

Flexible Retrieval in a Structured Environment

A THESIS SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Aniruddha Mahajan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

September, 2004

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a master's thesis by

Aniruddha Mahajan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Name of Faculty Advisor(s)

Signature of Faculty Advisor(s)

Date

GRADUATE SCHOOL

Flexible Retrieval in a Structured Environment

Aniruddha S. Mahajan

September 8, 2004

Abstract

Flexible retrieval is particularly useful for XML documents, which have distinct structural components. Traditional text retrieval deals with returning a set of documents (usually in rank order), whereas flexible structured retrieval deals with retrieving the best document components with respect to the content of a structured query. The user can request and have returned particular components or elements (e.g., paragraph, figure, abstract, bibliography, etc.) of the XML document. Retrieval takes place at the appropriate level of granularity. Even when no explicit structural constraint is specified, structured retrieval identifies appropriate elements, thus making the user's task more manageable.

The research described and implemented in this thesis is the ad-hoc flexible retrieval of XML documents. The user is able to exploit the structural nature of the XML data and restrict the search to specific structural elements within an XML collection. The flexible system is built upon our earlier system for structured retrieval, which is itself based upon the Extended Vector Space Model supported by Smart.

Acknowledgements

I would like to express my gratitude towards the people who helped me during my graduate and thesis work, people who took efforts to ensure that my thesis work was up to the mark.

My advisor Dr. Carolyn Crouch for her support and guidance throught my graduate career. She was always patient and helped in all aspects of this research.

Dr. Donald Crouch and Dr. Bassam Shaer who gave me valuable feedback on my work.

Archana Bellamkonda who worked with me for two years and helped me during the same.

Sudip Khanna and Poorva Potnis for their help with programs.

Thanks to Lori Lucia, Linda Meek and Jim Lutinnen for being ever cheerful and ready to help with all matters concerned; and to my friends with whom I had a lot of fun and who made my stay in Duluth more enjoyable.

Contents

1	Introduction	1
2	Data and Evaluation Procedures for Experimentation	3
2.1	Data for Experimentation	3
2.1.1	The Document Collection	4
2.1.2	Queries	5
2.1.3	Assessments	5
2.2	Evaluation of Results and Methods	8
2.3	The Extended Vector Space Model	10
2.3.1	The Vector Space Model	10
2.3.2	The Extended Vector Space Model	11
2.3.3	Smart	11
2.3.4	Adapting the Extended VSM to CO Queries	12
3	The Flexible Retrieval System	14
3.1	Flexible Retrieval	14
3.1.1	What is Flexible Retrieval?	14
3.1.2	Why Flexible Retrieval?	14
3.1.3	The Task	15
3.2	The Top Level Design of the Flexible Retrieval System	16

3.2.1	Overview	16
3.2.2	Top-Down Approach	16
3.2.3	Bottom-Up Approach	18
3.3	Implementation of the Flexible Retrieval System	19
3.3.1	Overview of The Flexible System	19
3.3.2	Input to the Flex system	20
3.3.3	The Tree Representation	20
3.3.4	Populating the Tree	22
3.3.5	Ranking the Results	25
3.3.6	Formatting the Results	27
4	Experiments and Results	28
4.1	Experiments and Tabulated Results	28
4.2	Results - Graphs	29
5	Conclusions and Future Work	35
5.1	Conclusions	35
5.2	Suggested Future Work	36
A	Appendix: Flexible Retrieval: Algorithm	40

List of Figures

2.1	Structure of a typical INEX article	4
2.2	An example of extended vector space model	12
3.1	Tree structure of a typical XML document	17
3.2	Data associated with each node of a tree	21
3.3	Assigning relevance, coverage values to the leaf nodes	24
3.4	A populated tree	25
4.1	Article level indexing, strict quantization, <code>inex_eval</code>	31
4.2	Article level indexing, gen. quantization, <code>inex_eval</code>	31
4.3	Flexible retrieval, strict quantization, <code>inex_eval</code> , <code>add(Rel)</code>	32
4.4	Flexible retrieval, gen. quantization, <code>inex_eval</code> , <code>add(Rel)</code>	32
4.5	Flexible retrieval, strict quantization, <code>inex_eval</code> , <code>add(Rel)*avg(cov)</code>	33
4.6	Flexible retrieval, gen. quantization, <code>inex_eval</code> , <code>add(Rel)*avg(cov)</code>	33
4.7	Flexible retrieval, strict, <code>inex_eval</code> , <code>add(Rel)*avg(cov)</code> , 1500 results	34
4.8	Flexible retrieval, gen., <code>inex_eval</code> , <code>add(Rel)*avg(cov)</code> , 1500 results	34

List of Tables

2.1	4 point scale to access exhaustivity	6
2.2	4 point scale to access specificity	6
2.3	Possible e-value and s-value combinations	7
2.4	Subvector weights as used during Smart retrieval	13
4.1	Experiments performed and average precision obtained	30

Chapter 1

Introduction

Information retrieval is a developing and dynamic field. Every day millions of queries are sent out on the web and various search engines respond to them by returning a list of webpages for each query. However most of these systems deal with text-based retrieval, i.e., the retrieval of documents containing unstructured text. These particular retrieval techniques pay no attention to the structure of the document being returned. With the rapid spread of XML [Extensible Markup Language] throughout the web, retrieval techniques which utilize the specific features of XML are being developed. This thesis focuses on the flexible retrieval of XML elements (i.e., the retrieval of the most relevant paragraphs, subsections, sections, abstracts of documents) in addition to the document itself where it is appropriate.

XML is a step ahead of HTML, which is the current language of choice for the web. XML allows the flexible development of user-defined document types. It provides a "robust, non-proprietary, persistent, and verifiable file format for the storage and transmission of text and data both on and off the web; and it also eliminates the more complex options of SGML, making it easier to program for". [1]

In an XML document, part of the information associated with the document is contained in the structure of the document, and the rest is contained in textual format

within the elements (document components). Thus an XML document has tags distinguishing its various parts or elements. For example, the tags <author>, <para> and <sec> represent author, paragraph and section, respectively. (Thus for this thesis, the XML representation of its author is <author>Aniruddha Mahajan</author>). Using XML instead of HTML has one clear, distinct advantage. Depending upon the user's particular need, s/he can request and have returned, particular elements (e.g., abstracts, sections or paragraphs) of the XML document instead of the whole document itself. Even when a structural constraint is not specified by the user, the retrieval system can return elements instead of whole articles, which makes the user's task more manageable since s/he has only to peruse elements instead of entire documents. Traditional text retrieval deals with returning a set of documents (usually in rank order), whereas flexible XML retrieval deals with retrieving the best documents or document components with respect to the content of a structured query.

This is exactly what this thesis aims to achieve. It describes the addition of a *flexible retrieval* capability to an existing XML retrieval system. The University of Minnesota, Duluth is a participant in the INEX [INitiative for the Evaluation of XML retrieval] Workshop [2]. The task described in this thesis is the flexible retrieval of XML documents as it pertains to the INEX ad-hoc task.

In the following sections, we discuss the data used in these experiments and the procedures used to evaluate retrieval results (Ch. 2) and look at the Flexible Retrieval System itself (Ch. 3). We then report our experiments and results (Ch. 4). Finally, conclusions and suggestions for future research are presented in Ch. 5.

Chapter 2

Data and Evaluation Procedures for Experimentation

The data used in these experiments is the test collection developed as part of the INEX 2002 and 2003 workshops. INEX promotes the development and evaluation of content-based XML retrieval systems. The evaluation procedures used were also developed at INEX and distributed to the participants in the workshop. As participants of the workshop we contributed to the development of queries and the assessment of documents.

2.1 Data for Experimentation

The INEX test collection consists of three distinct parts: the document collection itself, a set of topics or queries, and a set of relevance assessments.

2.1.1 The Document Collection

The INEX document collection consists entirely of IEEE Computer Society publications from 1995 - 2002. Each document has a complex XML structure; on average, an article contains 1,532 XML nodes (where a node is typically represented by a pair of tags). Figure 2.1 shows the overall structure of a typical INEX document consisting of a *front matter* (<fm>), a *body* (<bdy>) and a *back matter* (<bm>).

```
<article>
  <fm>
    <ti>IEEE Transactions on ...</ti>
    <atl>Construction of ...</atl>
    <au>
      <fnm>John</fnm>
      <snm>Smith</snm>
      <aff>University of ...</aff>
    </au>
  </fm>
  <bdy>
    <sec>
      <st>Introduction</st>
      <p>...</p>
    </sec>
    <sec>
      <st>...</st>
      <ss1>...</ss1>
    </sec>
  </bdy>
  <bm>
    <bib>
      <bb>
        <au>...</au><ti>...</ti>
        ...
      </bb>
    </bib>
  </bm>
</article>
```

Figure 2.1: Structure of a typical INEX article

All the articles are marked up in XML and follow a common schema i.e., DTD.

Typically the *body* (<bdy>) tag encloses the main content and is structured into *sections* (<sec>), *subsections* (<ss1>) and *sub-subsections* (<ss2>). Each of these logical unit starts with a title followed by a number of *paragraphs* (<p>).

The *frontmatter* contains the article's metadata such as the title, author, abstract, etc., and the *backmatter* consists of its bibliography.

2.1.2 Queries

Queries or topics are essentially of two types: content-only (or CO) queries and content-and-structure (or CAS) queries. The participants of each year's INEX workshop develop these queries, which are representative of the range of real user needs. CO queries, for which the retrieval system should retrieve relevant XML elements of varying granularity, are similar to standard IR queries and do not have any structural constraints. CAS queries, on the other hand, have explicit structural constraints and may require the system to return specific elements (e.g., abstract). A total of 30 CO and 30 CAS queries were selected for the participants to work on during INEX 2003.

2.1.3 Assessments

The Relevance Assessments form an important part of the INEX evaluation process. After the topics are selected, each participating group uses its system to retrieve the top n (e.g. 1000 or 1500) elements. The results are pooled on a per-query basis. The pooled documents are then manually assessed by the participants to determine the relevance of each element. This manual judgment is accomplished by making use of the detailed explanation provided in the query itself and judging the relevance of each element with reference to it. The resulting relevance assessments provide the basis for system evaluation.

Relevance is defined by INEX with reference to the following two dimensions [3]:

- Exhaustivity (e-value) - describes the extent to which the document component discusses the topic.
- Specificity (s-value) - describes the extent to which the document component focuses on the topic.

To assess exhaustivity, we adopt the 4-point scale as shown in Table 2.1. To assess specificity, we adopt the 4-point scale shown in Table 2.2.

Table 2.1: 4 point scale to access exhaustivity

e-value	Meaning
0	Not exhaustive, the element does not discuss the topic at all.
1	Marginally exhaustive, the element discusses a few aspects of the topic.
2	Fairly exhaustive, the element discusses many aspects of the topic.
3	Highly exhaustive, the element discusses most or all aspects of the topic.

Table 2.2: 4 point scale to access specificity

s-value	Meaning
0	Not specific, the topic is not at all a theme of the element
1	Marginally specific, the topic is a minor theme of the element if at all
2	Fairly specific, the topic is a major theme of the element.
3	Highly specific, the topic is the only theme of the element.

The important point to note here is that even though the e-value and the s-value are totally different values, they are not independent of each other. A non-exhaustive element (0 e-value) is also not specific (has a 0 s-value) and vice versa, thus restricting to 10 the combination of e-value and s-value. (In INEX 2002, the terms *relevance* and *coverage* were used instead of *exhaustivity* and *specificity* respectively. These terms, particularly relevance, were deemed misleading and thus replaced).

The 10 possible combinations of e-value and s-value that an element can have are shown in Table 2.3

Table 2.3: Possible e-value and s-value combinations

e-value	s-value	Meaning
3	3	Highly exhaustive, Highly specific
3	2	Highly exhaustive, Fairly specific
3	1	Highly exhaustive, Marginally specific
2	3	Fairly exhaustive, Highly specific
2	2	Fairly exhaustive, Fairly specific
2	1	Fairly exhaustive, Marginally specific
1	3	Marginally exhaustive, Highly specific
1	2	Marginally exhaustive, Fairly specific
1	1	Marginally exhaustive, Marginally specific
0	0	Not exhaustive, Not specific

All that is required for an element to have an e-value of 3 is that almost all aspects of the topic are discussed within the element. The element may contain other, irrelevant information as well. Similarly, an element can be assessed as highly-specific (s-value 3) even if it discusses just a few aspects of the topic involved. For example, a small element like a paragraph may have an e-value of 1 (as it contains very little information on our topic), yet will have an s-value of 3 as long as the concerned topic is the sole theme of the component. However, any document component that does not discuss the concerned topic at all (e-value 0) must have an s-value of 0 and vice versa.

The rules for assigning exhaustivity and specificity values to elements as defined by the *INEX 2003 Relevance Assessments Guidelines* are as follows:

- As a general rule it can be said that the exhaustivity level of a parent element is always equal to or greater than the exhaustivity level of its child elements. This is due to the cumulative characteristics of exhaustiveness. For example, the parent of a highly exhaustive element will always be highly exhaustive since the child element already discusses all or most aspects of the topic.
- The parent of non-exhaustive child elements (i.e., all children having e-values of 0) will also be non-exhaustive (e-value 0).
- With regard to specificity, an element has a positive s-value which is greater than or equal to the maximum s-value of all its child elements if one of its child elements has an s-value greater than 0. For instance, if a parent element has tiny child element with s-value 1 and a large child element with s-value 2, then the s-value of that parent element will be 1 or 2 depending on the judgment made by the human assessor.

2.2 Evaluation of Results and Methods

Evaluation means assessing the value of a system or product. It plays an important part in the development of retrieval systems as it stimulates their improvement. The predominant approach to evaluating a system's retrieval effectiveness is through the use of test collections constructed specifically for that purpose. A test collection usually consists of a set of documents, user requests (topics), and relevance assessments (i.e., the set of elements previously assessed as relevant for each query). Several large-scale evaluation projects have resulted in establishing Information Retrieval test collections. One of the largest evaluation initiatives is the Text REtrieval Conference (TREC), which yearly since 1992 runs numerous tracks based on an increasingly diverse set of tasks to be performed on continually growing test collections.

Besides a test collection, the evaluation of retrieval effectiveness also requires appropriate measures and metrics. A number of measures have been proposed over the years. The most commonly used are recall and precision. Recall corresponds to, in the above specification of effectiveness, a *system's ability to retrieve as many relevant documents as possible*, whereas precision pertains to a *system's ability to retrieve as few non-relevant documents as possible*.

Given $Retr$ as the set of retrieved documents and Rel as the set of relevant documents in the collection, recall and precision are defined as follows:

$$recall = \frac{|Rel \cap Retr|}{|Rel|}$$

$$precision = \frac{|Rel \cap Retr|}{|Retr|}$$

A recall - precision graph is typically used as a combined evaluation measure for retrieval systems. Such a graph, given an arbitrary recall value, plots the corresponding precision value. Traditional evaluation procedures cannot be applied directly to INEX because of the inherently different nature of XML retrieval. The evaluation metrics applied to traditional text retrieval approaches do not take into account the structural information and content conditions associated with XML retrieval.

We evaluate our approach by using the evaluation procedures [4] developed by INEX. Our results are compared with the results of other groups participating in INEX to evaluate the utility and effectiveness of our approach. The XML retrieval engines used by participants are evaluated based upon the INEX Test Collection [Documents, Queries, Assessments] and uniform scoring techniques, including recall/precision measures, which take into account the structural nature of XML documents. These metrics have been implemented within the *inex_eval* package distributed to INEX participants.

Inex_Eval: is the metric developed by INEX for the evaluation of CO as well as CAS queries. In order to apply the recall/precision metric, first the assessors' judgments for exhaustiveness and specificity have to be quantised into a single relevance value. Based on the quantised relevance values, procedures that calculate recall-precision curves for standard document retrieval can be applied directly to the results of the quantization functions. [5]

This package produces two sets of results for two different quantization methods. The *strict* quantization method allows only the elements which are both highly exhaustive and highly specific to be considered as relevant within the assessments. Thus all comparisons are made against a limited set of elements from within the assessments. The *generalized* quantization method allows all elements within the assessments to be open for comparison (i.e., elements ranging from marginally exhaustive, marginally specific to highly exhaustive, highly specific).

2.3 The Extended Vector Space Model

2.3.1 The Vector Space Model

The Vector Space Model [6] was developed by Salton and used as the basis for the Smart retrieval system [7]. In the vector space model, each document (and query) is viewed as a set of word types and is indexed as a weighted term vector. [The indexing procedure uses a stop list to remove commonly occurring words (*and*, *or*, *the*, *of*, etc.) and is followed by a stemming procedure to generate word stems from the remaining words (e.g., *rotation* and *rotate* are reduced to stem *rotat*)]. A weight is assigned to each word stem and each document is represented as a vector of weighted word stems. Thus, the vector space model represents the document collection as a vector space of dimension $m \times n$, where m is the number of unique word stems in the

document collection and n is the number of documents in the collection. The queries, like documents, are represented as weighted term vectors.

In the vector space model, the relevance of a document to a query is represented by the mathematical similarity of their corresponding term vectors. A commonly used measure is the cosine of the angle between the two vectors. The smaller the angle between the corresponding vectors, the greater the similarity of the two vectors.

2.3.2 The Extended Vector Space Model

The Extended Vector Space Model [8] proposes a method for representing, in a single extended vector, different classes of information about a document, such as author name, bibliographic information, etc. This model allows for the incorporation of objective identifiers along with the usual content identifiers in the storage and retrieval of documents. In the extended vector space model, a document vector consists of a set of subvectors, where each subvector represents a different concept type or c-type. Similarity between a pair of extended vectors is calculated as a linear combination of the similarities of corresponding subvectors. The extended vector space model as shown in Figure 2.2 is suitable for representing documents with different classes of information.

2.3.3 Smart

Smart [7] is our basic retrieval engine. The advantage of using Smart is that it is based on the Vector Space Model and supports the Extended Vector Space Model. Our version [9, 10], used for structured retrieval, differs from the extended vector space model as described in 2.3.4. This system is used as a foundation for our current system, which adds a flexible retrieval capability to it.

Legend

\underline{d}	a content term vector
\underline{o}	an objective identifier vector (such as author name)
\underline{c}	a citation vector
\underline{D}	extended document vector
\underline{Q}	extended query vector
α, β, γ	similarity coefficients (concept type weights)
v_i	weight of the i^{th} component in a v-type vector

Extended Vectors \underline{D} and \underline{Q}

$$\underline{D} = (d_1, \dots, d_n, o_1, \dots, o_m, c_1, \dots, c_k)$$
$$\underline{Q} = (d_1, \dots, d_n, o_1, \dots, o_m, c_1, \dots, c_k)$$

Similarity of \underline{D} and \underline{Q} – $Sim(\underline{D}, \underline{Q})$

$$Sim(\underline{D}, \underline{Q}) = \alpha[\text{content terms similarity}] + \beta[\text{objective identifier similarity}] + \gamma[\text{citation similarity}]$$

Figure 2.2: An example of extended vector space model

2.3.4 Adapting the Extended VSM to CO Queries

The extended vector model relies on subvectors which can be weighted for specific tuning of the system. The documents as well as queries used contain multiple concept types (c-types). Each of the c-types are meaningful units of information. *Subjective identifiers* have chunks of text which cannot be objectively compared. The comparison and thus correlation of these c-types is always subjective. Some examples of subjective identifiers are <bdy>, <abs>, etc. Other c-types called as *objective identifiers* can be directly compared and matched or not. Some examples of subjective c-types are <author>, <pub_yr>, etc. Objective identifiers are used for processing CAS queries only.

The search words present within each CO query are associated with each of the subjective c-types and a corresponding extended vector query is formulated. Individual similarities of each subjective query subvector with the corresponding document subvector are calculated using c-type to c-type matching. The final similarity between the query and the document is computed as a linear combination of the similarities

of corresponding subjective subvectors using the formula shown in Figure 2.2. We use tuned weighting of subvectors within the extended vector space model with the subvector weighting as shown in Table 2.4

Table 2.4: Subvector weights as used during Smart retrieval

Subvector	Weight
bdy: body	2.00
bibl_atl: bibliography article title	2.00
abs: abstract	1.00
kwd: keywords	1.00
atl: article title	1.00
bibl_ti: bibliography	0.00
ed_intro: editor introduction	0.00
ack: acknowledgements	0.00

Chapter 3

The Flexible Retrieval System

3.1 Flexible Retrieval

3.1.1 What is Flexible Retrieval?

A retrieval method which returns to the user components of documents (as opposed to the traditional methods which return whole documents) may be called *flexible retrieval*. Document components are the structures contained within the document (e.g., sections, subsections, abstracts, paragraphs, section titles, etc.). A good flexible retrieval system will return a mixture of all these document components (elements) as well as whole documents as deemed fit by the relevance calculations.

3.1.2 Why Flexible Retrieval?

Often users are very specific about what information they want. Although an entire document may not be devoted to that topic, the topic may well be discussed within smaller units (i.e., the components or elements of the document). Suppose the user wants information regarding, say, *flexible XML retrieval*. S/he may not find a document devoted entirely to that. There may be documents on information retrieval in

general whose sections, subsections or paragraphs discuss the material at hand. In such a case, it is inappropriate to return a whole document to the user as relevant because the whole document is not relevant to the query. The user has to search through the entire document to find the relevant matter.

Flexible retrieval is important in the sense that the system is able to return to the user the most relevant element (as determined by the retrieval engine). In INEX terms, the element returned should be *exhaustive* and *specific* about the query.

Traditional information retrieval lacks this capacity mainly because such systems are unable to retrieve components within the documents. XML documents have a clear structure which the system can utilize. The queries can also make use of the structure and incorporate particular requests as seen in CAS queries. E.g., a CAS query may ask for abstracts from documents relevant to the query. Thus flexible retrieval as a whole is better in two ways: (1) flexible retrieval is able to return better elements to the user, and (2) the user can formulate a query to request particular elements.

3.1.3 The Task

Earlier versions of our retrieval system [9], [10], based on the Extended Vector Space Model, returned only whole documents ranked according to relevance. So in effect the task is to modify the retrieval process so as to return elements (document components) along with documents. The list of elements to be returned includes *abstracts*, *paragraphs*, *sub-sections*, *sections*, *bodies*, *articles*, *figure-titles*, *section-titles* and *introductory paragraphs*. These should be returned in rank order according to their respective relevance values as determined by the system. The method used to determine rank should incorporate both *exhaustivity* (relevance) and *specificity* (coverage) into the calculations with the objective of producing as nearly a faithful representation

of the manual assessments as possible.

3.2 The Top Level Design of the Flexible Retrieval System

3.2.1 Overview

Our goal is to add a flexible retrieval capability to the current retrieval system [9, 10]. We are using Smart, a retrieval engine based on the Vector Space Model. Smart normally produces a ranked list of documents as output. The job of the flexible retrieval system is to produce instead a ranked list of document components or elements. Figure 3.1 shows the tree layout of a typical XML article. Note that the root of the tree is the article itself, while the leaf-nodes can be seen to be the paragraphs contained within the XML document structure. Each document (article) is represented by a similar tree. The flexible retrieval module needs to return the relevant document components (e.g., <sec>, <ss1>, <ss2>, <p>, <bdy>, <article>, <ip1>, <abs>) as shown in Figure 3.1.

There are two basic approaches to constructing a flexible retrieval module: top-down and bottom-up.

3.2.2 Top-Down Approach

This approach requires an indexing at the article (document) level by Smart. The top down approach, as the name implies, starts at the top of the tree with the relevant articles as returned by Smart and works its way down to the paragraphs. It is the simplest approach in terms of pre-processing of data as the rankings of documents (articles) is already available to us. The flex module would take the ranked docu-

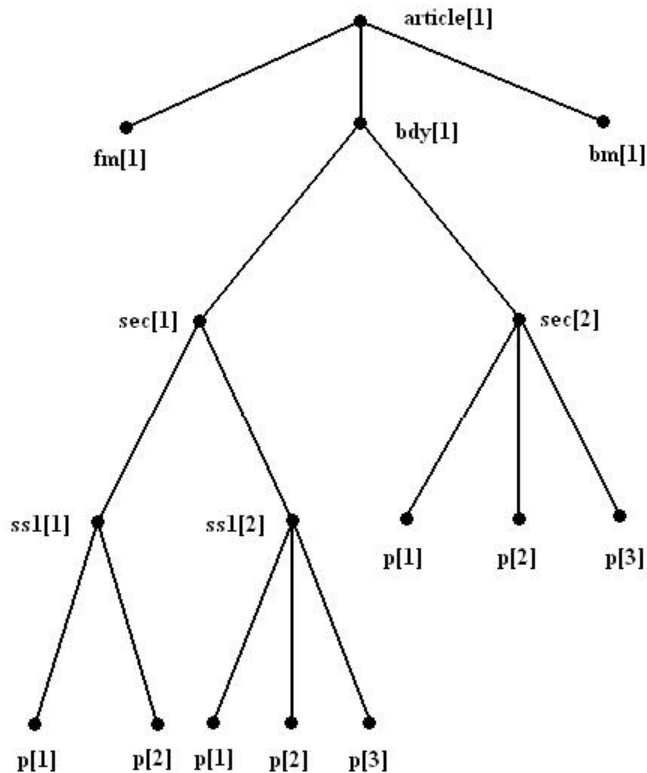


Figure 3.1: Tree structure of a typical XML document

ments as input, and output a ranked list of elements selected from these documents. However, this entails an extra stage within the module which must examine the document concerned and determine the relevance of each component within. This is a superfluous stage as the same document has already been processed by Smart. Still, this is necessary as Smart does not produce relevance values for each of the elements that make up the document and gives a single relevance value to represent the whole article. The last step here would be to rank the elements as per their relevance values and produce a listing of elements for each query.

The advantage of using this method is that we can use the current indexing of the document collection and do not need to index it at a different level. Also, the earlier system performed well in INEX 2003 [10] in returning a good set of relevant

documents. So in effect the top-down approach would be building on this firm base. The disadvantage is that our system must determine the relevance of each element within each document, repeating what has already been done by Smart.

3.2.3 Bottom-Up Approach

In the bottom-up implementation, our basic indexing unit (the basic unit of retrieval) is the paragraph. Under these conditions Smart considers each paragraph as if it were a separate document. Thus Smart indexes on that basis and returns a list of relevant paragraphs instead of articles. Each paragraph has a correlation with the query and Smart assigns it a correlation value. This value represents the calculated *relevance* of that paragraph to the query.

In the bottom-up approach we start at the leaf node level of the tree and work our way up until we reach the root (i.e., the apex or article node). We decide upon the levels of granularity that we want our system to analyze (eg., paragraphs, subsections, sections, body elements, articles etc). The flexible retrieval module starts with a ranked list of leaf nodes (paragraphs) and calculates the respective relevance value of each parent node as it works its way up the tree. Thus each parent node gets a relevance value based on the relevance values of its children. This continues until the article node is reached. When such calculations are done for each and every tree (i.e., each and every document or article), all elements in those trees have been assigned values which indicate their relevance. These elements are sorted to give a ranked list of document components.

To recap: The flexible retrieval module takes as input a ranked list of leaf nodes (paragraphs). As we continue to use the Smart system to do our retrieval for us, the output of Smart (which will be the input to the flexible retrieval module) must be paragraphs. This necessitates a new indexing performed at the paragraph level.

Here Smart views each paragraph as a separate document and calculates its relevance. Ultimately we end up with the output which is the documents as seen by Smart, i.e., paragraphs sorted in order of their correlation values.

The advantage of the bottom-up approach is that we can use Smart to perform the actual retrieval and calculation of relevance for each terminal node. It is a proven system and relevance is calculated exactly rather than approximately. For this reason alone, the bottom-up approach appears superior to the top-down method, and that is why it is selected in our implementation.

3.3 Implementation of the Flexible Retrieval System

3.3.1 Overview of The Flexible System

The flexible retrieval system implemented here works on a rank-ordered set of elements (paragraphs) given to it by Smart. These are leaf elements of the trees of articles in the document collection. The two main phases in the process are (1) building the tree structures, and (2) filling the empty trees with relevance values, i.e., *populating the trees*.

The system first constructs a tree representation of each article. All leaf elements have already been assigned relevance values by Smart. After the trees have been constructed, these relevance values associated with the terminal nodes are used to propagate a set of new values up each node of each tree, thus populating the tree. Finally, a list of elements from all trees with terminal nodes in the input list is gathered, sorted by correlation to the query, and returned. These are our flexible retrieval results for that particular query. All trees are *cleansed* before a new query is started.

3.3.2 Input to the Flex system

The flexible retrieval system needs 2 files as input:

To build the trees: a list of all possible leaf nodes, which means a list of all leaf nodes within all documents (articles) of the collection. This is a long list as one can imagine. It is used only in the building of trees. The tree-building module scans this file and constructs an empty tree for each document (article). [The creation of this file itself needs another program: one which maps the document number or Smart id of the paragraph to its *x-path* [11]]. Also, as the documents and the indexing do not change over time there is no need for the flex system to perform all the calculations and comparisons to create the trees every time. Thus, the empty trees are created only once, elements are written to disk and the data are read from the saved file at the start of each new run.

To populate the empty trees: a ranked list of leaf nodes (paragraphs). This is the output of the Smart retrieval system when the document collection is indexed at the paragraph level. This list contains the address (i.e., the x-path) of each retrieved element and its relevance value. This file is used to start populating the tree. Each relevance value in this file is used to fill in the corresponding leaf node in the tree.

Thus the first file is used to create the tree representations (generate the nodes of each tree) while the second file is used to start populating the trees.

3.3.3 The Tree Representation

Instead of actually constructing (many instances of) a tree data structure we are implementing the same in a Perl hash. A single hash can be used to encompass all the trees needed to cover all the represented documents. This is due to the fact that

the tree to which each element belongs (the root node, i.e., the article) can be gleaned from its x-path.

Thus the hash contains all the elements, i.e., all nodes of all trees concerned. The *key* for this hash is obviously the x-path of the element as it is unique for each element. Every key has two *values* - the relevance or exhaustivity of that node and its coverage or specificity. Thus this hash is actually a hash of arrays. Just as one would traverse a tree and locate the parents/child/sibling nodes . . . we can do the same for the hash. To find the parent/child/sibling, we look for the match to the appropriate regular expression. As the x-path is standardized, the regular expressions for matching the parent, children or siblings are the same for all nodes. We continue to call the hash a tree as it is easier to imagine and picture. Figure 3.2 shows the nodes of a tree structure and the data associated with each node. Note that all relevance and coverage values are initialized to zero.

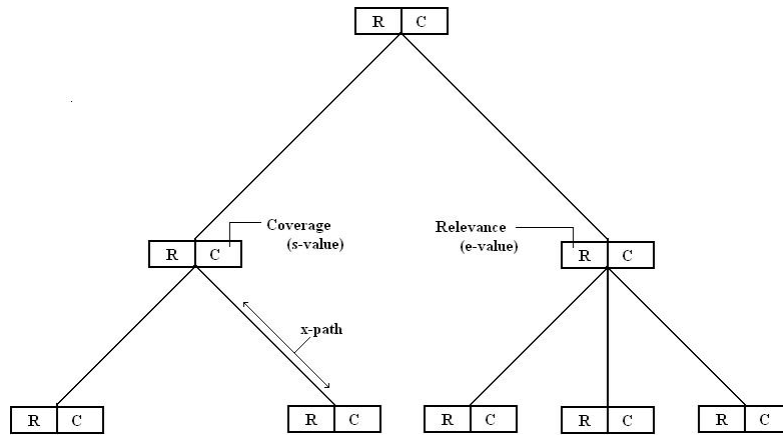


Figure 3.2: Data associated with each node of a tree

3.3.4 Populating the Tree

The second input file (i.e., the output of Smart) contains the leaf nodes (paragraphs) that are found relevant by Smart along with their respective relevance values. This file is read line by line (i.e., leaf node by leaf node) and the corresponding relevance (exhaustivity) value is fed into the hash. Thus when the whole file is read, some leaf nodes have been assigned e-values, the rest still are zero. We need two values for each node (exhaustivity and specificity), and yet only exhaustivity has been calculated. We have to explicitly generate the specificity values for the paragraphs. The parents of these nodes will then have both e and s values available for propagation.

One method of assigning s-values to is to assign each leaf node returned by Smart (each relevant leaf node) an s-value of 1, and the remaining leaf nodes (non-relevant ones) can continue to have an s-value of 0. Another method is to give the s-value of each leaf node a numerical value equal to its e-value. This is a better approach because the leaf node is essentially a paragraph. The higher its e-value, the the more relevant information it contains. Paragraphs by nature are small texts. So if a paragraph has a higher e-value, it means it contains more information on the query topic, leaving less space (due to small size) for other information. Thus we can say that for paragraphs the s-value is linearly proportional to the e-value.

Figure 3.3 illustrates the assignment of e-value and s-value to each leaf node in our tree. The paragraphs have been assigned e-values as given by Smart. For each leaf node the s-value is assigned the same e-value.

Once all leaf nodes have been assigned their respective values (a large number of them are not relevant and have a value of zero), the whole process of populating (i.e., assigning e-values and s-values to the rest of the nodes) the tree begins. For every leaf node, we find its parent and determine the relevance and coverage [e-value and s-value] for that particular node. The values (of the parent) are totally dependent on

the values of all its children. So what we actually do is find all the siblings of the selected child and process them to obtain the values for their parent. This process is repeated till all the nodes have been assigned values. The root node (i.e., the article node) is the last to be assigned a value in a tree. Remember that one tree represents a single document. A large number of documents are handled for each query.

Consider how the e-value and s-value of a non-terminal node are determined by our algorithm. We use a simple approach which follows the guidelines specified by INEX for e-value and s-value propagation within a document structure. We know that the e-value of a parent will be equal to or greater than the e-value of its child. This is because the e-value represents the amount of information that particular element provides about the query topic. The amount of information given by a parent will always be equal to or greater than its child. It will be equal if the child is the only relevant child of the parent. If there is more than one child which is relevant to the topic, then the amount of information contained within the parent is greater than that contained within any child. So the exhaustivity of a document component has a tendency to increase as its size increases. We simply add each child's e-value and assign the total to the parent.

Parent's e-value = Sum of e-values of all children.

Calculation of the coverage (s-value) is done separately. While e-value has a tendency to increase as we go up the tree (towards the root), the s-value has a tendency to decrease. This is because the s-value or specificity of a parent is always less than or equal to the maximum s-value of its children. The specificity (coverage) of a node is the measure of how specific the document component is with respect to the topic. As a parent will generally have more than one child and not all children are necessarily relevant to the topic, the parent often contains material which does not pertain to the topic as well. So specificity has a tendency to decrease as the size

of the document component increases. Thus we take the average of all the s-values of its children and assign it to the parent.

$$\text{Parent's s-value} = \text{Average of s-values of all its children.}$$

Figures 3.3 and 3.4 show a tree being populated using this method.

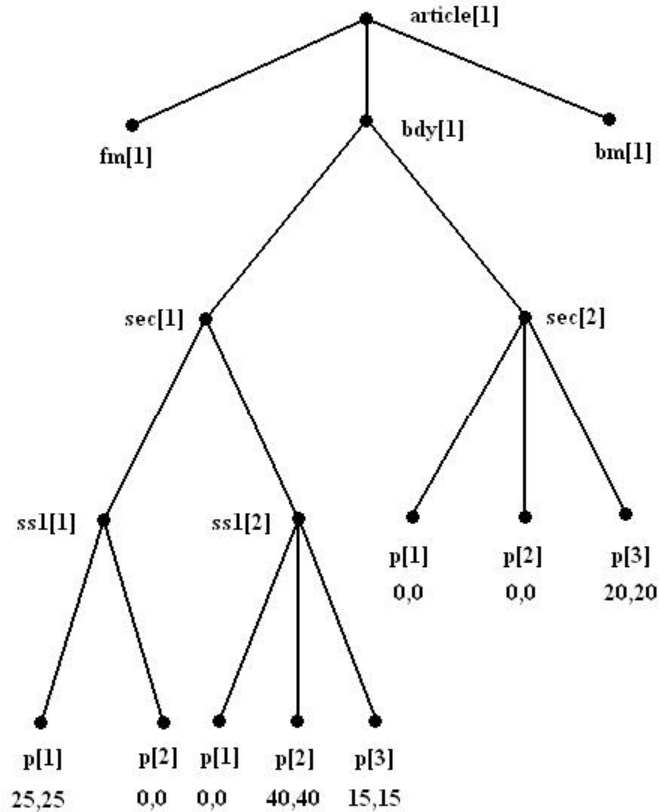


Figure 3.3: Assigning relevance, coverage values to the leaf nodes

This is the method of populating one tree. After all such trees (built from the paragraph level up) have been populated, we finally have values assigned to all concerned nodes. These values together tell us how relevant a particular node is to the topic.

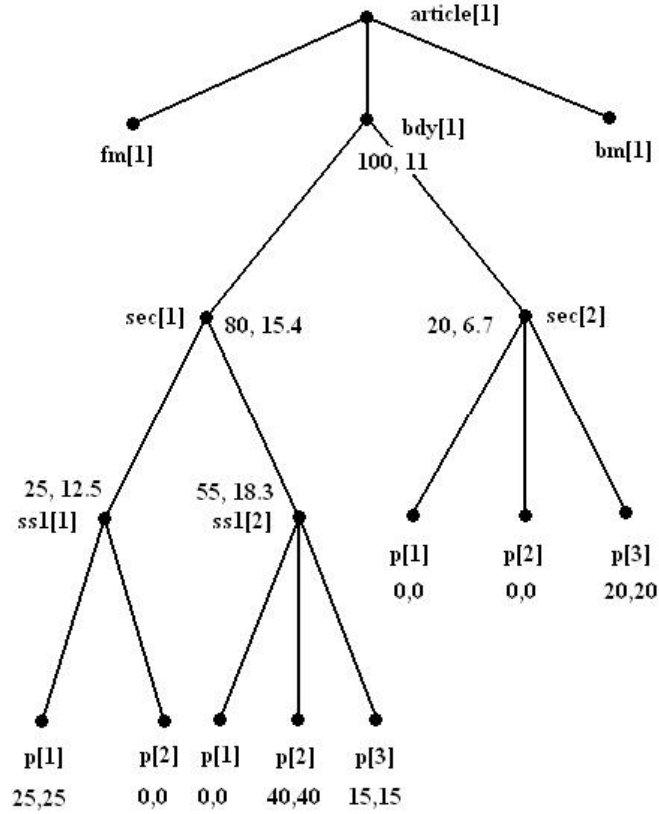


Figure 3.4: A populated tree

3.3.5 Ranking the Results

The e-value tells us how much relevant information is contained in that particular node, while the s-value tells us whether the node exclusively discusses our topic of interest or dwells upon other things as well (and if so, how much). The best possible document-component will be one which provides us with an exhaustive amount of relevant information and nothing else. Such a component will be highly exhaustive and highly specific. The higher both values are, the better the node. We need to keep in mind that as we go up the tree (towards the root), the e-value increases and the s-value decreases due to the inherent definitions of both values. We need a standard procedure to rank the document-components on the basis of these two values. We

considered the following options:

Relevance : Here e-value is the only value considered. This method considers only the relevant content of a document component. We know that greater the relevant content, the greater e-value. In this method the nodes are ranked according to decreasing e-value. However this method is biased towards the bigger elements such as article or body and thus the elements from the lower parts of the tree (like sections, subsections and paragraphs) have no representation in the top ranks.

Relevance and then Coverage : In this method, all elements are first sorted on relevance (e-value) and then (within this list) on coverage (s-value). So elements having the same e-value (or within a range) are sorted on the s-value. This means that relevance is given more importance, and among the equally (or similarly) relevant document components we prefer the more specific ones. However we find that this method does not differ much from the previous one as the e-value increases sharply while the s-value decreases gradually. So we still get the article and body elements at the top, relegating the rest to the bottom of the ranks.

Product : This method uses the product of the e-value and s-value which gives a fairly smooth curve and is not heavily biased towards the top elements. Since the e-value increases and the s-value decreases as we go up the tree, the product of these values can be said to be a fair estimate of the *goodness* of both. So a section may have a lower e-value than a body element, but as its s-value is higher, both compete fairly for the top ranks.

We use the third method, (i.e., the product) as a base for ranking the document components. A more effective method may be found later using logical and statistical

approaches.

3.3.6 Formatting the Results

Finally the results are formatted in accordance with INEX guidelines [12] and evaluated to compare the performance of our system against the manual relevance assessments performed.

Chapter 4

Experiments and Results

4.1 Experiments and Tabulated Results

Any system needs to be tested for effectiveness upon completion. The retrieval effectiveness of our system is tested using the INEX evaluation metrics. Both *inex_eval* and *inex_eval_ng* can be used to evaluate CO queries. CAS queries are evaluated using *inex_eval* only. The scope of this thesis pertains only to the CO queries and their retrieval. Retrieval performed on CAS queries also uses the flexible retrieval system developed here but due to the special circumstances associated with CAS retrieval, they are discussed under [13].

The experiments were performed on data indexed using Lnu-ltu weighting [14]. For retrieval of CO queries using our retrieval system, best results were obtained using Lnu-ltu weighting [9]. Table 4.1 shows experiments performed and their results. Results shown are for 100 retrieved elements/documents per query except for the last row where 1500 elements were returned. It displays (in order) the level of indexing (i.e., the node at which document collection is indexed by Smart), the type of weighting used, type of retrieval (i.e., document or flexible), the description of the method used in propagating the correlation values up the tree and the method used

for sorting the results, average precision as calculated by the strict method and average precision as calculated by the generalized method. The fourth column explains how the relevance (e-value) and coverage (s-value) were propagated up the tree. Let us elaborate on this:

Rel : Only relevance (not coverage) was calculated. This method is used for document retrieval (the earlier version of our retrieval system which retrieved documents and not components calculated only relevance and it was the sole criterion for ranking the documents). The issue of propagating values up a tree does not arise in this case.

add(Rel) : This method is used in a flexible retrieval where no coverage (i.e., s-value) was calculated for any nodes. Relevance (i.e., e-value) was propagated up the nodes of a tree. The e-value of a parent is the sum of the e-values of all its children. The final results are obtained as a sorted list of s-values of all nodes.

add(Rel) * avg(Cov) : This method calculates both e-value and s-value during flexible retrieval. Relevance of a node is propagated as the sum of the e-values of all its children while coverage is propagated as the average of the s-values of all its children. Final ranking is based on the product of the e-value and s-value of each node. This list is sorted in rank order.

4.2 Results - Graphs

The *inex_eval* evaluation tool developed by INEX evaluates the formatted results to give the average precision of our retrieved elements. It also plots a recall-precision

Table 4.1: Experiments performed and average precision obtained

Indexing	Weighting	Type	Desc	P-strict	P-gen
article	Lnu-ltu	document	Rel	0.0648	0.0251
paragraph	Lnu-ltu	flexible	add(Rel)	0.0717	0.0332
paragraph	Lnu-ltu	flexible	add(Rel) * avg(Cov)	0.0847	0.0376
paragraph	Lnu-ltu	flexible	add(Rel)*avg(Cov) 1500	0.1022	0.0712

graph for the results. Two values are calculated: the strict and the generalized approach. The figures below display this graph as well as the average precision calculated for these experiments.

Figures 4.1 and 4.2 display the recall-precision graphs plotted using the *inex_eval* evaluation tool for article retrieval. Figures 4.3 and 4.4 show the results for flexible retrieval as described in *add(Rel)*. Figures 4.5 and 4.6 show the flexible retrieval results for the approach described in *add(Rel)*avg(cov)*. Figures 4.7 and 4.8 show the flexible retrieval results for the approach described in *add(Rel)*avg(c-v)*. In this run 1500 elements are returned instead of the default 100.

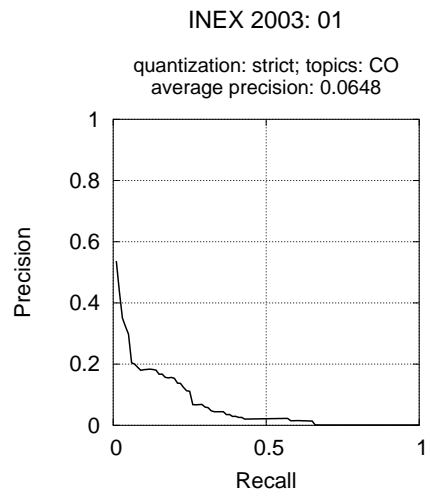


Figure 4.1: Article level indexing, strict quantization, `inex_eval`

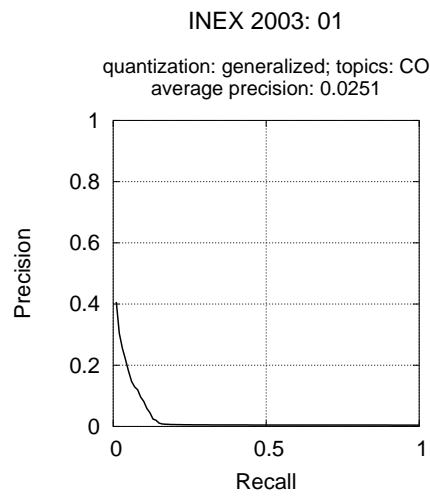


Figure 4.2: Article level indexing, gen. quantization, `inex_eval`

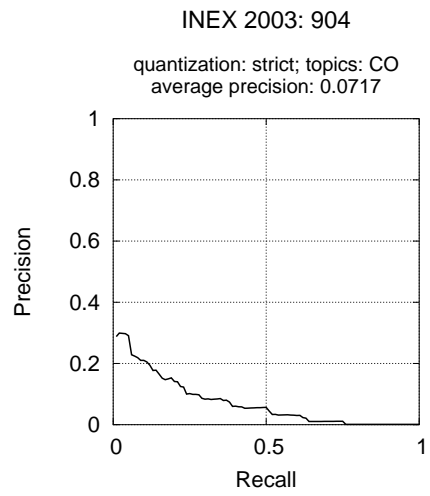


Figure 4.3: Flexible retrieval, strict quantization, `inex_eval`, `add(Rel)`

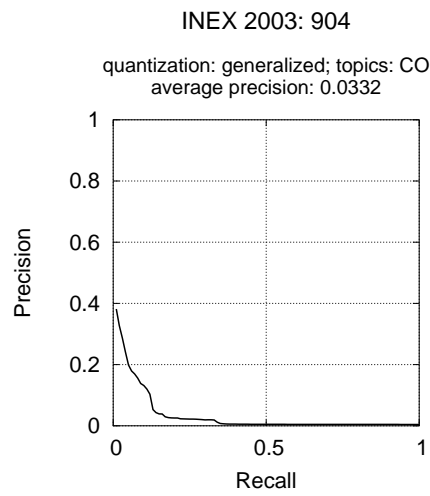


Figure 4.4: Flexible retrieval, gen. quantization, `inex_eval`, `add(Rel)`

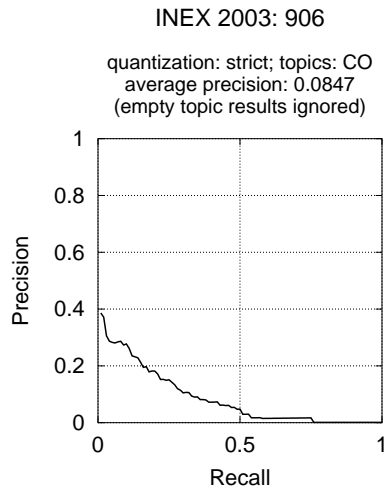


Figure 4.5: Flexible retrieval, strict quantization, `inex_eval`, $\text{add}(\text{Rel}) * \text{avg}(\text{cov})$

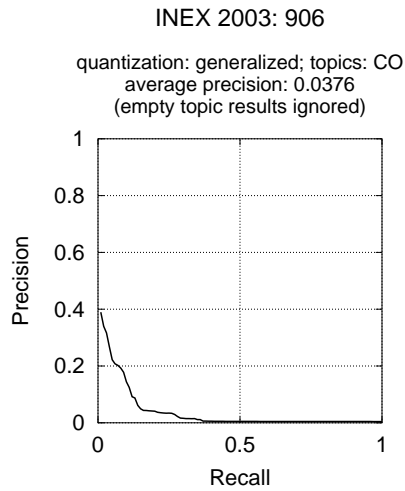


Figure 4.6: Flexible retrieval, gen. quantization, `inex_eval`, $\text{add}(\text{Rel}) * \text{avg}(\text{cov})$

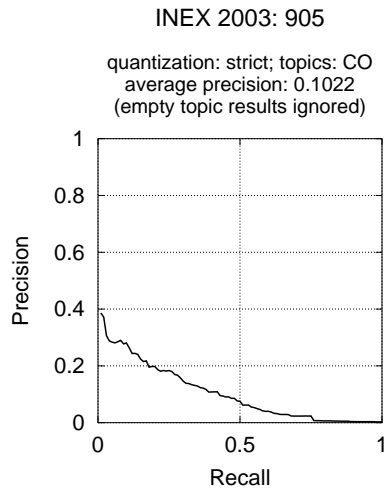


Figure 4.7: Flexible retrieval, strict, inex_eval, $\text{add}(\text{Rel}) * \text{avg}(\text{cov})$, 1500 results

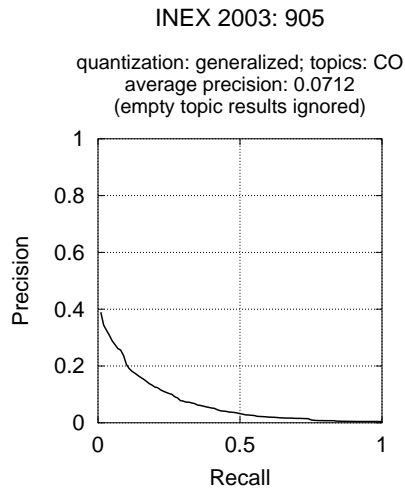


Figure 4.8: Flexible retrieval, gen., inex_eval, $\text{add}(\text{Rel}) * \text{avg}(\text{cov})$, 1500 results

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This is our initial approach towards the construction of a Flexible Retrieval System. The aim of this work was to create a basic, working Flexible Retrieval System. The system is still in the early stages of development. We evaluate our approach using the Content-Only (CO) and Content-and-Structure (CAS) queries and the document collection provided by INEX. The CO queries, having no structural restrictions or containment conditions, require that the most relevant elements be retrieved. The experiments performed on the CAS queries and their results are presented in [13]. The results obtained are better than the earlier version of the system in the sense of a higher average precision value as demonstrated in Table 4.1.

In this research, we analyze various approaches towards the building of the flexible retrieval system, the method by which the trees are built, how the relevance (e-value) and coverage (s-value) are calculated, the method by which these values are propagated up the tree, and the ranking of the elements retrieved by our system. Our experiments reveal that the best approach gives distinctly superior results over the earlier version of the system (document level retrieval) as well as our other approaches

towards flexible retrieval.

We conclude that flexible retrieval is necessary for element level retrieval and the currently implemented system achieves this goal while improving the average precision of the results.

5.2 Suggested Future Work

This system is the first implementation of a Flexible Retrieval System. The most important aspect of the problem that remains unsolved is to improve the precision of the system so that the user can get a closer match to the ideal results. A few suggestions are made below:

1. The paragraphs that are used within flexible retrieval process can be filtered by accepting only those paragraphs which consist of, say, at least n words. This is because paragraphs are essentially small elements and even if the keywords of the query appear within them, they can be considered highly relevant while not actually being so (i.e., the paragraph may just mention the concerned topic and not actually discuss it). As the length of a paragraph is short, it may be assigned a higher precision than actually appropriate. Such elements may be ranked higher than necessary and prevent actually relevant elements from occurring in the top ranks. Weeding out such paragraphs by specifying a minimum length would be a good approach towards increasing precision.
2. New ranking methods need to be explored, which would study and use the statistical and logical data offered by the current results. A new method of ranking may utilize a more complex ordering approach than the simple product of the e -value and s -value being taken right now.
3. Tree building can be restricted to just those trees (articles) whose paragraphs

occur in the high ranked document level Smart results. The rationale for this is that we already have ascertained the Smart document retrieval results to be good [10] and thus in this approach we would be generating the list of relevant elements from within those articles. A disadvantage is that this requires the system to maintain two indexings (at both the article and paragraph levels).

Bibliography

- [1] Flynn P., The XML FAQ. <http://www.ucc.ie/xml/>
- [2] Fuhr N., Malik S., Lalmas M. Overview of the INitiative for the Evaluation of XML Retrieval (INEX) 2003. In: *Proceedings of the INitiative for the Evaluation of XML Retrieval (INEX) 2003 Workshop*, Schloss Dagstuhl, Germany, 2003.
- [3] Kazai G., Lalmas M., Piwowarski B. INEX 2003 Relevance Assessment Guide. <http://inex.is.informatik.uni-duisburg.de:2003/internal/>
- [4] Kazai G., Report of the INEX 2003 Metrics Working Group. In: *Proceedings of the INitiative for the Evaluation of XML Retrieval (INEX) 2003 Workshop*, Schloss Dagstuhl, Germany, 2003.
- [5] INEX down, up load website, Evaluation. <http://inex.is.informatik.uni-duisburg.de:2003/internal/#eval>
- [6] Salton G., Wong A., Yang C. A Vector Space Model for Automatic Indexing. *Comm. ACM* 18(11), 613-620, 1975.
- [7] Salton G., editor. *The SMART Retrieval System - Experiments in Automatic Document Retrieval*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

- [8] Fox E. Extending the Boolean and Vector Space Models of Information Retrieval with P-norm Queries and Multiple Concept Types. Ph.D. Dissertation, Department of Computer Science, Cornell University, 1983.
- [9] Crouch C., Apte S., and Bapat H. Using the Extended Vector Model for XML Retrieval. In: *Proceedings of the First Workshop of the Initiative for the Evaluation of XML Retrieval (INEX)*, (pp. 95-98), Dagstuhl, Germany, 2002.
- [10] Crouch C., Apte S., and Bapat H. Adapting the Extended Vector Space Model for XML Retrieval. In: *Proceedings of the INitiative for the Evaluation of XML Retrieval (INEX) 2003 Workshop*, Schloss Dagstuhl, Germany, 2003.
- [11] XML Path Language (Xpath). <http://www.w3.org/TR/xpath>
- [12] Kazai G., *et al.* INEX03 Retrieval Task and Result Submission Format Specification. In: *Proceedings of the INitiative for the Evaluation of XML Retrieval (INEX) 2003 Workshop*, Schloss Dagstuhl, Germany, 2003.
- [13] Bellamkonda A., Automation of Structured Query Processing. M.S. Thesis, Department of Computer Science, University of Minnesota Duluth, 2004. <http://www.d.umn.edu/~ccrouch/>.
- [14] Singhal A., Salton G., Mitra M., Buckley C. Document Length Normalization. *Information Processing and Management*, 32(5), 619-633, 1996.

Appendix A

Appendix: Flexible Retrieval: Algorithm

Preprocessing: A list of all possible leaf nodes is prepared by an XML parser. This list is one of the inputs to the Flexible Retrieval System. The Document Collection is indexed at the paragraph level. Smart returns ranked list of paragraphs as output to a query. This is the second input to the program described below.

1. Accept as input the list of all leaf nodes within the document collection. Construct tree structures to represent articles from leaf nodes given. The x-paths of the nodes indicate their parents and help in construction of the tree for that article. Construct one tree to represent each article within the document collection.
2. Consider second input file, i.e. the output of Smart which is a list of paragraphs along with the correlation of each paragraph to the query. Read this list query by query. Assign the correlation value of each paragraph as the e-value and s-value of the corresponding node. For each query, perform the function to populate trees. Before starting on a new query, cleanse all trees, i.e. reinitialize

values within all nodes to zero.

3. Populating the trees: For each node within the tree structures, starting from leaf nodes with non-zero values assigned, search for its siblings. Also find the parent of these siblings. Assign e-value and s-value to the parent accordingly. Mark these nodes as processed. Continue searching for siblings and move up the tree when all nodes from a level with non-zero e-value have been processed until e-value and s-value have been assigned to the article node.
4. Sort all nodes from all trees according to the product of their e and s values. Write node x-path and e, s values to disk. Proceed to next query.