

Query Processing in a Flexible Retrieval Environment

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

Satyanarayana Murthy Ganapathibhotla

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

July 2006

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

Satyanarayana Murthy Ganapathibhotla

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

Abstract

Whereas traditional information retrieval aims at retrieving entire documents, structured retrieval deals with the retrieval of elements from those documents. To retrieve these elements at a granular level, they must be separately indexed. Maintaining such an index is costly. We have proposed a method for the dynamic retrieval of elements which requires maintaining only an index of leaf nodes of the document tree. The document trees associated with highly correlating leaf nodes within the tree is Lnu- weighted and is correlated with the ltu-weighted query using inner product. This process of retrieving elements on the fly is known as flexible retrieval. [Khanna, 2005].

Our original approach to flexible retrieval used only an estimate of the query at each element level (i.e. the paragraph query, weighted using paragraph level statistics [Khanna, 2005]). In this thesis, a method to generate a properly weighted query (given the query at paragraph level) is described. We show that the dynamic element retrieval achieves the virtually same results as an equivalent retrieval against an all-element index. The INEX 2004 and 2005 test collections are utilized for these experiments. We also discuss the number of leaf nodes that needs to be supplied to our flexible retrieval system in order to achieve performance along with the time and space requirements for dynamic element retrieval.

Acknowledgements

I would take the opportunity to sincerely thank my thesis advisor, Dr Carolyn Crouch for her invaluable guidance and support. I would especially thank for the long hours she had spent and the immense patience she had with me along the two years.

I would also thank Dr Donald Crouch for providing me with constant encouragement and timely feedback during my work. I would also like to thank Dr Robert Mc Farland for providing me with subtle insights in vector spaces and providing me with a good mathematical background.

I would like to thank the department faculty, Prof. Steve Holtz for providing the archives of Smart documentation. It has been really useful to us.

I would also thank Vishal Bakshi for his help especially in the times of pressing need during debugging. I would also thank Vikram Malik, Nachiket Kamat, and Aditya Mone for their support during experimentation.

I would also take the opportunity to sincerely thank the staff at the Department of Computer Science at the University of Minnesota Duluth - Lori Lucia, Linda Meek and Jim Luttinen for their constant support.

Finally I would thank my family and friends for their encouraging support and cheering me up throughout.

1. Introduction

Information retrieval is the science of returning relevant documents or components of these documents in response to a user's information need. Hence, there must be some way of representing documents and queries for effective retrieval. This way is best achieved by a model. One among these models is Salton's Vector Space Model [1]. In this model, both the documents and queries are represented as vectors, and the similarity between a document and a query is computed using a measure such as cosine or inner product.

This model works as long as a document is viewed as a continuous text with equally important information throughout. This may not be the case. Consider, for example, a journal article. The body of the document may prove to be the chief contributor to a user's information need, and other information, such as the title, abstract, keywords or bibliographic information, may prove adjunct to the body. The rationale might be as follows. A user usually concentrates on the body of the document, as it is more focused on the subject content. However, if he is in hurry, the user may simply read the abstract to judge the article's relevance. In such cases, a more suitable model might be the Extended Vector Space Model [2], in which a document vector is comprised of subjective subvectors (such as the body of the document) and objective subvectors (such as author surname). This model enables calculation of similarity among different portions of text. These portions, represented as subvectors, may also be called concept types (ctypes).

With the rapid growth in XML documents on the World Wide Web, people are

also interested in searching for information from these documents. XML enables a document to be structured, so a facility to retrieve parts of the document rather than the entire document itself is desirable. In order to promote the research with respect to XML elements, the INEX initiative was started in 2002 [3]. We at the University of Minnesota Duluth are participants in the INEX workshops. We use Smart [4] as our retrieval engine. Smart is based on the Vector Space Model.

The basic operations of Smart can be broken down into three stages: indexing, term weighting and retrieval. During indexing, a document is filtered using a stop list, stemmed and then converted to vector form. All the terms in the vector are then weighted using a suitable weighting scheme. During retrieval, the similarity between a document and a query is computed using a similarity measure. The details of these operations are described in Chapter 2.

Whereas traditional information retrieval aims at retrieving entire documents, structured retrieval deals with the retrieval of elements from those documents. To retrieve these elements at a granular level, they must be separately indexed. Maintaining a separate index for each element type or, alternatively, an all-element index composed of all elements of each type, is costly in terms of space. Hence we have proposed a method to dynamically retrieve elements by maintaining only an index of leaf nodes (i.e., paragraphs) [15]. We then build document trees from these paragraphs. Each element within the tree must be correlated with the query and ranked based on a similarity measure. This process of retrieving elements on the fly is known as *flexible retrieval*. [15].

This thesis describes the generation of properly weighted queries at all levels during flexible retrieval, given the query at the paragraph level. It also discusses the

number of leaf nodes that need to be supplied to our flexible retrieval system in order to achieve performance equivalent to that of the all-element index.

The details of flexible retrieval are described in Chapter 2. The experiments used to test our flexible retrieval system are described in Chapter 3, and the results are discussed in Chapter 4. Conclusions and suggestions for future research are included in Chapter 5.

2. Flexible Retrieval

We refer to the process of dynamically returning the most highly correlated elements from documents as *flexible retrieval* [15]. The importance of such a method lies in the fact that it promotes the retrieval of more focused elements – elements that relate to the information need. This avoids the user’s having to examine the entire document in order to get the relevant material. This is a more suitable methodology for XML documents, where every document is comprised of many distinguishable elements.

Others have also looked at the problem of element retrieval. The details of these methods can be found in [6], [7], [8] and [9]. Our approach to flexible retrieval started with [10] and [5]. We use two methods: one for flexible retrieval and the second to produce a baseline for evaluation purposes. A baseline is provided using the all-element index approach. In this method, we parse the documents to extract its elements and maintain an index for all such elements. The major disadvantage of this method is replication of data in the form of overlap of elements. It also occupies a lot of space on the disk.

First we consider the implementation of our flexible retrieval system, known as Flex. In this approach, *all* the basic textual units (mostly paragraphs but also including abstracts, section titles, figure captions, etc., which are also indexed as paragraphs) are considered as leaf nodes of a document tree. We index only these leaf nodes and retrieve them by using our search engine, Smart. The top n elements from this initial retrieval are fed to Flex, which then builds trees for all the documents with at least one leaf among

these top n retrieved nodes. During the process of tree building, the internal element nodes are constructed, term-weighted, and then correlated with the query at the element level using inner product. After all the trees have been built, the list of all elements is sorted based upon the similarity value and reported. Our original approach to flexible retrieval [5] used only an estimate of the query at each element level (i.e., the paragraph query, weighted using paragraph level statistics). In this thesis, a method to generate a properly weighted query to be used throughout (i.e., at each level of) the document tree (given the query at paragraph level) is described. We use *Lnu* weighting for element vectors and *ltu* weighting for query vectors.

In this chapter, the operation of our retrieval engine, Smart, is described along with a description of the *Lnu-ltu* term weighting methodology. The generation of element vectors and query vectors is covered later in the chapter.

2.1. Smart

Smart [4] is a search engine that was created at Cornell University under the supervision of Gerard Salton and developed by Chris Buckley and others. It is built on the concept of Vector Space Model, and it also supports Fox's Extended Vector Space Model [2]. We use Smart.13.0 as our basic search engine. The various operations of Smart are summarized in the three following sections.

2.1.1. Indexing

All the documents or elements are initially fed to Smart. Smart does an initial pre-parsing and the various subvectors or ctypes, (i.e., the different portions of the text) are

identified. It filters the documents based on a stop-list, stems the tokens, and assigns a unique identifier to each. (The token is also known as a term, a keyword or a concept.) It also assigns a unique identifier to each document. (This mapping between document identifier and the file location of the document is stored in the *textloc* file.)

Smart also creates dictionaries for each ctype, containing a mapping between a concept and its identifier for each ctype. It creates an inverted index, which contains the mapping between a concept and the set of documents (or elements) in which the concept occurs. The frequency of the concept in the document (or element) is stored in the inverted file. This is known as its term-frequency (*tf*). Collection frequency information is stored in the *collstats file* (one is produced for every ctype). Documents or elements are converted to vector form, with terms weighted using the term frequency information. (These are known as the *doc.nnn* vectors.)

2.1.2. Term Weighting

After the initial indexing phase of Smart, the vectors can be altered by choosing an appropriate weighting scheme. During indexing, the vectors are weighted using the term frequency (*nnn*) weighting scheme. Smart allows the use of several term weighting schemes, based on (1) term frequency, (2) inverse document frequency, and (3) the normalization procedure. (The details of the weighting schemes may be found in the Smart source library and <http://people.csail.mit.edu/jrennie/ecoc-svm/smart.html>.) The *Lnu* weighting scheme we have used was described in detail in [5] and the *ltu* weighting scheme is described in detail in section 2.3.

After a text element (e.g., document, section, paragraph) is indexed by Smart, each concept or word type along with its element frequency information is found in the

inverted index. A query is comprised of a set of word types. The duty of a search engine is to find the documents (or elements) which better fit the query. It could be said [11] that the job of a search engine is to separate the document collection into two classes: say A and B, one related to the query and the other which is not (a form of clustering problem). The search engine must then order the set in a descending order in terms of document similarity with respect to the query.

Consider first term frequency. Term frequency is defined as the number of times a term appears in a particular document or element. In Smart terminology, it is the weight of a term in a document vector (*doc.nnn*). It is also present in the inverted file, where the weight of a term (its frequency) in every document or element in which it occurs is specified. The larger the weight of a term, the greater is its importance in that document. The second factor of importance in term weighting is inverse document frequency (*idf*). This is defined as the number of documents (or elements) in which the term occurs. The third factor is normalization. The documents in a collection may or may not be of similar length. Concern arises when we have a collection with a distribution of elements such as paragraphs, subsections, sections and articles. Longer elements (or documents) have more terms and usually also terms with higher frequencies (owing to element length). Even though a shorter document may be more relevant than a longer one, retrieval would be biased towards the retrieval of the latter due simply to its length. In order to promote an unbiased retrieval, a factor known as normalization comes into play [12]. The various schemes which normalize documents are:

(1) Cosine normalization

In this method, weights are normalized according to the following formula

$$\sqrt{w_1^2 + w_2^2 + \dots + w_t^2},$$

where the w_i is the weight of a term in the vector. For more details, see [1].

(2) Maximum tf normalization

In this method, the weights are normalized according to the maximum term frequency in a document. An example would be Smart's augmented tf normalization factor given by the following formula:

$$(0.5 + 0.5 * tf / (\max tf - tf))$$

(3) Byte length normalization:

It normalizes weights based on the size of a document in bytes. It is used in the Okapi system. The details of the BM25 formula can be found in [13].

(4) Pivoted length normalization:

This method was applied in the *Lnu-ltu* weighting formula. (See section 2.3.4.1 for details.)

2.1.3. Retrieval

During the retrieval phase of Smart, element vectors are correlated with the query vectors using (in our case) inner product. A list of top ranked elements (known as the *tr* file) and a list of relevant elements in rank order (known as the *rr* file) are produced as a result.

2.2. Generation of Element Vectors in Flex

After an initial leaf node retrieval by Smart, empty trees are built (from document schemas) for those documents which have at least one element among the top n retrieved nodes. These empty trees are later fed to Flex to be fully constructed and correlated at each level with an appropriately weighted query. The operation of Flex begins with the class FlexDriver whose method reads a configuration file that contains information that is

input to Flex for building document trees. This information is handled by the class, Properties. It also contains information that is required for constructing queries to correlate with elements at each level. The *ltu* queries are constructed by the class FlexQuery which is described in a section 2.4.

The detailed process of generating element vectors and applying *Lnu* weighting to the terms can be found in detail in [5]. *Lnu* term weighting in general is described in detail in [12]. A brief summary of tree building is given here. The process of tree building begins with the leaf node construction. The information about a node in a document tree is maintained in the DocTree class. After the vectors of the leaf nodes are read into the tree, the class Flex.cc is responsible for constructing vectors at the next (higher) level by merging the vectors of the leaf nodes. Merging the child vectors involves taking the union of all the terms present in its child nodes. The parent vector then consists of all the terms that are present in its child nodes.

The weights initially assigned to the terms in the vectors are the term frequencies in that vector. For a given term that occurs in multiple child nodes, the term frequencies are summed to obtain the weight at the next level. These weights are then converted into *Lnu* weights. The process continues until the highest-ranked element, representing the body of the article, is generated. In this way, element vectors are generated in Flex.

2.3. Generation of Query Vectors in Flex

The generation of query vectors is the central aim of this thesis. The problem is a significant one owing to the nature of the weighting scheme we use. We use *ltu* weighting for queries and it requires global statistical information. First, a review of the *ltu*

weighting scheme is presented. Then the parameters that are required to obtain *ltu* weights are discussed. The method for incorporating this scheme in Flex is also presented.

2.3.1. *ltu* term weighting

The *ltu* term weighting mechanism (Figure 1) is comprised of three parts, as seen in Figure 1.

$$\frac{(1 + \log_e tf) (1 + \log_e(N / n_k))}{(1 - slope) + slope * (\# \text{ unique terms}) / pivot}$$

where *tf* is the term frequency

N is the total number of elements in the collection

n_k is the number of elements that contain this term

slope and *pivot* are empirically determined constants

unique terms is the number of unique terms that are present in a vector

Figure 1: *ltu* term weighting

- (1) The *tf* component, given by the formula: $(1 + \log_e tf)$, where *tf* is the term frequency obtained from within the query vector itself
- (2) The *idf* component, given by the formula: $(1 + \log_e(N / n_k))$, where *N* is the total number of elements in a collection. The value *n_k* is the number of elements in which a term appears. For a given term, *n_k* varies across the different ctypes in a

vector. The details are described in section 2.3.3.

(3) The normalization component, given by the formula:

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

This component is used to normalize the weight on a term in a query based on the two components, pivot and slope, as described in section 2.3.4.1.

2.3.2. Determining N

N is defined as the number of elements in a collection. It is a global statistic. There are eight subjective subvectors identified in the IEEE journal article collection, namely: body (*bdy*), article title (*atl*), abstract (*abs*), abstract (*abs*), editor's introduction (*edintro*), keywords (*kwd*), acknowledgements (*ack*), bibliographic article title (*bibl_atl*), and bibliographic title (*bibl_ti*).

Some articles may in fact not contain some ctypes. Hence, N , which is the number of elements in the collection, varies across different ctypes. Let, N_{bdy} represent the number of elements which contain the body subvector, N_{atl} the number of elements which contain the article title subvector, etc. This value of N is obtained during the initial parsing of the documents by counting these tags. The various values of N are described in Tables 1 and 2. The value of N used in calculating *ltu* query term weights in the body subvector is the sum of the individual N_i values at the paragraph, sub section, section and body levels.

Table 1: N values (2005 data)

<i>Ctype</i>	<i>Description</i>	<i>Paragraph</i>	<i>Subsection</i>	<i>Section</i>	<i>Article</i>
0	bdy	1378202	104746	94421	16440
1	atl	1379061	104788	94476	16452
2	abs	1077757	86026	66215	9986
3	edintro	57486	4187	3870	759
4	kwd	675282	50951	32839	4794
5	ack	532010	45904	33623	4952
6	bibatl	1052791	84534	64703	9891
7	biti	1059647	84925	65582	10028

Table 2: N values (2004 data)

<i>Ctype</i>	<i>Description</i>	<i>Paragraph</i>	<i>Subsection</i>	<i>Section</i>	<i>Article</i>
0	bdy	1030318	77852	69692	12049
1	atl	1031207	77894	69754	12062
2	abs	795315	63420	48782	7358
3	edintro	45448	3059	2713	550
4	kwd	496202	37759	24719	3769
5	ack	341125	30242	22348	3294
6	bibatl	767707	62107	47205	7201
7	biti	773457	62472	47883	7304

2.3.3. Determining n_k

The parameter n_k refers to the number of elements in which a term appears. Smart uses the collection statistic (*collstat*) files or the inverted index to find the correct value of n_k to use during term weight computation. These files are produced at the time of indexing. The only information that is directly available to us during flexible retrieval is the n_k value at the paragraph level (as only a paragraph index is maintained).

In order to obtain the n_k values at the other element levels (i.e., subsection, section and body) levels, we would ordinarily use the respective indices. However, we propose an alternate method which does not require indexing at the other levels to compute the n_k values. Along with the paragraph index, Smart produces a *textloc* file that contains information about the location of the element on the disk. Using a Perl script, we convert the *textloc* file into a *docid-docpath mapping* file which contains a mapping between the Smart identifier (*id*) of the element and its xpath (which is required for conversion from Smart to INEX format). The xpath contains the detailed information about the location of an element in a document. It provides a unique way of addressing an element location within a document. A typical xpath is:

```
an/1995/article[1]/bdy[1]/sec[1]/ss1[1]/p[1]
```

Recall that the inverted file contains a mapping between a term and the docids in which the term appears. Given the inverted index at the paragraph level, we obtain the n_k values as shown below.

Since processing these n_k values at query time is expensive, we front load the n_k calculations. After indexing the paragraphs, we calculate n_k for every term that appears in the inverted file and store it in a vector form (known as the n_stats vector). A typical n_stats vector for one term is represented in the Figure 2.

con-id	ctype	element-type	n_k value
1	0	1 (para)	4
1	0	2 (ss)	3
1	0	3 (sec)	2
1	0	4 (art)	1
1	0	5 (allele)	10

Figure 2: Structure of n_stats vector

The description of the above vector is as follows. Consider for example, a term (concept or con) with con-id = 1 and ctype = 0 (representing a term from the body of an article). The n_k value at paragraph level is stored under element-type 1, the n_k value at the subsection level is stored under element-type 2, at the section level under element-type 3, at the article level under element-type 4, and the n_k value at the all-element level (i.e., the sum) is stored under element-type 5. In this way, we create an n_stats vectors for every term (and every ctype).

For example, suppose we are given a concept id, say 37145, which occurs in 5 paragraphs as shown in Figure 3. The figure shows a typical inverted file entries for this con.

<i><u>docid</u></i>	<i><u>ctype</u></i>	<i><u>conid</u></i>	<i><u>weight</u></i>
1000	0	37145	1.0000
2000	0	37145	1.0000
3000	0	37145	2.0000
4000	0	37145	2.0000
5000	0	37145	1.0000

Figure 3: Inv vector for a con

We use the docid-docpath mapping file in order to find the xpaths corresponding to these docids. Consider the xpaths given in Figure 4.

<i><u>docid</u></i>	<i><u>xpath</u></i>
1000	an/1995/article[1]/bdy[1]/sec[1]/ss1[1]/p[1]
2000	an/1995/article[1]/bdy[1]/sec[1]/ss1[1]/p[2]
3000	an/1995/article[1]/bdy[1]/sec[1]/ss1[2]/p[1]
4000	an/1995/article[1]/bdy[1]/sec[2]/ss1[1]/p[1]
5000	an/1996/article[1]/bdy[1]/sec[1]/ss1[1]/p[1]

Figure 4: A typical docid-docpath mapping file

We process the xpaths of the paragraphs shown to produce the union of the subsection, section and article xpaths as seen in the Figure 5. Thus, based on only a paragraph index, we are able to determine the value of n_k at each higher (i.e., subsection, section, and body) level in the document tree. The n_k values are stored in a vector form in the `n_stats` file. (These values are used by the query processing class, FlexQuery.) The algorithm

describing the calculation of n_k at the upper levels of the document tree is given in Figure 6.

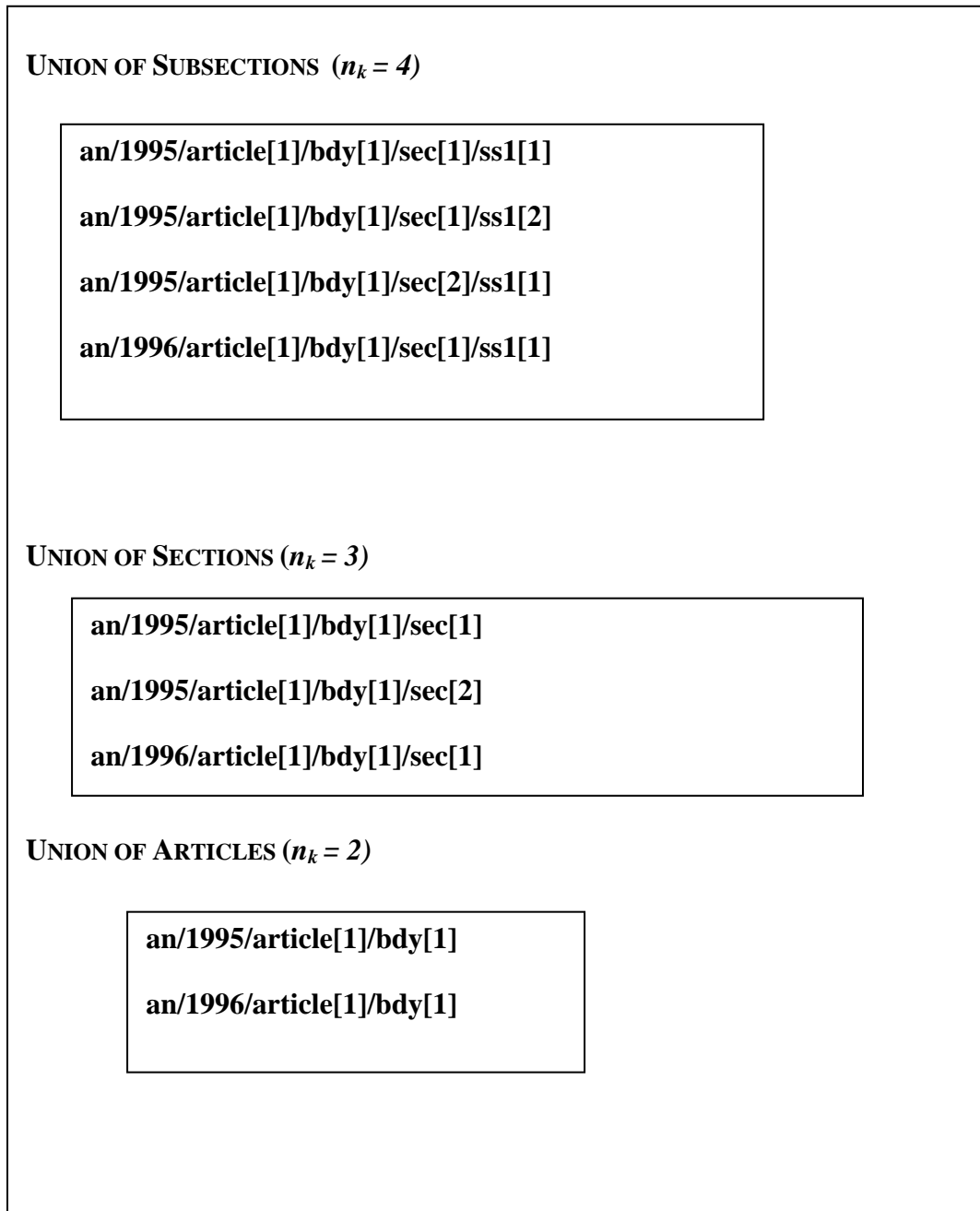


Figure 5: Processing xpaths

For each *ctype*, *i*

Open the inverted index belonging to the *ctype i*

For each *con*, *j* present in the inv file of type *i*

Get the docids in which *con j* appears

Get the corresponding xpaths for the docids

Process the xpaths by shortening them and collect the union of *subsections*, *sections* and *articles* in separate hashes.

n_k value at *subsection* level for *con j*, of *ctype i* = Number of elements in the *subsection hash*

n_k value at *section* level for this *con j*, of *ctype i* = Number of elements in the *section hash*

n_k value at *article* level for this *con j* of *ctype i* = Number of elements in the *article hash*

End of For

Close the inverted index belonging to the *ctype i*

End of For

Figure 6: Algorithm describing the calculation of n_k at upper levels, given n_k at the paragraph level

2.3.4. Normalization

2.3.4.1. Pivoted Normalization

According to Singhal, et.al, [12], a normalization scheme that takes into account the probability of retrieval and relevance is more efficient than another which does not. They studied the deviation of the retrieval pattern from the relevance pattern. Curves of probabilities of retrieval and relevance were plotted against document length to study the effect. They found that all the documents on one side of the point of intersection of the curves had a probability of relevance less than the probability of retrieval, and on the other side, the reverse was true. In order to balance the effect, they defined the document length at the point of intersection of these curves as the *pivot*. Using pivot, the probability of retrieval can be tilted towards the probability of relevance. The amount of tilting is determined by the *slope*, which can be trained on one set of data and queries and applied on collections with similar characteristics in terms of length. These parameters are depicted in the Figure 7. We use this normalization function for normalizing the weights on query terms as well as terms on documents. The estimation of these parameters is described in Chapter 3.

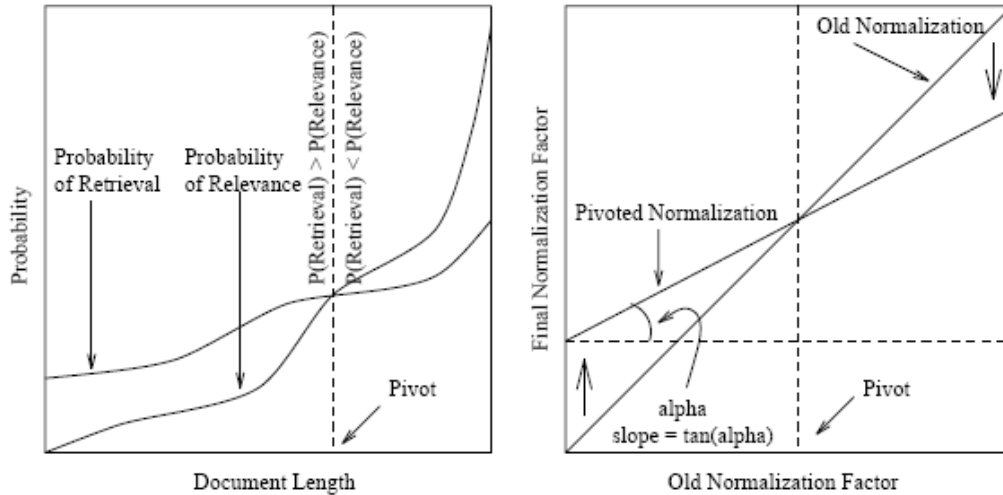


Figure 7: Pivoted length normalization (from [12])

2.3.4.2. Number of Unique words

The number of unique words that appears in the normalization function is specified in Smart using a parameter known as the “major-ctype.” If the major-ctype is set to 0, each subvector contained in the vector is normalized by the number of unique words present in the ctype 0 subvector. By default, the major ctype value is set to -1, which means that the vector is normalized based upon the number of unique words present in all the subvectors put together. As most of the contribution is due to the body subvector, we have set major ctype to 0 throughout our experiments.

2.4. Incorporation of Query Processing in Flex

We extended our original flexible retrieval system, Flex, in order to generate a properly weighted *ltu* query at run time. The driver class, FlexDriver, calls the process_query method belonging to the FlexQuery class to construct queries at all levels. The class FlexQuery interacts with the smart_interface library to read the n_k values from

the *n_stats* vectors. It reads the pivot, slope and the *N* values from the Properties class. It also reads the path of the *nnn*-weighted paragraph query. After getting the required information, it computes the *ltu* weighted queries at all levels, namely, the paragraph, subsection, section and article levels. It also creates an all-element query which is virtually identical to the *ltu* query produced during *ltu* term weighting in the all-element index. After creating the queries, it returns control to FlexDriver which calls Flex to populate a document tree. Flex interacts with the FlexQuery class by means of *get_X_query* functions, (where X may stand for paragraph, subsection, section, body, all-elements) which return queries at each level.

3. Experiments

The various experiments that we have conducted to test the performance of our system are described in this chapter. The methods that we use for element retrieval—the all-element index (base case) and flexible retrieval—are described very briefly. Then, we describe the various parameters used for testing are specified. Later, we describe the body only experiments and the extended vector retrieval experiments. Results are presented in Chapter 4. We have used INEX 2004 and 2005 IEEE journal articles, queries and relevance assessments as our test bed and *inex_eval* as the metric for evaluation of our experiments.

3.1. All-Element Index Retrieval

We get the IEEE collection from INEX in the form of a zipped tar file. It is uncompressed and parsed using a Java parser [5] which uses JAXP API. After parsing, we index paragraphs, subsections, sections and articles using Smart. We use relevance assessments provided by INEX and calculate pivot and slope values currently tuned to (*inex_eval*) average precision for our experiments. Term weighting is performed using the *Lnu-ltu* weighting scheme, and a Smart retrieval is done. We submit the first 1500 retrieval results of the all-element run and evaluate them using *inex-eval*.

3.2. Flexible Retrieval

The details of this procedure were described in Chapter 2. We use only the paragraph index for flexible retrieval. While parsing, we calculate the values of N at each

level (and for every ctype). After the indexing process, we calculate the n_k values for every con in the inverted index and store them (in the form of an `n_stats` vector) to be later fed to Flex. The paragraph vectors are term-weighted using the *Lnu-ltu* weighting scheme, the initial paragraph retrieval is performed, the empty trees are built, and Flex is run in order to populate the trees.

3.3. Parameters used in experimentation

The parameters used in our experiments are n , *pivot* and *slope*. The parameter n represents the number of leaf nodes input to Flex. An important factor in our experiments is the minimum number of trees which must be generated (i.e., in which relevant elements appear). Flex only builds trees for those documents which have one element (paragraph) in these top n nodes. Thus tree building time is minimized by choosing an appropriate value of n . The values of n utilized in these experiments are 250, 200, 150, 100, 50, 25, 10, 5, and 1.

The importance of *pivot* and *slope* were described in Section 2.3.4.1. *Pivot* is defined as the average number of unique words in an element. *Slope* was calculated empirically (tuned to the average precision over 1500 elements with respect to Smart evaluation for paragraphs, subsections, sections and bodies). We used two sets of values for *pivot* and *slope*—generic values from TREC (with *pivot* set to 110 and *slope* set to 0.2) and what we call *tuned values*, tuned to the all-element index. The various values of *pivot* and *slope* are shown in Table 3.

Table 3: Pivot and slope values

SNO	ELEMENT TYPE	PIVOT	SLOPE
1	Paragraph	25	0.01
2	Subsection	93	0.045
3	Section	156	0.007
4	Article	559	0.01
5	All-Element	43	0.03

3.4. Body-only experiments

These experiments were conducted by setting just the body subvector weights to 1 with the remaining subvector weights set to 0. The values of precision at different ranks and the average precision are recorded in the tables for specified values of n , pivot and slope, and different quantization measures under *inex_eval*. For details about quantization measures, refer to INEX 2005 metrics paper [14].

3.5. Extended vector experiments

A set of experiments based on the use of extended vectors were also performed. There are eight identifiable subvectors in INEX 2004 and 2005 data. An important factor in extended vector retrieval is subvector weighting, so our first step was to establish meaningful subvector weights for use with the all-element index and then apply the same

weights to Flex. We first describe subvector weighting and then the extended vector experiments.

3.5.1. Subvector weighting

Table 4 shows the various combinations of subvector weights that were applied first to the all-element index and then to extended vector retrieval using Flex. First we observe that *bdy* is the chief contributor among all subvectors. Removing *abs*, *kwd*, *ack* and *bibl_ti* improved our results. Various subvector weight combinations were tested on *bdy*, *atl*, *edintro* and *bibl_atl*; it was found that combination 1 as shown in Table 4 is the best combination of subvector weights found to date. We then apply these subvector weights to extended vector flexible retrieval by running Flex. The results of Flex are described in section 4.

3.5.2. Extended vector retrieval using Flex

We can view the extended vector experiments in two ways:

- (1.) Hybrid retrieval: Initial Smart paragraph retrieval yields input to Flex to produce (extended vector) elements. Subvector weighted as indicated in Section 3.5.1.
- (2.) Pure extended vector retrieval: Extended vector retrieval selects the initial set of paragraphs input to Flex. Retrieval is performed using extended vectors.

Both approaches were examined and results are reported in the appendix.

Table 4: Subvector weight combinations

Description	bdy	atl	abs	edintro	kwd	ack	bibl_atl	bibl_ti	AvgP
equal weights	1	1	1	1	1	1	1	1	0.01947
only bdy	1	0	0	0	0	0	0	0	0.06396
only atl	0	1	0	0	0	0	0	0	0.00491
only abs	0	0	1	0	0	0	0	0	0.00119
only edintro	0	0	0	1	0	0	0	0	0.00081
only kwd	0	0	0	0	1	0	0	0	0.00007
only ack	0	0	0	0	0	1	0	0	0.00039
only bibl_atl	0	0	0	0	0	0	1	0	0.00084
only bibl_ti	0	0	0	0	0	0	0	1	0.00086
no bdy	0	1	1	1	1	1	1	1	0.00217
no atl	1	0	1	1	1	1	1	1	0.01978
no abs	1	1	0	1	1	1	1	1	0.01993
no edintro	1	1	1	0	1	1	1	1	0.01831
no kwd	1	1	1	1	0	1	1	1	0.01867
no ack	1	1	1	1	1	0	1	1	0.01925
no bibl_atl	1	1	1	1	1	1	0	1	0.01891
no bibl_ti	1	1	1	1	1	1	1	0	0.01986
combn 1	2	0	0	1	0	0	0	0	0.06651
combn 2	3	0	0	1	0	0	0	0	0.06644
combn 3	4	0	0	1	0	0	0	0	0.06551
combn 4	4	1	0	1	0	0	0	0	0.04457
combn 5	4	1	0	2	0	0	0	0	0.04329
combn 6	4	1	0	0	0	0	0	0	0.04319
combn 7	3	2	0	0	0	0	0	0	0.04063
combn 8	4	2	0	1	0	0	0	0	0.04051
combn 9	3	2	0	1	0	0	0	0	0.04038
combn 10	4	2	0	0	0	0	0	0	0.04006

4. Results

The results of our experiments and their analysis are described in detail in this chapter. First, we present the body-only experiments using TREC and tuned pivot and slope, under both strict and generalized quantizations. We also conducted a similar set of experiments for extended vector retrieval and hybrid retrieval using the 2004 and 2005 INEX collections. See the Appendix for details. In each table below, the first column gives the value of n , i.e., the number of nodes input to Flex, intermediate columns give precision at rank, and the last column indicates average precision. The first row indicates the base case run, i.e., all-element retrieval.

4.1. Body-only experiments

Here, we describe the results of body-only experiments, which constitute the most important experiments performed. (The corresponding extended vector experiments are described in the Appendix). Experiments are performed on both INEX 2004 and 2005 collections. Experiments are evaluated using both generalized and strict quantizations of *inex_eval*. According to [14], for users who are interested in highly exhaustive and highly specific elements, strict quantization is applied. In order to evaluate elements based on varying degrees of relevance, generalized quantization is applied. All experiments were performed using two values of pivot and slope—TREC and tuned.

4.1.1. 2004 data

The tables show identical experiments for both TREC and tuned values of pivot

and slope. Results in Tables 5 and 6 are evaluated using generalized quantization; Tables 7 and 8 use strict quantization.

Table 5: 2004 data TREC pivot and slope generalized quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.3897	0.3088	0.2735	0.2529	0.1999	0.1523	0.0697	0.0354	0.0625
250	0.3971	0.2882	0.2684	0.2412	0.1982	0.1506	0.0714	0.0413	0.0652
200	0.3971	0.2882	0.2699	0.2430	0.1990	0.1507	0.0732	0.0421	0.0658
150	0.3971	0.2971	0.2684	0.2438	0.1996	0.1513	0.0769	0.0435	0.0675
100	0.3971	0.2985	0.2706	0.2452	0.2022	0.1535	0.0827	0.0441	0.0701
50	0.3971	0.2985	0.2691	0.2441	0.2006	0.1590	0.0842	0.0414	0.0725
25	0.3971	0.2897	0.2713	0.2485	0.2031	0.1737	0.0786	0.0343	0.0723
10	0.3676	0.2838	0.2654	0.2375	0.2129	0.1571	0.0578		0.0676
5	0.3382	0.2868	0.2632	0.2500	0.1954	0.1341	0.0385		0.0608
1	0.2794	0.2353	0.2110	0.1772	0.1216	0.0769			0.0433

It is observed that the best average precision (AvgP) is found when $n = 50$; 10 input nodes produce an AvgP required to that produced by the all-element index. The best P @ 20 is found in the base case; 5 input leaf nodes produce a comparable result to that of the all-element index.

Table 6: 2004 data tuned pivot and slope generalized quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.3676	0.3324	0.3037	0.2643	0.2063	0.1566	0.0717	0.0360	0.0632
250	0.3971	0.3279	0.3029	0.2610	0.2087	0.1600	0.0757	0.0412	0.0676
200	0.3971	0.3279	0.3051	0.2618	0.2096	0.1620	0.0769	0.0421	0.0683
150	0.3971	0.3279	0.3059	0.2621	0.2100	0.1650	0.0793	0.0435	0.0701
100	0.3750	0.3294	0.3074	0.2632	0.2146	0.1679	0.0841	0.0440	0.0733
50	0.3750	0.3279	0.3162	0.2643	0.2178	0.1724	0.0854	0.0413	0.0755
25	0.3971	0.3235	0.3125	0.2662	0.2168	0.1784	0.0788	0.0343	0.0748
10	0.3897	0.3368	0.2890	0.2485	0.2221	0.1632	0.0578		0.0737
5	0.3676	0.3206	0.2824	0.2551	0.2001	0.1360	0.0385		0.0662
1	0.2941	0.2529	0.2221	0.1790	0.1222	0.0769			0.0473

It is observed that the best AvgP is found when $n = 50$; and 5 input nodes produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 is found when $n = 25$; and 25 input nodes produce a comparable result to that of the all-element index. Tables 5 and 6 indicate that experiments with tuned values of pivot and slope perform better than experiments with TREC values in terms of AvgP.

Table 7:2004 data TREC pivot and slope strict quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.2000	0.1440	0.1240	0.1120	0.0848	0.0628	0.0268	0.0141	0.0586
250	0.2000	0.1200	0.1240	0.1040	0.0848	0.0616	0.0270	0.0164	0.0591
200	0.2000	0.1200	0.1240	0.1040	0.0840	0.0616	0.0272	0.0168	0.0590
150	0.2000	0.1200	0.1240	0.1060	0.0840	0.0616	0.0286	0.0177	0.0601
100	0.2000	0.1280	0.1280	0.1100	0.0840	0.0632	0.0322	0.0177	0.0618
50	0.2000	0.1280	0.1280	0.1080	0.0832	0.0656	0.0323	0.0153	0.0624
25	0.2000	0.1200	0.1280	0.1060	0.0872	0.0672	0.0302	0.0122	0.0617
10	0.1600	0.1120	0.1120	0.0920	0.0752	0.0536	0.0205		0.0540
5	0.1200	0.1200	0.1120	0.1060	0.0768	0.0472	0.0130		0.0512
1	0.1200	0.1200	0.1080	0.0940	0.0552	0.0300			0.0483

It is observed that the best AvgP is found when $n = 50$; and 25 input nodes produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 is found in the base case run and a result comparable to the base case can be obtained by feeding $n = 100$ leaf nodes to Flex.

Table 8:2004 data tuned pivot and slope strict quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.2000	0.2000	0.1720	0.1200	0.0848	0.0584	0.0271	0.0141	0.0722
250	0.2000	0.1760	0.1760	0.1200	0.0832	0.0604	0.0282	0.0163	0.0725
200	0.2000	0.1760	0.1760	0.1200	0.0832	0.0616	0.0285	0.0167	0.0725
150	0.2000	0.1760	0.1720	0.1200	0.0832	0.0640	0.0301	0.0176	0.0735
100	0.2000	0.1840	0.1720	0.1200	0.0856	0.0636	0.0326	0.0176	0.0754
50	0.2000	0.1840	0.1760	0.1180	0.0888	0.0672	0.0331	0.0153	0.0766
25	0.2000	0.1840	0.1680	0.1140	0.0864	0.0692	0.0305	0.0122	0.0747
10	0.1600	0.1440	0.1320	0.0960	0.0768	0.0580	0.0205		0.0676
5	0.1600	0.1600	0.1320	0.1020	0.0768	0.0480	0.0130		0.0629
1	0.1600	0.1280	0.1240	0.0940	0.0560	0.0300			0.0519

It is observed that the best AvgP is found when $n = 50$; 25 input nodes produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 can be found in the base case and also when $n = 100$; results comparable to the base case run can be obtained by feeding $n = 100$ leaf nodes to Flex.

From Tables 7 and 8, it can be concluded that experiments using tuned values of pivot and slope perform better than experiments using TREC values. From Tables 5—8, i.e., the body-only experiments on 2004 data, it can be concluded that tuned values of pivot and slope perform better as compared to TREC values. The number of leaf nodes to be fed to Flex can be kept as low as 50 for decent values of average precision.

4.1.2. 2005 data

The tables show identical experiments for both TREC and tuned values of pivot and slope. Tables 9 and 10 are evaluated using generalized quantization; Tables 11 and 12 use strict quantization.

Table 9: 2005 data TREC pivot and slope generalized quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.4224	0.3241	0.2991	0.2841	0.2364	0.1921	0.0943	0.0472	0.0739
250	0.4224	0.3448	0.2991	0.2978	0.2398	0.1909	0.0937	0.0512	0.0766
200	0.4224	0.3362	0.2983	0.2944	0.2398	0.1922	0.0951	0.0512	0.0767
150	0.4224	0.3310	0.2957	0.2901	0.2378	0.1915	0.0957	0.0498	0.0756
100	0.4224	0.3345	0.2974	0.2944	0.2347	0.1866	0.0928	0.0456	0.0723
50	0.4224	0.3362	0.3000	0.2897	0.2290	0.1830	0.0811	0.0364	0.0629
25	0.4224	0.3397	0.3043	0.2841	0.2241	0.1785	0.0711	0.0289	0.0535
10	0.4224	0.3328	0.3034	0.2836	0.2110	0.1542	0.0499		0.0408
5	0.4224	0.3414	0.3276	0.2871	0.2019	0.1360	0.0347		0.0332
1	0.4483	0.3207	0.2500	0.1797	0.1060	0.0681			0.0185

It is observed that the best AvgP value is found when $n = 200$; 150 input leaf nodes are required to produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 is found when $n = 5$; 5 input leaf nodes produce a result comparable to the all-element index.

Table 10: 2005 data tuned pivot and slope generalized quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.3621	0.3241	0.2966	0.2668	0.2241	0.1847	0.0912	0.0456	0.0698
250	0.3448	0.3362	0.3069	0.2694	0.2257	0.1857	0.0932	0.0504	0.0733
200	0.3448	0.3345	0.3043	0.2681	0.2291	0.1869	0.0946	0.0507	0.0740
150	0.3448	0.3345	0.3095	0.2651	0.2305	0.1884	0.0956	0.0495	0.0732
100	0.3448	0.3362	0.3164	0.2694	0.2286	0.1873	0.0925	0.0456	0.0696
50	0.3534	0.3345	0.3190	0.2698	0.2302	0.1828	0.0817	0.0364	0.0616
25	0.3793	0.3276	0.3095	0.2711	0.2271	0.1786	0.0712	0.0288	0.0523
10	0.3793	0.3190	0.3009	0.2759	0.2141	0.1549	0.0499		0.0404
5	0.3879	0.3483	0.3164	0.2802	0.2050	0.1366	0.0347		0.0327
1	0.4138	0.3276	0.2362	0.1819	0.1055	0.0681			0.0180

Here we see that the best AvgP is found when $n = 200$; 100 input leaf nodes are enough to obtain a comparable result with the all-element index. The best P @ 20 is

found when $n = 5$; 5 input leaf nodes produce a comparable result to that of the all-element index.

From Tables 9 and 10, we conclude that for the INEX 2005 collection, under generalized quantization, experiments with TREC values of pivot and slope perform better than experiments with tuned values. The rationale may be that the tuned values of pivot and slope are tuned with respect to strict quantization of *inex_eval*. We find results produced by TREC values of pivot and slope are substantially better than those produced by tuned values (i.e., tuned to generalized quantization).

Table 11: 2005 data TREC pivot and slope strict quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.1481	0.0667	0.0852	0.0815	0.0689	0.0511	0.0219	0.0096	0.0318
250	0.1481	0.0741	0.0741	0.0852	0.0674	0.0496	0.0215	0.0106	0.0333
200	0.1481	0.0741	0.0778	0.0833	0.0681	0.0507	0.0219	0.0101	0.0337
150	0.1481	0.0741	0.0778	0.0833	0.0689	0.0511	0.0227	0.0100	0.0340
100	0.1481	0.0741	0.0778	0.0870	0.0681	0.0500	0.0227	0.0091	0.0335
50	0.1481	0.0741	0.0741	0.0759	0.0630	0.0478	0.0174	0.0068	0.0263
25	0.1481	0.0741	0.0778	0.0741	0.0644	0.0496	0.0154	0.0059	0.0247
10	0.1481	0.0815	0.0852	0.0759	0.0585	0.0378	0.0087		0.0182
5	0.1481	0.0889	0.0889	0.0722	0.0511	0.0285	0.0059		0.0182
1	0.1852	0.0741	0.0667	0.0370	0.0178	0.0089			0.0099

Here, the best AvgP is found when $n = 150$; 100 input leaf nodes produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 is found when $n = 100$; 100 input leaf nodes are required to produce a result comparable to the all-element index.

Table 12: 2005 data tuned pivot and slope strict quantization

n	P@1	P@5	P@10	P@20	P@50	P@100	P@500	P@1500	AvgP
-	0.1852	0.0889	0.0704	0.0611	0.0511	0.0426	0.0200	0.0093	0.0640
250	0.1852	0.1037	0.0778	0.0611	0.0541	0.0437	0.0201	0.0102	0.0668
200	0.1852	0.1037	0.0778	0.0611	0.0541	0.0448	0.0212	0.0100	0.0672
150	0.1852	0.1037	0.0815	0.0630	0.0548	0.0467	0.0221	0.0100	0.0677
100	0.1852	0.1037	0.0815	0.0667	0.0548	0.0478	0.0223	0.0091	0.0678
50	0.1481	0.0889	0.0815	0.0556	0.0526	0.0433	0.0173	0.0069	0.0251
25	0.1481	0.0889	0.0815	0.0593	0.0563	0.0467	0.0155	0.0058	0.0237
10	0.1481	0.0963	0.0741	0.0667	0.0570	0.0381	0.0087		0.0185
5	0.1481	0.1111	0.0741	0.0630	0.0519	0.0285	0.0059		0.0203
1	0.1111	0.0815	0.0519	0.0370	0.0178	0.0089			0.0080

Best AvgP is when $n = 100$; and 100 input leaf nodes produce an AvgP equivalent to that produced by the all-element index. The best P @ 20 can be found when $n = 10$; and $n = 5$ input leaf nodes produce a comparable result to that of the all-element index. From tables 11 and 12, we conclude that under strict quantization, (in almost all the cases), experiments with tuned values of pivot and slope perform better than experiments with TREC values.

From Tables 9, 10, 11 and 12, (i.e. the body-only experiments of the 2005 data), it can be concluded that tuned values of pivot and slope perform better (when compared to TREC values of pivot and slope) for strict quantization, and the reverse occurs in the case of generalized quantization. The value of number of leaf nodes fed to Flex can be kept at 100 for decent values of average precision.

4.1.3. Tree Generation Stats

Tree Stats 2004

While Flex is building the trees, we calculate the amount of time it takes to build a tree and also the average number of trees built per query. As seen from the tables, that it takes fraction of a second to generate a single tree. Also, the average number of trees built per query ranges from approximately 8 to 146 for corresponding values of n of 10—250.

Table 13: Tree Stats 2004

Pivot and Slope	n	Avg time per query	Avg num of trees built per query
TREC	250	7.7059	145.2060
Tuned	250	7.7059	145.2060
TREC	200	8.1471	119.1470
Tuned	200	7.6177	119.7470
TREC	150	5.7353	92.0000
Tuned	150	6.0000	92.0000
TREC	100	3.5000	63.9412
Tuned	100	3.4706	63.9412
TREC	50	1.9706	34.1176
Tuned	50	1.8824	34.1176
TREC	25	1.0294	18.3235
Tuned	25	1.0588	18.3235
TREC	10	0.4412	7.8529
Tuned	10	0.4706	7.8529
TREC	5	0.2647	4.2647
Tuned	5	0.2647	4.2647
TREC	1	0.0882	1.0000
Tuned	1	0.0882	1.0000

Tree Stats 2005

Table 14: Tree stats 2005

Pivot and Slope	n	Avg time per query	Avg num of trees built per query
TREC	250	8.2250	149.0250
Tuned	250	8.2250	149.0250
TREC	200	17.4750	122.4500
Tuned	200	17.9500	122.4500
TREC	150	13.8750	95.0000
Tuned	150	13.4500	95.0000
TREC	100	3.6250	65.8750
Tuned	100	3.6500	65.8750
TREC	50	1.9500	35.2250
Tuned	50	2.0000	35.2250
TREC	25	1.0750	19.5250
Tuned	25	1.1000	19.5250
TREC	10	0.5000	8.5500
Tuned	10	0.5000	8.5500
TREC	5	0.2750	4.6000
Tuned	5	0.2500	4.6000
TREC	1	0.0500	1.0000
Tuned	1	0.0500	1.0000

We also present the space requirements for all-element and flexible retrieval using both the 2004 and 2005 data. The tables compare the disk space requirements of the all-element index versus the paragraph index.

2004 data - space requirements (body-only vectors)

Table 15: 2004 data - space requirements

Files	All-Element Index	Paragraphs (Flex)
Dictionaries	39 MB	18 MB
Inverted Index	402 MB	202 MB
Doc vectors	437.2 MB	185.9 MB
Collstats	106 MB	40 MB
Schemas		203 MB
n_stat_vectors		23.8 MB
Total	984.2 MB	672.7 MB

2005 data space requirements (body-only vectors)

Table 16: 2005 data - space requirements

Files	All-Element Index	Paragraphs
Dictionaries	44 MB	19 MB
Inverted Index	547 MB	277.8 MB
Doc vectors	1402.47 MB	540.3 MB
Collstats	211 MB	132 MB
Schemas		289 MB
n_stat_vectors		27.8 MB
Total	2204.47 MB	1285.9 MB

As seen from the tables, the dictionary, inverted index, the document vectors and the collstats occupy twice the amount of space in all-element index as they do for flexible retrieval (using the paragraph files as input). Even with the additional files required by Flex (schemas and n_stat vectors) the total space required for dynamic retrieval is nearly

half that required by the all-element index. For extremely large collections, it would be feasible to use flexible retrieval in order to prevent replication of data in terms of the all-element index.

5. Conclusion and Future Work

Flexible retrieval minimizes the space that mammoth indices occupy on a hard disk. We have established that flexible retrieval can in fact be implemented by producing element vectors at each level along with the corresponding query vectors. Given n_k at the paragraph level, n_k can now be exactly determined at other (upper) levels in the document tree. This value can in fact be calculated in advance and later accessed using Smart utilities, thereby minimizing the query processing time. This calculation of n_k also enables us to use this value in other formulas involving inverse document frequency. As a result of this research, we have also established the minimum number of input nodes required to build a sufficient number of trees so that a particular value of AvgP or P @ r can be realized. For 2004 data, we required 50 leaf nodes to produce a result similar to the all-element index retrieval in terms of AvgP, and for the 2005 data about 100 leaf nodes to produce a result similar to the all-element index retrieval with respect to AvgP.

More testing on pivots and slopes tuned to different metrics may produce interesting results. Testing subvector weights under different metrics, choosing an optimal one through normalization among each individual lists, and then summing the results across all the lists may produce an optimal set of subvector weights. It is possible that n , i.e., the number of leaf nodes fed to Flex can reasonably be varied on a query basis. By their nature, some queries may require that very few trees be generated. Others, similarly, may require many. This would provide us with interesting insights about the dependency of the characteristics of a query term weight (such as n_k) on the number of

relevant documents in which it appears. Based on the number of elements in which a query term appears, it may be possible to determine in advance, how many trees need to be built for that query to achieve an optimal performance.

Bibliography

- [1] Gerard Salton, A.Wong, and C.S.Yang. A vector space model for information retrieval. *Journal of the American Society for Information Science*, 18(11):613-620, November 1975.
- [2] 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.
- [3] 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.
- [4] Salton, G., editor. The SMART Retrieval System – Experiments in Automatic Document Retrieval, Prentice-Hall, Englewood Cliffs, NJ, 1971.
- [5] Khanna, S. Design and Implementation of a Flexible Retrieval System, MS Thesis, University of Minnesota Duluth, 2005.
- [6] 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.
- [7] Govert,N; Abolhassani,M; Fuhr,N; GroBjohann,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.
- [8] Mass,Y; Mandelbrod M. Component ranking and Automatic Query Refinement for XML Retrieval. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [9] Sigurbjornsson,B; Kamps,J; Rijke,M. The University of Amsterdam at INEX 2004. *INEX 2004 Workshop Preproceedings*, Schloss Dagstuhl, December 6-8, 2004.
- [10] Mahajan, A. Flexible retrieval in a structured environment. MS Thesis, University of Minnesota Duluth, 2004.
- [11] Baeza-Yates, R., Ribeiro-Neto, B. (eds.), *Modern Information Retrieval*, ACM Press, New York (1999).
- [12] Singhal, A; Buckley, C; Mitra, M. Pivot Document Length Normalization. *ACM SIGIR*, 1996.
- [13] Robertson, S; Walker, S; Beaulieu, M. Okapi at TREC-7. *Automatic adhoc, filtering*,

VLC and interactive track.

[14] Kazai, G; Lalmas, M; INEX 2005 Evaluation Metrics. *Advances in XML Information Retrieval and Evaluation: Fourth Workshop of the Initiative for the Evaluation of XML Retrieval (INEX 2005)*, Schloss Dagstuhl, 28-30 November 2005, Springer-Verlag, Lecture Notes in Computer Science, Vol 3977, pp. 16-29, 2006.

[15] Crouch, C; Khanna, S; Potnis, P; Doddapaneni, N; The Dynamic Retrieval of XML Elements. *Advances in XML Information Retrieval and Evaluation: 4th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2005*, Dagstuhl Castle, Germany, 28-30 November 2005, Springer-Verlag, Lecture Notes in Computer Science, Vol 3977, pp. 268-281, 2006.

Table of Contents

1. INTRODUCTION	1
2. FLEXIBLE RETRIEVAL	4
2.1. Smart	5
2.1.1. Indexing	5
2.1.2. Term Weighting.....	6
2.1.3. Retrieval.....	8
2.2. Generation of Element Vectors in Flex.....	8
2.3. Generation of Query Vectors in Flex	9
2.3.1. <i>ltu</i> term weighting.....	10
2.3.2. Determining N	11
2.3.3. Determining n_k	13
2.3.4. Normalization	18
2.4. Incorporation of Query Processing in Flex	19
3. EXPERIMENTS	21
3.1. All-Element Index Retrieval	21
3.2. Flexible Retrieval.....	21
3.3. Parameters used in experimentation	22
3.4. Body-only experiments.....	23
3.5. Extended vector experiments	23
3.5.1. Subvector weighting	24
3.5.2. Extended vector retrieval using Flex	24
4. RESULTS	26
4.1. Body-only experiments.....	26
4.1.1. 2004 data.....	26
4.1.2. 2005 data.....	29
4.1.3. Tree Generation Stats.....	33
Tree Stats 2004	33
Tree Stats 2005	34
2004 data - space requirements (body-only vectors)	35
2005 data space requirements (body-only vectors).....	35
5. CONCLUSION AND FUTURE WORK	37
BIBLIOGRAPHY	39

List of Figures

Figure 1: Itu term weighting	10
Figure 2: Structure of n_stats vector.....	14
Figure 3: Inv vector for a con	15
Figure 4: A typical docid-docpath mapping file.....	15
Figure 5: Processing xpaths	16
Figure 6: Algorithm describing the calculation of n_k at upper levels, given n_k at the paragraph level.....	17
Figure 7: Pivoted length normalization (from [12])	19

List of Tables

Table 1: N values (2005 data).....	12
Table 2: N values (2004 data).....	12
Table 3: Pivot and slope values	23
Table 4: Subvector weight combinations.....	25
Table 5: 2004 data TREC pivot and slope generalized quantization.....	27
Table 6: 2004 data tuned pivot and slope generalized quantization	27
Table 7:2004 data TREC pivot and slope strict quantization	28
Table 8:2004 data tuned pivot and slope strict quantization.....	29
Table 9: 2005 data TREC pivot and slope generalized quantization.....	30
Table 10: 2005 data tuned pivot and slope generalized quantization	30
Table 11: 2005 data TREC pivot and slope strict quantization	31
Table 12: 2005 data tuned pivot and slope strict quantization.....	32
Table 13: Tree Stats 2004	33
Table 14: Tree stats 2005	34
Table 15: 2004 data - space requirements.....	35
Table 16: 2005 data - space requirements.....	35