

Design and Implementation of a Flexible Retrieval System

A thesis
submitted to the faculty of the graduate school
of the University of Minnesota
by

Sudip Khanna

in partial fulfillment of the requirements
for the degree of
Master of Science

September 2005

Department of Computer Science
University of Minnesota Duluth
Duluth, MN 55812
USA

UNIVERSITY OF MINNESOTA

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

Sudip Khanna

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.

Carolyn J. Crouch

Name of Faculty Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

© Sudip Khanna 2005

Abstract

Traditionally, the field of information retrieval has focused on retrieving information from unstructured documents. But with the tremendous growth of the web and the emergence of XML as a preferred standard for storing documents, the focus is now shifting to retrieving information from structured documents. Structured documents allow us to partition the content into different document elements. These elements can divide the content into mutually-exclusive parts or give a tree-like hierarchical structure to the document.

Flexible retrieval is the task of retrieving specific, relevant portions of documents in response to a user query. The query may or may not explicitly specify structural constraints. The research described in this thesis aims to develop an approach for flexible retrieval that requires neither indexing every document element in the collection nor evaluating queries against multiple indices. We use a single index of non-overlapping leaf nodes from document trees and propose a measure that can be used to compare elements at different levels in the document tree.

Acknowledgements

I would like to take this opportunity to convey my sincere thanks to my advisor Dr Carolyn Crouch for her constant support and for sharing her immense experience in this field. She was always patient and helped me in all aspects of my work.

I would like to thank Dr Donald Crouch and Dr Barry James for their valuable feedback on my work.

I would also like to thank Poorva Potnis and Nagendra Doddapaneni for being supportive co-workers as well as invaluable critics of my work. I thank GMVS Murthy and Vishal Bakshi for their help with the experiments.

I would like to thank the staff at the Department of Computer Science at the University of Minnesota Duluth – Lori Lucia, Linda Meek and Jim Luttinen for their help, especially during the summer.

Finally I would like to thank all my friends who made my stay in Duluth unforgettable.

Table of Contents

1. Introduction.....	1
2. Background.....	3
2.1 Document collection.....	3
2.2 Query collection.....	4
2.3 Relevance Assessments	6
2.4 Evaluation metrics	7
2.5 Smart.....	7
2.6 Approaches to Flexible Retrieval	8
3. The Flexible Retrieval System.....	11
3.1 Flexible retrieval.....	11
3.2 Overview of the system	12
3.3 Design of the Flexible Retrieval System	13
3.4 Implementation of the Flexible Retrieval System	21
4. Experiments and Results.....	25
4.1 Experiments	25
4.2 Results.....	32
4.3 Parameters for INEX 2005	33
5. Conclusions and Future Work	34
5.1 Conclusions.....	34
5.2 Suggestions for future work.....	35
Bibliography	36
Appendix A.....	38

List of Figures

Fig 2.1 overall structure of a typical document	4
Fig 2.2 typical INEX CO and CAS queries	5
Fig 2.3 4-point scale to measure exhaustivity and specificity	6
Fig 2.4 recall and precision measures	7
Fig 3.1 an extended vector representation of a document	13
Fig 3.2 an overview of the Flexible Retrieval System	14
Fig 3.3 merging <i>nmn</i> vectors to generate vectors for inner nodes	15
Fig 3.4 pivoted normalization	16
Fig 3.5 <i>Lnu</i> element term weighting formula	17
Fig 3.6 <i>ltu</i> query term weighting formula	18
Fig 3.7 calculating score for a document element	19
Fig 3.8 calculation of score for two vectors of different lengths	20
Fig 3.9 document structure representation	22
Fig A.1 class diagram for Flexible Retrieval System	38
Fig A.2 sequence diagram for Flexible Retrieval System	39

List of Tables

Table 3.1 slope and pivot values at different levels in document tree	17
Table 3.2 sub-vector weights used for calculating final score of a document element ...	19
Table 4.1 effect of sub-vector weights on mean precision	26
Table 4.2 effect of extended vectors on mean precision	27
Table 4.3a effect of minimum size of elements on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights	29
Table 4.3b effect of minimum size of elements on mean precision with only <i>bdy</i> sub-vector	29
Table 4.4a effect of input number of leaf nodes on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights and inner product similarity measure ...	30
Table 4.4b effect of input number of leaf nodes on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights and cosine as similarity measure	30
Table 4.4c effect of input number of leaf nodes on mean precision with only <i>bdy</i> sub-vector and inner product as similarity measure	31
Table 4.4d effect of input number of leaf nodes on mean precision with only <i>bdy</i> sub-vector and cosine as similarity measure	31
Table 4.5 comparison of current system with previous version and the base run	33

1. Introduction

Traditionally, the field of information retrieval has been focused on retrieving information from unstructured documents. But with the tremendous growth of web and the emergence of XML as a preferred standard for storing web documents, the focus is now shifting to retrieving information from structured documents. This is also known as structured retrieval.

Structured documents allow us to partition the content into different document elements, each of which can be explicitly marked and identified. These elements might carry additional semantic information about their content or might just be used to provide overall structure to the document. These elements could partition the content into mutually-exclusive parts or give a tree-like hierarchical structure to the document, where elements have a parent-child relationship (i.e., their content overlaps). To increase the usefulness of a structured document collection, we must allow queries that refer to this structure.

An information retrieval system working with a structured document collection must allow users to explicitly refer to the document structure in their queries. A traditional IR system can be extended to handle such queries by first processing the queries and then filtering the retrieval results such that all retrieved elements satisfy the structural constraints.

However, the more difficult to handle queries are those for which the user does not explicitly specify any structural constraints, but the system is required to retrieve

specific relevant portion(s) of documents in response [1]. This is referred to as Flexible Retrieval [12]. The IR system, in this case, needs to retrieve from the document collection at different granularity levels and compare elements from different levels for relevance.

Flexible retrieval warrants evaluating a document element from two different aspects – relevance (exhaustivity) and coverage (specificity). Relevance is a measure of how much of the query's information need is satisfied by this element and coverage refers to how focused this element is on the information need. A flexible retrieval system needs to retrieve elements that have both high relevance and high coverage.

This thesis is aimed at designing and implementing a flexible retrieval system for a collection of hierarchically-structured documents. This could be a static collection of XML documents in an electronic library or a dynamically changing set of HTML documents on the web.

In Chapter 2, we discuss the data used for experimentation as well as various past approaches to flexible retrieval. Chapter 3 describes our approach to flexible retrieval and its implementation details. We discuss the experiments designed to evaluate our approach in Chapter 4 and their results in Chapter 5. Finally, conclusions and suggestions for future research are provided in Chapter 6.

2. Background

The data used in our experiments and the metrics used to evaluate our approach are provided by INEX. The INitiative for the Evaluation of XML retrieval (INEX) [1] is an international effort that was set up in 2002 to provide an infrastructure for evaluating various approaches to content-oriented retrieval of XML documents. INEX 2004 provides a large XML test collection consisting of publications of the IEEE Computer Society between 1995 and 2002 and various metrics to evaluate the results obtained from different approaches.

Our research group at the University of Minnesota Duluth participates in the ad-hoc retrieval and relevance feedback tracks at INEX.

2.1 Document collection

The document collection provided by INEX consists of about 12,000 IEEE articles published between 1995 and 2002. All these documents are formatted as XML and conform to a single DTD. A document, on average, contains about 1500 elements and is primarily divided into three parts – front matter, body and back matter. Front matter contains information such as the article title, author name, abstract, etc., and the back matter contains the bibliography. Body contains the bulk of the document’s content. Fig 2.1 shows the overall structure of a typical document. More details can be found in [1].

```
<article>
  <fm>
    <atl> Pioneering Work in ...</atl>
    <au>
      <fnm>Thomas M.</fnm>
      <snm>Stout</snm>
    </au>
    <abs> ... </abs>
  </fm>
  <bdy>
    <sec>
      <st>Introduction</st>
      <p> ... </p>
    </sec>
    <sec>
      ...
    </sec>
  </bdy>
  <bm>
    <bib>
      <bibl> ... </bibl>
    </bib>
  </bm>
</article>
```

Fig 2.1 Overall structure of a typical document

2.2 Query collection

INEX 2004 supports two types of topics [1]:

- Content-only (CO) topics: topics that ignore the document structure, focusing only on the information need.
- Content and Structure (CAS) topics: topics that contain explicit references to XML structure.

Fig 2.2 shows typical CO and CAS queries. In INEX 2005, these queries can be interpreted in multiple ways. A CO query may additionally have structural hints provided by user to guide the retrieval system. These are known as CO+S queries.

```

<inex_topic topic_id="91" query_type="CO" ct_no="7">
  <title>Internet traffic</title>
  <description>
    What are the features of Internet traffic?
  </description>
  <narrative>
    A relevant document/component describes the features of the internet
    traffic, i.e. it contains information on traffic evolution, its measurement
    and its possible congestion
  </narrative>
  <keywords>
    internet, web, traffic, measurement, congestion
  </keywords>
</inex_topic>

<inex_topic topic_id="149" query_type="CAS" ct_no="148">
  <title>
    //article[about(./(abs|kwd), "genetic algorithm")]
    //bdy//sec[about(.,simulated annealing)]
  </title>
  <description>
    Find sections about simulated annealing in articles that
    mention genetic algorithms
  </description>
  <narrative>
    ...
  </narrative>
  <keywords>genetic, simulated annealing, optimization</keywords>
</inex_topic>

```

Fig 2.2 typical INEX CO and CAS queries

CAS queries can be interpreted in the following ways: SSCAS, SVCAS, VSCAS and VVCAS. The first letter indicates strict (S) or vague (V) interpretation of *where-to-*

look-for constraints and the second letter indicates strict (S) or vague (V) interpretation of *what-to-return* constraints.

The query collection for INEX 2004 consists of 40 CO and 34 CAS queries.

2.3 Relevance Assessments

Relevance assessment is one of the most important aspects of the INEX operation. These are manual assessments of every document element in the collection for relevance to every query in the collection. This is done by INEX participants at the start of each year as explained in [13]. The set of document elements found relevant to a query by human assessors form the *ideal* retrieval result for that query. This set of relevant document elements is then used to evaluate retrieval results from the IR systems.

Relevance in INEX is defined according to the following two dimensions:

- Exhaustivity (E) – describes the extent to which the document element discusses the topic of request
- Specificity (S) – describes the extent to which the document element focuses on the topic of request.

Exhaustivity and Specificity are measure on a 4-point scale as shown in Fig 2.3.

E=0 : Not exhaustive	S=0 : Not specific
E=1 : Marginally exhaustive	S=1 : Marginally specific
E=2 : Fairly exhaustive	S=2 : Fairly specific
E=3 : Highly exhaustive	S=3 : Highly specific

Fig 2.3 4-point scale to measure exhaustivity and specificity

2.4 Evaluation metrics

Various metrics have been proposed to measure the effectiveness of a retrieval strategy, but the most commonly used are recall and precision [12]. Recall tests the ability of a system to retrieve as many relevant documents as possible whereas precision refers to the ability to retrieve as few non-relevant documents as possible. Fig 2.4 shows how recall and precision are calculated for a set of retrieved documents.

$$\text{Recall} = \frac{\text{Number relevant documents retrieved}}{\text{Total relevant documents in the collection}}$$
$$\text{Precision} = \frac{\text{Number relevant documents retrieved}}{\text{Total documents retrieved}}$$

Fig 2.4 recall and precision measures

Since the notion of relevance is divided into two components in INEX, we need a way of combining these two measures into a single value for relevance. This is done using the quantization methods. Various quantization methods [9] have been used that evaluate retrieval systems with respect to their capability of retrieving highly exhaustive and/or highly specific document elements.

2.5 Smart

Smart 13.0 [11] is our basic retrieval system. It provides us with the capability to index a document collection and retrieve highly correlating documents. An advantage of using Smart is that it uses Vector Space Model (VSM) and supports the Extended Vector Space

Model. Section 3.2 discusses how the Extended VSM can be used to represent a structured document.

2.6 Approaches to Flexible Retrieval

Many approaches to XML retrieval [1] provide insight into the current research in this area. Grabs and Schek [4] extend the XML query language XPath with IR functionality to introduce a new language called XPathIR. It lets users specify the retrieval model to be used and the minimum Retrieval Status Value (RSV) of an element for it to be considered relevant for the query.

In [4], Flexible XML retrieval is implemented by first identifying the basic indexing nodes, either automatically, in which case all leaf nodes are considered separate entities, or by having an administrator decide these nodes. Once the indexing nodes have been identified, they are represented and retrieved using the Vector Space Model.

INEX CAS queries which contain only one structural constraint are handled as single-category retrieval. Consider, for example, `//ack[about(., "research funded america")]`. In such queries, the granularity of retrieval is at the level of the basic indexing nodes and only a basic vector retrieval is required. CAS queries with multiple structural constraints are handled as multi-category retrieval. Consider `//(abs|kwd)[about(., "genetic algorithm")]`. The granularity of retrieval for such queries is given by all elements that satisfy the constraints (e.g., *abs* and *kwd* in this case). The statistic of inverted frequency is first modified by merging statistics for each of the categories that satisfy the constraints and then a conventional vector space retrieval is carried out using these modified statistics.

INEX CO queries are treated as nested retrieval, where the granularity of retrieval could be at any level in the document tree since no constraints are imposed on what element to retrieve. It essentially is a weighted sum of constrained single-category retrieval results, where the constraints are such that an element only contributes towards the score of an ancestor element.

Gövert, et. al., [5] present an implementation of the XIRQL query language [6] called HyREX. To process the INEX CO topics, HyREX implements a strategy by which the document element that satisfies the information need in the most specific way is retrieved. This is done using the concept of index nodes. Standard term weighting formulas are used to weight the index nodes and these weights are then propagated up the document tree to calculate term weights for inner nodes. Term weights are down-weighted as they propagate up the tree so as to avoid giving highest weights to most general elements. However, the amount by which the weights are reduced is collection dependent and needs to be calculated empirically.

The INEX CAS topics are handled by first converting them into XIRQL using the concept of data types and vague search predicates and then these XIRQL queries are processed using HyREX.

Another approach, as suggested by Mass, et. al., [7] is to maintain separate indices for elements at different granularity levels. They have separate indices for paragraphs, sub-sections, sections and articles. A query is evaluated against each of these indices and their results are merged to get a ranked list of elements. This solves the problem of distorted statistics for nested elements, but introduces a deficiency since not all elements at all granularity levels are indexed. To compensate for this deficiency, the score of a

retrieved element is scaled by the score of its containing article. This approach has the disadvantage that each query needs to be evaluated against multiple indices before we get a ranked list of relevant elements for the query.

Sigurbjornsson, et. al., [8] present another approach in which two separate indices are maintained – one containing only articles and one containing elements at all granularity levels (e.g., paragraphs, sub-sections, sections, articles, etc.). A language-based modeling approach that uses a linear interpolation of three language models – one for the element, one for its containing article and one for the entire collection is used for retrieval. The three language models are estimated using statistics from the two indices. An element index is used for estimating the element model and article index is used for estimating the article and document models. They also incorporate the document length in their model by using it in the estimation of prior probabilities.

Evaluation results for official INEX 2004 submissions show that the approaches taken by [7] and [8] do considerably better than other approaches for the CO task. However, both these approaches involve maintaining multiple indices and evaluating queries against these indices. Our approach is more similar to that taken by [5] but uses as few collection-dependent parameters as possible and the actual correlation values for all document elements rather than estimated values. We hope to achieve the performance of a multi-index approach by using a single index of leaf nodes and going bottom-up to populate the document tree.

3. The Flexible Retrieval System

3.1 Flexible retrieval

Flexible retrieval is the task of retrieving the most specific document element that satisfies the query's information need [1]. As described in Section 2.3, a highly specific document element focuses entirely on the information sought in the query and discusses nothing else. On the other hand, a highly exhaustive element discusses everything that the query needs and may or may not have additional information. A flexible retrieval system, therefore, needs to consider not only the exhaustivity of a document element but also its specificity.

A flexible retrieval system can retrieve portions of the document and hence must scan the document collection at different levels of granularity and assign scores to document elements. This score should reflect the exhaustivity and specificity of that element for it to be useful in comparing elements at different levels in the document tree.

The aim of flexible retrieval, as explained in [12], is to reduce the time a user spends browsing through search results looking for the information he needs. Traditional IR systems retrieve entire documents, forcing the user to look inside these documents and find content that answers his information need. A flexible retrieval system seeks to rank at the top those elements that are not only relevant to the query but are also completely focused on the query, thereby reducing the time the user has to put in to find what he is looking for. The following sections in this chapter describe the design and implementation of our flexible retrieval system.

3.2 Overview of the system

Our approach to flexible retrieval is to index only the leaf nodes from the document trees. We have manually identified the set of elements to be treated as leaf nodes and indexed them. For the INEX document collection, these are the paragraphs (p,p1,p2,p3,ip1,ip2,ip3,ip4,ip5,ilrj), lists (la,lb,lc,ld,le,l1,l2,l3,l4,l5,l6,l7,l8,l9,list), figure captions (fgc), tables (tbl), section titles (st) and abstracts (abs). These are the content-bearing elements that partition the document into mutually exclusive parts. Some of these elements are not leaf nodes according to the DTD (e.g., tables and lists) but we treat them as leaf nodes because all their child nodes are too small to be meaningful units of retrieval.

Smart, based on the Vector Space Model, represents indexed elements as a vector of stemmed and weighted terms from those elements. A document collection with d leaf nodes is thus represented by d vectors, each having n_i terms where n_i is the number of unique terms in the i^{th} leaf node.

The Extended Vector Space Model, as described in [10], can be used to produce a vectorized representation of structured documents. As shown in Fig 3.1, an extended vector consists of sub-vectors, where each sub-vector represents content from different parts of the document. Our index of leaf nodes consists of extended vectors with eight sub-vectors. These sub-vectors represent content from the eight subjective parts of the document – article title (atl), abstract (abs), keywords (kwd), editor’s introduction (edintro), body (bdy), bibliographic title (bibl_ti), bibliographic article title (bibl_atl) and acknowledgements (ack). The queries are indexed in a similar way.

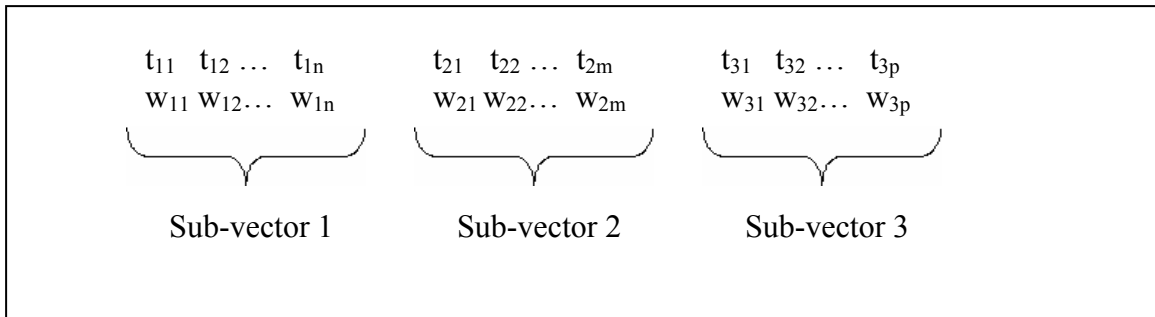


Fig 3.1 an extended vector representation of a document

An initial Smart retrieval run is done on this extended vector index of leaf nodes and a set of *highly correlating* leaf nodes (containing query terms) is retrieved for each query. All documents containing a retrieved leaf node are then populated in a bottom-up fashion. Vector representations of all inner nodes in the tree are generated from the vector representation of the leaf nodes, and a score based on the correlation of these vectors with the query is assigned to all the nodes in the tree. This score is then used to produce a ranked list of document elements in response to the query. Fig 3.2 gives an overview of the system.

3.3 Design of the Flexible Retrieval System

The flexible retrieval part of the system comes into the picture after the initial Smart retrieval run. It is responsible for populating a document tree, given the vector representation of its leaf nodes. A document tree is built only for those documents that have at least one leaf node retrieved in the initial retrieval.

1. Parse out the leaf nodes from the original XML documents
2. Index the leaf nodes and queries with Smart
3. Run an initial Smart retrieval to get highly correlating leaf nodes
4. For each document that contains a retrieved leaf node
 - a. Get its document schema
 - b. Generate vector representation for inner nodes
 - c. Calculate score for every node in the tree
5. Return a ranked list of elements from all these documents

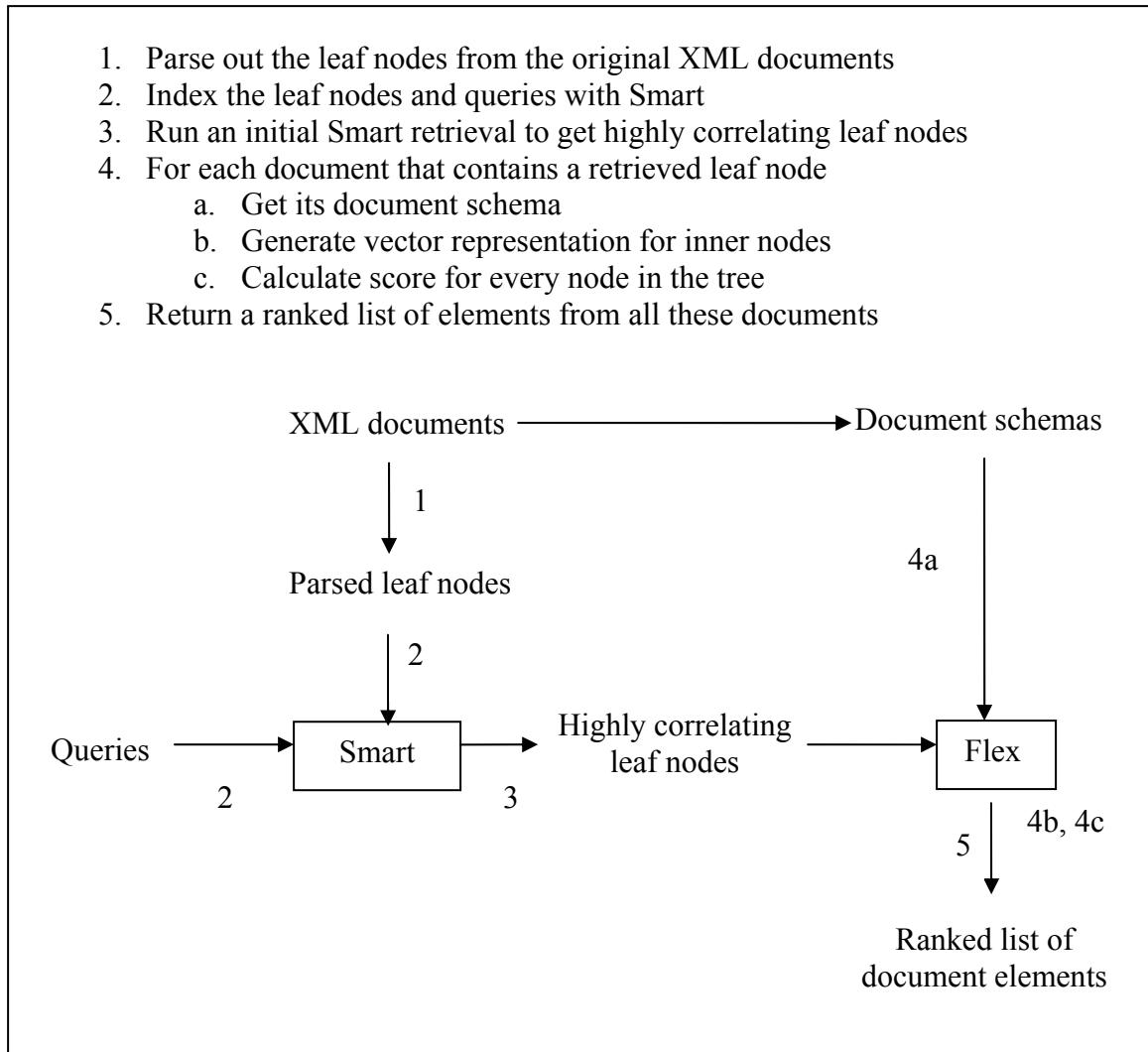


Fig 3.2 an overview of the Flexible Retrieval System

The index

Our Smart index of the leaf nodes uses the *nnn* weighting scheme to weight terms in the extended vector. In this weighting scheme, the weight assigned to a term is equal to its term frequency in the document element.

The representation for inner nodes

The vector representation for an inner node in the document tree is generated by merging the vectors of its child nodes. The vector for an inner node, therefore, contains terms from

all its child node vectors and the weight of each unique term is that of the term in the child node vector. If a term occurs in multiple child nodes, we simply sum its weight in all child nodes to get its weight in the parent node vector. (We can do this since term weights in vectors of the leaf nodes represent term frequency.) The inner node vector thus provides the same representation of its content that leaf node vectors provide for the leaf nodes. Fig 3.3 shows a simple example of how this works.

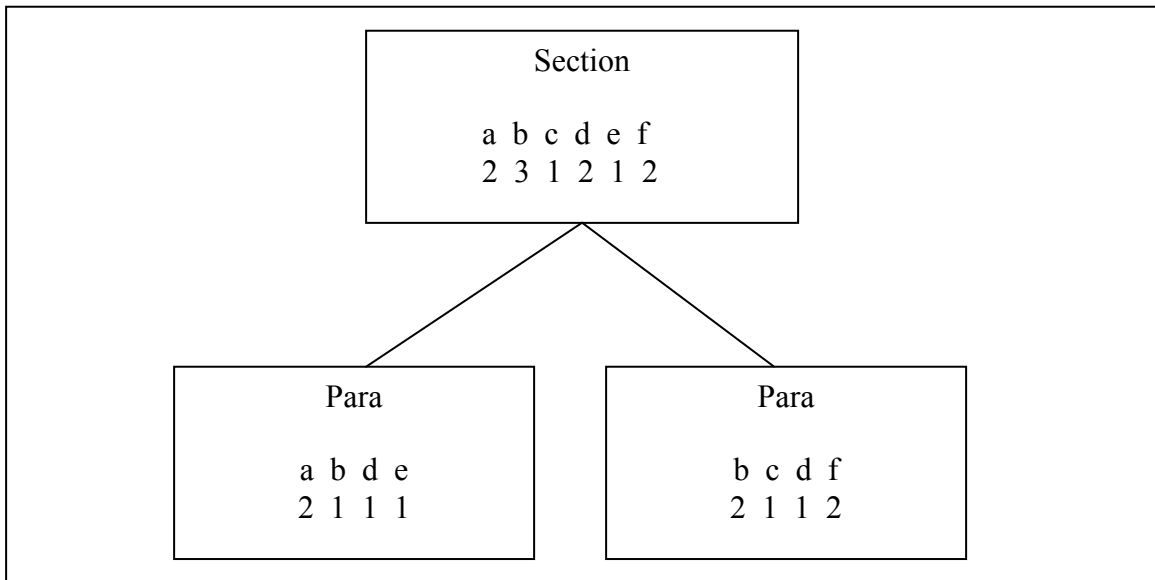


Fig 3.3 merging *nnn* vectors to generate vectors for inner nodes

The weighting scheme

The *nnn* weighting scheme used in the vector representation of document elements provides a very raw representation of their content. It takes only the occurrence frequency into account without considering the length of the element itself. Hence, it cannot be used to calculate a score for the element. We need a weighting scheme that takes the element length into consideration and normalizes term weights based on it.

Failure to consider length and normalization issues would produce scores that are biased towards larger elements. This is due to two reasons – larger elements have more

terms and larger elements have terms with higher term frequency. As a result, the probability of retrieval is more than the probability of relevance for larger elements and vice-versa for smaller elements when we use inner product as the similarity measure.

As explained in [14], the probability of retrieval of a document is inversely proportional to the value of the normalization factor for it. Therefore, we need to increase the normalization factor for larger elements and decrease it for smaller ones. This is done using the constants *slope* and *pivot*. Pivot represents the document length for which the probability of relevance is equal to the probability of retrieval. The normalization factor is now *pivoted* at pivot and *tilted* so that documents on one side of pivot get larger normalization factors and those on the other side get smaller normalization factors. The amount of *tilting* required is represented by *slope*. Using this pivoted normalization, we hope to reduce the difference between the probability of relevance and the probability of retrieval for all documents. Fig 3.4 illustrates this idea.

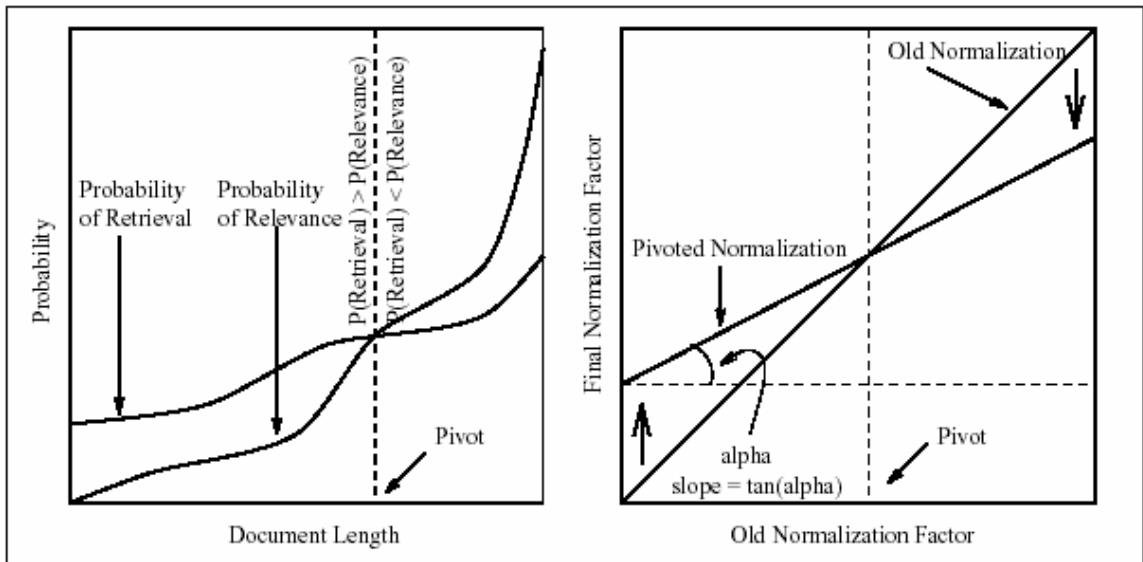


Fig 3.4 pivoted normalization (from [14])

To achieve useful term weights for inner nodes, the *nnn* weights on element vectors are converted to *Lnu* weights using the formula shown in Fig 3.5.

$$\frac{\frac{1 + \log(\text{tf})}{1 + \log(\text{average tf})}}{(1 - \text{slope}) + \text{slope} * (\# \text{ unique terms}) / \text{pivot}}$$

where, tf is term frequency (as in the *nnn* vector)
average tf is the average term frequency of all terms in this vector
unique terms is the number of distinct terms in this vector
slope & pivot are empirically determined constants.

Fig 3.5 *Lnu* element term weighting formula (from [14])

Slope and pivot are empirically determined constants that need to be calculated at every level in the document tree since the length characteristics of elements is different at each level. We have used different slope and pivot values at four levels in the document tree – paragraphs, sub-sections, sections and articles. These values for the INEX 2004 document collection are listed in Table 3.1.

	Slope	Pivot
Paragraph	0.06	24
Sub-section	0.01	92
Section	0.10	154
Article	0.03	538

Table 3.1 slope and pivot values at different levels in document tree

The *Lnu* weighting scheme thus produces term weights that are normalized according to the element length and reduces the undue advantage that longer elements have over shorter ones. Another advantage of the *Lnu* weighting scheme is that it does

not require a global statistic, such as the inverse document frequency. Hence, we can locally calculate *Lnu* term weights for an element vector, given only its *nnn* term weights and the values of slope and pivot.

The query vectors are weighted using the *ltu* weighting scheme. Fig 3.6 shows the formula for *ltu* weights.

$\frac{(1 + \log(\text{tf})) * \log(N/n_k)}{(1 - \text{slope}) + \text{slope} * (\# \text{ unique terms}) / \text{pivot}}$
<p>where <i>tf</i> is the term frequency <i>N</i> is the collection size <i>n_k</i> is the number of documents that contain this term <i>slope</i> & <i>pivot</i> are empirically determined constants <i># unique terms</i> is the number of distinct terms in this vector</p>

Fig 3.6 *ltu* query term weighting formula

The score

A score is assigned to each document element based on its *Lnu* vector representation and its correlation with the query. We use the measure of inner product between the element vector and the query vector to calculate this score.

A final rank-ordered list of document elements is produced as follows. Each element vector is expanded into its extended vector representation (i.e., sub-vectors) consisting of the eight subjective sub-vectors. The final score of an element is obtained by first calculating the inner product between corresponding sub-vectors and then combining these values linearly to get a single value. Fig 3.7 gives an example.

Element vector	a b c d e a b x x y z
	2 1 3 1 1 1 2 1 1 1 1
Query vector	a b x a b x a b x
	1 1 2 1 1 2 1 1 2
Inner product	2*1 + 1*1 1*1 + 2*1 + 1*2 1*2
	(sub-vector1) (sub-vector2) (sub-vector3)
Sub-vector weights	0.5 0.3 0.2
Final score =	0.5*(2*1 + 1*1) + 0.3(1*1 + 2*1 + 1*2) + 0.3(1*2)

Fig 3.7 calculating score for a document element

Sub-vector weights are used to assign relative importance to the different parts of a document. Table 3.2 shows the sub-vector weights that we have used on the eight sub-vectors in our index. Chapter 4 describes the experiments conducted to arrive at this set of weights for the sub-vectors.

Sub-vector	weight
bdy	3
atl	2
abs	1
edintro	0.5
kwd	0.5
ack	0
bibl_atl	0
bibl_ti	0

Table 3.2 sub-vector weights used for calculating final score of a document element

This score accounts for both exhaustivity and specificity of the document element. This is due to the *Lnu* weighting scheme used on the element vectors. Referring to Fig 3.5, we can see that a term occurring with equal frequency in a long and a short vector would get a smaller *Lnu* weight in the long vector than in the short vector. Thus, based on

our assumption that shorter elements are more specific than longer ones, we can say that *Lnu* term weights are a representative of the element's specificity.

The inner product is a measure of the element's exhaustivity, but since term weights used to calculate the inner product reflect specificity, the overall score is a measure of both and thus can be used to compare elements at different granularity levels. Fig 3.8 show how two element vectors of different lengths, having equal frequencies of query terms, get different scores due to *Lnu* weighting scheme.

<u><i>nnn</i> weights</u>	<u>avg tf</u>	<u><i>Lnu</i> weights</u>	<u>query</u>	<u>score</u>
a b c d e f 2 3 3 1 4 3	2.67	a b c d e f 0.68 0.84 x x x x	a b 1 1	1.52
a b c 2 3 3	2.67	a b c 0.97 1.19 x	a b 1 1	2.16

Lnu weights calculated assuming slope = 0.5 and pivot = 4

Fig 3.8 calculation of score for two vectors of different lengths

The Flex module

The Flex module implements functionality for maintaining a document schema in memory, reading *nnn* vectors for its leaf nodes from Smart index, merging them to generate *nnn* vectors for the inner nodes, converting all vectors to *Lnu* vectors and calculating a score for each of the document elements.

The module is divided primarily into two C++ classes – Flex and DocTree. The FlexDriver program is used to read document schemas from text files into memory and to invoke Flex class's methods on it to populate the document tree. DocTree is used to represent a single document tree. A utility class called Properties is used to hold values of

various parameters that can be used to fine tune the module. These parameters are discussed in Chapter 4.

Relevance Feedback

Issues with using relevance feedback in a flexible retrieval system are discussed in [15].

3.4 Implementation of the Flexible Retrieval System

Parsing XML documents

The INEX document collection consists of XML documents. We need to parse these XML documents and extract the content of nodes identified as leaf nodes. We do this using the Simple API for XML (SAX) which is part of the Java API for XML Processing (JAXP).

This parsing of XML documents produces new, smaller documents, each having the content of only one leaf node from the original XML document. In addition, these new documents also carry the eight subjective sub-vectors identified in Section 3.2.

Another outcome of parsing the XML documents is a representation of the document structure which can be stored in text files and later used by Flex. It basically is a preorder traversal of the document tree, with additional information about the number of child and sibling nodes at each node. Fig 3.9 shows this representation for a sample document tree.

Indexing the leaf nodes

The parsed set of documents, representing leaf nodes of the original XML documents, is indexed using Smart. Smart uses the extended vector representation to store each component of the document as a separate sub-vector in the extended vector.

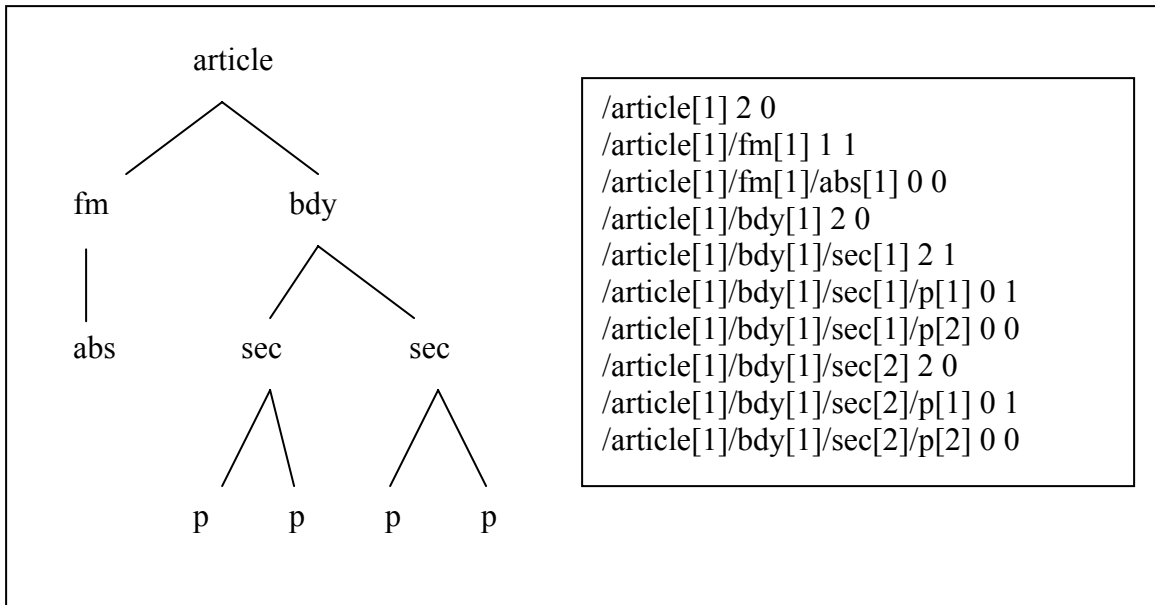


Fig 3.9 document structure representation

Initial retrieval run

An initial Smart retrieval run is done on this extended vector index of leaf nodes and a set of highly correlating leaf nodes is retrieved for each query. This set of retrieved elements gives us an idea of which documents contain relevant content for the query and the process of flexible retrieval is then applied to these documents.

'Flex'ible retrieval

Once the initial Smart retrieval run is done, we read the document trees of those documents in memory that contain at least one retrieved leaf node. The vector representation of the corresponding leaf nodes is then read from Smart index and stored along with the node in the document tree. It is important to note that we read the vectors of all the leaf nodes and not just of those retrieved in the initial run. We now have a set of document trees along with the vector representation of all their leaf nodes.

We then proceed to generate a vector representation of the inner nodes in the document tree. Note at this point the vectors use *nnn* weighting scheme. Hence, we can add the term weights of those terms that occur in more than one child node and keep the term weights of those that occur in only one child node to generate a *nnn* vector for the parent node. This vector is stored along with the node in the document tree. The immediate parent of each leaf node is populated in this way.

nnn vectors are first converted to *Lnu* vectors according to the formula in Fig 3.5 and then inner product is used to calculate the correlation of the element vector with the query vector. This gives us a score, which as discussed in the previous section, may in a sense be considered to represent both relevance and specificity of the element.

We repeat this process of generating *Lnu* vectors and calculating a score for every element in the document tree, by moving up the tree. (This is done for every document that has a highly correlating leaf node according to the initial retrieval run.) The end result of flexible retrieval is a rank-ordered list of document elements.

The implementation of the Flexible Retrieval System is divided primarily into two C++ classes – Flex and DocTree. Flex implements methods for generating *nnn* vectors for inner nodes, converting them to *Lnu* vectors and calculating the score for all nodes in the document tree. It also interfaces with Smart through the smart_interface library to read *nnn* vectors for leaf nodes from the index. The smart_interface library is written in C and provides methods for reading Smart vectors.

FlexDriver is the driver program that is responsible for reading the document structure from text files into memory and invoking Flex class's populate_tree method on this tree along with all the appropriate parameters. The various parameters used and their

possible values are discussed in Chapter 4. A utility class called Properties is used to hold the values of these parameters.

DocTree class represents a single document tree. FlexDriver creates an instance of this class and passes it on to Flex. Flex works with only one DocTree instance at a time. The nodes in a DocTree object can store its xpath, *nnn* vector, score and the number of child nodes.

FlexDriver maintains a list of elements from those documents that have been processed by Flex. Once all these documents have been processed, it writes the list of elements, sorted by their score, to the output file. Additional scripts are required to convert the output of the Flexible retrieval system to INEX format, which can then be submitted for evaluation.

Appendix A gives a detailed view of the classes and their interaction in the form of UML diagrams.

4. Experiments and Results

We have conducted several experiments to fine tune the parameters used by our system as well as to evaluate its performance. Retrieval effectiveness of our Flexible Retrieval System was tested using the INEX evaluation metrics which is implemented in `inex_eval` as well as available online. We have used the document collection and queries made available under INEX 2004 for our experiments since we had access to the relevance assessments for that collection.

4.1 Experiments

Some of the parameters whose values needed to be calculated empirically are the sub-vector weights, the values of slope and pivot, minimum size of an element for it to be considered by the Flexible Retrieval System and the number of leaf nodes to be fed to Flex. Following is a description of the experiments conducted to calculate their values. 1000 leaf nodes were fed to Flex and a ranked list of 1500 elements was produced by it in all cases, unless otherwise noted.

Sub-vector weights

The Extended Vector Representation of an element in our Smart index contains content from different parts of the element. However, the text from the leaf nodes is only represented by the *bdy* sub-vector while the remaining sub-vectors have content that is common to all leaf nodes from that document, and is in a way, representative of the entire document (e.g., the article title, abstract, keywords, etc.). Therefore, ideally we might like

to retrieve using only on the *bdy* sub-vector. But our experiments with different sub-vector weights show that we can improve our results if we include some of the other sub-vectors in our retrieval run.

Table 4.1 show the mean precision value as calculated by *inex_eval* for different sub-vector weights. GR represents the generalized recall.

	bdy	atl	abs	edintro	kwd	ack	bibl_atl	bibl_ti	avgRP	GR
equal weights	1	1	1	1	1	1	1	1	0.04	
only bdy	1	0	0	0	0	0	0	0	0.04	
only atl	0	1	0	0	0	0	0	0	0.02	
only abs	0	0	1	0	0	0	0	0	0.01	
only edintro	0	0	0	1	0	0	0	0	0	
only kwd	0	0	0	0	1	0	0	0	0	
only ack	0	0	0	0	0	1	0	0	0	
only bibl_atl	0	0	0	0	0	0	1	0	0.01	
only bibl_ti	0	0	0	0	0	0	0	1	0.01	
no bdy	0	1	1	1	1	1	1	1	0.01	
no atl	1	0	1	1	1	1	1	1	0.04	
no abs	1	1	0	1	1	1	1	1	0.05	
no edintro	1	1	1	0	1	1	1	1	0.04	
no kwd	1	1	1	1	0	1	1	1	0.05	
no ack	1	1	1	1	1	0	1	1	0.04	
no bibl_atl	1	1	1	1	1	1	0	1	0.06	
no bibl_ti	1	1	1	1	1	1	1	0	0.05	
weights 1	2	2	0	0	0	0	0	0	0.06	39.50
weights 2	2	2	0.5	0	0	0	0	0	0.07	41.03
weights 3	2	2	0.5	0.5	0	0	0	0	0.07	40.83
weights 4	2	2	1	0.5	0	0	0	0	0.07	41.03
weights 5	2	2	1	0.5	0.5	0	0	0	0.07	40.90
weights 6	3	2	1	0.5	0.5	0	0	0	0.07	42.22
weights 7	3	2	0.5	0.5	0.5	0	0	0	0.07	41.59
weights 8	3	3	1	0.5	0.5	0	0	0	0.07	41.14

Table 4.1 effect of sub-vector weights on mean precision

As can be seen from the table, the sub-vector weights of $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ give us the maximum mean precision of 0.07 and generalized recall of 42.44. We have used these weights for calculating the score for each element in Flex.

We can also use these weights for an extended vector retrieval of leaf nodes.

Table 4.2 compares the performance of the Flexible Retrieval System in these cases.

Leaf node retrieval	Flexible retrieval	avgRP
only <i>bdy</i>	only <i>bdy</i>	0.0607
only <i>bdy</i>	extended vector	0.1002
extended vector	extended vector	0.1007

Table 4.2 effect of extended vectors on mean precision

As can be seen from Table 4.2, we get similar results when we use extended vector retrieval for flexible retrieval with either only *bdy* or extended vector retrieval of leaf nodes. However, there is considerable difference in the number of documents that Flex works on in the two cases. As already discussed, Flex only works with those documents that have at least one retrieved leaf node.

An initial retrieval of leaf nodes considering only *bdy* sub-vector retrieves leaf nodes from, on average, 518.59 distinct documents. Thus, Flex works with 518.59 documents for each query. On the other hand, using extended vectors for the initial retrieval run retrieves leaf nodes from only 274.18 distinct documents. About 91% of these documents were also used in the first case.

Based on these statistics, we can say that using extended vectors for initial retrieval produces a more focused set of leaf nodes. In other words, we have leaf nodes

from a smaller and probably relevant set of documents. This reduces the number of documents that Flex needs to populate, without adversely affecting the performance.

Slope and Pivot

Experiments were conducted to empirically determine the values of slope and pivot that work best for elements at different granularity levels. These values are shown in Table 3.1. A consistent value of 0.2 for slope and 110 for pivot were used for a separate index consisting of every element in the collection, irrespective of its granularity level. We call this the all-element index.

Minimum size for relevant elements

Some of the elements in our index of leaf nodes are very small and can never satisfy a query's information need. Hence, we do not want such elements to be returned in the ranked list of elements produced by Flex. However, we want to keep them in our index because they provide valuable information about the relevance of their parent nodes e.g., a highly correlating section title on its own may not be a good element to retrieve, but it contributes to the relevance of its parent section element.

Hence, Flex needs a parameter for the minimum size of elements so that it can ignore all elements smaller than this size. Tables 4.3a-b shows the results of the set of experiments carried out to determine this size. These experiments were conducted using only the *bdy* sub-vector as well as with the extended vector using sub-vector weights found best in the previous section. Both inner product and cosine similarity measures were used to compare their effectiveness in retrieving relevant document elements.

As can be seen from the table, there is not a significant improvement in mean precision after ignoring elements with less than 10 or 20 terms. In fact, the performance

goes down if we ignore elements with 35 or less terms. Hence, we do not ignore elements based on their size and consider all elements to be retrievable if they get a good score.

Minimum size	Slope/Pivot			
	Separate	Consistent	Separate	Consistent
	Inner product		Cosine	
size 0	0.1007	0.0700	0.0472	0.0472
size 10	0.0999	0.0692	0.0468	0.0468
size 20	0.0990	0.0686	0.0516	0.0515
size 35	0.0930	0.0653	0.0483	0.0482
size 50	0.0839	0.0565	0.0474	0.0565

Table 4.3a effect of minimum size of elements on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights

Minimum size	Slope/Pivot			
	Separate	Consistent	Separate	Consistent
	Inner product		Cosine	
size 0	0.0607	0.0506	0.0336	0.0335
size 10	0.0605	0.0500	0.0332	0.0331
size 20	0.0602	0.0492	0.0370	0.0369
size 35	0.0578	0.0462	0.0365	0.0365
Size 50	0.0538	0.0404	0.0368	0.0369

Table 4.3b effect of minimum size of elements on mean precision with only *bdy* sub-vector

Number of leaf nodes input to Flex

In all the experiments discussed so far, only 1000 leaf nodes were input to Flex from the initial retrieval run. As discussed in Chapter 3, Flex only works with those documents that have at least one leaf node retrieved in the initial run. Thus, the number of leaf nodes fed to Flex restricts the number of documents that it works on.

It is possible that feeding a set of leaf nodes with higher mean precision to Flex would produce a better rank-ordered list of document elements. One way of increasing the mean precision of our initial retrieval of leaf nodes is to consider only those leaf nodes that come from *good* documents. We know from [17] that a Smart retrieval on an only-document index retrieves relevant documents. We could use this result to filter our initial retrieval of leaf nodes. However, this requires maintaining two indices – one at the document level and one at leaf nodes level. More details of such processing on the list of retrieved leaf nodes are discussed in [16].

Tables 4.4a – 4.4d shows the effect of the number of input leaf nodes on the mean precision of the final ranked list of elements produced by Flex.

Input number	Slope/Pivot			
	Separate		Consistent	
	Smart	Processed	Smart	Processed
1000	0.1007	0.1000	0.0700	0.0704
1500	0.0997	0.0992	0.0699	0.0705
2000	0.0992	0.0986	0.0699	0.0704
5000	0.0979	-	0.0699	-

Table 4.4a effect of input number of leaf nodes on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights and inner product as similarity measure

Input number	Slope/Pivot			
	Separate		Consistent	
	Smart	Processed	Smart	Processed
1000	0.0472	0.0480	0.0472	0.0479
1500	0.0471	0.0479	0.0471	0.0479
2000	0.0471	0.0478	0.0470	0.0478
5000	0.0470	-	0.0470	-

Table 4.4b effect of input number of leaf nodes on mean precision with extended vectors using $\langle 3,2,1,0.5,0.5,0,0,0 \rangle$ sub-vector weights and cosine as similarity measure

Input number	Slope/Pivot			
	Separate		Consistent	
	Smart	Processed	Smart	Processed
1000	0.0607	0.0596	0.0506	0.0509
1500	0.0600	0.0593	0.0506	0.0508
2000	0.0596	0.0586	0.0505	0.0507
5000	0.0587	-	0.0505	-

Table 4.4c effect of input number of leaf nodes on mean precision with only *bdy* sub-vector and inner product as similarity measure

Input number	Slope/Pivot			
	Separate		Consistent	
	Smart	Processed	Smart	Processed
1000	0.0336	0.0337	0.0335	0.0336
1500	0.0326	0.0329	0.0326	0.0328
2000	0.0322	0.0323	0.0321	0.0322
5000	0.0316	-	0.0315	-

Table 4.4d effect of input number of leaf nodes on mean precision with only *bdy* sub-vector and cosine as similarity measure

As can be seen from the tables, increasing the number of input leaf nodes to Flex always brings down the mean precision. This is probably because Flex is now working on a larger number of leaf nodes as well as documents and not all of these are highly correlated with the query.

On the other hand, using the set of leaf nodes that has been filtered as per the discussions in [16] provides for only a marginal improvement in mean precision. This is probably due to the fact, which can be verified by looking at the relevance assessments, that most of the highly correlating leaf nodes come from a few documents. Since Flex considers all elements of a document that has at least one highly correlating leaf node, all

these leaf nodes were already considered by Flex. Hence, pre-processing of the retrieved list of leaf nodes does not produce significant improvement in mean precision of the final ranked list of elements.

4.2 Results

It is clear from the experiments conducted that we get the best results from Flex when we input 1000 leaf nodes and do not filter elements based on their size. Also, results in Table 4.3a and Table 4.3b indicate that using inner product as the similarity measure produces a ranked list of elements that has higher mean precision than that produced by using the cosine similarity measure.

Using different values of slope and pivot for elements at different granularity levels contributes significantly towards improving the mean precision. This is especially true when we use inner product as the similarity measure, as is evident from Tables 4.4a and 4.4c.

It is also clear from the values in Table 4.1 that the sub-vector weights of $\langle 3, 2, 1, 0.5, 0.5, 0, 0, 0 \rangle$ provide significant improvement over using only *bdy* sub-vector. Hence, we use these weights for calculating the score of elements in Flex.

Now that we have values for all parameters that give us the best results from Flex, we can compare its performance to that of our previous system, as described in [12], and to our base run. Our base run consists of evaluating INEX queries against a Smart index of all elements from the document collection at all granularity levels. We call this the all-element index. It has a vector representation of every element in the collection, even if

their content overlaps. Table 4.5 shows the value of mean precision as calculated by `inex_eval` for these retrieval runs.

Retrieval run	avgRP	Description
Flexible retrieval (last year)	0.0507	Sum of exhaustivity and average of specificity propagated to parent node
All element index	0.0691	Slope=0.2, pivot=110 Extended vector retrieval with weights <3,2,1,0.5,0.5,0,0,0>
Flexible retrieval (current system)	0.1007	Slope and pivot as in Table 3.1 Extended vector retrieval with weights <3,2,1,0.5,0.5,0,0,0> 1000 leaf nodes input to Flex

Table 4.5 comparison of current system with previous version and the base run

As is clear from the values in Tale 4.5, our current approach, with the parameter values as determined in Section 4.1, is a significant improvement over previous versions as well as the base run.

4.3 Parameters for INEX 2005

Although these parameters were calculated using data for INEX 2004, they can still be used for INEX 2005. This is because the INEX 2005 document collection has more documents from the same domain. The document structure and the length characteristics of the collection do not change. However, a different set of topics may force us to reconsider our sub-vector weights.

5. Conclusions and Future Work

5.1 Conclusions

We have made significant improvement in the performance of our system as compared to that in [12]. This is primarily due to the fact that we now carry a vector representation of the elements along with their structure in the document tree. The score assigned to an element is based on its actual correlation with the query vector rather than an estimation of the correlation.

The *Lnu* weighting scheme used on the element vectors also contributes significantly to the improved performance. It takes element length into account and hence, a similarity score based on these weights may be considered to be a measure of both exhaustivity and specificity. Since this weighting scheme does not require any global statistic, we can use it and work with only one document tree at a time. This is an important aspect of this weighting scheme which would become even more significant when the system is scaled to handle a larger document collection.

As discussed in Chapter 4, the present system not only out-performs its previous version, but is also better than the base case which uses an all-element index. Hence, we have been able to surpass the performance of an all-element index by using an index of the non-overlapping leaf nodes and taking a bottom-up approach to assigning scores to all document elements.

We have presented, in this thesis, an approach to Flexible Retrieval that neither requires indexing every document element in the collection nor requires evaluating queries against multiple indices.

5.2 Suggestions for future work

The current system has a few parameters which are collection dependent (e.g., the values of slope and pivot and the sub-vector weights). An important step in making this system collection-independent would be to either take a completely different approach and avoid using these parameters or automate the process of estimating their values as and when the collection changes.

The current version of Flexible Retrieval System produces a rank-ordered list of document elements based on a single score that we believe represents both exhaustivity and specificity of the elements. There is a need to experiment with approaches that calculate separate values of exhaustivity and specificity and use these values to obtain the final ranked list of elements. Using an explicit value of specificity might result in some improvement in the mean precision of the results.

Bibliography

- [1] Fuhr, N; Malik, S; Lalmas, M. Overview of the INitiative for the Evaluation of XML retrieval (INEX) 2003. *INEX 2003 Workshop Proceedings*, Schloss Dagstuhl, December 15-27, 2003.
- [2] Bapat, H. Adapting the Extended Vector Space Model for Structured XML Retrieval. MS Thesis, University of Minnesota, Duluth, 2003.
- [3] Apte, S. Using the Extended Vector Space Model for Content Oriented XML Retrieval. MS Thesis, University of Minnesota, Duluth, 2003.
- [4] Grabs, T; Schek, H. ETH Zurich at INEX: Flexible Information Retrieval from XML with PowerDB-XML. *Proceedings of the First Workshop of the INitiative for the evaluation of XML Retrieval (INEX)*, Schloss Dagstuhl, December 9-11, 2002.
- [5] Gövert, N; Abolhassani, M; Fuhr, N; Großjohann, K. Content-oriented XML retrieval with HyREX. *Proceedings of the First Workshop of the INitiative for the evaluation of XML Retrieval (INEX)*, Schloss Dagstuhl, December 9-11, 2002.
- [6] Fuhr, N; Großjohann, K. XIRQL: A Query Language for Information Retrieval in XML Documents. In: Croft, W.; Harper, D.; Kraft, D.; Zobel, J. (eds.): *Proceedings of the 24th Annual International Conference on Research and development in Information Retrieval*, 172–180. ACM, New York. 2001
- [7] Mass, Y; Mandelbrod M. Component ranking and Automatic Query Refinement for XML Retrieval. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [8] Sigurbjornsson, B; Kamps, J; Rijke, M. The University of Amsterdam at INEX 2004. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [9] Vries, A; Kazai G; Lalmas, M. Evaluation Metrics 2004. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [10] 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.
- [11] Salton, G., editor. *The SMART Retrieval System – Experiments in Automatic Document Retrieval*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

- [12] Mahajan, A. Flexible retrieval in a structured environment. MS Thesis, University of Minnesota Duluth, 2004.
- [13] Kazai, G; Lalmas, M; Piwowarski, B. INEX 2004 Relevance Assessment Guide. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [14] Singhal, A; Buckley, C; Mitra, M. Pivoted Document Length Normalization. *ACM SIGIR*, 1996.
- [15] Potnis, P. Relevance Feedback in a Flexible Environment. MS Thesis, University of Minnesota Duluth, 2005.
- [16] Doddapaneni, N. Effective Structured Query Processing. MS Thesis, University of Minnesota Duluth, 2005.
- [17] Crouch, C; Apte, S; Bapat, H. Adapting the Extended Vector Space Model for XML Retrieval. *INEX 2003 Workshop Proceedings*, Schloss Dagstuhl, December 15-27, 2003.

Appendix A. UML Diagrams

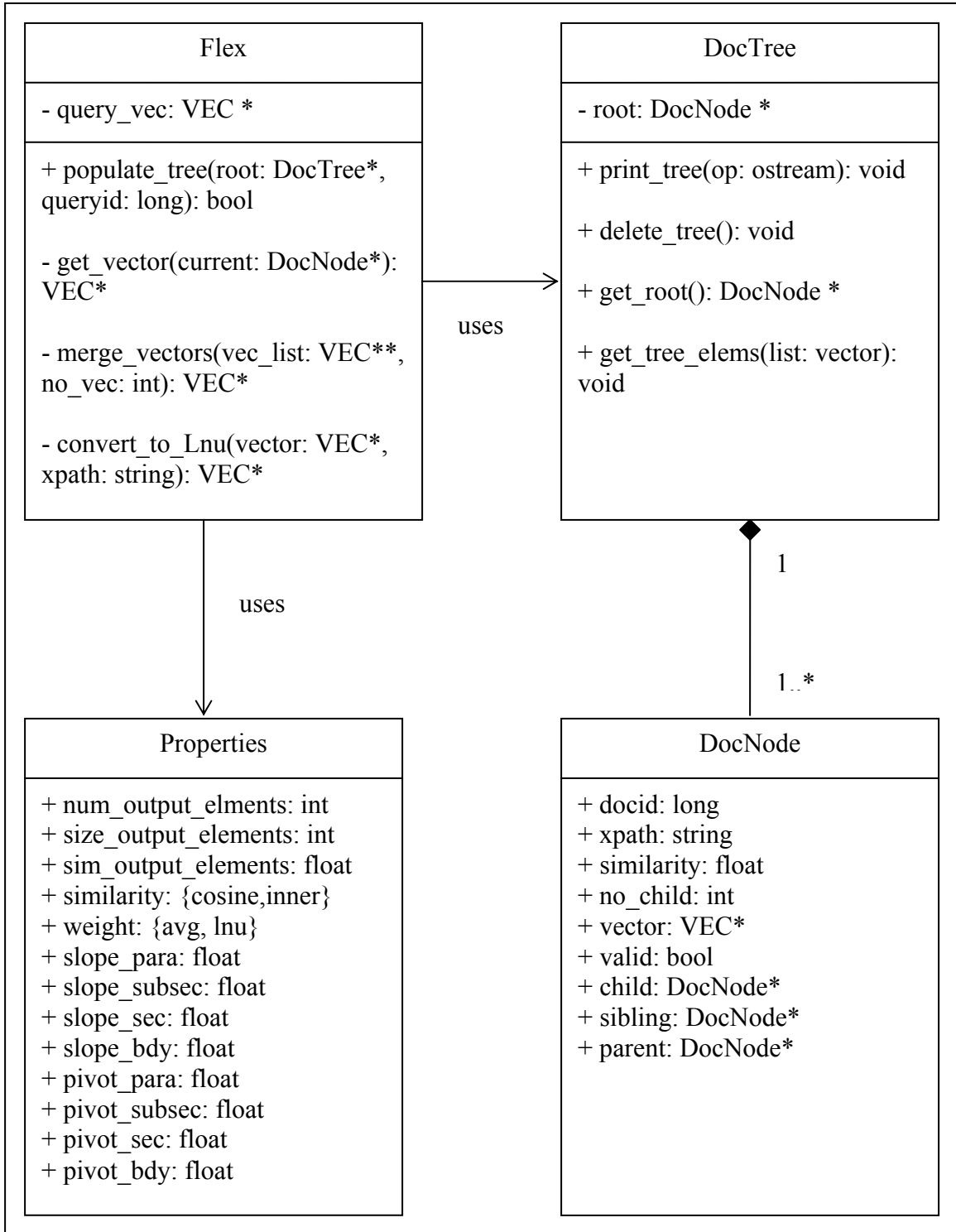


Fig A.1 class diagram for Flexible Retrieval System

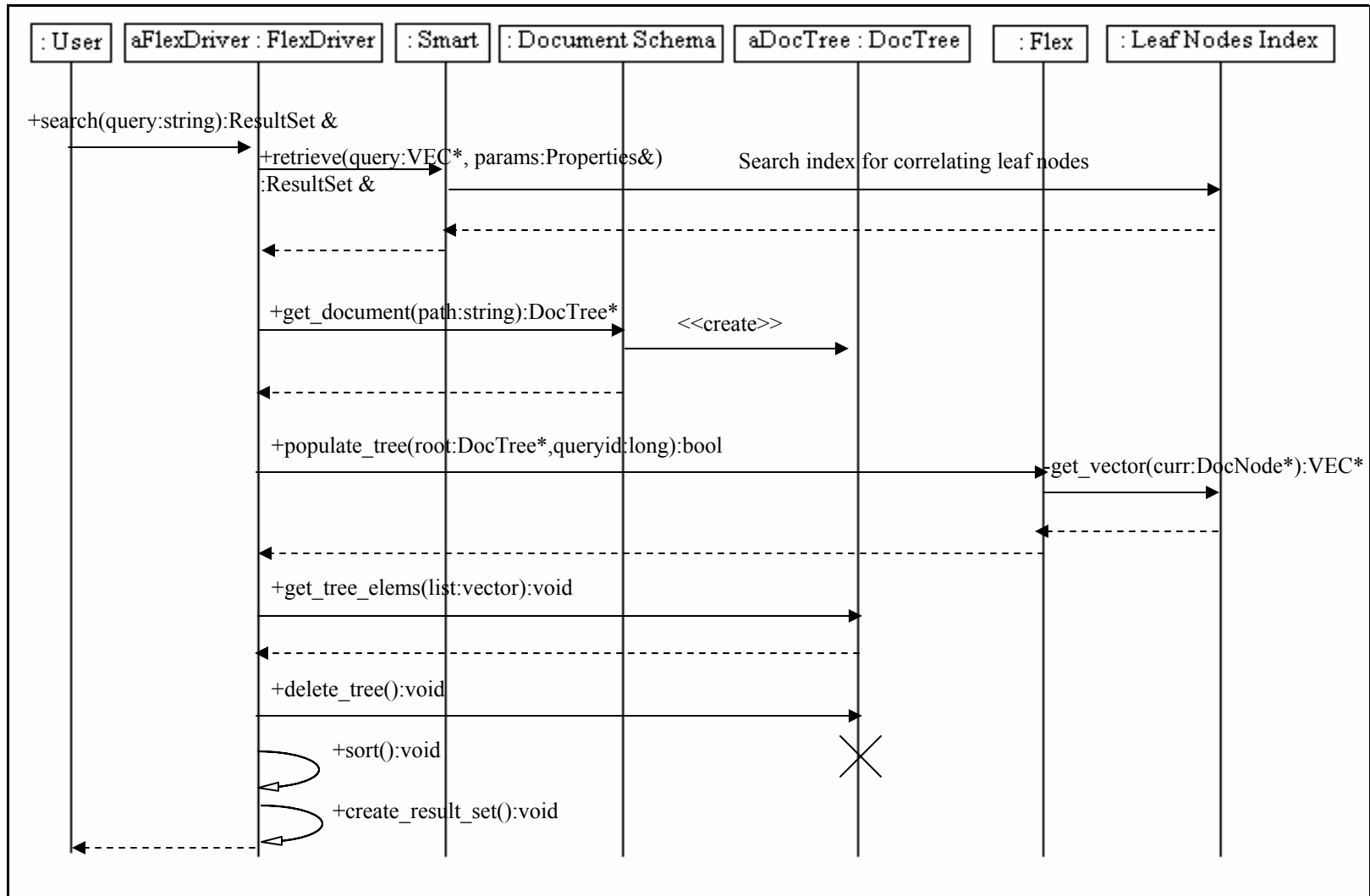


Fig A.2 sequence diagram for Flexible Retrieval System