Compression and Fast Indexing for Multi-Gigabyte Text Databases*

Alistair Moffat[†] Justin Zobel[‡]

Abstract

In the last two years we have developed improved techniques for indexing and retrieval of text data, including algorithms for inversion, for compression of the data and index, and for efficient ranking. These techniques were, however, tested on relatively small databases. In this paper we describe the application of these techniques to a large (2 Gb) heterogeneous text database. Our experiments show that compression performance does not degrade with the increase in volume of data, and that response times remain small, confirming that the new techniques are suitable for large quantities of data.

Keywords: information retrieval, text compression, index construction.

CR Categories: E.4 [Coding and Information Theory]: *data compaction and compression*; H.3.1 [Information Storage and Retrieval] Content Analysis and Indexing—*indexing methods*; H.3.2 [Information Storage and Retrieval] Information Storage—*file organisation*; H.3.3 [Information Storage and Retrieval] Information Search and retrieval—*search process*;

1 Introduction

Large document collections present many problems for practical information retrieval. To avoid unnecessary accesses to the text of the collection during query evaluation, comprehensive indexes are required. It must also be possible to create and access these indexes in a reasonable amount of time, and to store them, and the data itself, in a reasonable amount of space.

*A preliminary version of this paper was presented at the First Australian Workshop on Information Retrieval, Monash University, November 1992.

†Department of Computer Science, The University of Melbourne Parkville, Victoria 3052; alistair@cs.mu.oz.au.

‡CITRI, Department of Computer Science, RMIT, 723 Swanston St, Carlton, Victoria 3053; jz@kbs.citri.edu.au.

In the last two years we have developed techniques that allow the text to be stored compressed and provide fast access to document collections via compressed indexes [2, 15, 16]; create indexes with an in-memory inversion technique [11]; and permit economical ranking of large document collections [12, 17]. These techniques rely on the large main memories of modern machines to store the vocabulary of the document collection, allowing rapid access to index information and a compression model for the text. On the small document collections initially available to us, the largest of which was 132 Mb, compressed text and index together typically required 35–45% of the space required for the source text alone; query response times were at most a few seconds; and the peak memory requirement during query processing was less than a megabyte.

The experiments described in this paper are concerned with proving these techniques on a large document collection, that provided for the DARPA-sponsored TREC project [7], an international effort to develop information retrieval techniques on about 2 Gb of data extracted from the TIPSTER database. The TREC collection contains documents on science, technology, and economics, collected from five sources: the Associated Press news service; the US Department of Energy; the US Federal Register; the Wall Street Journal; and from computer magazines and journals published by Ziff. There are 894,406 distinct words before case-folding and stemming (reducing to 538,497 distinct index terms after case-folding and stemming), more than 352,000,000 word-occurrences in total, and 742,985 documents ranging in length from a few tens of bytes to some that are over 1 Mb. Thus, by any measure, this collection is large.

We first describe the structure of text databases with inverted file indexes, and how documents are ranked with regard to a query. Then, in Section 4, we describe the compression of the text of the documents, and in Section 5 describe the representation of the inverted file and associated information. The result of applying these techniques to the TREC data is described in Section 6.

2 Text databases

Text databases provide access to text documents on the basis of their content. Access is assumed to be via an inverted file index [6, 8, 16]. In such a scheme each distinct word in the database is held in a *vocabulary*, which may be an array, or may be a disk-based search structure such as a B-tree. For each word in the vocabulary the *inverted index* stores an *index entry*, a list of the identifiers of documents containing the word. Interleaved with the identifiers are the number of occurrences of each word in each document. That is, each index entry contains, for one word, a list of "document-number, frequency-in-document" pairs describing the documents that contain that word. More detailed index structures, for example, include word-positional information and support word-proximity queries [4, 3], but we do not consider them in this paper.

The vocabulary stores, for each term, the address in the inverted file of the corresponding entry, allowing direct access to the index. The vocabulary also stores two values for each term: f_t, the number of documents containing term t; and F_t, the total number of appearances of term t in the collection.

Since the index entries contain ordinal document identifiers rather than disk addresses, a *document mapping* is needed to convert identifiers into addresses. In simplest form, a document mapping is an array of addresses, with document identifiers used as array indices. For a large collection this arrangement will require either a non-trivial amount of memory space, or, at the cost of an extra disk access per record, can be stored on disk in an auxiliary file. Similarly, an *index mapping* is required to turn an ordinal word number into the address of the corresponding index entry, and this is stored in the vocabulary.

Other components of the database are the *compression model*, necessary for decoding if the text of the collection is to be stored in compressed form, and the *document lengths*, used for ranking. The actual makeup of these two components is discussed later. In our experiments the compression model was held in memory, together with a small index into the vocabulary and an array of *approximate lengths*, also described below; all of the other structures were stored on disk. However, the document lengths and document mapping are stored interleaved in a single *document information* file, since the document lengths are used to help decide which documents to access; similarly the index mapping is stored interleaved on disk with the records of the full vocabulary. Figure 1 shows the relationship between these various components and the flow of information during the processing of a query.

For simplicity, the database is assumed to be static. That is, the problems presented by update operations are not considered, and so the primary concern is with the cost of access to structures rather than the cost of maintaining them. This restriction is valid in quite a wide range of situations. For example, one application where such techniques are useful is the distribution of archival data such as legal transcripts on CD-ROM.

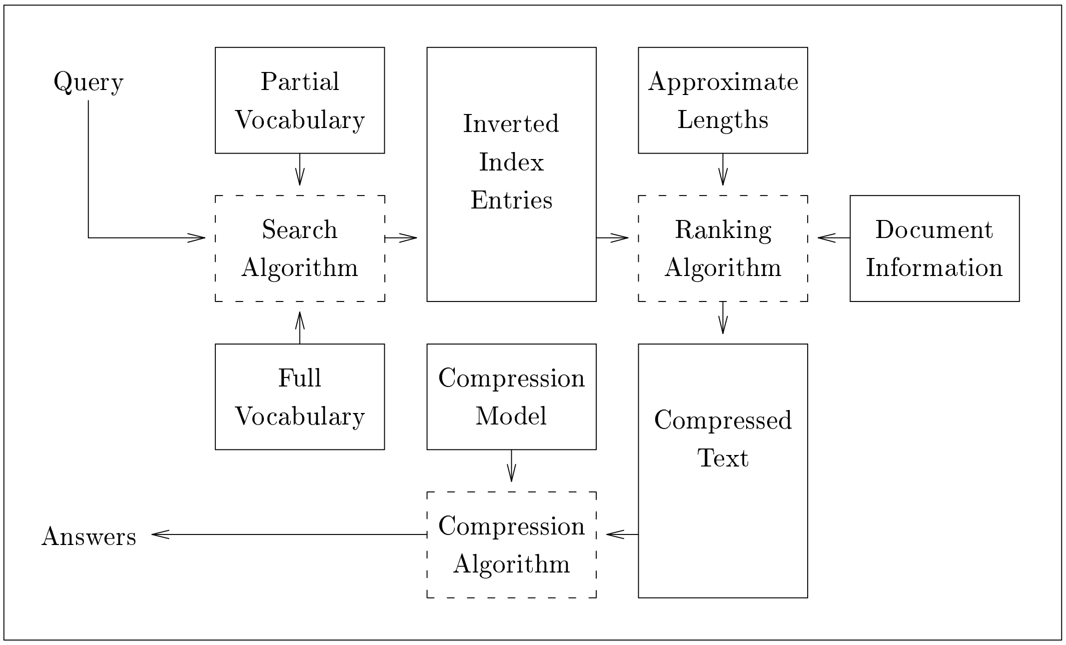

Figure 1: Organisation of text databases. The partial vocabulary, approximate lengths, and compression model are stored in memory. All other components are stored on disk.

3 Query evaluation

Given an inverted file index of the format described, it is straightforward to find the documents that match a Boolean query. To answer a conjunction (that is, find the documents that contain every one of a list of words), each word in the conjunction is found in the vocabulary, then the index entries for these words are fetched and intersected. Union and subtraction are used for disjunction and negation respectively. These operations result in a list of answer documents that can then be retrieved using the document mapping, decompressed using the compression model, and presented to the user.

Many queries to text databases are, however, inherently informal: they are requests to find documents about a topic rather than requests to find documents containing particular words. The process of matching documents to an informally phrased query, returning as answers the documents that most closely match the query, is called *ranking*. As an example, consider the cosine measure, one of the more effective ranking techniques [1, 13, 14]. The cosine measure evaluates the relevance of a document to a query via the function

$$cosine(q,d) = \frac{\sum_t w_{q,t} \cdot w_{d,t}}{\sqrt{(\sum_t w_{q,t}^2)} \cdot \sqrt{(\sum_t w_{d,t}^2)}} \ ,$$

where q is the query, d is the document, and $w_{x,t}$ is the weight of word t in document or query x. The expression $\sum_t w_{x,t}^2$ is a measure of the *length* of document or query x in terms of the weight and number of words in x. Note that this formal definition of length bears no relation to the informal usage of length to describe the size in bytes of the document.

One function for assigning weights to words in document or query x is to use the frequency-modified inverse document frequency, described by

$$w_{x,t} = f_{x,t} \cdot \log_e(N/f_t) \ ,$$

where $f_{x,t}$ is the number of occurrences of word t in x, N is the number of documents in the collection, and f_t is the number of documents containing t. This is commonly called the $\mathit{tf} \cdot \mathit{idf}$ weighting. Note that this function includes both global information, from the word weight $\log_e(N/f_t)$, and local information, from the term frequency $f_{x,t}$. The information used by the cosine measure is similar to that used by many ranking techniques.

The values required to compute the sum $\sum_t w_{q,t} \cdot w_{d,t}$ are held in the inverted index. To answer the query, an accumulator with initial value zero is created for each document in the database; the index entry for each word in the query is retrieved; and, for each "word-number, document-number, frequency-in-document" triple, a cosine contribution is added into the corresponding accumulator. A similar technique is also used by Harman and Candela [8]; of necessity, it supposes that the document collection is static, since changes to any document, or the addition of new documents, can change the length of every document in the collection.

The query length $\sum_t w_{q,t}^2$ is, for a given query, a constant; it does not affect the ranking and so need not be calculated. For each document in a static database the document length, $\sum_t w_{d,t}^2$, is also a constant, and can be precomputed and stored at the time the database is created. Thus, to compute the final ranking after the accumulators have been calculated, the document length for each document with a non-zero accumulator value must be fetched, and the accumulator value divided by this length. Finally, the k documents corresponding to the k largest cosine values are then retrieved and displayed, where k is a constant between, say, 10 and 200.

For each of the queries supplied with TREC, over 90% of the documents contained one or more of the terms even after information-less *stopwords* were removed from the queries, meaning that over 90% of the document lengths needed to be accessed to process each query. Moreover, at over 2 Mb, the table of document lengths is large, and in practice space limitations might prevent it being held in memory. It is possible, however, to keep limited precision *approximate lengths* in memory, thereby largely eliminating the cost of fetching document lengths from disk [17]. For example, suppose that b bits of main memory storage is the maximum allowed for each document length x, where $L \leq x \leq U$ and L and U are lower and upper bounds respectively on the document lengths. Then if $B = [(U + \epsilon)/L]^{2^{-b}}$, where ϵ is a very small number, the function $f(x) = \lfloor \log_B(x/L) \rfloor$ yields integers c between 0 and $2^b - 1$ for all legal values of x, and $g(c) = L \cdot B^c$ can be used to compute approximations to x. The use of a geometric code for assigning values c is more effective than the simpler linear code in which the interval L to U is divided into 2^b equally sized buckets. This is because the lengths are used multiplicatively, and so a geometric code gives the same relative error in the final calculated value irrespective of the value x being approximated.

The c values stored can then be used in either of two ways. First, the approximate lengths can be used to reduce the number of exact document lengths needing to be retrieved from disk. Second, *cosine* can be computed with $g(c)$, yielding an approximate ranking. With as few as two bits per length, there was only a small decline in retrieval performance in the five small document collections for which relevance judgments were available at the time [17]. The effect of applying these techniques to the TREC collection are described in Section 6.

4 Text compression

Using a word-based model of the text, the space required to store the documents comprising a database can be reduced to less than 30% of the original size [2]. Each word occurrence in the text is replaced by a Huffman code [9], the length of which is dependent on the frequency of the word. The intervening "non-words" are similarly coded against a vocabulary of non-words. Thus, by alternately decoding a word, then a non-word, and so on down the compressed text, the exact form of the input document can be reconstructed. For example, consider the phrase

"To be or not to be, that is the question!"

This phrase would be broken into a sequence of 10 word tokens, each followed by a non-word token. Table 1 shows some typical codes assigned to these various tokens (in fact, these are the codes created during the processing of TREC, and so are, of course, not an optimal encoding for this particular phrase). Using these codes, the phrase would be represented as the bitstring

011011101110 1 11000011 1 11000111 1 101101000 1 111001 1 11000011 0111
1101011 1 1101000 1 1111 1 0110001010100 0000000010000010000

where the intervening blanks are purely indicative. Using this code, the 42 characters of the phrase are stored in 113 bits. Note that the tokens "the" and " " (a single space) are the most frequent word and non-word in TREC, with probabilities of occurrence of 4.5% and 68.1% respectively; and that "!" appears as a token (that is, not followed by whitespace or other non-alphanumeric characters) only 349 times in the TREC collection, giving rise to a very long code.

Making use of the assumption that the database is static, the compression is performed in two passes. The first pass counts the frequency of every word and non-word in the collection, calculates a Huffman code for both distributions, and writes a lexicon file that contains all of the words used in the text and the corresponding code-word lengths, and all of the non-words and their code-word lengths. The second pass reads the whole collection again, compressing each document, and recording in the document mapping the address at which

Token	Code Length	Code
"To"	12	011011101110
"be"	8	11000011
"is"	7	1101000
"not"	9	101101000
"or"	8	11000111
"question"	13	0110001010100
"that"	7	1101011
"the"	4	1111
"to"	6	111001
" "	1	1
", "	4	0111
"!"	19	0000000010000010000

Table 1: Example codes for words and non-words from the TREC compression model

each compressed document starts. Because the compression is performed with a static model, documents can be decompressed individually, given only the address of the compressed document. For further details of the scheme used the reader is referred to the description in [15].

The decompression process is very fast. Huffman codes are particularly amenable to a table-based "look up and copy" implementation, and each such decoding step costs only a handful of shift and test operations per bit of input and generates several output bytes. Furthermore, the particular assignment of codewords—a *canonical* code, since all choice is removed and only a single set of codes can result—means that decoding can be effected without an explicit code tree being constructed. Indeed, in our implementation the decoding model requires less than 8 bytes of memory per word, including all indexing information, and decodes faster than the popular utility *Compress*.

As a result of the efficient decoding, the compression regime has only limited impact on retrieval time. Moreover, the small amounts of time needed for decompression are partially and sometimes fully offset by reduced disk traffic.

Because every word and non-word decompressed is retrieved from the compression model, it is crucial that this model be held in main memory. If the compression model was stored on disk, two or more disk accesses would be required for each word in each document retrieved, dominating all other retrieval costs and severely degrading performance. The scheme is thus only effective when sufficient main memory is available for the compression model, and, if memory is restricted, a simpler model should be used instead. On the smaller databases, the compression model did not exceed about 500 Kb. We were very interested to see how much space would be required to hold the model for the TREC collection.

5 Inverted file compression

The inverted file index entries should also be compressed, since without compression the inverted file might take as much space as the original text itself. Good compression of index entries can be achieved by numbering documents sequentially from 1, sorting index entries, representing each sequence of identifiers as a sequence of differences or gaps, and then using compact representations of the generally small integers that result [2, 4, 3, 16].

For example, a word might appear in the first, fourth, seventeenth, ..., documents, with an index entry of

$$1, 4, 17, 91, 113, \ldots$$

This can be represented by the sequence of differences

$$1, 3, 13, 74, 22, \ldots \ ,$$

which can then be compressed.

Several different techniques for compressing these runlengths have been described, in two broad classes [2]. *Global* methods use the same encoding for all entries, and so have the advantage of being general, but are insensitive to the frequency of each term. *Local* methods code each entry using a code that is adjusted to take into account one or more parameters of the distribution of gap values for that term. The most useful parameter is probably the mean gap, or, equivalently, the frequency of the term.

One simple global method is the gamma code of Elias [5], in which positive integer x is represented by coding $\lfloor \log_2 x \rfloor$ in unary followed by the value of $x - 2^{\lfloor \log_2 x \rfloor}$ in binary. For example, $x = 15$ has a unary prefix part of "1110" and a binary suffix part of "111", yielding a total code of "1110 111". Some gamma codes for other values of x are shown in Table 2. Note that, although the code is variable length, every stream of bits can be uniquely decoded into a stream of positive integers, and the code has the "prefix-free" property.

x	gamma(x)	Golomb(x), $b = 6$
1	0	0 00
2	10 0	0 01
3	10 1	0 100
4	110 00	0 101
7	110 11	10 00
15	1110 111	110 100
63	111110 11111	1111111110 100

Table 2: Examples of gamma and Golomb coding

Use of a variable length prefix code allows the frequent small runlengths to be represented more succinctly than the less frequent long runlengths, and is a marked improvement over a standard binary encoding.

Further advantage is given by use of a local code, because of the high variability of word frequency, and the effect this has on the average runlength within the entry. For example, the word "the" can be expected to have a sequence of runlengths where each difference is very small, usually one. On the other hand, a rare word such as "zygodactylous" will have gaps that are usually much larger, but also sometimes small, since rare words will tend to occur in clusters of related documents within the collection. Hence, a local compression method that adjusts its codes according to word frequency can be expected to perform better than a global method, and this is indeed the case. In our implementation the "local Golomb" code has been used [2]. In this code, integers $x \geq 1$ are represented by coding $q = \lfloor (x-1)/b \rfloor$ in unary, and then $r = x - 1 - b \times q$ in binary. To minimise the expected length of the encoding for each indexed word, the parameter b should be chosen as $b = 0.69 \times N/p$, where N is the number of documents in the collection and p is the number of documents containing the indexed word; that is, the best value of b is about 69% of the average run-length. The unary representation of q (that is, q one bits followed by a zero bit) will consume $q+1$ bits, and the binary representation of r will require at most $\lceil \log_2 b \rceil$ bits. When b is not a power of two, some of the binary suffix codes can be one bit shorter. For example, with $b = 6$ there are six possible r values—0 through 5—and these are represented as 00, 01, 100, 101, 110, and 111 respectively; and if and $x = 15$ is to be coded, then the unary prefix component is "110" and the binary suffix is "100", representing a remainder of 2. Other Golomb codes for $b = 6$ are shown in Table 2.

The term "frequency-within-document" values $f_{d,t}$ are efficiently represented by use of the gamma code [2]. These values are interleaved with encoded runlengths in the index entries.

After compression, the index entries for a text database will typically occupy less than 10% of the space required for the text itself. Decompression is fast, with about 100,000 document identifiers decoded per second on a 25 MIP Sun SPARC 2. None of these compression methods require any significant memory resources, once the index entry has been retrieved.

These inverted file encodings are also useful in the inversion process required during database construction, where again it is possible to capitalise on the large main memory offered by current workstations and the ability to perform two passes over a static database [11]. The most common previous technique was to write a file of "word-number, document-number" pairs in document number order and sort the file into word number order. This approach is viable for small collections, but the space requirements of the temporary files involved may exceed the space of the original text, prohibiting their use on large collections.

There are very few systems on which there is 2 Gb of surplus disk space available to allow the inversion of a 2 Gb database; certainly, there were none available to us! Another approach is the disk-based method described by Harman and Candela [6, 8]; they report that the inversion of an 806 Mb database on a minicomputer required 313 hours and 163 Mb of disk work space to produce an inverted index of 112 Mb. The previous largest test of the new inversion process required 45 minutes to invert a 132 Mb collection, using 12 Mb of memory. We were thus very interested to see whether the in-memory approach would be viable for the TREC collection.

It was decided to index every term, including very common terms such as "the". An index on these terms allows any Boolean or ranked query to be evaluated without a post-retrieval scan, and, with the application of compression, even the longest index entry ("the") required under 2 Mb, and they rapidly decreased in size from there. All words were, however, stemmed before indexing, using Lovins's algorithm [10]. This meant that the index vocabulary stored fewer words than the compression model, and reduced the number of pointers that had to be stored in the index.

6 Experimental results

This section describes the application of these techniques to the 2055 Mb TREC collection. All results are for a 25 MIP Sun SPARC 2 with 256 Mb of main memory and no other active user processes, and with all files stored on a local disk to avoid any networking delays.

Compression

The two stages of the compression process took about three hours each, generating a compression model and a compressed database of 4.2 Mb and 605.1 Mb respectively (Table 3). The second pass also generated a file containing the document mapping, a little under 3 Mb. That is, including both of these auxiliary files, the document collection was reduced from 2055 Mb to 612 Mb, or 29.8%, a disk saving of 1400 Mb.

The aggregate encoding rate, about 100 Kb/second, was the same rate as previously observed for smaller collections. Decoding also ran at the previously measured speed of 50–100 (compressed) Kb per second—that is, about 300–400 uncompressed kilobytes per second.

One obvious (but nevertheless disruptive) problem arose with all of the

Pass	Output files	Size Mb	Size %	CPU Time (Hr:Min)	Memory (Mb, peak)
First pass:					
Compression	Compression model	4.2	0.2	2:37	25.6
Inversion	Vocabulary	6.4	0.3	3:02	18.7
Overhead				0:19	2.5
Total		10.6	0.5	5:58	46.8
Second pass:					
Compression	Compressed text	605.1	29.4	3:27	25.6
	Document mapping	2.8	0.1		
Inversion	Inverted index	132.2	6.4	5:25	162.1
	Inverted index mapping	2.1	0.1		
	Document lengths	2.8	0.1		
	Approximate lengths	0.7	0.0		
Overhead				0:23	2.5
Total		745.8	36.3	9:15	190.2
Overall Total		756.4	36.8	15:13	190.2

Table 3: Files in compressed TREC database ($b = 8$). Percentages are relative to the size of the original document collection (2055.3 Mb)

programs. A 2 Gb file contains 16 *billion* bits, and in several places where bytes or bits were being counted it became necessary to change the data type being used, since even 32-bit unsigned integers were inadequate. Fortunately, double precision floating point values on the Sun have 52 bits of integer precision before truncation errors intrude, and, after the necessary variables were converted to doubles, the programs ran successfully.

The need to store the coding model meant that the compression passes were expensive on memory. With more than 900,000 unique symbols (884,406 words and 37,514 non-words) each consuming 6–8 bytes of characters, a 4 byte string pointer, a 4 byte frequency count, and an additional storage requirement of about 12 bytes per word for the dynamic search structure, each pass required 25 Mb of data structures. These figures reinforce our contention that database creation needs to be performed on a reasonably well configured machine, even if query processing is to be performed elsewhere.

Inversion

The inversion process was performed as two passes in tandem with the two compression passes. The first stage, counting term frequencies, had similar time and memory requirements to the first component of the compression process, but could not actually share data structures because of differing definitions of "words" and the need, in the index, for stemming.

The second pass of the inversion process was where problems were expected. The in-memory technique requires the allocation of memory a little larger than the final size of the compressed inverted file, and previous experiments had indicated that this would be roughly 10% of the size of the input text [11]. That is, use of more than 200 Mb of the 256 Mb of available memory was anticipated. However, the long TREC documents—averaging 470 words rather than the 90 for the previous largest collection—meant that the inverted file was smaller than anticipated, and, as it turned out, the inversion process ran smoothly to completion in 162 Mb, and contributed five and a half hours to the time of the combined second pass. One would not want to do this every day, but these resource requirements are within the capabilities of a typical $50,000 workstation. The alternative, a sort-based inversion, would take roughly the same time. It would, however, require the use of a temporary file of 141,420,851 ten-byte records, or more than 1 Gb of disk. Furthermore, if the system sort was used, a further 1 Gb of temporary work files would be required.

Improvements to the inversion process have been implemented that allow the second pass to operate in the same amount of memory as required by the first pass dynamic vocabulary structure, with only slightly slower performance. With this improved implementation and using four separate passes in total to compress and index the text, the retrieval system can be built in under 30 Mb of memory and in less than 18 hours.

Using the "local Golomb" code, the final inverted file required 132 Mb, or 6.4% of the input text. Of this, about 40 Mb is the gamma coded "frequency-within-document" values. Hence, if only Boolean queries are to be supported, the inverted file can be reduced to under 100 Mb.

The second pass of the inversion process also produced three other files—the index mapping for the inverted file, giving the address for each entry; the file of exact document lengths (later merged with the document mapping to form the document information file); and a file of approximate document lengths. The sizes of these, plus all of the other components of the compressed database, are shown in Table 3.

Performance on Boolean queries

Performance on Boolean queries is dominated by the amount of data to be decoded—first, from the inverted index, and then from the text itself. For example, on a short query such as "*sex maniac*", the bulk of the time is spent decoding documents. On that query, 0.1 seconds is spent accessing and decoding the 6.9 Kb of the two inverted file entries, and then a further 2.7 CPU seconds is incurred in the accessing and decoding of the 13 answers. The total elapsed time for the query is 4 seconds. There is, however, a total of 1.1 Mb of text in those thirteen answers (362 Kb compressed) and so the response times are reasonable. As an extreme query, the cost of searching for "*and how it of one said that the they what when who why will*" was also measured. This query takes 7 elapsed minutes, of which 1 minute is spent accessing and decoding inverted file entries (4.3 Mb); and 6 minutes elapsed (and 3 minutes CPU) are spent accessing and decoding the 22.3 Mb comprising the 2953 documents that match, a total of 77.7 Mb of answers. Given the absurd nature of the query, the response is entirely acceptable.

One area of current investigation is the use of techniques that allow inverted file entries to be intersected with only partial decoding being necessary, using self-indexing inverted file entries of the form described in [12]. Preliminary results show, in cases where the number is answers are small but there is a large volume of inverted file entries being processed, that substantial reductions in processing time are possible.

Performance on ranked queries

The data for the TREC experiment consists of three components: the 2 Gb of text; 50 test queries to be evaluated against the database; and relevance judgments for each query, which are the judgments of human assessors as to whether each document is relevant to each query. The cost of making these assessments has been a major bottleneck in information retrieval research, and, for example, the largest collection previously available contained just 3,204 documents amounting to less than 2 Mb of text, and 64 queries. In TREC, because of the impossibility of making the complete set of relevance assessments, the only documents assessed for each query were the top 100 documents retrieved by each of the approximately 15 participating research groups. At the time the experiments described here were carried out there were about 90,000 relevance judgments in total, covering 47 of the 50 queries. That is, an average of about 2,000 documents had been examined for each query, and a decision made as to whether or not they were relevant. On average, 350 answers were found for each query. There may be other answers that were not found, but the likelihood of this was pessimistically assumed to be negligible.

```
<top>
<head> Tipster Topic Description

<num> Number:  054

<dom> Domain:  International Economics

<title> Topic:  Satellite Launch Contracts

<desc> Description:
Document will cite the signing of a contract or preliminary agreement, or
the making of a tentative reservation, to launch a commercial satellite.

<narr> Narrative:
A relevant document will mention the signing of a contract or preliminary
agreement, or the making of a tentative reservation, to launch a
commercial satellite.

<con> Concept(s):
1.  contract, agreement
2.  launch vehicle, rocket, payload, satellite
3.  launch services, commercial space industry, commercial launch industry
4.  Arianespace, Martin Marietta, General Dynamics, McDonnell Douglas
5.  Titan, Delta II, Atlas, Ariane, Proton

<fac> Factor(s):

<def> Definition(s):
</top>
```

Figure 2: One TREC query before preprocessing

For the experiments the 50 test queries were transformed (by removal of SGML tags, stop words, and punctuation) into lists of words to which the cosine measure could be applied. Figure 2 shows a typical query before this transformation; and Figure 3 shows the list of query terms generated. The top 200 ranked documents were then retrieved and decompressed for each query.

Clearly, there were some words, such as "document", that should have been added to the stop-list; the presence of this word alone in every query meant that many of the irrelevant answers were about software engineering and documentation standards. All of the results listed here are from the initial processing run that was carried out before any of the relevance judgements were released,

```
document cite signing contract preliminary agreement making tentative
reservation launch commercial satellite relevant document mention
signing contract preliminary agreement making tentative reservation
launch commercial satellite contract agreement launch vehicle rocket
payload satellite launch services commercial space industry
commercial launch industry arianespace martin marietta general
dynamics mcdonnell douglas titan delta ii atlas ariane proton
```

Figure 3: The same TREC query after preprocessing

with no attempt made to iteratively refine the query by hill-climbing on the relevance judgements.

Processing of these queries was remarkably quick. Table 4 shows, for the three stages of processing each query, the average amount of data involved and the average time required. (The searching algorithm and the ranking algorithm operate jointly, and their times cannot be separately identified). As can be seen, on average each of the queries is processed in about 20 seconds of CPU time. Given that each query involves approximately 2.1 million document numbers and 600,000 non-zero accumulators, this result is very pleasing. Note that the times listed in Table 4 include the cost of writing the retrieved documents to disk for later analysis by a postprocessing program, but do not include the time required by the postprocessing.

Operation	File	Disk Accesses	Data (Kb)	CPU Time (sec)	Elapsed (sec)
Searching	Full Vocabulary	43	172		
Ranking	Inverted Index	43	1708		
	Document Information	219	7	16.6	18.2
Compression	Compressed text	200	203		
	Answers	—	729	3.7	7.2
Total		505	2819	20.3	25.4

Table 4: Ranked queries on TREC ($b = 8$)

On average the queries require about 500 disk accesses and the transfer of 2.8 Mb of data. At perhaps one hundred accesses per second on the disk, and a transfer rate of 1 Mb/sec, the input/output operations contribute about 5–8 seconds to the elapsed query time, accounting for the observed difference between CPU time and elapsed time. One hundred accesses per second for the file containing the compressed text is optimistic for purely random accesses, since a transfer of several Kb from a random point in a large file in the Unix file system will typically require three or more seeks, each costing perhaps 10 to 20 milliseconds. However, the bulk of the accesses to the document information file are in sequential order, and the accesses into the compressed text are batched into groups that are performed in sequential order.

For small queries response is even more impressive. For a query involving four terms—*"Australia wheat tariff subsidy"*—and retrieval of the top ten documents, 2.6 seconds of CPU time are required; and, if output is directed into a file, the total elapsed time for the query is under 4 seconds.

The use of approximate lengths in-memory using $b = 8$ (that is, storing an 8-bit approximation for each of the full 32-bit weights) means that on average only 219 exact weights are required to identify the top 200 documents for each of the 50 TREC queries. Depending on whether the exact lengths would have been stored in memory or on disk, with the use of approximate lengths either the memory required has been reduced by over 2 Mb, at the expense of 19 disk operations; or, alternately, at a cost of 725 Kb of memory, the need to sequentially read the entire exact lengths file has been avoided, a saving of several seconds per query.

Use of the limited precision lengths to provide the final ranking was also tested. In this method the exact lengths are completely ignored, meaning that the cosine measure is approximate, and some variation in recall effectiveness can be expected. In previous experiments on smaller collections this degradation was minimal for even very low values of b, the number of bits used to represent each weight. Two experiments were performed on TREC to evaluate the change in retrieval effectiveness.

First, of the top 200 documents retrieved by the "exact" cosine measure, the number that were not retrieved by the approximate ranking was counted. Table 5 shows, for various values of b, the maximum number of documents that "dropped out" of the top 200 answers for any single query, and the mean number of dropouts averaged over the 50 queries. Not considered in this experiment is whether the documents that dropped out—or indeed, those that dropped in to replace them—were in fact relevant, and so the results are at best a crude measure of the usefulness of the technique. However, use of approximate weights down to $b = 8$ appears to have little effect on the set of answer documents.

As a second experiment, the 47 partial judgments were used to compare the number of relevant answers using the exact ranking and the approximate ranking technique. These results are shown in Table 6. Note that relevance

b	Maximum dropouts	Average dropouts
12	1	0.4
10	4	1.1
8	9	3.7
6	36	16.2
4	91	55.4
2	195	135.0

Table 5: Effect of varying b—number of dropouts, 50 queries, 200 answers

judgments have not been provided for all documents, giving rise to a range for precision—most queries returned some documents that were unjudged. If it is to be pessimistically assumed that these documents are not relevant, the precision should be taken to be the lower value. For example, with 8-bit approximate weights and the top 30 answers retrieved for each query, the average percentage (of the 30 answers) of definitely relevant documents was 48.8%; 50.5% (= 100.0% − 49.5%) of the documents retrieved were definitely not relevant; and a further 0.7% of the retrieved documents were unjudged for the query.

The final column of Table 6 shows an 11pt recall-precision average, calculated by averaging the best ratio of precision obtained after 0%, 10%, ..., 100% of the relevant answers had been retrieved. For this figure it has been pessimistically assumed that all non-judged documents are not relevant, and also that all precisions outside the top 200 retrieved documents are zero. The TREC experiment adopted this approach of taking an 11pt average based only on 200 documents rather than a complete ranking because of the computational cost of evaluating a full ranking of all 750,000 documents, and this convention has been continued in this paper. Thus, if more documents are retrieved, retrieval effectiveness measured in this way will increase. For example, with the top 1,000 documents ranked, the 11pt average for exact ranking increases to 24.6%.

Clearly, although approximate ranking is finding different documents, in many cases the "dropins" are no less relevant than the "dropouts", and the use of approximate ranking has little effect on precision, even when the document lengths are stored in as few as 4 bits.

b	Percentage precision Number of answers					Pessimal effectiveness (11pt av., %)
	5	15	30	100	200	
exact	55.7–56.2	52.1–52.3	48.8–49.6	40.7–44.1	34.2–43.3	17.4
12	55.3–55.7	51.9–52.2	48.8–49.6	40.7–44.1	34.1–43.3	17.3
10	55.3–55.7	51.8–52.1	48.9–49.7	40.6–44.1	34.1–43.2	17.4
8	54.9–55.7	51.2–51.5	48.8–49.5	40.6–44.1	34.2–43.3	17.5
6	55.7–56.6	50.8–51.3	47.5–48.4	40.6–44.5	34.1–44.0	17.7
4	57.9–59.1	50.2–50.9	46.5–48.6	38.0–45.6	31.2–48.0	16.0
2	42.6–46.4	32.6–42.4	29.9–46.6	21.3–57.7	15.7–65.8	7.9

Table 6: Effect of varying b—percentage precision, 47 queries, 200 answers

Memory requirements

Table 7 shows the maximum memory that might be required during the processing of any query. The peak memory requirement is about 10.9 Mb, since not all components are required to be co-resident, and, in particular, the accumulators and answer buffer are never simultaneously allocated. If necessary, this could be further reduced by writing the output text directly to disk rather than into an answer buffer, in which case the array of accumulators becomes a major cost. For this reason we have also been examining methods whereby the space required by the accumulators can be reduced [12]. Another problem currently being investigated is whether similar performance—both in terms of compression efficiency and speed of decoding—can be obtained with reduced memory requirements for the model.

Structure	Memory Space (Mb)
Compression model	6.31
Partial Vocabulary	0.03
Approximate Weights ($b = 8$)	0.71
Accumulators	2.83
Inverted file buffer (longest entry)	1.00
Compressed text buffer	1.00
Answer buffer (longest document)	2.49
Peak Requirement	10.88

Table 7: Memory requirements during query processing

7 Summary

We have built a compressed retrieval system for around 2 Gb of text. The complete system requires less than 40% of the size of the input text alone, and was created in just over 15 CPU-hours on a well-configured workstation. Retrieval performance for simple techniques such as the cosine measure is fast, and queries can be processed effectively on low-end workstations in seconds. These results are confirmation that large-scale information retrieval need not be expensive, and that the techniques described in this and other papers have significantly improved the affordability of multi-gigabyte information retrieval systems.

Acknowledgements

We would like to thank Tim Bell, Alan Kent, Ron Sacks-Davis, Ross Wilkinson, and Ian Witten for their comments and support, and Lachlan Andrew, Gary Eddy, and Neil Sharman for their assistance with the implementation. This work was supported by the Australian Research Council.

References

[1] S. Al-Hawamdeh and P. Willett. Comparison of index term weighting schemes for the ranking of paragraphs in full-text documents. *Int. J. of Inf. and Lib. Research*, pages 116–130, 1990.

[2] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Jour. of the American Society for Information Science*, 44(9):508–531, October 1993.

[3] A. Bookstein, S. T. Klein, and D. A. Ziff. A systematic approach to compressing a full-text retrieval system. 28(6):795–806, 1992.

[4] A. Bookstein, S.T. Klein, and T. Raita. Model based concordance compression. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 82–91, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.

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

[6] E. A. Fox, D. K. Harman, R. Baeza-Yates, and W. C. Lee. Inverted files. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 3, pages 28–43. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[7] D. K. Harman, editor. *Proc. TREC Text Retrieval Conference*, Washington, November 1992. National Institute of Standards Special Publication 500-207.

[8] D. K. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Jour. of the American Society for Information Science*, 41(8):581–589, August 1990.

[9] D. S. Hirschberg and D. A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, April 1990.

[10] J. B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computation*, 11(1-2):22–31, 1968.

[11] A. Moffat. Economical inversion of large text files. *Computing Systems*, 5(2):125–139, 1992.

[12] A. Moffat and J. Zobel. Fast ranking in limited space. Technical Report 93/11, Department of Computer Science, The University of Melbourne, Parkville 3052, Australia, May 1993.

[13] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Massachusetts, 1989.

[14] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[15] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. In *Proc. Australasian Computer Science Conf.*, pages 1077–1089, Hobart, Australia, January 1992.

[16] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. Int. Conf. on Very Large Databases*, pages 352–362, Vancouver, Canada, August 1992.

[17] J. Zobel, A. Moffat, and R. Sacks-Davis. Memory-efficient ranking of document collections. Technical Report TR-92-53, Collaborative Information

Technology Research Institute, RMIT and The University of Melbourne, Melbourne, Australia, August 1992.