12-2010

Dynamic indexing

Viswada Sripathi
University of Nevada, Las Vegas

DYNAMIC INDEXING

By

Viswada Sripathi

Bachelor of Technology, Computer Science and Engineering
Jawaharlal Nehru Technological University, India
May 2008

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2010

ABSTRACT

Dynamic Indexing

by
Viswada Sripathi

Dr. Kazem Taghva, Examination Committee Chair
Professor, Department of Computer Science
University of Nevada, Las Vegas

In this thesis, we report on index constructions for large document collections to facilitate the task of search and retrieval. We first report on classical static index construction methods and their shortcomings. We then report on dynamic index construction techniques and their effectiveness.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to sincerely thank my committee chair, Dr. Kazem Taghva for all his support and guidance throughout this thesis research. Without his guidance and persistent help, completion of this thesis would not have been possible.

I sincerely thank my graduate coordinator Dr. Ajoy K Datta for his help and invaluable support through my masters program and also for being my committee member. I extend my gratitude to Dr. Laxmi P. Gewali and Dr. Venkatesan Muthukumar for accepting to be a part of my committee. A special thanks to Mr. Ed Jorgensen, Ms. Leslie Nilsen, Mr. Darren Paulson, Ms. Donna Ralston and Ms. Sonia Taylor for all their help and support during my work under them. I would also like to take this opportunity to thank the staff of Computer Science department for their help.

I would also like to extend my appreciation towards my parents, brother, cousins and all my friends and family members for always being there for me through all phases of my work, for their encouragement and patience and giving me their invaluable love and support without which I would never be where I am today.

CHAPTER 1

INTRODUCTION

Information Retrieval (IR) is the process of extracting, representing, storing and capturing the required information [1]. However, the field of IR includes several systems of any type of unstructured data such as multimedia objects used by many users every day. An information retrieval system uses phrases to index, retrieve, organize and describe documents. Information Retrieval came into existence in the 1950s [2]. Information retrieval systems, generally called search engines, are now an essential tool for finding information in large scale, diverse, and growing corpuses such as the Internet. Information Retrieval is an essential aspect of Web search engines, when the data consists of information found on the Web. The process of indexing and retrieving text documents is known as document retrieval. The purpose of information retrieval (IR) is to provide satisfactory information needs to the users. For a given query, documents are retrieved which consists of similar query terms, based on having some number of query terms present in the document [3]. The retrieved documents are then ranked according to the frequency of occurrence of the query terms, host domain, link analysis. The purpose of information retrieval is to match the requested item partially or completely and provide the most accurate matching results. The likelihood of the relevance of the item depends on the extent of the match in IR [4].

In typical information retrieval process Figure 1.1 [5], the user

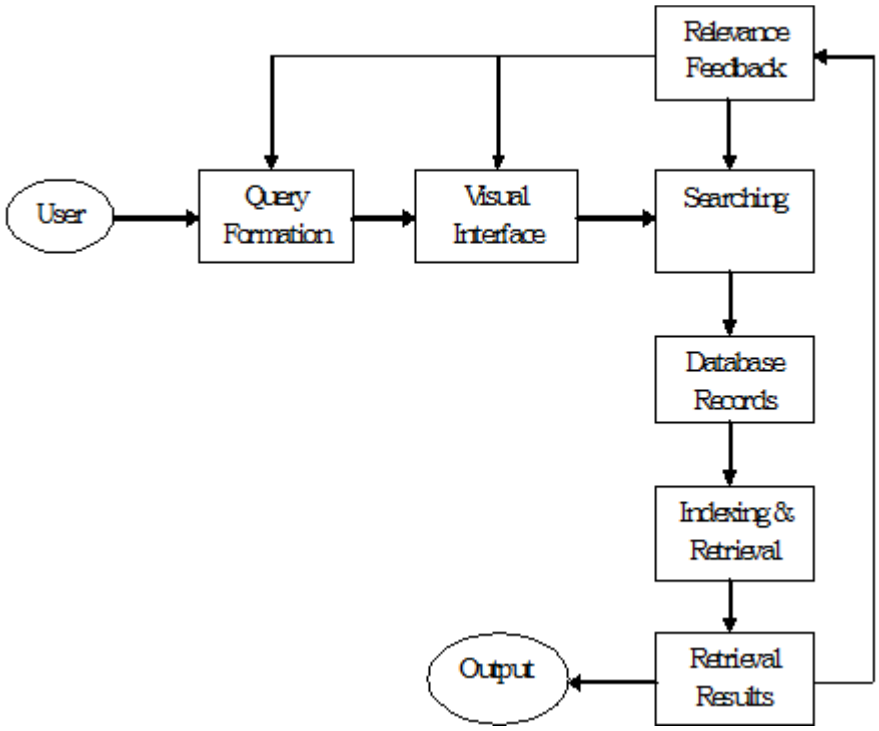

Figure 1.1 Process of Information Retrieval

gives a query and the contents of the query are searched. In the documents collected the stop words and stemming are removed and a database is formed. The purpose of indexing is to provide an efficient way to search from a large collection of the database. In order to generate meaningful retrieval results, recent retrieval systems have incorporated users' relevance feedback to modify the retrieval process. Finally the retrieval results are displayed with the aid of an indexing scheme.

Several models have been proposed to retrieve information. The three most commonly used retrieval models are the vector space model, the probabilistic model, and the inference network model [2].

The documents and queries are represented as vectors in the Vector Space Model. The success or failure of the vector space model mainly depends on term weighting [21]. Terms represent words, phrases, or any other keywords used to identify the contents of a text. This model involves constructing a vector that represents terms in the document and another vector that represents terms in a query. Next, a method must be chosen that represents the closeness between document vector and query vector. The traditional method of determining closeness between the vectors is to use the size of the angle between them. This angle can be measured using Cosine Rule. The angle between the two vectors would be zero i.e. $\Theta=0$ if the two vectors are identical which implies $\cos\Theta = 1$.

The Probabilistic Model was first presented by Maron and Kuhns in 1960 and later many different probabilistic models were proposed with different probability estimates [6]. The Probabilistic Model is based on the estimation of the probability of the relevance of the documents for a given query. In other words this model clarifies the question: what is the probability that this document is relevant to a given query [7]. The documents are ranked according to decreasing probability of relevance hence, it is known as Probabilistic Ranking Principle (PRP) [20].

The Inference Network (IN) model has the skill to execute ranking given many sources of inference by performing a combination of evidence [8]. The IN model is basically used to model documents, the document contents, and the query. The Document Network (DN) and the Query Network (QN) are the two sub-networks in the IN model. During indexing the DN is produced and it is static during retrieval. QN is produced from the query text during retrieval. The retrieval result is extracted by performing two processes. The complete IN is formed by attaching QN to DN during the attachment process and this is done when there is a similarity in the concepts of both the networks. The formation of the probability relevance to the query in the evaluation process is done by evaluating the complete IN for each document node. During the evaluation the document node's one output is initialized to 1 and rest of the other document nodes are initialized to 0. This process is applied for each document node in turn until the entire network is evaluated. Finally, the final node I is used to produce the ranking and also the probability of document relevance.

In this thesis, we start with the importance of search engines in everyday life. Next we discuss about indexing and its types. Finally we discuss about dynamic indexing, its features and importance.

CHAPTER 2

SEARCH ENGINES

Finding information has always been very difficult. After the invention of computers it has become much easier for the users to find information [9]. Internet plays a major role in the information retrieval. We can find information on the web using search engines. Search engine's purpose is to search a given query from a collection of documents and return list of documents where the query is found. In the recent years, World Wide Web search engines have vastly become a primary source for electronically retrieving information. The information maybe some sort of images, web pages, information and other types of files like media files. Once the data has been gathered, the search engines construct lexicons and indexes. When a user enters a query into the search engine the user will expect the results that match the given query. The most commonly used search engines are 'Google', 'Yahoo', and 'Alta Vista'. Search engines use 'spider' or 'crawler' to fetch the list of documents which match the given query. A crawler is an automated software agent which reads each and every site. Later the data for each web page is stored in an index. The purpose of an index is to get fast and accurate results. Whenever a query is given it is not that you are searching it in the search engine; here you are actually searching the index which is created by the search engine. As we have noticed sometimes when a query is being searched we get some dead links in the

search results. This is because the index might be created when those links were working and the index might not been updated after that so, it displays the dead links.

2.1 Vector Space Model

We already discussed a brief introduction to the vector space model in Chapter1. As mentioned earlier one of the most popular and common way to measure the similarity between document vector and query vector is known as cosine rule. The Cosine rule for ranking can be calculated as mentioned below [10].

$$\text{Cosine}(Q, D_d) = \frac{1}{W_q W_d} \sum_{t=1}^{n} w_{q,t} \cdot w_{d,t}$$

Where,

$$W_q = \sqrt{\sum_{t=1}^{n} w^2_{q,t}} \quad \text{and} \quad W_d = \sqrt{\sum_{t=1}^{n} w^2_{d,t}}$$

Here, $w_{q,t}$ represents the query term weights and $w_{d,t}$ represents the document term weights respectively. There are many different algorithms to weigh these terms and which one to choose depends on the characteristics of the collection [22]. In vector space model, each document will be represented by a vector in n-dimensional space and the query is also represented as a n-dimensional vector for any query weight, document weight or cosine measure.

Now consider a small collection of documents and calculate the cosine similarity measure to rank the documents. Here the values in the brackets indicate the number of times a term appears in a document

[11]. Term weights can be calculated in different ways. Here we use the following formulae to calculate them. The table below shows the collection of documents.

Document ID	**Text**
Doc 1	book(2) pencil(3) pen(1)
Doc 2	book(1) flower(2) ribbon(1) box(3)
Doc 3	pencil(4) ribbon (2)
Doc 4	pencil(1) pen(3) flower (5)
Doc 5	book(1) pencil(2) flower(1) ribbon(3)

Table 2.1.1 Document Collection

Using the values in the above collection we calculate the values of w_t, $f_{d,t}$, $r_{d,t}$, $w_{d,t}$, $w_{q,t}$, W_d. Here in this example the total number of documents in the collection is 5.

$w_{q,t}$ (weight of query vector) = $r_{q,t} \cdot w_t$

$w_{d,t}$ (weight of document vector) = $r_{d,t}$

$r_{d,t}$ (relative term frequency) = $1 + \log_e f_{d,t}$

$r_{q,t}$ (query term frequency) = 1

w_t (weight of the term t) = $\log_e (1+\frac{N}{f_t})$

W_d (weight of the document) = $\sqrt{\sum_{t=1}^{n} w^2_{d,t}}$

W_q (weight of the query) = $\sqrt{\sum_{t=1}^{n} w^2_{q,t}}$

Where,

N - Number of documents in the collection

f_t - Number of documents that contain term t

The below table below indicates the document vectors with calculated values of W_d, $w_{d,t}$, f_t, w_t and $r_{d,t}$.

Doc ID	**book**	**pencil**	**pen**	**flower**	**ribbon**	**box**	**W_d**
Doc1	1.69	2.09	1.0	0.0	0.0	0.0	2.86
Doc2	1.0	0.0	0.0	1.69	1.0	2.09	3.03
Doc3	0.0	2.38	0.0	0.0	1.69	0.0	2.91
Doc4	0.0	1.0	2.09	2.60	0.0	0.0	3.48
Doc5	1.0	1.69	0.0	1.0	2.09	0.0	3.03
f_t	3	4	2	3	3	1	
w_t	0.98	0.84	1.25	0.98	0.98	1.79	

Table 2.1.2 Document Vectors

The table below shows the cosine similarity measure i.e. Cosine (Q, D_d) for two queries {box} and {pencil, box} from the collection of documents. Here the W_q values are calculated for the given two queries. The ranking is simple for a single query term {box} as it appears only

once in the document collection. For the second query we need to calculate the cosine similarity measure for both the terms in the query i.e. pencil and box.

Doc ID	**box** $W_q = 1.79$	**pencil, box** $W_q = 2.18$
Doc 1	0.0	0.28
Doc 2	0.68	0.56
Doc 3	0.0	0.31
Doc 4	0.0	0.11
Doc 5	0.0	0.21

Table 2.1.3 Cosine Similarity Measures

As discussed earlier, the cosine similarity measure is based on the ranking so, the documents are sorted in the descending order of their measure [10]. For the query {box}, the top ranked document would be Document 2. Similarly for the query {pencil, box} order of ranking would be Document 2, Document 3, Document 1, Document 5 and Document 4 respectively.

The next step is indexing, but before we perform indexing some preprocessing steps must be performed to facilitate fast and accurate information retrieval. Indexing plays an important role in information

retrieval otherwise without it the search engine has to scan all the documents which results in waste of time and computing power. Indexing increases the performance and speed in searching a query from a collection of documents. As mentioned above the preprocessing steps include tokenization, removal of stop words and stemming.

Before preprocessing, collect all the documents to be indexed. In the preprocessing steps, the first step is tokenization. Tokenization is a process where sentences are broken into words known as tokens [12]. Tokens can be represented in XML. During the process of tokenization all the unnecessary characters like punctuations are eliminated [1]. Let us consider a collection of documents and perform tokenization. The example below shows a list of sentences 'The box consists of toys.' 'So, take it.'

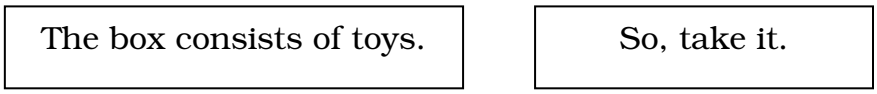

Figure 2.1.1 Collection of documents

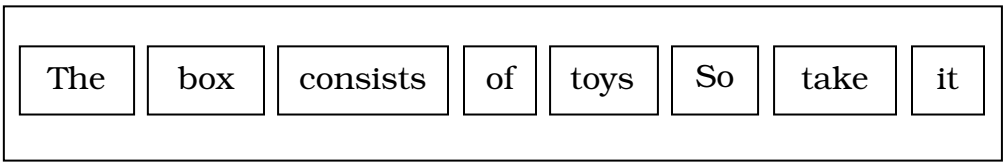

Figure 2.1.2 Output of Tokenization

After performing tokenization these sentences are chopped into a list of tokens 'The', 'box', 'consists', 'of', 'toys', 'So', 'take', 'it as shown in figure 2.1.2.

In the next step, stop words are removed from the previous step. Stop words are the frequently occurring words that are not searchable. These words include 'the', 'a', 'is', 'of', 'be', 'as', 'and', 'has' etc. As these words are not necessary search engines do not record these extremely common words in order to save index space and to speed up the searches. The figure below shows the elimination of stop words i.e. stop words are removed.

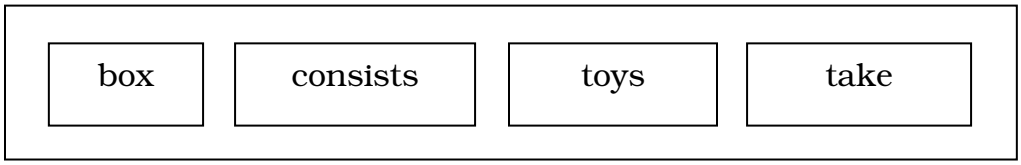

Figure 2.1.3 Output after removing stop words

The final step is stemming. Stemming is a process of reducing the words into their base or root form [13]. Stemming algorithms reduce words for example 'brighter', 'brighten', 'brightest' to their root form 'bright'. Several types of stemming algorithms are available but they differ in their performance and accuracy. A common algorithm known as Porter's Algorithm is available in several programming languages on the web [1]. After performing stemming the pre-processing steps are

completed. Now the document collection can be indexed. The figure below shows the final list of words to be indexed.

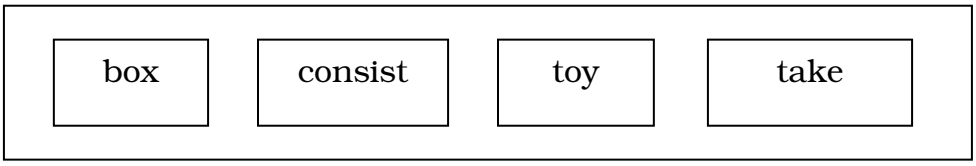

Figure 2.1.4 Output after Stemming

Now the next step after completing the pre-processing steps is to perform indexing. Indexing helps the search engine provide accurate results. The figure below [1] shows an inverted index for the collection of documents in table 2.1.1. The inverted index is created after the stemming process. The inverted index is constructed for the unique terms or tokens known as index terms. For constructing an inverted index first the terms are sorted in an alphabetical order. In the next step the corresponding posting for the first term i.e. 'book' is stored in the memory.

The postings of the remaining terms are compared against the postings in the memory. The final result must be the list of documents which has all the terms in the query. For example consider an example query 'pen, flower' then the result will be Document 4. We can say that indexing plays a major role in information retrieval.

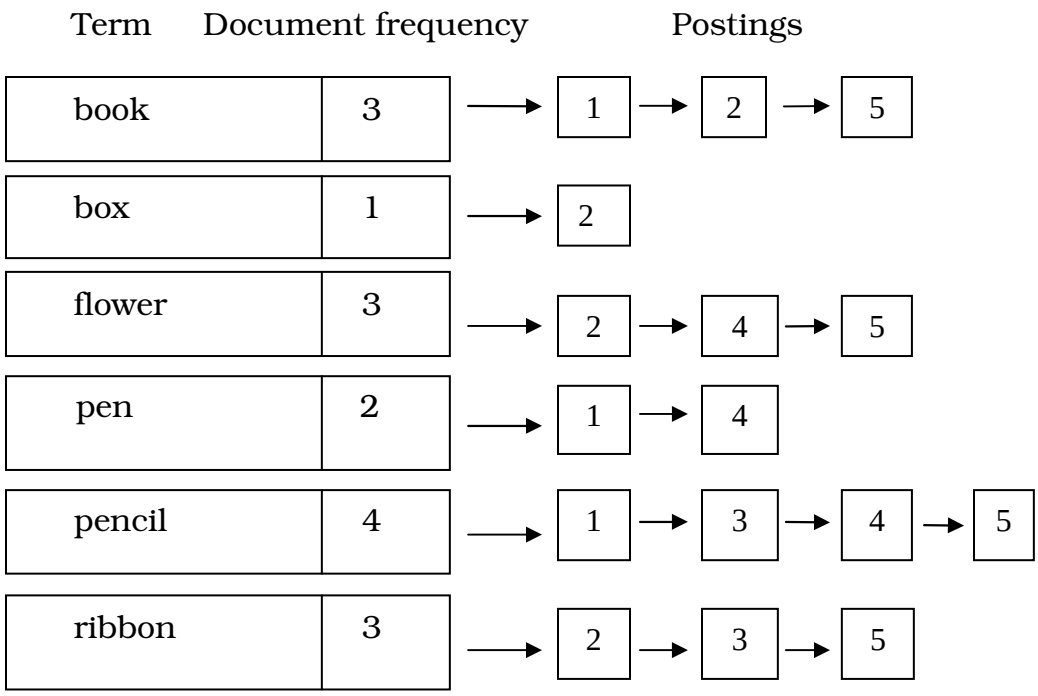

Figure 2.1.5 Inverted Index of collection

CHAPTER 3

INDEXING

As we discussed earlier information retrieval is defined as exploring the information from large documents like the World Wide Web (WWW). Using Indexing the required information is collected, parsed and stored to provide high speed and exact information retrieval [14]. Indexer is the machine that is responsible for indexing. Mostly the information retrieval designing is based on the characteristics of the hardware used. The examples and algorithms discussed in this chapter are taken from 'Managing Gigabytes'. We start with the review of the hardware basics.

3.1 Hardware Basics

The data in memory is accessed much faster than the data on disk. The time taken by a disk head to relocate to a place where the data is located is known as seek time [14]. The data must not be transferred from disk during the positioning of the disk head. Therefore it is much faster to transfer a large chunk of data from disk to memory than to transfer a lot of small chunks. We can say that disk input/output is block based as we are reading and writing entire blocks. Here the size of the blocks is 8 KB to 256 KB. The IR systems use servers with some several GB of main memory, sometimes tens of GB. The disk space available is several times larger to the order of the magnitude. Fault tolerance machines are very expensive so, regular machines can be

used as they are much cheaper. The table below shows hardware assumptions [14].

symbol	**Statistic**	**value**
s	average seek time	5 ms = 5 x 10^{-3} s
b	transfer time per byte	0.02 µs = 2 x 10^{-8} s
	processor's clock rate	10^{9} s^{-1}
p	low-level operation (e.g., compare & swap a word)	0.01 µs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

Table 3.1.1 Hardware Assumptions

3.2 Index Construction

The most challenging task while building a database is construction of an index. As we think it not that easy to construct an index, it gives rise to many problems. The process of building an index is known as the inversion of the text. As we all know inversion is nothing but reverse of a given thing or turning something upside down. To

construct an index the same procedure is used. Inversion is a familiar term used by all in many fields. For example consider a mathematician performing transposition operation which is nothing but inverting a matrix. This process of transposition is also used while constructing an index. Consider a collection of six documents as shown below [15].

Document	**Text**
1	pease porridge hot, pease porridge cold
2	pease porridge in the pot
3	nine days old
4	some like it hot, some like it cold
5	some like it in the pot
6	nine days old

Table 3.2.1 Document Collection

In the table above each line indicates a document. The text in each of the documents contains index terms and each index term appears in some of the lines. Here we express the document collection as frequency matrix where each row corresponds to one document and each column

corresponds to one word. The table below shows the frequency matrix where each document collection is summarized in one row of this frequency matrix. It shows the frequency matrix for the given collection of six documents with all the terms and the document numbers. Here rows indicate each document listed in the collection [15].

	Term												
	cold	days	hot	in	it	like	nine	old	pease	porridge	pot	some	the
1	1	-	1	-	-	-	-	-	2	2	-	-	-
2	-	-	-	1	-	-	-	-	1	1	1	-	1
3	-	1	-	-	-	-	1	1	-	-	-	-	-
4	1	-	1	-	2	2	-	-	-	-	-	2	-
5	-	-	-	1	1	1	-	-	-	-	1	1	1
6	-	1	-	-	-	-	1	1	-	-	-	-	-

Table 3.2.2 Frequency matrix

Number	Term	Document					
		1	2	3	4	5	6
1	cold	1	-	-	1	-	-
2	days	-	-	1	-	-	1
3	hot	1	-	-	1	-	-
4	in	-	1	-	-	1	-
5	it	-	-	-	2	1	-
6	like	-	-	-	2	1	-
7	nine	-	-	1	-	-	1
8	old	-	-	1	-	-	1
9	pease	2	1	-	-	-	-
10	porridge	2	1	-	-	-	-
11	pot	-	1	-	-	1	-
12	some	-	-	-	2	1	-
13	the	-	1	-	-	1	-

Table 3.2.3 Transposed equivalent of frequency matrix

To create an index the matrix must be transposed i.e. inverted to form a new version in which the rows are the terms. The inverted file can be created by building a transposed frequency matrix in memory. In the next step read the text in the order of the document column by column at a time and write the matrix to disk row by row in the order of the terms.

The table 3.2.3 above shows the transposed equivalent of frequency matrix. The table consists of terms and the corresponding term numbers and the document numbers. It shows some values which indicate the number of times each term occurs in each document. Here in the above table the document collection consists of thirteen words and there are six documents.

3.2.1 Algorithm to create an inverted file:

1. Given a collection of N documents and n terms.

   For each document $1 \leq d \leq N$.

   For each term $1 \leq t \leq n$.

   Set $f[d, t] \leftarrow 0$

2. For each document D_d

   a. Read the document parsing it into terms.

   b. For each index term $t \in D_d$

Set f [d, t] ← f [d, t] + 1

3. For each term 1 ≤ t ≤ n

    a. Start a new inverted file entry
    b. For each document if f [d, t] > 0 then add <d, f [d, t]> to the entry.
    c. Append this to inverted file.

Using an inverted frequency matrix, it is easy to construct an index. As all this approach seems to be easy, but in reality this process is difficult to implement because of the size of the frequency matrix. As the size of the document increases, the size of the frequency matrix also increases. For example consider that the text Bible has to be inverted. Collection Bible is the King James Version of the Bible, with each verse taken to be a document, including the book name, chapter number and verse number. The Bible contains 31, 101 documents and 8,965 distinct terms. If for each entry in the frequency matrix a four-byte integer is allowed then, the matrix will occupy 4*8,965*31,101 bytes of main memory. This is barely managed on a large machine as it comes to more than 1 Gigabyte. For TREC (Text Retrieval Conference) collection, the size of the matrix becomes more difficult if a four byte integer is allowed for each entry i.e. 4*535,346*741,856 bytes or 1.4 Terabytes [15].

Supposing that one byte is sufficient to record each within-document frequency $f_{d,t}$ (for TREC it is not adequate) does not help either the space requirements for the two collections which are 250 Mbytes and 350 Gbytes respectively and the algorithm still is not viable. Boolean matrix is sufficient if only a Boolean access is required. The frequencies can be reduced to 31 Mbytes and 46 Gbytes but it still requires a large amount of memory. A machine with large virtual memory can be used and the operating system can be responsible to page the array into and out of memory as required. There will be one page fault for each pointer in the index due to the column-by-column access when the matrix is created. To build a Bible index it requires about 700,000 page faults at the rate of 50 page replacements per second, which requires 1400 seconds i.e. about 4 hours [15].

The virtual memory subsystem of a processor implements the virtual address spaces provided to each process [16]. Each process has one page table and during the execution process it is completely loaded into the main memory. There are few page tables which cannot be fully held in main memory as their processes as very large. For example each process can have a virtual memory of up to 2^{32} = 4 Gbytes in a 32 bit x 64 architecture. For example consider a two-level scheme with 32 bit address. Consider 4 Kbyte pages then the offset part of virtual address is 12 bits in size then this will leave 20 bits as the selector of the page directory and a table with 2^{20} entries is not practical. If each page table

requires 4 bytes, then a page table with 2^{20} entries requires 4 Mbytes. Page fault occurs when a page is not in the main memory and later that page should be loaded by the operating system.

In TREC Collection

Number of documents= $5*10^6$

Number of distinct terms= $1*10^6$

To read the entire text, parse and filter through the dictionary takes 5 hours. During this time, the temporary file is written, containing 400 million 10-byte records.

cold <t, d, $f_{d,t}$ > takes 12 bytes

This takes half hour. The temporary file is sorted, if for 48 Mbytes of main memory, k ≈ 4,000,000. Use quick sort, 1.2 k log k ≈ 110 seconds. Total sorting is 3 hours. During this internal sorting, the entire temporary file is both read and written, so another hour should be allowed to cover reading and writing. Sorting the temporary file takes 13 hours. Finally, the temporary file is again read, and written to disk. This takes $1\frac{1}{2}$ hour. So the complete inversion takes 20 hours.

Algorithm

To produce an inverted file for a collection of documents [15].

1. /* Initialization */

Create an empty dictionary structure S. Create an empty temporary file on disk.

2. /* Process text and write temporary file */

   For each document D_d in the collection $1 \leq d \leq N$

   a. Read D_d, parsing it into index terms.
   b. For each index term $t \in D_d$

      i. Let $f_{d,t}$ be the frequency in D_d of the term t
      ii. Search S for t
      iii. If t is not in S, insert it
      iv. Write a record <t, d, $f_{d,t}$ > to the following temporary file, where t is represented by its term number in S.

3. /* Internal sorting to make runs */

   Let k be the number of records that can be held in main memory.

   a. Read k records from the temporary file.
   b. Set into non-decreasing t order and for equal values of t, non-decreasing d order.
   c. Write the sorted run back to the temporary file.
   d. Repeat until there are no more runs to be sorted.

4. /* Merging */

Pair wise merge run in the temporary file until it is one sorted run.

5. /* Output the inverted file */

   For each term t, $1 \leq t \leq n$

   a. Sort a new inverted file entry.

   b. Read all triplets < t, d, $f_{d,t}$ > from the temporary file for t.

   c. Append the inverted file entry to the inverted file.

The sorting algorithm is not efficient for large collections [15]. For the example inversion, each of these contains about 10 x 400 million bytes, which requires a total of 8 Gbytes of disk space at the peak of the process. This accounts to more than 20 times the size of the index that is eventually produced and 60 percent larger than the text being inverted. Of course the text being inverted is probably stored compressed and also the temporary disk space required is more than twice the space required to store raw collection. As the requirement of disk space is more we can say that the sort based inversion is suitable for moderate collection of documents of size between 10 to 100 Mbyte ranges. This is not applicable for large collections which are gigabyte range.

For the document collection pease, porridge, ... sort-based inversion is performed and the values are retrieved.

Term	Term Number
cold	4
days	9
hot	3
in	5
it	13
like	12
nine	8
old	10
pease	1
porridge	2
pot	7
some	11
the	6

Table 3.2.4 Sort based inversion for the text

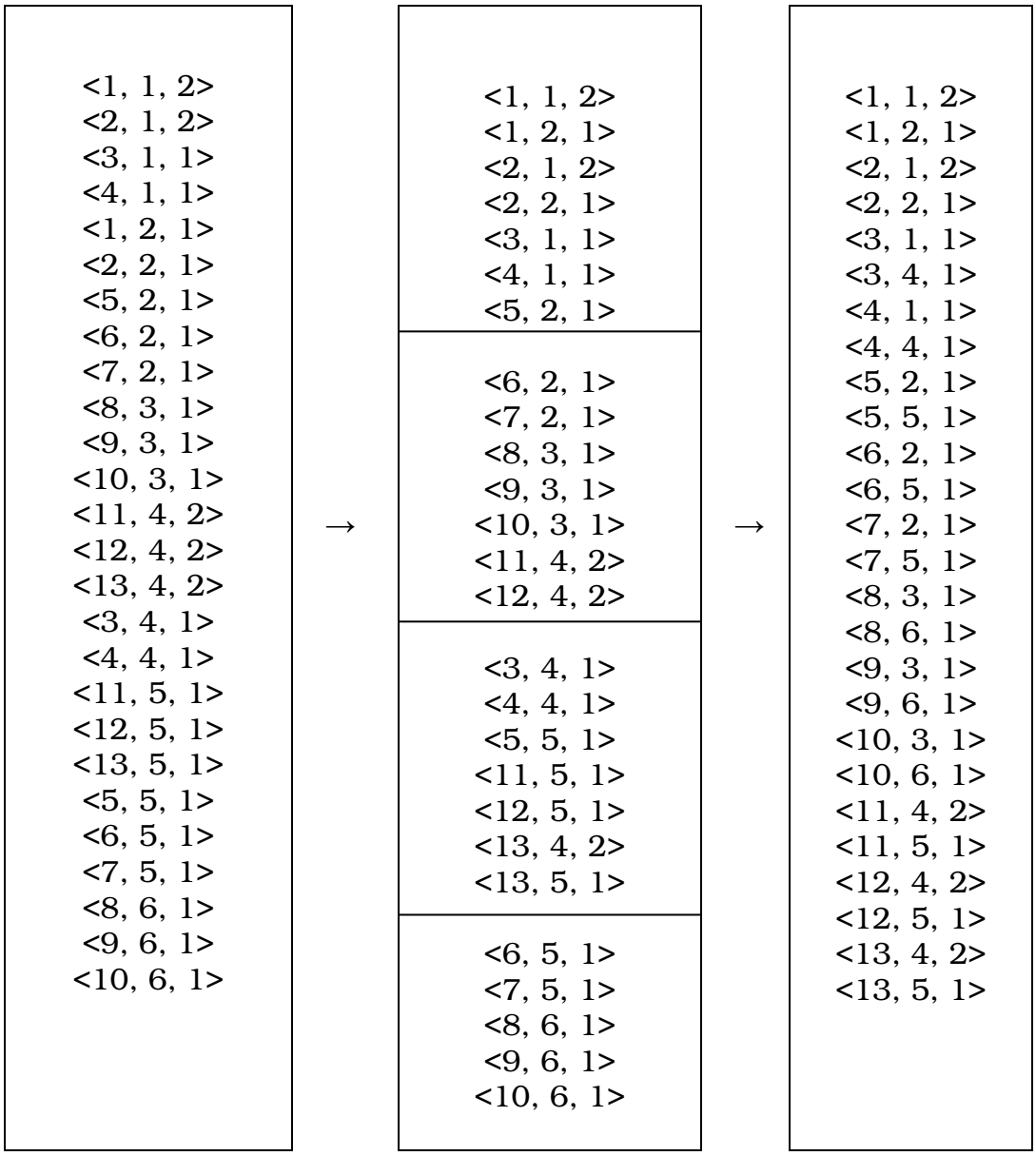

Table 3.2.4 Sort based inversion

CHAPTER 4

DYNAMIC INDEXING

In the previous chapter we discussed that the document collections were static. Most of the indexing techniques are 'static' as they are performed in two phases [19]. To build temporary internal files, input files are read during the first phase. In the next phase these temporary internal files are optimized to prepare for retrieval. Once the optimization is finished the indices are static so, it is not possible to add new documents without rebuilding the whole new index. Also, the queries for the retrieval of documents cannot be completed until the second phase of indexing is performed.

To overcome the limitations caused by static indexing techniques, dynamic indexing has been introduced. Now we discuss about the document collections which are dynamic. Static indexing can be used for document collections which do not change and remain same and we find such collections in rare cases. Each time for a query to be retrieved the indexes which are present in the index files are checked without the optimization of the internal files. In dynamic indexing, the postings for the words are stored in an index file which is organized into a set of fixed length blocks. These blocks are the ones which pack the postings for much of the words together with free space being more or less. The block numbers for each posting are stored in an address record table. A free

block list is kept for the blocks which contain enough amount of free space and which store information related to them.

Dynamic indexing helps in providing methods for indexing a collection of documents in a single phase. Using this, the queries are retrieved without optimizing and generating internal files. The postings for a word are stored sequentially in memory in order to retrieve the postings from memory by performing less number of input/output operations and allow retrieval at all times. According to one aspect of dynamic indexing, the words found in the documents of a database are allocated with blocks of index file to the postings. The index file is allocated with a predetermined initial block size and further the block is divided into blocks with decreasing sizes successively. For a successive level, each block is divided into n blocks of equal size [19]. The size of the initial block is the sum of the sizes of blocks in each of the successive levels.

There are many collections where documents are added, deleted and updated i.e. which change frequently. Whenever new documents are added to the database then the collection of indexes becomes large and it takes time for the index file to get updated. Blocks of index files are allocated to the postings for words that are contained in the index file in an information retrieval interface. The word in the first block of the index file is updated by the information retrieval interface. The postings which

are updated consist of some additional postings for the word in the documents which are added to the database [17].

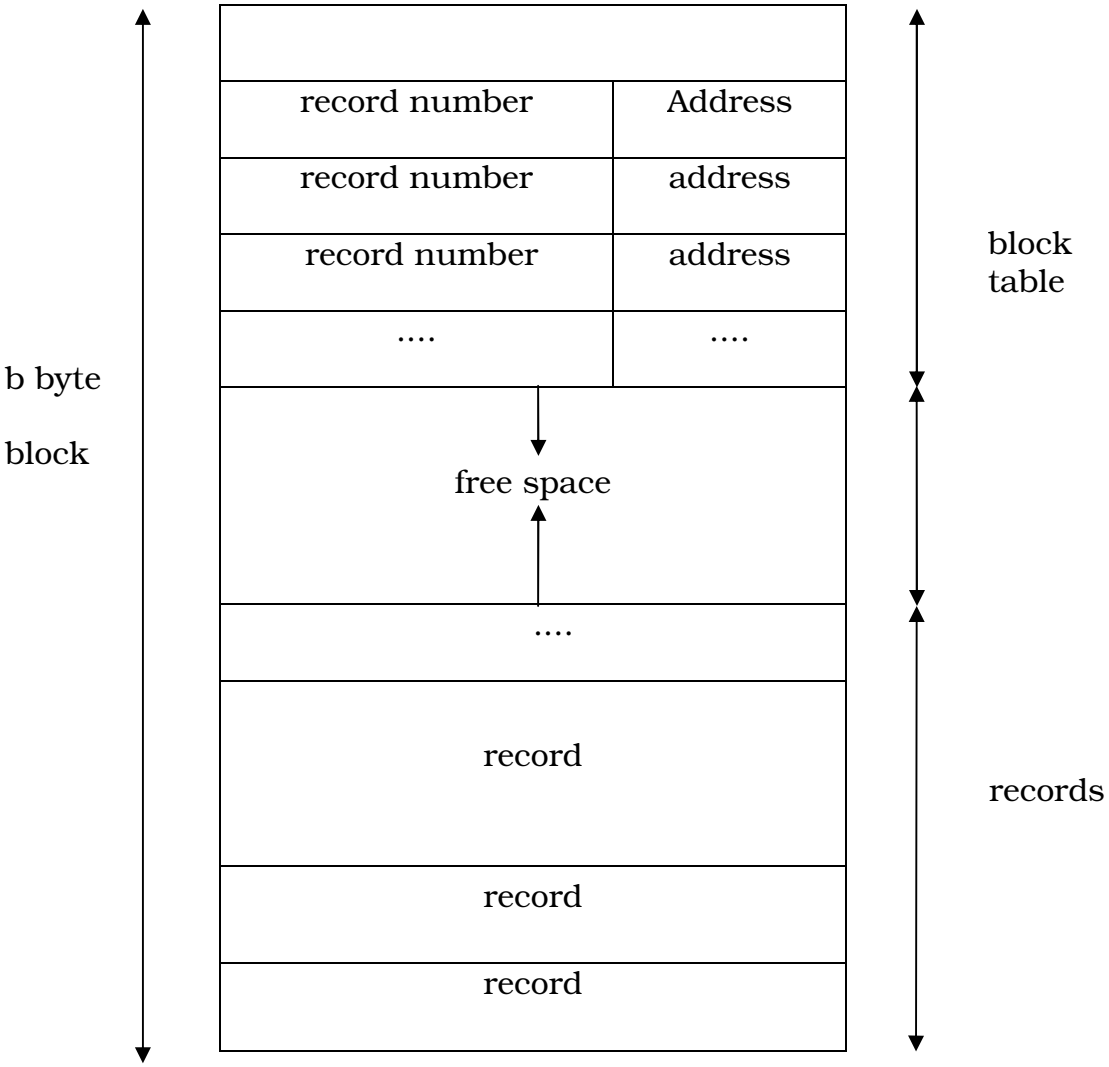

Figure 4.1 Block Structure

From a free block list a second block is searched by the information retrieval interface which is free to accommodate the updated postings for the word. The free block list contains information which

indicates whether or not a block is free. The postings for the word are moved from the first block to the second block by the information retrieval interface.

In the figure above, each block contains a block address table, some records and some free space. The number of records stored in the block and also for each record stored, the record number and an address within the block for the record is listed by the block address table [19]. The records themselves are packed at the other end of the block from the block table, and there is some free space between the block table and the records. In the memory, a record address table is maintained that stores, for each record number, the block number currently containing that record. Also in memory is a free list that describes blocks that currently have an amount of free space greater than a given tolerance. Finally, the current last block of the file is kept in main memory rather than on disk.

The record address table is used to find the correct block number to access a record with given ordinal record number. The block address table searches for the record number and the whole block is read into memory. This yields the address of the record within the block, so the record can then be located and the contents used. Now consider the problem of extending a specific record. First, the block which consists of the record is retrieved. The extended record can still be accommodated, if the block contains sufficient free space such that the records are linearly shifted in the block to make the correct space, the extension added to the

record, the block table updated, and the altered block written. This may also have some effect on the free list.

If there is insufficient free space within the original block, then the smallest record can be deleted whose removal leaves enough space for the extended record from the block. If there is no such record, then the record being extended is removed. Again, the block table and free list should be updated, and the block must be written back to the disk. In this case, however, still extant is a record that has no block i.e. either a record that was removed to make space for the extension or the newly extended record itself. This record is treated as an insertion.

A record is inserted by consulting the free list and determining if there is any block in the file that has space. If there is, that block is retrieved from disk; the record is inserted; the block table, record address table, and free list are all updated and the block is written back to the disk. In this case, however, still extant is a record that has no block i.e. either a record that was removed to make space for the extension or the newly extended record itself. This record is treated as an insertion.

To insert a record, the free list is consulted to determine if there is any block in the file that has space for it. If there is, that block is retrieved from disk; the record is inserted; the block table, record address table, and free list are all updated; and the block is written back to disk. If there is not if all the blocks on the free list are sufficiently full that they cannot absorb this record attention is switched to the last record, it is

inserted and the various tables are updated. If it cannot, the last block is written and perhaps added to the free list, and a new, completely empty, last block created in memory. Finally, the record can be inserted into this empty block.

In most cases, a record extension can be carried out with one block read and one block write. The worst that can be required is four disk operations: a block read to been removed from that block; a block read to retrieve a block that does have enough is sufficiently high that in this raw form the scheme is not likely to be useful.

For example consider collections like The Complete Works of Shakespeare, dictionaries, encyclopedia etc which have undergone many changes with new information is being discovered and added. For such collections, each time the posting lists and the dictionary should be updated whenever there are any changes made to the collection. These modifications can be done to the index by reconstructing it from the beginning. This can easily be done if the modifications are small and the delay caused in searching new documents is acceptable.

We can say a collection to be dynamic for one of the two ways. To append a new document to the existing collection an 'insert' operation has to be used which adds a new document without changing the previous collection. When a document contains many words and is inserted into a database, then the postings of all these words are expanded in a manner of multipoint insertion rather than a simple

append operation. Also, the 'edit' operation is important using which changes can be made to the existing collection and unnecessary documents can be removed. The problem of reconstructing a new index can be solved by maintaining two indexes; one is a small auxiliary index for storing new documents which is stored in main memory and second is a large main index. The required information is retrieved by performing a search process in both the indexes and the final results are merged. There is an invalidation bit vector which stores all the deleted documents. The search final results are displayed after removing the deleted documents. We can say a document to be updated when it performs insertion and deletion operations. This process helps the information retrieval system to dynamically index a collection of documents in the database.

The auxiliary index is merged into the main index whenever it becomes too large and the cost of merging depends upon the storage of the index in the file system. The merge includes only extending each of the auxiliary index postings list with its corresponding postings lists of the main index, if each postings list is stored as a separate file. The auxiliary index is mainly used to reduce the number of disk seeks that are necessary over time. We require M_{ave} disk seeks to update each document separately. Here M_{ave} represents the average size of the vocabulary of documents in a collection. An additional load is put on the disk for with an auxiliary index, when the main index and auxiliary index

are being merged. Large number of files cannot be handled by most of the file systems, because of this one-file-per-postings-list scheme is infeasible. To overcome this, the entire postings list can be concatenated i.e. the index is stored as one large file [14].

4.1 Algorithm Logarithmic Merging

LMERGEADDTOKEN (indexes, Z_0, token)

1 $Z_0 \leftarrow$ MERGE (Z_0, {token})

2 if $|Z_0| = n$

3 then for i $\leftarrow$ 0 to ∞

4 do if I_i □ indexes

5 then $Z_{i+1} \leftarrow$ MERGE (I_i, Z_i)

6 (Z_{i+1} *is* a temporary index on disk)

7 indexes $\leftarrow$ indexes − {I_i}

8 else $I_i \leftarrow Z_i$ (Z_i becomes the permanent index I_i)

9 indexes $\leftarrow$ indexes □ {I_i}

10 BREAK

11 $Z_0 \leftarrow \Phi$

LOGARITHMICMERGE ()

1 $Z_0 \leftarrow \Phi$ (Z_0 is the in-memory index)

2 indexes $\leftarrow \Phi$

3 while true

4 do LMERGEADDTOKEN (indexes, Z_0, GETNEXTTOKEN ( ))

In this algorithm each token is added to Z_0, the in-memory index by LMERGEADD TOKEN. The LOGARITHMICMERGE initializes Z_0 and then indexes. Here each posting is processed [T/n] times as it is touched during each [T/n] merges. Here n represents the size of the auxiliary index and T represents the total number of postings. Here the docIDs are considered and the representation of terms is neglected. Hence we can say that Θ (T^2/n) gives the overall time complexity. For this purpose, it can be said that the postings list is nothing but a list of docIDs.

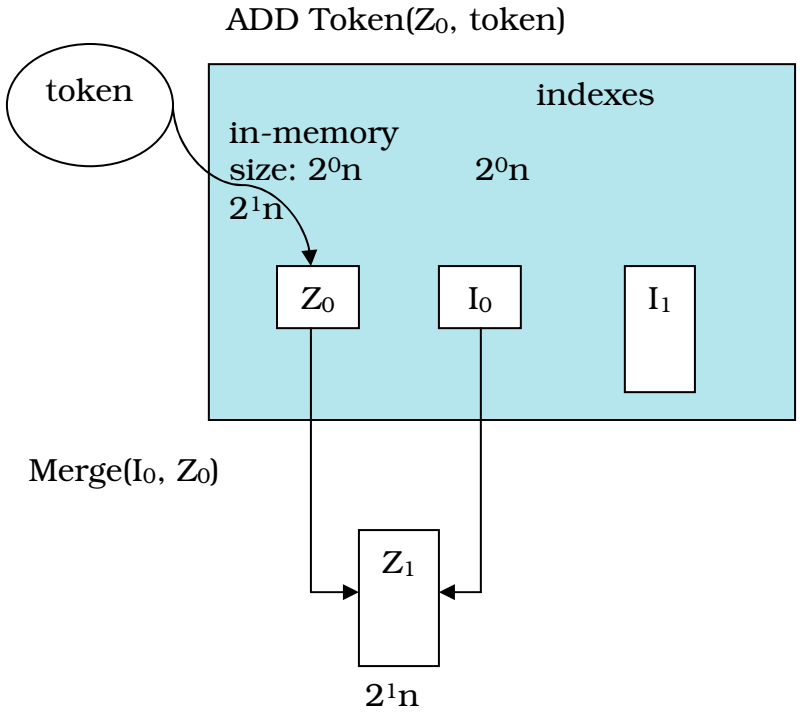

Figure 4.2 Logarithmic merging of I_0 and Z_0

The overall time complexity of Θ (T^2/n) can be made much better by advancing $\log_2$ (T/n) with indexes I_0, I_1, I_2, I_3,.... with sizes 2^0 x n, 2^1 x

n, 2^2 x n, 2^3 x n …. On each level the postings are processed only once and are percolated up this sequence of indexes. We call this scheme as logarithmic merging.

The logarithmic algorithm is discussed above. As discussed above an in-memory auxiliary index can accumulate up to n postings which we call as Z_0. After reaching a limit n, a new index I_0 is created on the disk.

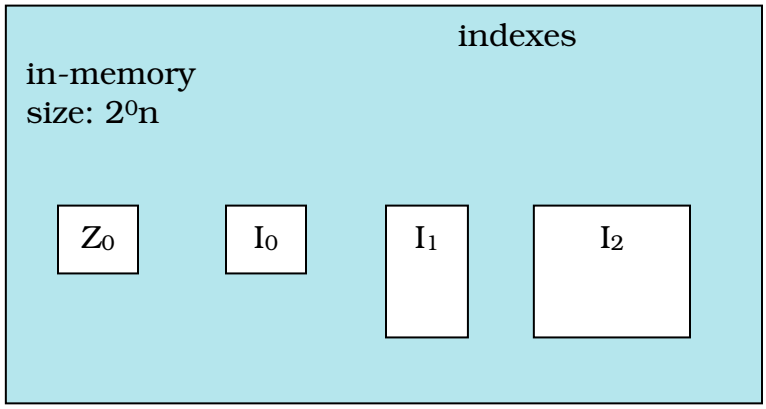

Figure 4.3 In-memory and indexes

and the 2^0 x n postings in Z_0 are transferred into this new index I_0. If Z_0 is full, a new index known as Z_1 of size 2^1 x n is created by merging Z_0 with I_0. If there is no I_1 existing then Z_0 is stored as I_1 or if I_1 exists then Z_1 is merged with I_1 into Z_2 and so on. The in-memory Z_0 is queried by servicing all currently valid indexes I_i and search results on disk and merging the results [18].

Each posting is processed only once on each of log (T/n) levels hence, the overall index construction time is Q (T log (T/n). This efficiency gain can be traded for a slowdown of query processing and the results from log (T/n) indexes need to be merged as it is opposed to the main and auxiliary indexes. The very large indexes should be merged occasionally in the auxiliary scheme and this results in slow down of the search system during the merge. This process occurs less frequently and the indexes present in a merge on an average are small.

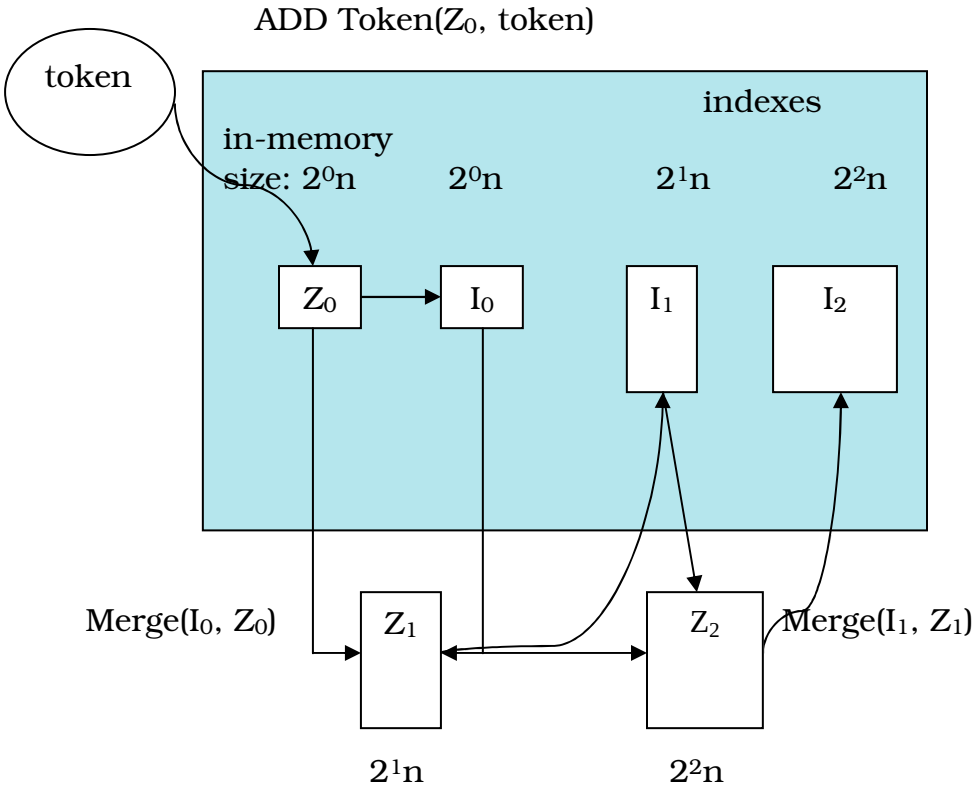

Figure 4.4 Logarithmic merging of (I_0, Z_0) and (I_1, Z_1)

The figure 4.5 below shows a block structure where each block represents a portion of memory for the index file [19]. Each level consists

of blocks which are of equal sizes. The index file initially consists of a block with some predetermined size and it is divided into various blocks of successive sizes. The figure ranges from high level to low level. Here

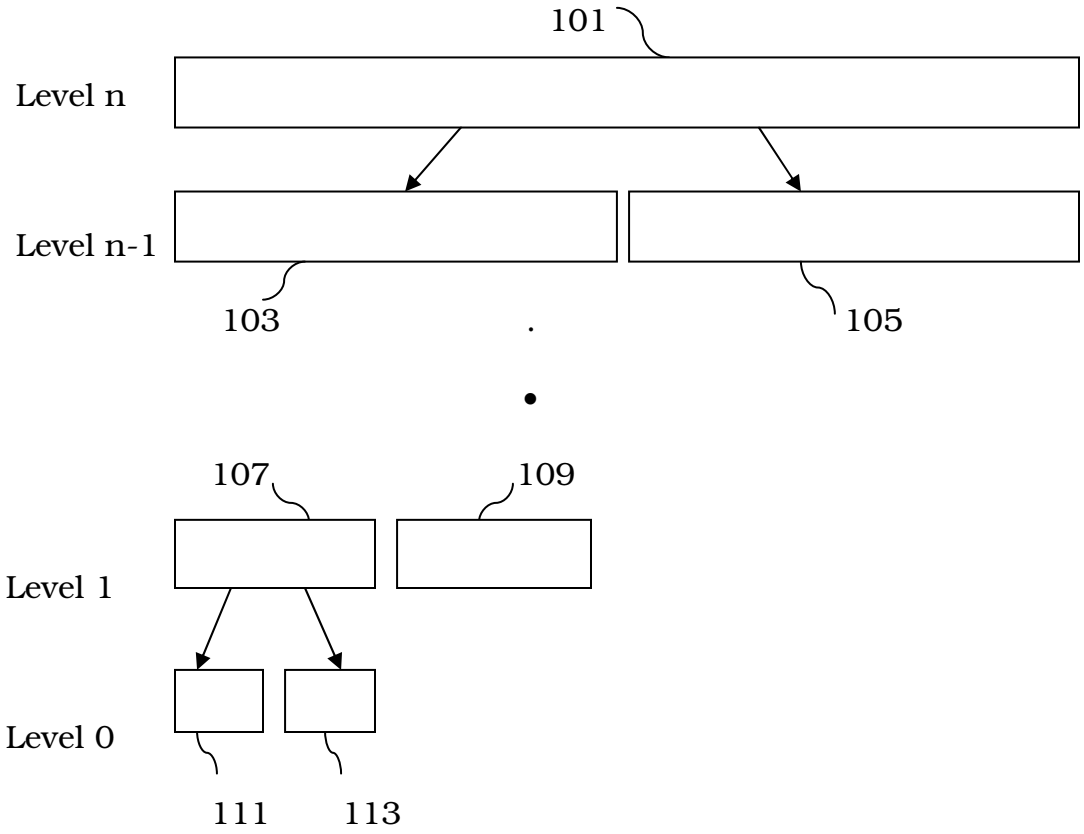

Figure 4.5 Structure of index file

the higher level is level n and the lower level is level 0. The higher level consists of a single block which is large in size whereas the lower level consists of smaller blocks with minimum size. The amount of memory that is wasted during fragmentation can be minimized by the smaller blocks with minimum size. Here the block 101 is in the higher level i.e.

level n and in level n-1; it is portioned into two blocks 103 and 105 of equal sizes. Each higher block is partitioned into two blocks of equal sizes. In the level 1 the block 107 is partitioned into two blocks 111 and 113 of equal sizes in the lower level i.e. level 0. The blocks 103, 105, 111 and 113 are the child blocks whereas the blocks 101 and 107 are the parent blocks. The size of parent blocks is twice the size of the child blocks as each parent is divided into two child blocks. The size of the block in higher level n is 2^n, the size of the block in lower level 0 is 2^0 i.e. 1 and the size of the block in level 1 is 2^1 i.e. 2. The parent block is thrice the size of the child block if it has 3 children. The size of the block in lower level 0 will be 1 and the size of the block in the higher level n will be 3^n. When the index file is opened then the information is read from the secondary memory into the main memory and when the index file is closed then the information kept in the main memory is written back to the secondary memory.

For example, consider an index file allocating a block to the postings list for a word from the document collection. Consider the document collection as shown below. There are four documents in the collection.

doc1	pen, pencil, box, cap
doc2	cap, duck, ball
doc3	pen, duck, box, drum
doc4	pencil, box, ball, cap

Now consider the word ‘box’ is seen in many documents. It appears in doc1, doc3 and doc4 respectively so, the postings of ‘box’ are [doc1, doc3, doc4]. The index file is partitioned into blocks to store the postings of ‘box’.

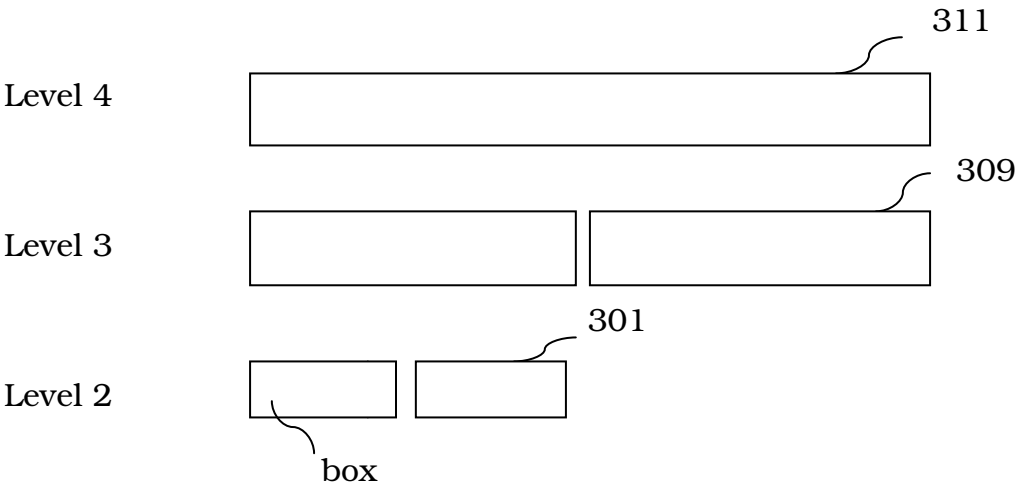

Figure 4.6 Index file allocating blocks

The block structure is partitioned to allocate the word ‘box’ in level 2. The largest block 311 in the level 4 is deleted and a new block 309 and 301 are added in the level 3 and level 2 respectively.

The figure 4.7 below shows the indexing file allocation two words. Now consider the word ‘duck’ which is present in doc2 and doc3 and the postings for ‘duck’ are [doc2, doc3] respectively. The block 301 is partitioned into blocks 303 and 305 in level 1. The posting for ‘duck’ are allocated to block 303 in level 1. The block 301 is removed from the free block list and block 305 id added in level 1.

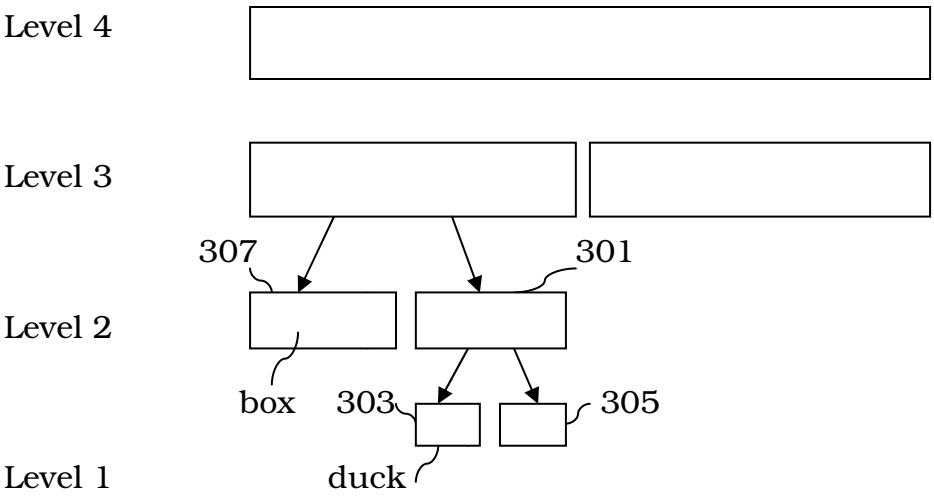

Figure 4.7 Index file allocating blocks with two words

The maintenance of collection wide statistics becomes complicated when there are multiple indexes. For example, the spelling correction algorithm gets affected which selects the corrected alternative with the most hits [14]. It is no longer a simple lookup for the correct number of hits for a term with multiple indexes and an invalidation bit vector. In logarithmic merging, the aspects of an IR system i.e. query processing; index maintenance, distribution etc. are more complex. Some of the large search engines allow a reconstruction from scratch strategy due to the complexity of dynamic indexing. So, they do not construct indexes dynamically; instead a new index is built from scratch periodically. Finally while processing a query the old index is deleted and searched using the new index.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The main objective of this thesis was to survey the importance of indexing and especially dynamic indexing in retrieving information. We discussed about the various procedures involved in retrieving information. First we discussed about search engines and vector space model in chapter 2 and discussed the importance of indexing in retrieving information. In chapter 3 we discussed about different types of indexing and their drawbacks. Finally we discussed about dynamic indexing and how it is used in retrieving information from large document collections. The document collections require frequent changes and this can be done using dynamic indexing and modifications made in the collections can immediately be visible in the index.

Dynamic indexing technique is mainly focused on large document collections and to reconstruct the index from scratch when new documents are added to the database and the old one is deleted. Different operations can be used in building the index like insert, delete, update etc.

The document collections which require frequent changes can be modified using dynamic indexing and modifications made in the collections can immediately be visible in the index. In future it can be capable of making indexed documents available for query immediately

after they are indexed, which typically can take a small fraction of a second.

BIBLIOGRAPHY

1. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, 'Introduction to Information Retrieval', Chapters 1, 6, 8, 9, 11 & 12, Cambridge University Press, 2008. http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html.

2. Amit Singhal, 'Modern Information Retrieval: A Brief Overview', IEEE Data Engineering Bulletin, Volume 24, pages 35-43, 2001.

3. C. J. Van Rijsbergen, 'Information Retrieval', Second Edition, Chapters 1, 6 & 7, Information Retrieval Group, University of Glasgow, London: Butterworths, 1979.

4. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 'Managing Gigabytes', Second Edition, Chapter 4, Morgan Kaufmann Publishers, Inc, San Francisco, May 1999.

5. Ricardo Baeza Yates, Berthier Riberio Neto, 'Modern Information Retrieval', Chapter 1, Addison Wesley, Addison Wesley Longman, 1999.

6. K. Sparck Jones, S. Walker and S. E. Robertson, 'A Probabilistic model of Information Retrieval: Development and Comparative Experiments', Part 1, Information Processing and Management: an International Journal, Pergamon Press, Inc, Volume 36, Issue 6, January 2000.

7. ChengXiang Zhai, 'A Brief Review of Information Retrieval Models' October 2007.

8. Andrew Graves, 'Video retrieval using an MPEG-7 Based Inference Network', Department of Computer Science, University of London, August 2002.

9. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 'Managing Gigabytes', Second Edition, Chapter 10, Morgan Kaufmann Publishers, Inc, San Francisco, May 1999.

10. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 'Managing Gigabytes', Second Edition, Chapter 4.4, Morgan Kaufmann Publishers, Inc, San Francisco, May 1999.

11. Kazem Taghva, 'Database Management Systems', Lecture Notes, Fall 2009, University of Nevada, Las Vegas.

12. HappyCoders,'TokenizingJavasourcecode'(n.d.)
http://www.java.happycodings.com/Core_Java/code84.html

13. Martin Porter, 'The Porter Stemming Algorithm', Jan 2006.
http://tartarus.org/~martin/PorterStemmer/.

14. Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze, 'Introduction to Information Retrieval', Chapter 4, Cambridge University Press, 2008.
http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html.

15. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 'Managing Gigabytes', Second Edition, Chapter 5, Morgan Kaufmann Publishers, Inc, San Francisco, May 1999.

16. Ulrich Drepper, 'Article on Virtual Memory', October 2007.
https://lwn.net/Articles/253361/

17. Ian H. Witten, Alistair Moffat, and Timothy C. Bell, 'Managing Gigabytes', Second Edition, Chapter 5.7, Morgan Kaufmann Publishers, Inc, San Francisco, May 1999.

18. Hatena:: Diary:: Naoya, 'Logarithmic Merging', May 2009.
http://d.hatena.ne.jp/naoya/20090512/logarithmic_merging
http://bloghackers.net/~naoya/ppt090512logarithmic_merging.ppt

19. Frank Smadja, Haifa (IL), 'Dynamic Indexing Information Retrieval or Filtering System', February 2004.

20. S. E. Robertson, C. J. van Rijsbergen and M. F. Porter, 'Probabilistic models of Indexing and Searching', Proceedings of the 3rd annual ACM conference on Research and development in information retrieval, Cambridge, England, Page(s): 35 - 56, June 1980.

21. Dik L. Lee, Huei Chuang, Kent Seamons, 'Document Ranking and the Vector- Space Model', IEEE, Volume 14, Issue 2, Page(s): 67 - 75, March/April 1997.

22. Kazem Taghva, Julie Borsack and Allen Condit, 'Effects of OCR Errors on Ranking and Feedback Using the Vector Space Model', Information Science Research Institute, UNLV, Inf. Proc. and Management, 32(3): 317-327, 1996.

VITA

Graduate College
University of Nevada, Las Vegas

Viswada Sripathi

Degrees:
Bachelor of Technology in Computer Science and Engineering, 2008
Jawaharlal Nehru Technological University, India

Master of Science, Computer Science, 2010
University of Nevada, Las Vegas

Thesis Title: Dynamic Indexing

Thesis Examination Committee:
Chair Person, Dr. Kazem Taghva, Ph.D.
Committee Member, Dr. Ajoy K. Datta, Ph.D.
Committee Member, Dr. Laxmi P. Gewali, Ph.D
Graduate College Representative, Dr. Muthukumar Venkatesan, Ph.D