Space and Time Improvements for Indexing in Information Retrieval

Willie Rogers, Gerald Candela, and Donna Harman
National Institute of Standards & Technology
Building 225, Room A216
Gaithersburg, MD 20899
{rogers, harman, jerry}@magi.ncsl.nist.gov

Abstract

When indexing large text collections minimizing the indexing time and the disk storage used to create an index remains important. Indexing optimizations applied to a prototype retrieval system at NIST are discussed in this paper. These include the organization of the index, the use of virtual memory facilities to improve indexing time, an index addressing scheme to decrease index size, and the implementation of term position information extensions using compression. These improvements provided a large decrease in indexing time and moderate decrease in index size for indices without term position extensions. Indices using term position extensions had a more moderate increase in space/time efficiency.

1 Introduction

As computers grow exponentially faster, and disk drives become more compact and inexpensive, it seems that efficiency should be less important. However, this is not so, at least in the information retrieval community. If available disk space is growing, so is the amount of text to process. Whereas 25 years ago it was a major undertaking to process the 1400 Cranfield abstracts, today research involves test collections of over a million documents and real-world applications process far larger amounts of text. The space and time optimization issues remain important.

The need for time optimization is the most obvious. Users expect very fast search response times, on the order of 1 or 2 seconds. But equally critical is the need for optimization of indexing time.

Efficiency of creating (as opposed to updating) an index is certainly necessary for processing the large TIPSTER collection (Harman 1993). Indexing runs taking several days are prone to system problems, and liable to impede research. In real-world applications, the ability to quickly re-index large collections overnight or over a weekend means that most data can be viewed as nearly-static, allowing efficiencies in index storage space.

Indexing time has two particular components: the time to create an index and the time to update such an index. This paper addresses only the first component, assuming a static or nearly static data collection. For approaches to indexing highly dynamic collections, see Schäuble (1993) and Anick and Flynn (1993).

In addition to time optimization, there

are space optimization issues to consider, both in memory and disk storage. Most researchers (and many users) rely on workstations with 16 or 32 megabytes of memory, and disk space seldom seems adequate for indexing. This means that the space needed to create the index must be minimized and the space needed to store the final indices should be as small as possible. The size of the final index impacts the search time and also determines the hardware needs for retrieval. For example, distributed searching is most "usable" if the index can be distributed across multiple workstations (eliminating heavy network traffic). Also, applications using CD-ROM require small indices for most effective use.

The prototype retrieval system being developed at the National Institute of Standards and Technology (NIST) has always emphasized the need for small indices, minimal memory and disk requirements for index creation, and minimal search time. Section 2 of this paper describes the basic NIST system (the PRISE system), including the indexing techniques. The TIPSTER collection was much larger than collections previously indexed using the PRISE indexing programs. To build the index in a reasonable amount of time, the indexing programs required much faster routines than the current system provided. The techniques involved to do this are described in section 3. The very large number of documents in the TIPSTER collection caused an explosion of the final index size, and section 4 discusses how this problem was resolved. Section 5 discusses the inclusion of positional information in the index, and section 6 discusses the effect of these inclusions on search time.

2 Indexing in The NIST PRISE System

The NIST PRISE system was initially developed as a prototype testing vehicle for demonstrating that the statistical ranking techniques developed in past laboratory experiments (Salton and McGill 1983) could be implemented efficiently, and that users would accept this manner of text retrieval (Harman and Candela 1990). The efficiency part of this work required devising fast indexing algorithms that operated from workstations (without tape drives) and creating fast searching operations (with 1 second response time for about 1 gigabyte of text searching).

The PRISE indexing technique is based on a two-step process that does not need an explicit sorting step. The first step parses the text of the collection and produces the basic inverted file (intermediate postings file) and binary term tree (intermediate dictionary); and the second step adds the term weights to the inverted file and reorganizes that file for maximum efficiency (see Figure 1).

The creation of the basic inverted file avoids the use of an explicit sort by using a term-based right-threaded binary search tree (Knuth 1973). As each term is identified by the text parsing program (`build`), it is looked up in the binary tree, and either is added to the tree, along with related data, or causes tree data to be updated. The data contained in each binary tree node is the current number of postings (the number of records containing one or more instances of the term) and an offset to where the postings for that term begin. Each node also contains a left link and a right link that point to other nodes in the tree.

Figure 2 shows an illustration of the right-threaded tree. The root node of the tree contains the term "high" which occurs twice in the text of the collection. The postings for that term start at the offset `1464` in the postings file. Similarly, the node referenced by the left link of the root node contains the term "billion" which also occurs twice. The postings for that term start at `4549`.

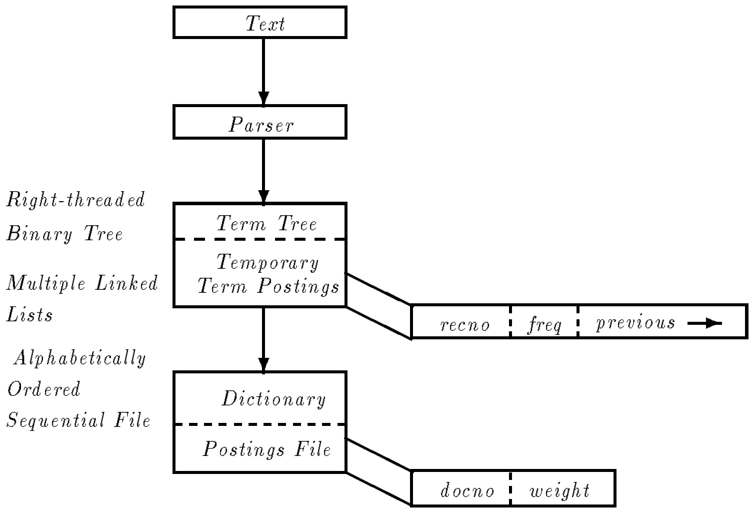

Figure 1: Flowchart of PRISE Indexing Method

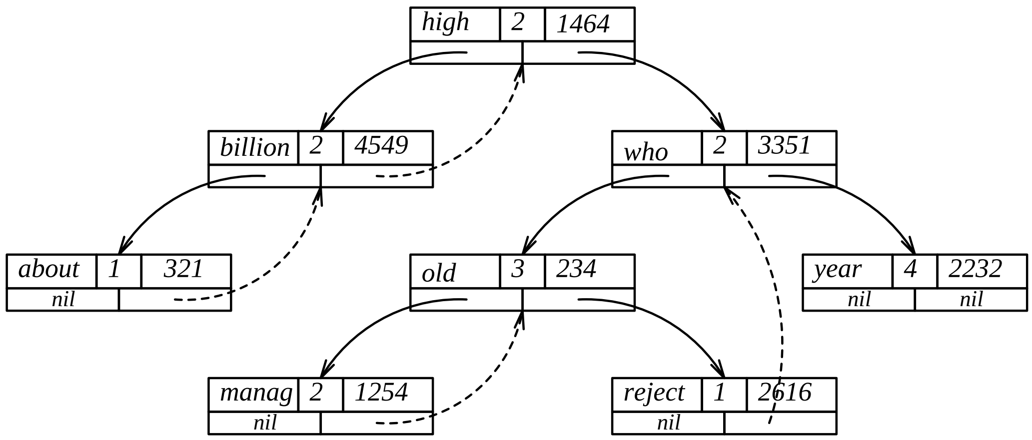

Figure 2: Right-Threaded Binary Tree

The postings are stored as multiple linked lists, one linked list for each term, with the lists stored in one large postings file. Each element in the postings file consists of a record address (the location of a given term), the term frequency in that record, and a pointer to the previous element in the linked list for that given term (the first element has a null pointer). By storing the postings in a single file, the postings are easily accessed by following the links. As the postings for each term are stored in reverse order, the entire list does not need to be read for each addition, but only once for use in creating the final postings file (step two).

Note that both the binary tree and the postings list are capable of further growth. This is important in indexing large databases where data is usually processed from multiple separate files over a short period of time. The use of the binary tree and linked postings list could be considered as an updatable inverted file. Although these structures are not as efficient to search, this method could be used for creating and storing supplemental indices for use between updates to the primary index.

The binary tree and postings file are saved for use by the term weighting routine called `rebuild` (step two). This routine walks the binary tree and the linked postings list to create an alphabetical term list (dictionary) and a sequentially-stored final postings file. To do this, each term is consecutively read from the binary tree (this automatically puts the list in alphabetical order), along with its related data. A new sequentially stored postings file is allocated, with two elements per posting. The linked postings list is then traversed, with the frequencies being used to calculate the term weights. The last step writes the record and corresponding term weights to the newly-created sequential postings file. This final file only needs two elements per posting (document number & weight) since no link pointer is required. Using a sequentially stored postings list in place of a linked list saves storage and reduces access time, as input can be read in multi-record buffers, one buffer usually holding all records for a given term. The sequentially–stored postings could not be created in step one because the number of postings is unknown at that point in processing, and input order is text order, not inverted file order.

The final index files therefore consist of the dictionary and the sequential postings file. Each element of the dictionary contains the term, its IDF (inverse document frequency) weight, the number of postings of the term in the entire text collection, and the location of its postings in the postings file. Each element in the postings file contains a record identifier and the term weighting for the given term in that record.

Table 1 gives some statistics showing the differences between the "old" and the new (PRISE) indexing schemes. The "old" indexing scheme refers to a version of a traditional indexing method in which records are parsed into a list of words within record locations, the list is inverted by sorting, and finally the term weights are added.

Note that the size of the final index file is relatively small, approximately 8% of the input text size for a 50 megabyte database, and around 14% of the input text size for the 806 megabytes[2]. This size remains constant when using the new indexing method as the format of the final indexing files is unchanged. p The working storage (the storage needed to build the index files) for the new indexing method is not much larger than the size of the final index files themselves, and substantially smaller than the size of the input text. However, the amount of working storage needed by the old indexing method would have been approxi-

[2]The 359 megabyte text collection is one of the subsets of the full 806 megabyte text collection, see Harman and Candela 1990 for more details of these data sets.

Indexing Statistics						
Text Size (megabytes)	Indexing Time (hours)		Working Storage (megabytes)		Index Storage (megabytes)	
	old	new	old	new	old	new
1.6	0.25	0.50	4.0	0.7	0.4	0.4
50.0	8.00	10.50	132.0	6.0	4.0	4.0
359.0	N/A[1]	137.00	-	70.0	52.0	52.0
806.0	-	313.00	-	163.0	112.0	112.0

Table 1: Indexing Statistics

mately 933 megabytes for the 359 megabyte database, and over 2 gigabytes for the 806 megabyte database, an amount of storage beyond the capacity of many environments.

The new method takes more time for the very small (1.6 megabyte) database because of the method's additional processing overhead. As the size of the database increases, however, the process time has an $n \log n$ relationship to the size of the database. The traditional method contains a sort which is $n \log n$ (best case) to n^2 squared (worst case), making processing of the very large databases likely to have taken longer using the old method, and considerably longer if a tape sort was required because of the large amount of working storage.

3 A Method for faster indexing through the use of virtual memory facilities

Indexing the TIPSTER collection uncovered many deficiencies in the PRISE system's indexing routines. Optimizations were applied to both the first phrase of indexing (performed by the `build` program) and the second phase of indexing (performed by the `rebuild` program.)

Several simple optimizations were applied to the `build` program including the use of buffered I/O routines to speed up reading the input text and the use of a DFA scanner to speed up elimination of stop words(Fox 1992).

The `rebuild` program was somewhat harder to optimize; the `rebuild` program's method of randomly accessing the disk-based intermediate postings file made it relatively immune to the optimizations which improved the speed of the `build` program. This was primarily due to the large number of seeks required to traverse the linked lists in the postings file.

This postings traversal becomes very slow as the index becomes larger. In this case, the primary performance inhibitor is the I/O system interface used to access the postings file. Because the addresses of posting entries for a particular term are usually far apart in the intermediate postings file (due to the order of occurrence of corresponding terms in the original text), straight-forward optimizations such as the use of a buffered I/O system are ineffective.

Using virtual memory facilities provides a method of optimizing production of inverted file indexes by optimizing disk access. Virtual memory systems are commonly used in operating systems to allow computers to run programs larger than available memory. This is achieved through the use of external storage; only the parts

character pattern	temporary postings file
a[a–m]	tpost00
a[n–z]	tpost01
b	tpost02
c[a–m]	tpost03
c[n–z]	tpost04
d	tpost05
...	...
v	tpost24
w	tpost25
x	tpost26
y	tpost26
z	tpost26

Table 2: Mapping of terms to corresponding postings files.

of the program currently being accessed by the computer's cpu reside in physical memory. To improve the response time of such programs when external storage is accessed, optimal caching and disk access algorithms have been developed. These algorithms are a common feature of modern operating systems. Many operating system manufacturers have provided program interfaces to these virtual memory systems to reduce the overhead of traditional I/O services and to speed access to files and data on external (random access) storage devices. The use of these optimal (virtual memory) algorithms can significantly decrease the time necessary to create an inverted file index.

The BSD Unix(tm)[3] virtual memory system uses the `mmap()` system call to map a file into memory allowing it to be accessed as if it were memory, utilizing the faster, low-level paging facilities of the operating system. This avoids the overhead of executing calls supplied by a higher level I/O system.

[3] Unix is a trademark of AT&T

Initially, the BSD Unix(tm) system call `mmap()` was employed to speed up the building of the final postings file by memory mapping the intermediate postings file. When mapped, the postings file can be accessed as if it were a linked list of postings in memory. On small databases (of less than 10 megabytes), this new version of `rebuild` proved to be very fast, up to an order of magnitude faster than the non–memory version. Unfortunately, this performance degraded significantly for databases above 100 megabytes. The performance loss was so great that the non–memory mapped version was faster! It is likely that the degradation in performance was the result of the small memory size of the machine (32 megabytes) and the large distance between the locations of the postings (sometimes across many page boundaries) causing a large number of page faults.

The solution to the large database problem was to lexically separate the file into smaller sections that could be mapped and accessed with a minimum of page faults. During the `build` phase, the postings of each term were placed in one of a number of temporary postings files based on the ordinal value of the term. The lexical grouping of the posting files is based loosely on frequency of words in the English language. Table 2 shows a partial mapping of term patterns to the corresponding temporary postings file. In this table the first and second characters of the term are used to determine the file in which the postings of the term will be placed. For example, the postings for terms *aardvark*, *abbie*, *airstream* and *ambassador* would be inserted in the file *tpost00.* while *angeles*, *assault*, *ayatollah*, and *aztec* would be placed in the file *tpost01.* Similarly, the terms *xerograph*, *yorktown*, *zeppelin* would be placed in *tpost26.*

This is not an optimal mapping for all indices but works reasonably well for many. It is likely that with more investigation, a better mapping for general indices could be found. For example, a mapping could be based on the frequency of words in a dictionary created from a random sample of documents in the collection to be indexed.

The size of the temporary postings files are such that no single file is greater than twice the memory size of the machine. During the second phase (`rebuild`), the tree is traversed lexically to create the final dictionary and consolidate the posting lists. During this traversal each temporary postings file is memory mapped and visited once (in lexical order) as the dictionary tree is traversed. This is similar to distribution sorting using binary search trees proposed by Cooper and Lynch (1984) and an earlier method proposed by Cooper, Dicker, and Lynch (1980) except that during the read phrase of sorting the postings file is memory mapped.

Only one file is mapped at a time; the file is loaded as program memory pages and is accessed that way, utilizing the speed of direct memory access. Because each file is mapped almost *entirely* in memory, random access performance is greatly enhanced. `Build` and `rebuild` use a common set of routines to implement the grouping of the postings files.

By modifying a single mapping table in the `mmap` support module and recompiling both programs, the grouping of the postings files can be changed consistently. A startup file containing the file groupings can be used instead to allow more flexibility. Figure 3 shows the full index creation procedure.

Table 3 lists the timing results of index creation using various methods of indexing on the Cranfield collection, Federal Register(TIPSTER Disk 1), and Wall Street Journal(TIPSTER Disk 1). Most of the experiments were executed on a SPARCstation[4] 10 with 32MB of main memory. Final products of index creation were independent of the optimization methods used. The disk-based version reflects system timings using the improvements shown in Table 1. Note that whereas the memory mapped version (second row) worked well for the small collection, it degraded for larger collections. The final indexing version (multiple memory mapped) solved the indexing problem for larger collections with a small increase in build time and a significant decrease in rebuilding time. Because of its larger number of records, the Wall Street Journal took twice as long to index as the Federal Register.

Care must be taken when using virtual memory routines with other memory allocation routines on BSD Unix(tm) systems. The use of `mmap` can often conflict with these routines. Programs are also very sensitive to memory usage by other programs on the same machine. This can be alleviated somewhat by using more (i.e. smaller) intermediate postings files.

Programs using `mmap()` may not be portable to other operating systems. However, similar functionality is available in VAX/VMS and Unix(TM) System V Release 4. More recently, (Krieger and Stumm 1994) have proposed a generalized application-level interface to exploit I/O performance improvements such as mapped file I/O.

4 A Smaller Uncompressed Index Format

In the original postings format the system used 32 bits for each document posting, 16 bits for the document number and 16 bits for the document's weight. This allowed document numbers and weights as large as 65535 decimal.

[4] NIST does not in any way endorse the Sun SPARCstation series of workstations. SPARCstation is a trademark of Sun Microsystems Corporation.

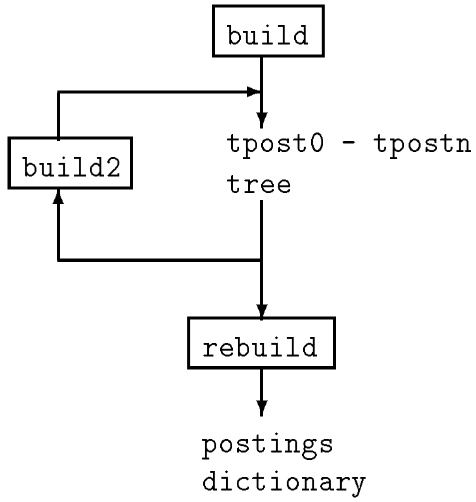

Figure 3: New index creation procedure

	Cranfield 1.6 megabytes		Federal Register 258 megabytes		Wall Street Journal 276 megabytes	
method	build	rebuild	build	rebuild	build	rebuild
disk based	16s	26s	22m 18s	6h 10m 3s	34m 21s	2d 3h 2m 8s [a]
memory mapped	15s	4s	22m 18s	6h 43m 5s		-
multiple memory mapped	18s	5s	21m 18s	6m 35s	48m 17s	13m 52s

[a] The index for this collection was produced on a Sparcstation 2 that is roughly 2.5 times slower than the Sparcstation 10 used to produce the other results.

Table 3: Indexing times of disk based, memory mapped, and multiple memory mapped indexing methods.

posting field widths				
version	document index field width	maximum document index	weight field width	maximum weight
1	16 bits	65535	16 bits	65535
2	32 bits	4294967296	32 bits	4294967296
3	20 bits	1048576	12 bits	4096

Table 4: largest integers expressable by supported posting field widths.

postings organization		Cranfield 1400 records 1.6 megabytes		Federal Register 26207 records 258 megabytes		Wall Street Journal 98736 records 276 megabytes	
docno (bits)	weight (bits)	dictionary (5059 terms)	postings	dictionary (86155 terms)	postings	dictionary (95839 terms)	postings
16	16	128290	319272	2431248	19538384	-	-
32	16	128768	478908	2446406	29307576	2769398	100803984
20	12	128290	319272	2431248	19538384	2748387	65155176

Table 5: Actual Dictionary and Posting Sizes

The original postings format was later revised for large databases; in this case, the size of a document posting was increased to 64 bits, 32 bits for the document number and 32 bits for the document's weight (see Table 4.) This allowed indexing of collections with more than 65535 records such as the Department Of Energy (DOE) abstracts on TIPSTER Disk 1 which contains 226,087 records, and Wall Street Journal articles from Disk 1 which contains 98,736 records (Harman 1993). Both of these collections were too large for the previous format to handle (see Table 5.)

A newer posting format tailored for TIPSTER databases was implemented in the Fall of 1992. The size of a document posting was reduced back to 32 bits; the document number field has been reduced to 20 bits, and the weight field to 12 bits. The document number field is large enough (2^{20} = 1048576) to produce an index for all the data of both TIPSTER disks. This postings scheme allows postings file sizes of the same order as the original scheme with much larger collections.

5 Modifications for Term Position Information

The indexing described in sections 2-4 contained minimal information about each document. As the documents were indexed, only the total frequency of each term was saved; no term position information was preserved. This allowed a very small index, with only a single weight per document term, and only one posting of a term per document, even if a term appeared multiple times. The large-granularity indexing produced using this method is adequate for statistical ranking systems where effective retrieval is not dependent on the positional information that is critical to Boolean retrieval systems. However, occasionally situations requiring a finer granularity of indexing can occur (Burkowski 1990). Additionally, research has been started at NIST using Natural Language Processing (NLP) techniques that require positional information.

A recent extension to the index format has been the addition of term position information. Whereas the current index format could have been extended to include term position information, it was felt that only minimal index growth should occur. It was decided to adapt the term position compression techniques described by Linoff and Stanfill to the existing PRISE indexing technique. Linoff and Stanfill use a variable length numerical (*n-s*) encoding suggested by Elias (1975) for use in representing lists of increasing integers. This encoding allows the packing of values into a smaller space than would be otherwise possible, and is especially useful when the values encoded are fairly small. Similar encodings have been suggested by Moffat and Zobel (1992a, 1992b).

The positional information for each term is placed in a file separate from the posting file described in section 2 and the address of that information is placed in an address field in the posting entry in the term's posting list. This separation allows searching using docno/weight information only or with additional term position information. With minimal modification, the existing search engine works with the new index (using docno/weight information only). The separation also simplifies construction of the indexing programs. The format of the modified postings record is shown in the top part of figure 4. The bottom part of figure 4 shows the new positional information entry. The position information entry contains the byte length of the term position record, the number of integers encoded, and the encoded term position information. Term position information consists of a list of position addresses in the form of a *tag* followed by corresponding *address information*. The address information consists either the section number and word position number or the word position number alone (for words occurring multiple times in the same section). A tag preceding the address determines what kind of address follows. The tags are used to determine when redundant information has been omitted. This method of encoding is called the Prefix Omission Method (POM). The primary difference between this format and the one described by Linoff and Stanfill is that only *section* and *word* position information is encoded [5].

The following example in figure 5 illustrates the format for two terms in a four paragraph document. The term *AIDS* occurs four times: in the second and eleventh words of the first paragraph, the nineteenth word of second paragraph, and the thirteenth word of the fourth paragraph[6]. The positions of the term would be represented by these four sub-sequences of integers: {*2,1,2*}, {*1,11*}, {*2,2,19*}, {*2,4,13*}. The first number in each sub-sequence is the tag denoting the information contained in the sub-sequence. The tag *2* denotes section and word position information, and the tag *1* denotes word position information only.

To prepare the data for the *n-s* encoding the position data is converted to run-lengths. Values of corresponding non-initial positions are replaced by the differences between adjacent positions (Linoff and Stanfill 1993). This increases the frequency of low-valued integers, which improves the effectiveness of the *n-s* encoding technique. The integer sub-sequences are now: {*2,1,2*}, {*1,9*}, {*2,1,19*}, {*2,2,13*}. The number of integers in all three sub-sequences is eleven. The final sequence of integers that represents the position information is: {*11,2,1,2,1,9,2,1,19,2,2,13*}. These integers in the sequence are encoded and then the *length* of the encoded information in bytes followed by the encoded sequence is written to the file containing the positions.

The encoding procedure follows these steps: first create integer sequences using tags and Prefix Omission Method, then reduce magnitude variation through run-length encoding, and then compress using *n-s* encoding.

The indexing routines used to implement term position information are not optimized. In particular, the overhead of encoding the positional information increases indexing time in the `build` phase. Additionally, the modified postings entries are memory mapped in the `rebuild` phase but the term position information is not. Ta-

[5] Linoff and Stanfill's term position format encodes paragraph, sentence, and word positions.

[6] positions of punctuation are not counted.

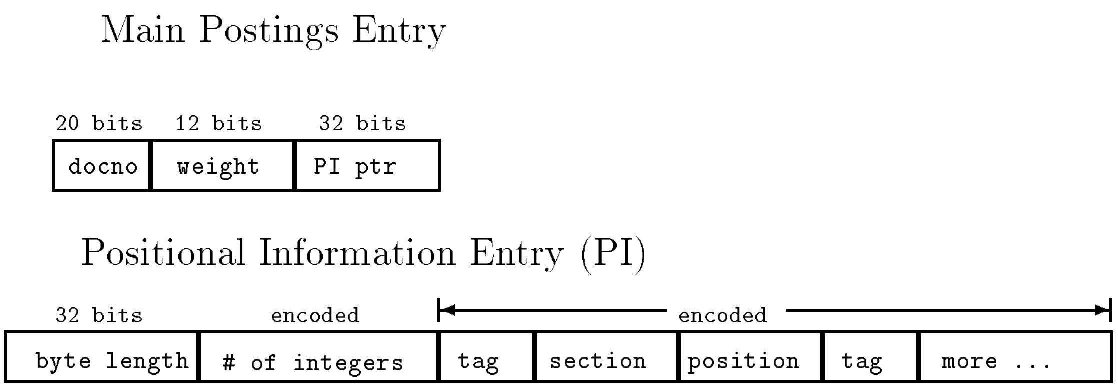

Figure 4: positional information format

ARC - AIDS Related Complex. A set of symptoms similar to AIDS.

AZT - Azidothymidine, a drug for the treatment of Acquired Immune Deficiency Syndrome, its related pneumonia, and for severe AIDS Related Complex.

TPA - Tissue Plasminogen Activator - a blood clot-dissolving drug.

treatment - any drug or procedure used to reduce the debilitating effects of AIDS or ARC.

term	section	word	sequence
AIDS	1	2	{2,1,2}
		11	{1,11}
	2	19	{2,2,19}
	4	13	{2,4,13}
drug	2	4	{2,2,4}
	3	9	{2,3,9}
	4	3	{2,4,3}

Figure 5: Sample text accompanied by a table listing two terms occurring in the text and their corresponding tpi sequences

ble 6 shows the difference in times between the Multiple Memory Mapping and Multiple Memory Mapped with Unmapped Term Position Information. The addition of positional information increases the size of index significantly. The positional indices can be as large as 50% to 100% of the size of the corpus text. Table 7 show the difference in sizes between old indices and indices containing term position information.

6 The Effect of Index Modifications on Search Efficiency

The indices created using the multiple memory map techniques are identical to the ones created using the older methods, and therefore, the search times for these indices are the same as times of indices created using the older methods. The use of bitmasks in the search engine to support the 20-16 bit postings format has a negligible effect on search time. Similarly, due to the structure of the postings file, the search times of indices constructed with positional information are minimally affected when not using positional information, although the use of positional information in the future will certainly increase search time.

7 Conclusion

The indexing method using term-based partitioning and virtual memory I/O has significantly decreased the time necessary to index large collections. Also, the modified postings format has allowed the size of an individual posting to remain the same as the first posting format while addressing a larger number of documents.

The current implementation of term position information provides reasonable compression. Further space saving could be gained by using the same encoding to compress the document numbers (and possibly weights and term position information addresses) in the primary postings file.

These changes were implemented with minimal or no effect on search efficiency. Future work will focus on improving the speed of dictionary lookup and document accumulation in the search engine.

References

Anick, P. and R. A. Flynn (1993). Integrating a dynamic lexicon with a dynamic full-text retrieval system. In R. Korfhage, E. Rasmussen, and P. Willett (Eds.), *SIGIR 93: Proceedings of Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, pp. 136–145. Association for Computing Machinery: ACM Press.

Burkowski, F. J. (1990, September). Surrogate subsets: A free space management strategy for the index of a text retrieval system. In J.-L. Vidick (Ed.), *SIGIR 90: 13th International Conference on Research and Development in Information Retrieval*, Brussels, Belgium, pp. 211–226. Association for Computing Machinery: Presses Universitaires De Bruxelles.

Cooper, D., M. E. Dicker, and M. F. Lynch (1980). Sorting of textual data bases: A variety generation approach to distribution sorting. *Information Processing & Management 16*(1), 49–56.

Cooper, D. and M. F. Lynch (1984). The use of binary search trees in external distribution sorting. *Information Processing & Management 20*(4), 547–557.

Elias, P. (1975, March). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21*(2), 194–203.

	Cranfield		Federal Register		Wall Street Journal	
text size	1.6 megabytes		258 megabytes		276 megabytes	
method	build	rebuild	build	rebuild	build	rebuild
multiple memory mapped	18s	5s	21m 18s	6m 35s	48m 17s	13m 52s
above with term position information	-	-	4h 12m 51s	37m 17s	9h 39m 16s	1h 41m 7s

Table 6: Indexing Times of Memory Mapped with and without Unmapped Term Position Information

	Cranfield		Federal Register		Wall Street Journal	
	1400 records		26207 records		98736 records	
text size	1.6 megabytes		258 megabytes		276 megabytes	
postings organization	dictionary 5059 terms	postings	dictionary 86155 terms	postings	dictionary 95839 terms	postings
original	128290	319272	2431248	19538384	2748387	65155176
positional	129101	1249393	2460425	125083992	2794646	278061746

Table 7: Positional Dictionary and Postings Sizes

Fox, C. (1992). Lexical analysis and stoplists. In W. B. Frakes and R. Baeza-Yates (Eds.), *Information Retrieval, Data Structures and Algorithms*, Chapter 7, pp. 102–130. Englewood Cliffs, New Jersey 07632: Prentice Hall.

Harman, D. (1993). Overview of the First Text REtrieval Conference (TREC-1). In D. K. Harman (Ed.), *The First Text REtrieval Conference (TREC-1)*, pp. 1–20. National Institute of Standards and Technology.

Harman, D. and G. Candela (1990). Retrieving Records from a Gigabyte of Text on a Minicomputer using Statistical Ranking. *Journal of the American Society for Information Science 41*(8), 581–589.

Knuth, D. E. (1973). *The Art of Computer Programming, Fundamental Algorithms*, Volume 1. Reading, Massachusetts: Addison Wesley.

Krieger, O. and M. Stumm (1994, March). The Alloc Stream Facility: A redesign of application-level stream i/o. *Computer 27*(3), 75–82.

Linoff, G. and C. Stanfill (1993). Compression of indexes with full positional information in very large text databases. In *SIGIR 93: Proceedings of Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 88–95. Association for Computing Machinery.

Moffat, A. and J. Zobel (1992a, March). Coding for compression in full-text retrieval systems. In *Proceedings IEEE Data Compression Conference (Snowbird, Utah)*, pp. 23–32. IEEE.

Moffat, A. and J. Zobel (1992b, June). Parameterized compression for sparse bitmaps. In *Proceedings, SIGIR (Copenhagen, Denmark)*, pp. 274–285. Association for Computing Machinery.

Salton, G. and M. J. McGill (1983). *Introduction to Modern Information Retrieval.* New York, NY: McGraw-Hill Book Company.

Schäuble, P. (1993). Spider: A multiuser information retrieval system for semistructured and dynamic data. In R. Korfhage, E. Rasmussen, and P. Willett (Eds.), *SIGIR '93: Proceedings of Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, pp. 318–327. Association for Computing Machinery: ACM Press.