SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(8), 891–903 (AUGUST 1995)

Adding Compression to a Full-text Retrieval System

JUSTIN ZOBEL
Department of Computer Science, RMIT University, P.O. Box 2476V, Melbourne 3001, Australia
(email: jz@cs.rmit.edu.au)

AND

ALISTAIR MOFFAT
Department of Computer Science, The University of Melbourne, Parkville 3052, Australia.
(email: alistair@cs.mu.oz.au)

SUMMARY

We describe the implementation of a data compression scheme as an integral and transparent layer within a full-text retrieval system. Using a semi-static word-based compression model, the space needed to store the text is under 30 per cent of the original requirement. The model is used in conjunction with canonical Huffman coding and together these two paradigms provide fast decompression. Experiments with 500 Mb of newspaper articles show that in full-text retrieval environments compression not only saves space, it can also yield faster query processing – a win–win situation.

KEY WORDS: full-text retrieval; data compression; text compression; Huffman coding; word-based model

INTRODUCTION

Full-text retrieval systems are in widespread use in applications as diverse as library catalogues, newspaper archives and office automation systems. These *document databases* tend to become very large over a period of time and are essentially static, making them prime targets for compression. In this paper we describe, by way of a case study, an efficient compression regime for use with full-text retrieval systems. One interesting aspect of this implementation is that recent innovations in compression – adaptive modelling and arithmetic coding – proved to be unsuitable, because of the need for random access into the database and high computational costs. Instead a semi-static model and a form of Huffman coding are used.

The compression scheme has been implemented and tested against a collection of nearly 175,000 articles covering several years of the *Wall Street Journal*. This *WSJ* collection, 508 Mb of text in all, is part of the larger *TREC* collection and has been used in a wide range of other information retrieval experiments.[1] Our results here show that, not only is it possible to substantially reduce the space consumed by a large collection, but it is possible to do so *without* affecting query response time. Indeed, in a variety of the cases we consider, query response time is actually improved by the application of compression.

This result is in marked contrast to another recent implementation of compression for text databases, by Witten, Bell and Nevill,[2] whose proposal achieves similar compression

CCC 0038–0644/95/080891–13
©1995 by John Wiley & Sons, Ltd.

Received September 1991
Revised October 1994 and January 1995

to ours, but results in slow retrieval. Older implementations[3,4,5] used simpler compression methods and did not achieve good compression on text.

THE COMPRESSION SCHEME

The last decade has seen significant advances in the area of data compression. Rissanen and Langdon showed that compression consists of two distinct activities: *modelling*, in which a probability is assigned to each possible next symbol in the stream of symbols being represented; and *coding*, where the actual next symbol is represented by a code relative to the probability distribution supplied by the modeller.[6] This logical separation has meant that a wide range of models for compression can now be considered.[7] Modern compression methods are typically *adaptive*, allowing the compression to be carried out in one pass. Over a long text the parameters in an adaptive model converge to the values that would have been assigned had a preliminary scan of the text been made, and although the compression is slightly less efficient during the early stages of the encoding, there is no need for parameters to be transmitted in a prelude transmission, and the net difference between one-pass and two-pass modelling is negligible.

The eighties also saw significant advances at the coding level. For adaptive coding it is now usual to prefer arithmetic coding[8] to Huffman coding,[9] because arithmetic coding, with its ability to allocate 'fractional' bits, is close to optimal even when the probability distribution describing the symbols is strongly biased in favour of one symbol; and because adaptive arithmetic coding is no slower than adaptive Huffman coding. However, for the problem of full text retrieval from document databases, adaptive modelling using arithmetic coding is not the most effective compression technique.

Choice of model

Instead of an adaptive model we chose to use a *semi-static* model, in which the encoder makes a preliminary scan over the text to be compressed before performing the actual compression in a second pass. There were two reasons for this choice. First, adaptive models are inherently sequential, since the code used for each symbol of the input stream is a function of both the initial configuration of the compression scheme and of every preceding symbol. This make it impossible to commence decoding at intermediate points in the compressed text.

The second disadvantage of adaptive models is that it is usual to start them in a bland state, assuming nothing about the particular text that is to be processed. As a consequence, during the initial stages of the compression, while the model is 'learning' the parameters of the text, the compression is relatively poor. On a long text this effect is generally small, and is quickly overcome by the efficiency of using a more accurate model. However on short texts, ranging up to perhaps 10 Kb, the learning overhead dominates the compression, especially with the complex models required for good compression.

In a document database these effects combine. To allow the documents in the database to be individually retrieved it is necessary, with an adaptive code, to encode each document individually; but individually each document might only be a few hundred or few thousand bytes long and so compression efficiency deteriorates. On the other hand, if a semi-static model is used each document in the database can be compressed using model parameters that, although not optimal individually for every document in the collection, are optimal for the collection as a whole. Furthermore, decompression is significantly faster in a semi-static model as there is no need for the model parameters and codeword assignments to be updated once they have been initialized.

The disadvantages of a semi-static model are that two passes of the input text must be made and that the model parameters must be stored as part of the database. However both of these disadvantages are relatively unimportant in a static document database, where it is assumed that considerable effort can be invested to create the database provided retrieval is fast; that the collection will not change over time; and that the total size of the database will be large.

We have chosen to use a word-based compression scheme.[10,11] There were three reasons for this. First, we expected the documents in the database to primarily be ASCII text files, for which this model is tailored. Previous experiments have shown this model to give particularly good compression for English text.[7,11,12] Second, use of long tokens means that the decode-time ratio of bytes of output to symbols processed is high, and that on a per-byte basis the cost of decoding is small. The final reason for using a word-based model is that structures associated with the model include a list of words (and non-words) appearing in the documents of the database, which has the useful side-effect of making a full vocabulary of the database available to the retrieval mechanism.*

To effect the desired compression, input documents were considered to consist of a strictly alternating sequence of words (defined to be strings of alphanumeric ASCII characters) and non-words (all other characters). Words (or non-words) longer than some predefined length – 15 characters in the current implementation – and words containing more than some predefined number of embedded numeric characters – currently four – were split into two or more parts, with zero-length non-words (respectively, words) inserted where necessary to maintain the strict alternation. The restriction on the number of numeric characters in any word was used to avoid long runs of integers, such as page numbers, each becoming individual words with frequency one. Two complete vocabularies and sets of statistics were maintained, one for words and one for non-words.

Choice of coding method

More than anything else, the decoding process must be fast. Witten, Bell and Nevill reported a decoding speed of about 500 characters per second on a Sun SPARCstation 1 and estimated that perhaps 5,000 characters per second would be possible if enough main memory was allocated to store the entire model in uncompressed form.[2] For large-scale implementations these decoding rates are too slow and in designing a new system it was clear that it would be worthwhile to give up some of the compression performance if doing so would allow a significant increase in decoding speed. One of the contributing factors for the slow decompression reported by Witten et al. is the use of arithmetic coding. To create a scheme capable of fast decoding it was apparent that more traditional prefix codes must be preferred.

The method chosen was Huffman coding.[9] Huffman coding has fallen out of favour in recent years, primarily because of the lack of optimality on skew symbol distributions and because adaptive Huffman coding requires more resources than adaptive arithmetic coding. The second of these objections can be dispensed with immediately, since the coder was to be coupled with a semi-static model. The first objection is the more serious and could

* A vocabulary might also be required by the indexing method, but in this indexing vocabulary terms are usually case-folded and stemmed to their root form and so some information is lost. The compression vocabulary stores, of necessity, unprocessed terms.

Table I. Huffman vs arithmetic coding for *WSJ*

	Words	Non-words
Number of distinct tokens	289,101	8912
Number of tokens	86,785,488	86,958,743
$p_{\max}$	0·043	0·703
Entropy, bits per token	11·171	2·461
Huffman code, bits per token	11·202	2·590
Inefficiency of Huffman code	0·28%	5·24%
Overall inefficiency of Huffman code	1·17%	

potentially mean that significantly worse compression might result from using Huffman coding rather than arithmetic coding, even for the same model. However for the type of text expected, this objection can also be countered. Gallager[13] has shown that the inefficiency caused by this rounding of probabilities is bounded above by

$$p_{\max} + \log_2 \frac{2\log_2 e}{e} \approx p_{\max} + 0 \cdot 086,$$

where $p_{\max}$ is the probability of the most likely symbol and the inefficiency is measured in 'bits per symbol in excess of self-entropy'.

In English text the most common word is usually 'the', with a probability under 10 per cent. Moreover, the zero-order entropy of the word distribution in most text is in excess of 10 bits per symbol. The distribution for non-words is less uniform and it is not uncommon for the most frequent non-word – usually a single blank – to have a probability as large as 80 per cent, in a non-word probability distribution with a self-entropy of as little as two bits. Nevertheless, when joined together in a strictly alternating sequence, Gallager's result limits the relative inefficiency for the combined compression model to less than

$$\frac{(0{\cdot}10 + 0{\cdot}086) + (0{\cdot}80 + 0{\cdot}086)}{10 + 2} = 9\%.$$

That is, a Huffman coded word-based model should be at most nine per cent inefficient compared with an arithmetically coded word-based model. In return for the speed improvements expected this penalty is well worthwhile; and this is just an upper bound. Table lists, for words and non-words, the various parameters of the *WSJ* collection described above. As can be seen, the overall cost of using Huffman coding is an inefficiency of just 1·2 per cent relative to the entropy of the model. The total number of non-word tokens shown in Table is greater than the number of word tokens because most of the documents in the collection both started and ended with non-words.

The actual coding method used was the Huffman–Shannon–Fano code.[14,15] There are several advantages of this particular form of Huffman coding, sometimes known as *canonical* Huffman coding. First, the space required during decoding is small: four bytes for each token, plus space for the tokens themselves, plus a few hundred bytes of auxiliary arrays. Second, decoding is fast, involving a shift and a comparison for each bit of compressed input, plus a single array access to retrieve the token once a code has been parsed. Third, memory reference patterns are highly localized, and if insufficient physical memory is available at most one page fault per token will take place during decoding, compared with as many as one page fault per bit for a traditional tree-based Huffman decoder. And fourth,

Table II. Canonical Huffman coding

Symbol number	Symbol	Codeword	Integer value of prefix bits				
			$len = 1$	$len = 2$	$len = 3$	$len = 4$	$len = 5$
1	C	00000	0	0	0	0	0
2	D	00001	0	0	0	0	1
3	A	0001	0	0	0	1	
4	B	0010	0	0	1	2	
5	G	0011	0	0	1	3	
6	E	01	0	1			
7	F	10	1	2			
8	H	11	1	3			
first[*len*]			2	1	2	1	0
base[*len*]			9	6	6	3	1

only the length of the Huffman code need be stored in the dictionary, rather than the code itself, a saving of 3 bytes per token.

To see how canonical Huffman coding operates, suppose that an alphabet of eight symbols – ‘A’ through ‘H’, say – are determined by Huffman’s algorithm to have codelengths of 4, 4, 5, 5, 2, 2, 4 and 2 respectively. Table shows the codewords allocated in a canonical code, with the symbols now ordered by decreasing codelength. Also shown is the integer resulting when each codeword is considered to be a *len*-bit value, for *len* in the range 1 through to the length of that codeword.

The key to fast decoding is the array *first*, which stores for each codelength the integer value of the first valid codeword of that length. For example, the first 5-bit codeword is 00000 and so $first[5] = 0$. Similarly, $first[4] = 1$, since the first 4-bit code, that of symbol ‘A’, corresponds to the integer 1. In the example there are no valid 1-bit codes and so $first[1] = 2$. The *first* array allows use of a simple loop that adds bits to an integer until a valid code is built up, as illustrated in Figure 1. Using the example code shown in Table , if the input bit stream is 1100111..., then c, an integer prefix of the codestring, will take the values 1 when $len = 1$, which is still less than $first[1]$; and then 3 when $len = 2$, at which point the decoding loop exits and a symbol can be output. To determine which symbol this code corresponds to, the array *base*, which indicates the symbol number for the first codeword of each length, is consulted. In this case the symbol is ‘H’, since $base[len] + c - first[len]$ for $len = 2$ and $c = 3$ identifies the eighth symbol in the table. Started again on the suffix 00111..., the sequence of values is $c = 0$, less than $first[1] = 2$; $c = 0$, less than $first[2] = 1$; $c = 1$, less than $first[3] = 2$; and $c = 3$, which is greater than $first[4] = 1$ and so a 4-bit code is decoded. In this case calculating $base[len] + c - first[len]$ for $c = 3$ and $len = 4$ yields symbol 5, a ‘G’.

The array *first* contains one entry for each codelength, and so, in our implementation, requires 32 words. Since this is the only array accessed during the dominant decoding loop – there is, for example, no pointer dereferencing or tree traversal on a per-bit basis – the loop executes extremely quickly. Then, once a codeword has been determined, a small number of further operations are required to locate and output the corresponding symbol, but these operations are performed once per symbol rather than once per bit. Hence, provided that the symbols are long – which is another reason to prefer the word-based model – overall decoding speed is high.

```
c := nextinputbit()
len := 1
while c < first[len] do {
    c := 2*c + nextinputbit()
    len := len + 1
}
/* Integer c is now a legitimate codeword of len bits */
output symbol[base[len] + c - first[len]]
```

Figure 1. Decoding a canonical Huffman code

Dynamic collections

The discussion so far has assumed that the collection was static and that novel words would never be encountered. However the mechanism described can also be adapted for use with dynamic collections, to which new documents are appended.

To achieve this flexibility an extra ‘escape’ code was included as both a word and a non-word, to allow the encoder to signal to the decoder that a token had been encountered that was not in the dictionary. Following the transmission of the escape code the encoder then specified, as a sequence of 8-bit bytes, the length in characters of the offending token and then the characters comprising the token. There are a wide variety of other static models that could be used to transmit new words, but we were happy with the ad-hoc method of using unweighted 8-bit character codes. Using this method we estimate that the database might grow by as much as 100 per cent before noticeable compression degradation would take place, since most of the words in new documents would still be in the existing dictionary with appropriate codes already allocated. The number of entirely novel words to be spelt out byte by byte would be small, and only after a substantial number of insertions would it be necessary to completely rebuild and recompress the database. Rebuilding the database is costly, but ten rebuildings would see the collection grow by a factor of perhaps 1,000, and could be expected to cover the full life-cycle of the database. Moffat, Sharman and Zobel consider dynamic text collections in more detail, and give methods that allow ten-fold expansion in size with almost no compression degradation.[16]

Storing the model parameters

Each of the two lists of tokens was stored in sorted order using front-coding, that is, only the characters of each word that differed from those of the previous word were actually recorded. Rather than use an exact model, the length of the matching prefix was coded into a 4-bit nibble, the length of the differing suffix into a second 4-bit nibble and the length of the Huffman code was stored as an 8-bit byte. The suffix characters then followed, entirely uncompressed. Each word thus required two overhead bytes, plus the characters of the word that differed from those of the previous word. A short prelude to each list described the total number of bytes comprising the tokens, the length of the maximum code, and the number of tokens with codes at each possible code length, all stored as 32-bit integers.

A wide variety of additional ad-hoc schemes could have been used in addition to the structure described. However we made little additional effort to optimize the storage of the vocabulary, since we expected it to be just one or two percent of the final compressed database. Moreover, the decompressor must be able to quickly load the vocabulary if system startup is to be fast.

Table III. Times and effectiveness of compression of *WSJ*

	Compress	*GZip*	*PPMC*	*Huffword*
Pre-process text (CPU s)	–	–	–	2,714
Encode (CPU s)	1,174	3,658	7,236	2,083
Decode (CPU s)	545	303	8,195	565
Dictionary size (Mb)	–	–	–	1·29
Compression (%)	43·1	36·8	24·1	28·2
Including dictionary (%)	–	–	–	28·4

Other considerations

Each compressed document was prefixed by a coded integer indicating the number of uncompressed bytes the string represented; this added a small overhead to the representation, but meant that there was no need for a separate ‘end-of-string’ symbol. The γ code[17] was used for this, requiring $1 + 2\lfloor \log_2 (n + 1) \rfloor$ bits to code integer n. A 1-bit flag followed, to indicate whether the first token was a word or a non-word and then the codes for the tokens comprising the string.

The implementation allowed for a longest Huffman code of 32 bits. Although in pathological situations 33-bit codewords can be forced by as few as 10 million tokens, in our experience 32-bit codewords are adequate for several gigabytes of text. Beyond this, 64-bit codes or some form of length-limited code[18,19] must be used. The use of length-limited codes would permit databases of up to 2^{32} *distinct* words to be represented, albeit with some considerable compression degradation.

During both encoding and decoding it is assumed that sufficient memory is available for all of the model to be permanently resident. In the two encoding passes, 16 bytes per token are required for indexing structures and to store the codes, in addition to the characters of the tokens themselves. The decoder requires only four extra bytes per token; nevertheless, even this can become a substantial amount of memory, and for *WSJ* approximately 3·3 Mb is required for the decoding model. One area of recent investigation has been methods whereby this memory requirement can be reduced.[16]

In total the source code for the two compression passes and the decompressor required about 1,500 lines of C code. All of the routines were written to handle binary data such as object files and could equally be used to store text or other non-text data.

Encoding and decoding performance

To demonstrate the usefulness of the compression scheme a stand-alone driver program was written to compress and decompress files. Table tbl-wsjspeed shows the time taken on *WSJ* by the three phases of the system: pre-processing, compression and decompression. By way of comparison, we also tested the standard Unix utility *Compress*; the newer *GZip*; and the *PPMC* program.[20] All timings are in CPU seconds on a Sun SPARC 10 Model 512; compression is measured in percentage remaining of the original text. As can be seen from the table, decoding using the semi-static word-based *Huffword* method is fast, running at just under one megabyte per second, or, taking machine speed into account, roughly 40 times faster than the speed estimated by Witten, Bell and Nevill.[2] Decoding is about the same speed as the utility *Compress*, but not as fast as *GZip*; while compression on this text is a little worse than a fifth-order *PPMC*.

Adaptive methods such as *GZip* and *PPMC* could be applied to text databases by compressing blocks of records, choosing the blocksize to be large enough that compression performance approaches the limiting values listed in Table . However, to access a given record on average half a block would need to be decoded. This extra decoding effort would substantially increase response time, even for methods such as *GZip* that permit very fast decoding. Moreover, the *Huffword* method already obtains better compression than all but the best of the adaptive methods.

Table also shows that the space required by the front-coded dictionary for a large database is insignificant. For *WSJ*, with over 500 Mb of uncompressed data, the dictionary required just $1 \cdot 3$ Mb, about 1 per cent of the compressed text. This ratio justifies our contention that there is little need to look for additional gains in the representation of the dictionary.

THE DATABASE SYSTEM

We have also embedded the compression system as a transparent layer within the `mg` full-text retrieval system* and carried out experiments to determine the cost (or otherwise) of doing so. Using the *WSJ* collection we built two retrieval systems: *WSJ*, a 'before' system that stores the text uncompressed; and *WSJ–C*, an 'after' system that uses the *Huffword* compression mechanism.

To provide content-based querying the `mg` system employs an indexing method based upon compressed inverted files. This indexing method allows a complete index to every word and number within relatively modest space overheads and supports both Boolean and ranked queries. For details of the indexing method the interested reader is referred to Moffat and Zobel;[21] what should be noted here is that the indexing and text compression components of the system are completely independent of each other. Indeed, the text compression subsystem was first developed using a quite different full-text retrieval mechanism.

Processing of a query in a full-text database that uses inverted file indexing involves four stages:

1. Parsing of the query into terms and lookup of the terms in the index vocabulary.
2. Retrieval of index lists and calculation of a list of answer documents.
3. Retrieval and, if necessary, decompression of answer documents; and
4. Presentation of answers to the user.

Of these, the process we are most interested in is step 3, since steps 1, 2 and 4 are the same whether or not the collection is compressed. There are several factors that might affect the time taken by step 3, depending upon the presence or absence of compression: the time required by the disk to seek to a particular record location; the transfer time needed to fetch that record once located; and the CPU time needed to perform decompression. Compression reduces transfer time at the expense of CPU time. What is not so obvious is that compression also reduces seek times, since the file is smaller and so either fewer disk blocks are required, or shorter inter-record track distances are traversed. In extreme cases, compression might mean that one less level of indexing blocks is necessary in systems such as Unix, in which files are stored as a tree of blocks.

To see how these three factors are interrelated, consider the following example. Suppose that on some hypothetical machine a random seek takes 12 milliseconds (ms), transfer is at

* The software for this system, which includes the compression code described here, is available from `munnari.oz.au` `[128.250.1.21]` in the directory `/pub/mg`.

the rate of 2 Mb/s and decompression can be achieved at a rate of 1 Mb/s. Suppose also that the compression rate achieved is 4:1 and that a seek in the compressed file requires two-thirds of the corresponding uncompressed time, since it only has to cover 25 per cent of the distance. Suppose also that the minimum transfer is of a block of 8 Kb and that if a record spans a block boundary both blocks must be completely fetched. Then if an output record of 4 Kb is fetched in the uncompressed system the time required is 12 ms (seek) plus 4 ms (transfer of first block), plus $(4/8) \times 4 = 2$ ms (probability of transfer of second block times cost of transfer for second block), a total of 18 ms. On the other hand, in the compressed system the time taken is 8 ms (shorter seek) plus 4 ms (first block) plus $(1/8) \times 4 = 0{\cdot}5$ ms (less likely to require second block) plus 4 ms (decoding), which is 16·5 ms. Hence, given this particular hardware configuration and the simplistic model assumed for these calculations, queries that result in a list of documents of average size 4 Kb execute faster with the compressed system than they do with the uncompressed system.

It is also worth noting that the trend over the last decade has been for processor and memory performance to improve at a faster rate than disk performance and that continued change in this direction swings the balance further in favour of the use of compression. Indeed, if the point is reached at which text can be decompressed from memory at the same rate as it can be streamed from disk (and at present this requires only a further two-fold improvement in processor speed relative to disk transfer rates) then compression will become mandatory for high performance systems.

RESULTS

We ran a number of experiments to determine whether or not the interrelationships between seek, transfer and decoding rates could be quantified in practice. In all of these experiments the time taken by steps 1, 2 and 4 of the query-processing paradigm outlined above was minimized by the use of a raw query mode in which ordinal document numbers were specified to the `mg` commandline rather than query terms and by discarding the retrieved text without displaying or writing it. Also, for all of the experiments the system was configured so that all of the decoding model – about 3·3 Mb – was resident in main memory and read a single time when the query-processing program was initiated.

The experiments considered were:

1. Retrieve a small number of answers, in a cluster.
2. Retrieve a small number of answers, scattered randomly in the collection.
3. Retrieve a large number of answers, in a cluster.
4. Retrieve a large number of answers, scattered randomly in the collection.
5. Retrieve a small number of long answers.
6. Retrieve a large number of long answers; and
7. Retrieve every document.

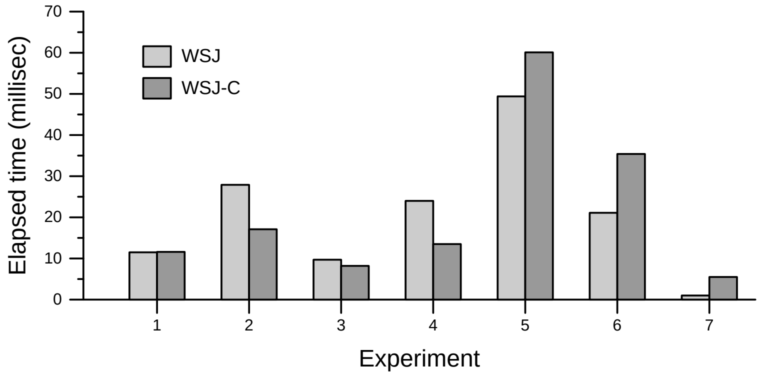
Retrieval of a small number of randomly scattered documents should maximally favour the compressed system, since the savings in seek time and transfer time can be expected to dominate. On the other hand, when the documents are clustered, seek times should be small anyway and the savings brought about by compression of limited impact. Similarly, retrieval of long documents should maximally favour the system storing uncompressed data; and when many documents are being decoded, the average distance between answers will be small anyway and so again CPU time might dominate. For these reasons, the seven experiments were considered to cover the full spectrum of possible usage of such a database and not in any way favour the compressed system.

For the *WSJ* collection, with roughly 175,000 documents, we deemed a small number of answers to be 10 and a large number to be 1,000. These correspond to an average of about 30 Kb and 3 Mb of text respectively. Rather than try to construct actual queries that achieved the desired distribution of answers, synthetic queries were supposed and lists of random document numbers were generated in each category. For the 'small' experiments 1,000 queries were constructed and for the 'large' experiments 10 queries were constructed. These were used to directly drive the text retrieval and decoding mechanism. The 'random' document numbers (experiments 2 and 4) were generated using a pseudo-random number generator, without replacement, that selected from the whole of the *WSJ* collection. The 'clusters' of document numbers (experiments 1 and 3) were commenced at a randomly chosen location and completed by assuming that each subsequent document had probability of 1 in 10 of being an answer, until the required number of documents had been selected. Hence, the 'large, clustered' experiments each spanned about 10,000 documents, or around 6 per cent of the collection. A 'small number of long documents' (experiment 5) was achieved by ordering the collection by decreasing document length and taking the longest ten documents to be the first query, the next group of ten documents to be the second query and so on, until 100 queries had been formed. The first query in this set resulted in more than 1 Mb of text being decoded, and, averaged over the 100 queries, the average document length was over 18 Kb. A 'large number of long documents' (experiment 6) was taken to be the same set of 1,000 document numbers; this experiment consisted of just a single query. Experiment 7 was effected by building into the retrieval mechanism a loop that retrieved every stored document.

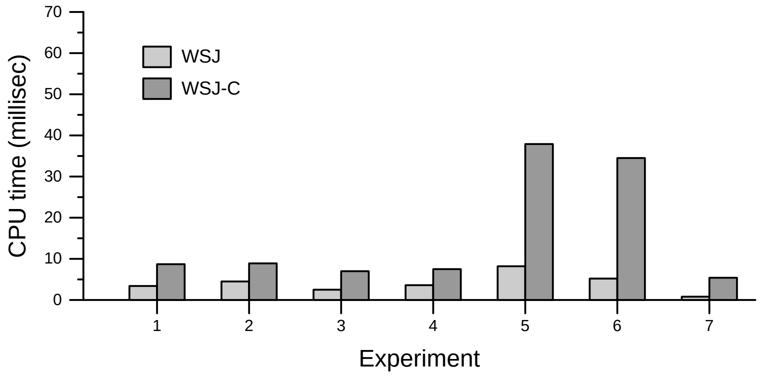
For all of the experiments the list of document numbers was sorted and documents were fetched in increasing document number order. Making a uni-directional pass through the text file retrieving answers minimizes seek time; and Boolean query-processing using an inverted file index produces exactly such a sorted list of answers.[21]

Figure 2 summarizes the results obtained when these query sets were executed. All times plotted are in milliseconds per answer and are the average of 50 complete executions of the respective experimental query sets except for experiment 7, for which 5 runs were performed. Hence, the times for experiments 1 to 4 are the average over 500,000 document-retrievals; for experiments 5 and 6 the averages are over 50,000 document-retrievals; and for experiment 7, the average is over 800,000 retrievals.

Execution rates are again for C source code executed on a Sun SPARC 10 Model 512 and CPU ('user' and 'system' time added together) and elapsed times are as reported by the Unix command `/usr/bin/time`. The programs themselves did not use any asynchronous disk operations and all data transfer followed the sequence 'need data; seek to the specified address; initiate read; wait for read to complete; use that data'. A separate file of document addresses was used to locate the start of each document in the main text file, but all accesses to this 600 Kb index were performed before any accesses to the main file, so that there would be no interleaving of operations on two different files. Finally, to minimize any caching effects, the runs were completely interleaved: run 1 on experiment 1 on *WSJ*, then on *WSJ–C*; then run 1 of experiment 2 on the two collections and so on. That is, each experiment was executed at regular intervals throughout the overall testing, which took approximately 20 hours. Experiments were run using local disks and on a machine with no other active user processes and so the differences between CPU times and elapsed times genuinely reflect the cost of accessing the two main text files.



(a) Elapsed time (milliseconds per answer)



(b) CPU time (milliseconds per answer)

Figure 2. Retrieval times, with and without compression, average of 50 runs

The disk used for all of the experiments has an average seek time of 12 ms; has 2,975 0·5-kilobyte sectors per cylinder (spread over 25 surfaces); blocks of 16 sectors; and has a spin rate of 5,400 revolutions per minute. A total of 2,570 cylinders hold 3·7 Gb.

In all of the experiments the compressed *WSJ–C* database required more CPU time per answer than does the uncompressed *WSJ* system, simply because of the need to decompress the text. However the elapsed times – that is, including the cost of all disk seek and data transfer operations – for the compressed system were *better* than the elapsed times for the uncompressed system for the first four experiments. In experiments 2 and 4 especially, in which there is a great deal of random seeking, the compressed system enjoyed a distinct performance advantage. But even when the answers were clustered in experiments 1 and 3 there was a small gain in query throughput to be achieved by the use of compression.

Experiments 5 and 6, on the other hand, reflected the cost of processing long documents and substantial amounts of CPU time can be required if answers are longer than a few kilobytes. The 18 Kb average document length in these queries resulted in around 30 msec of decoding effort, greater than the cost of seeking for and fetching the record in the uncompressed file. The first query in experiment 5 required more than 1·5 seconds of CPU time in *WSJ–C*.

We spent some time exploring possible explanations for the particularly good behaviour of the compressed *WSJ–C* database on the random queries (experiments 2 and 4). Although the model predicted that the compressed system would outperform the uncompressed system, the margin was rather more than had been expected. Possible contributors are the buffering strategy employed by the operating system, the extent to which the operating system asynchronously pre-fetches data after each read and the degree to which each of the two files resided on contiguous areas of disk.

To attempt to determine if the latter factor – physical layout on the disk – was influencing the results, we made copies of the two databases to fill up the remaining disk space (to 94 per cent full and 98 per cent full for *WSJ* and *WSJ–C* respectively) on the drive and then ran the experiments on the copies. We also ran experiments on a copy of *WSJ–C* copied onto the disk partition that contained *WSJ*. Since the original databases were never moved, in all of these experiments the copied databases would have been located on physically different parts of the disk from the originals and possibly would have been more fragmented. Nevertheless, the results obtained were almost identical to those presented in Figure 2. It would appear that layout and physical location of the blocks containing the *WSJ* and *WSJ–C* databases is not a major factor in the performance difference.

The difficulty we had in establishing the exact reasons for the results we have measured highlight the complexity of modern operating systems and the consequent need for great care when performing experiments. We have no doubt that, in the experiments performed, the compressed system was faster: the results were consistent within a small range and were repeatable. Our doubt is in the exact combination of factors that caused the result. Longer seeking time accounts for at least part of the difference, but it is also true that we were quite surprized by the magnitude of the improvement observed in experiments 2 and 4.

Although we have focussed on compression methods for the text of a retrieval system and used synthetic queries to test the proposed system, it is worth noting that compression also plays a large part in the storage of the index in the `mg` retrieval system. For the *WSJ* collection a full inverted index, storing the document number of every appearance of all words and numbers (including so called stop words such as ‘the’) required just 33 Mb, about 6 per cent of the space required by the raw text. The complete *WSJ* retrieval system, including compressed text, compressed index, and all other auxiliary files, totalled 180 Mb, or less than 36 per cent of the original text.

CONCLUSION

We have shown that it is possible to substantially compress document databases while retaining fast access to individual documents. We achieved this saving with a semi-static word-based model and Huffman coding; for data encoded using this combination fast decompression is possible and for typical queries the compressed database was *faster* than the equivalent uncompressed database.

The disadvantage of the semi-static approach is that two passes over the data are required for compression. We believe, however, that this is of minimal importance in the context of a document database system, in which changes are rare. Furthermore, if updates do take place, the inconvenience of periodically rebuilding the compressed database is far outweighed by the advantage of saving, in the examples we have considered, over 70 per cent of the space required to store the collection.

ACKNOWLEDGEMENTS

Gary Eddy, Neil Sharman and Tim Shimmin provided programming assistance; and Tim Bell answered our many questions about the workings of the Witten-Bell-Nevill scheme described in Reference 2. We also thank the referees for their comments and support.

REFERENCES

1. D. K. Harman, ed., *Proc. TREC Text Retrieval Conference*, Washington, November 1992. National Institute of Standards Special Publication 500-207.
2. I. H. Witten, T. C. Bell and C. G. Nevill, 'Indexing and compressing full-text databases for CD-ROM', *Journal of Information Science*, **17**, 265–271 (1992).
3. G. V. Cormack, 'Data compression on a database system', *Communications of the ACM*, **28**, (12), 1336–1342 (1985).
4. S. T. Klein, A. Bookstein and S. Deerwester, 'Storing text retrieval systems on CD-ROM: Compression and encryption considerations', *ACM Transactions on Information Systems*, **7**, (3), 230–245 (1989).
5. D. G. Severance, 'A practitioner's guide to data base compression', *Information Systems*, **8**, (1), 51–62 (1983).
6. J. Rissanen and G. G. Langdon, 'Universal modeling and coding', *IEEE Transactions on Information Theory*, **IT-27**, 12–23 (1981).
7. T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ, 1990.
8. I. H. Witten, R. Neal and J. G. Cleary, 'Arithmetic coding for data compression', *Communications of the ACM*, **30**, (6), 520–541 (1987).
9. D. A. Huffman, 'A method for the construction of minimum redundancy codes', *Proc. IRE*, **40**, (9), 1098–1101 (1952).
10. J. Bentley, D. Sleator, R. Tarjan and V. Wei, 'A locally adaptive data compression scheme', *Communications of the ACM*, **29**, (4), 320–330 (1986).
11. A. Moffat, 'Word based text compression', *Software—Practice and Experience*, **19**, (2), 185–198 (1989).
12. R. N. Horspool and G. V. Cormack, 'Constructing word-based text compression algorithms', *Proc. IEEE Data Compression Conference*, eds., J. A. Storer and M. Cohn, Snowbird, Utah, March 1992, pp. 62–81. IEEE Computer Society Press, Los Alamitos, California.
13. R. G. Gallager, 'Variations on a theme by Huffman', *IEEE Transactions on Information Theory*, **IT-24**, (6), 668–674 (1978).
14. J. B. Connell, 'A Huffman-Shannon-Fano code', *Proc. IEEE*, **61**, (7), 1046–1047 (1973).
15. D. Hirschberg and D. Lelewer, 'Efficient decoding of prefix codes', *Communications of the ACM*, **33**, (4), 449–459 (1990).
16. A. Moffat, N. B. Sharman and J. Zobel, 'Static compression for dynamic texts', *Proc. IEEE Data Compression Conference*, eds., J.A. Storer and M. Cohn, Snowbird, Utah, March 1994, pp. 126–135. IEEE Computer Society Press, Los Alamitos, California.
17. P. Elias, 'Universal codeword sets and representations of the integers', *IEEE Transactions on Information Theory*, **IT-21**, (2), 194–203 (1975).
18. A. S. Fraenkel and S. T. Klein, 'Bounding the depth of search trees', *The Computer Journal*, **36**, (7), 668–678 (1993).
19. A. Moffat, A. Turpin and J. Katajainen, 'Space-efficient construction of optimal prefix codes', *Proc. IEEE Data Compression Conference*, eds., J.A. Storer and M. Cohn, Snowbird, Utah, March 1995. IEEE Computer Society Press, Los Alamitos, California. To appear.
20. A. Moffat, 'Implementing the PPM data compression scheme', *IEEE Transactions on Communications*, **38**, (11), 1917–1921 (1990).
21. A. Moffat and J. Zobel, 'Self-indexing inverted files for fast text retrieval', *ACM Transactions on Information Systems*. To appear. Preliminary version in *Proc 5'th Australasian Database Conference*, Christchurch, New Zealand, January 1994, pp. 79–91.