmp/books.dtd">
...
</hyrex>
```

The <access> Element
This element must be present exactly once in a DDL file. It tells HyREX where to find the documents of a document class. This element has one attribute, classname, which refers to a HyREX document access class implementing a method to access documents in a certain way.
A few currently available classes:

Example:
HyREX::HyPath::Document::Access::XMLstream This document access class extracts subtrees of XML files. Each such subtree is considered to be a document in its own right.
HyREX::HyPath::Document::Access::Tar This document access class extracts files from tarballs (*.tar and *.tar.gz files).

The <convert> Element
This optional element is allowed to be present exactly once in a DDL file. It tells HyREX that your documents must be converted to XML. This element has one attribute, classname, which refers to a HyREX document convert class implementing a method to convert documents in a certain way.
The <summary> Element
Internally in HyREX, a query result is a weighted list of paths, where each path describes a node (XML element or XML attribute, usually) in an XML document. Paths look like book [3]/chapter [1] (first chapter in third book document). A summary is supposed to contain information that helps the user to identify the document. Summaries are automatically extracted from the XML documents according to the rules given in the <summary> element in the DDL. For example, for book summaries the elements title, author, year, and perhaps publisher might be useful.

The <datatype> Element
A data type in HyREX specifies which search predicates can be used in a query. This has an impact on the kinds of queries that users can formulate.
The <inodes> Element
Within the optional <inode> element one can specify so called index nodes. Index nodes are such nodes are the roots of subtrees in XML documents which serve as valid answer w. r. t. relevance oriented retrieval requests.
Index nodes (of course a document may have more than one index node, the root of a given document always is an index node) are specified by means of path expressions.
In the following example all 'section' nodes in the documents are treated as index nodes (in addition to the root node of the document):

```
<inodes>
<query query="//section"/>
</inodes>
```

The <structure> Element
The class specified in the <datatype> elements says how to index the values stored in certain regions of the XML documents. The classes specified in the <structure> element, however, say how to index the structural information in the XML documents. Currently, HyREX supports only one class for indexing the structural information:

Example: HyREX::HyPath::Structure::Tree
```
<structure classname="HyREX::HyPath::Structure::Tree">
<parameter name="compress" value="10"/>
</structure>
```

This class builds an external access path where the structural information for each document is stored separately. This class knows one parameter compress which specifies the effort used to determine an optimal compression for the structural information. Unfortunately the value depends on the number of documents; the range of values is from 1 to the number of documents you want to index. (HyREX does not know this figure before being finished with indexing; therefore it is not possible to provide a relative value or a percentage.)
The details of the DDL file structure is provided in the manual available with HyREX package.

4.4 How Index Is Stored

The elements, attributes and the text parts are assigned handle numbers (see section 1.3) while parsing the documents. A document id is assigned to each document of a particular class i.e., the documents are numbered according to the lexicographic order of their file names. In the DDL file if the access is so chosen that each such subtree of a document is considered to be a document in its own right then we need two keys [docid , handle] to identify a document of a class. The inverted list of each document class is stored separately. The words to be indexed are sorted ASCIIbetical order. The positions of occurances of the words are stored as follows:

```
[Word1]        [docid  tf  list_of_hanldes]  [docid  tf  list_of_handles]  [ …
[Word2]        [docid  tf  list_of_hanldes]  [docid  tf  list_of_handles]  [ …
…….
```

here tf is the term frequency.
The lists of all occurances are encoded according to some suitable encoding scheme to reduce storage space. Together with this the term frequency and inverted document frequency are stored separately such that once you get the docid from the inverted list the tf and idf are also known. The number of documents in a class, average document length, and the maximum term weight obtained in a document is also stored. All these are computed during the indexing stage.

4.5 How Structure Is Stored

XML structure tree or ***XS-tree*** data structure is used to store the structure of a xml document. It is highly compressed so that the XS-tree of the whole document collection can be kept in main memory. The XS-tree of each of the documents in constructed at the indexing phase (XS-tree is explained in section 1.3). From the index we can get the handle numbers and docid. So we take the document tree of the corresponding docid and output the exact path to the subtree in the document where the query terms occur. Consider the following XS-tree of the document in figure 1.4.
Array1=(Books,@value,author,#PCDATA,title,#PCDATA,Chapter,heading,#PCDATA,#PCDATA,…)
Array2=(1,2,2,3,2,3,2,3,4,3,…).

4.6 The Query Form

In an HTML query form, there can be a number of input fields. The question is, how to map this fairly flat structure onto the complex structure of XIRQL queries. This is done in a simple manner; only a fairly narrow class of XIRQL queries can be issued with HyGate. Here is an example of a XIRQL query that’s possible with HyGate:

```
/book[title $stemen$ "retrieval" $or$ author $soundex$ "fuhr"]
```

In general, such a query will consist of a prefix (here /book) followed by square brackets. Inside the square brackets there is a list of clauses separated by or. Here, there are two clauses, title $stemen$ "retrieval" and author $soundex$ "fuhr".

Each clause is a triple consisting of a *path condition, a search predicate,* and *a comparison value* (the last one is enclosed in double quotes). For example, the clause title $stemen$ "retrieval" has title as the path condition, $stemen$ as the search predicate and retrieval as the comparison value.
The above explanation is a bit simplified. Actually, it is possible for the user to enter several words into each search field. A word may begin with the + character, which indicates a mandatory condition, whereas the other conditions are optional. There are several methods for generating a query from the user input.

wsum The “wsum” method constructs a weighted sum from the user input, for example:

```
        /book [wsum ( 1.0, title        $stemen$      "retrieval",
                      5.0, author       $soundex$     "fuhr"     ) ]
```

Here, query conditions marked as mandatory by the user (via +) are given the weight 5.0 whereas the normal query conditions are given the weight 1.0. (HyREX will then normalize the weights internally such that they sum up to one.)
This method has the disadvantage that it might return documents for which none of the mandatory query conditions are fulfilled. However, if any mandatory query condition is fulfilled, then the corresponding document will appear near the top of the ranking list.

strict_bool The “strict_bool” method constructs a nested Boolean expression from the user input, for example:

```
/book [  ( title  $stemen$      "retrieval"     $and$   title    $stemen$      "information" )
                  $and$ (author   $soundex$    "fuhr"  $or$  author    $soundex$ "smith" )   ]
```

Here, mandatory query conditions are combined with and and optional query conditions are combined with or, and the mandatory and optional parts of the query are combined with and.
This method has the disadvantage that at least one of the optional query conditions must be fulfilled. In the extreme, if the user just types in +retrieval and fuhr, the two query conditions will be connected with and which is clearly the wrong thing to do. (However, connecting the mandatory part and the optional part with or has its own problems!)Thus, this method may return fewer documents than intended by the user.

4.7 Overview Of HyREX Indexing Algorithm

HyREX provides a program ‘hyrex_index’ to actually index the XML documents. For each of the document classes we need to write a separate DDL file and run this program with that DDL file supplied as command line argument.

The algorithm for indexing is as follows:
STEP 1:
For each of the documents in the document class do the following:

1: Find the document name, size, base directory, class name.

2: XML::Parser is used to test whether the document is syntactically correct according to the supplied DTD and build the XS-tree to store the structure information. Also store the handle number and the corresponding element name or attribute name or the text parts.

3: Create the document summary if summary element is provided in the DDL file.

4: Consider each handle and its corresponding entry stored in step 2 and apply index filters (for example remove stop words, apply stemmer etc).The filtering rules are fixed by the datatype specified in the DDL files.

5: Insert the filtered words and the corresponding handles in a temporary file.

STEP 2:
Sort the index keys in ASCII order and encode their handles so obtained.

STEP 3:
Close the first indexing phase and initialize the multiway-merge sort to merge indices of the documents.

STEP 4:
Take a key value pair and decode it, compute df, idf, tf and compute BM25 term weight store the max weight of a term in a particular document.
Also store triplets [$docid, $tf, $ list_of_hanldes] in encoded form in the new inverted list.

4.8 Overview Of HyREX search algorithm

HyREX provides a command line search tool ‘hyrex_search’ which is to be invoked with the DDL file name in command line.
The algorithm for search is as follows:
STEP 1:
Break the query into the following components:
[$predicate, $value, $pathexpr]
i.e., [search predicate, the word or phrase to be searched, path expression in the search]

STEP 2:

Check if the predicate given in the query matches some predicate in the supplied datatype by the ddl file.

STEP 3:
Filter the query terms using the same filtering rules as in indexing.

STEP 4:
```
Check if the filtered query terms lie in the index of the documents.
If not start from step 2 for the other query terms (in the wsum query).
Else
   Decode the information about the files from the inverted list.
   for ( i = 1 ; i <= $document_ frequency ; i++)
   {
          decode [$tf,$docid,$handles ]
          Compute BM25 term weight for all occurrences of the term in different documents.
          store as results:
          [appropriate section of doc where found the searched word, sum of term weights]
   }
```

STEP 5:
Decode the exact result paths using the corresponding stored XS-tree for that doc.

STEP 6:
Report results in descending order of result weights.

4.9 HyREX @ INEX 2002

We tried to use the *inex_eval executable 2003* for assessment of HyREX on 2002 IEEE collection , as they claim that this can also compute 2002 assessments when it is run with suitable parameters. But we found some problem in running the code. So we did the assessments manually on few queries and taking the top 20 retrieved results and checked with the available assessment files.
Using the Evaluation metrics in INEX 2002 (refer to [2]) and using strict quantization function for relevance and coverage viz,

$$f_{strict}(rel, cov) = \begin{cases} 1 \ if \ rel = 3 \ , cov = E \\ 0 \ otherwise \end{cases}.$$

The precision being computed by

$$P(rel \mid retr)(x) = \frac{x.n}{x.n + j + s.i/(r+1)} ,$$

here, n is the total number of relevant document components with regard to the user request in the collection; x ∈ [0, 1] denotes an arbitrary recall value. Given that the user stops viewing at the ranking after a given number of relevant document components NR. Let l denote the rank from which the NRth relevant component is drawn. Then j is the number of non-relevant document components within the ranks before rank l, s is the number of relevant components to be taken from rank l, and r and i are the numbers of relevant and non-relevant components in rank l, respectively.
we computed precession-recall for few queries:

INEX 2002 CO-query topic 32

Recall	Precession
0.05	0.0614
0.25	0.0587
0.50	0.0319
0.75	0.0239
0.80	0.0122

INEX 2002 CO-query topic 34

Recall	Precession
0.05	0.0891
0.25	0.0689
0.50	0.0542
0.75	0.0259
0.80	0.0127

Chapter 5

Pseudo-Relevance Feedback With HyREX

5.1 Pseudo-Relevance Feedback Algorithm

Let Q_0 denote the initial query as given by the user. Let α, β, γ denote the constants of Rocchio algorithm discussed in section 3.2.In HyREX by a single document we mean a subtree that is identified by its docid and the handle number. Let's assume that the top 10 retrieved documents by hyrex_search are really relevant.

Now what should be the dimension of the vector in the Rocchio equation (section 3.2, page 18) .Conceptually, it's the entire vocabulary in the inverted list of the document collection. But there is no need to consider all of them as a most of the terms are neither occurring in Q_0 nor in any of the top 10 relevant terms. Considering all terms will unnecessarily increase running time of the query reformulation step. So we need to find some term selection criteria. We can take two approaches:

(1). Pick terms with highest total weight in top 10 relevant documents. (2). Pick terms with highest document frequency in top 10 relevant documents.

Secondly we have to find what the non-relevant documents are. Here again we can take two different strategies:

(1).Assume all documents other than the top 10 as non-relevant. (2).Assume documents ranked more than K as non-relevant, where we may fix K=500

Algorithm:

Step1: Run hyrex_search on Q_0 and fin the top 10 relevant documents. We suppose that each of them is equally relevant.

Step2: For each of the above documents find the text they contain i.e., take [docid , handle] and find the xml file represented by the docid, then find the subtree in the file represented by the handle. Store all the text parts so found in a string S.

Step3: Filter the text in S using the same filtering algorithm as is done for both indexing and searching and get a set of words S'.

Step4: Sort the set of words in ASCII order.

Step5: For each of these sorted words find their inverted list entries.

Step6: Compute BM25 term weights for each of these documents (i.e., [docid , handle] pairs). Also compute the document frequencies.

Step7: Sort the terms according to one of the strategies i.e document frequency and BM25 weight or only BM25 term weights. Select top ranked terms from the sorted list and make the list S''.

Step 8: Let d = |S''|.Check if all the query terms lie in S''.If not add them and let x be the number of terms added and S''' be the final list. So D=d+x is our required dimension. Consider three vectors of dimension D.(1) query **Q** (2) relevant_document_average **R**(3) Non_relevant_document_average **N** all initialized to zero.

Step9:If the query is weighted normalize the weights and assign to the corresponding component in the vector **Q.** else assign equal weights 1/n to each component corresponding to query terms , n being the number of initial query terms.

Step10: Initialise Number_ of_ non-relevant documents = 0

```
For (each terms in S''' find the inverted list entries E) {
          For (each document in E check if it is relevant) {
               If (Relevant) {
                add its BM25 weight to the corresponding component in R
                }
               Else {
                    add its BM25 weight to the corresponding component in N
                    if(document is encountered first time) {
                    ++Number_ of_ non-relevant documents
                    }
                }
          }
}
```

Step 11: Use Rocchio formula to compute new query **Q'.**

$$\mathbf{Q'} = = \alpha\mathbf{Q} + \beta(1/|C_r|) \sum_{\forall \mathbf{d_j} \in \mathbf{C_r}} \mathbf{d_j} - \gamma(1/|C_{nr}|) \sum_{\forall \mathbf{d_j} \in \mathbf{C_{nr}}} \mathbf{d_j}.$$

Where C_r, C_{nr}, d_j have their usual meaning described in previous chapter.

Step 12: Select terms occurring with positive weights in the reformulated query vector **Q'** and generate a weighted query **Q''** with those terms.

Step 13: Invoke hyrex_search with **Q''** as new query.

5.1 Results

We did the assessments manually on few queries and taking the top 20 retrieved results and checked with the available assessment files. We found some improvements.

INEX 2002 CO-query topic 32

Recall	Precession
0.05	0.0872
0.25	0.0774
0.50	0.0735
0.75	0.0386
0.80	0.0184

INEX 2002 CO-query topic 34

Recall	Precession
0.05	0.0891
0.25	0.0801
0.50	0.0648
0.75	0.0212
0.80	0.0112

References

[1]Content-oriented XML retrieval with HyREX Norbert Gövert ,University of Dortmund, Germany Mohammad Abolhassani,Norbert Fuhr Kai Großjohann,University of Duisburg-Essen Germany

[2] FUHR, N., G¨OVERT, N., KAZAI, G., AND LALMAS, M. 2002. INEX: INitiative for the Evaluation of XML retrieval. In *Proceedings of the SIGIR 2002 Workshop on XML and Information Retrieval*,R. Baeza-Yates, N. Fuhr, and Y. S. Maarek, Eds. http://www.is.informatik.uni-duisburg.de/bib/xml/Fuhr_etal_02a.html.

[3] Understanding Inverse Document Frequency:On theoretical arguments for IDF. Stephen Robertson, Microsoft Research 7 JJ Thomson Avenue Cambridge CB3 0FB UK, (and City University, London, UK)

[4] Abolhassani, M.; Fuhr, N.; Gövert, N.; Großjohann, K. (2002). *HyREX: Hypermedia Retrieval Engine for XML*. Research report, University of Dortmund, Department of Computer Science, Dortmund, Germany.

[5] XIRQL: An XML Query Language Based on Information Retrieval Concepts NORBERT FUHR and KAI GROßJOHANN University of Duisburg-Essen.

[6] Index Compression vs. Retrieval Time of Inverted Files for XML Documents , Norbert Fuhr Norbert Govert, University of Dortmund, Germany ,http://ls6-www.cs.uni-dortmund.de/ir/projects/hyrex/.

Book reference:
Modern Information retrieval , Ricardo Baeza – Yates ,Berthier Riberiro-Neto