Compressing the Digital Library

Timothy C. Bell[1], Alistair Moffat[2], and Ian H. Witten[3]

[1]*Department of Computer Science, University of Canterbury, New Zealand, tim@cosc.canterbury.ac.nz*
[2]*Department of Computer Science, University of Melbourne, Australia, alistair@cs.mu.oz.au*
[3]*Departments of Computer Science, University of Calgary, Canada, and University of Waikato, New Zealand; ian@cpsc.ucalgary.ca*

Abstract

The prospect of digital libraries presents the challenge of storing vast amounts of information efficiently and in a way that facilitates rapid search and retrieval. Storage space can be reduced by appropriate compression techniques, and searching can be enabled by constructing a full-text index. But these two requirements are in conflict: the need for decompression increases access time, and the need for an index increases space requirements. This paper resolves the conflict by showing how (a) large bodies of text can be compressed and indexed into less than half the space required by the original text alone, (b) full-text queries (Boolean or ranked) can be answered in small fractions of a second, and (c) documents can be decoded at the rate of approximately one megabyte a second. Moreover, a document database can be compressed and indexed at the rate of several hundred megabytes an hour.

Keywords: Compression, indexing, full-text retrieval, inverted files, query processing.

1. Introduction

Digital libraries contain many gigabytes of text, images, and other data such as video and audio recordings. The large size of these collections poses two problems. First, they require considerable storage space; and second, the simple act of accessing the library to fulfill some information requirement can be very slow. In conventional applications these two problems are dealt with by using data compression and indexing respectively. Data compression reduces the amount of disk required to store information by recoding it in a more efficient manner. Indexing is used in information retrieval systems so that fast access can be provided to the documents stored, in the same way that the index of a conventional book provides fast access to the pages in which the main concepts of the book are mentioned. The challenge facing implementors of digital libraries is that these two are in tension—data compression saves space, but at the expense of added access time; and indexing methods provide fast access, but usually at the expense of considerable amounts of additional storage space. Indeed, a complete index to a large body of text can be larger than the text itself—after all, it might store the location of every word in the text.

This paper shows that it is possible to make compression and indexing work together efficiently. We describe compression methods suited to large amounts of text that allow both random-access decoding of individual documents and fast execution; we show how the index can itself be compressed so that only a small amount of overhead space is required to store it; and we show how the application of compression techniques allows efficient construction of the index in the first instance. The result is a system that can take a large body of text, and convert it to a compressed text *and* index that together take up less than half the space occupied by the original data. Combining the two techniques incurs little penalty in access speed—in fact, the access speed can even be improved, since there is less data to be read in from slow secondary storage devices. Moreover, the initial indexing and compression processes can be effected on a mid-range workstation at a rate of several hundred megabytes an hour.

The ideas described in this paper have been implemented in a public-domain full-text retrieval system called MG, which stands for “Managing Gigabytes.”[1] The MG system is able to index and compress text, and is also able to deal with bilevel images such as scanned documents and with gray-scale images such as scanned photographs, all in the same framework. MG provides both fast access and efficient storage. For example, on a test database of 100,000 newspaper articles totaling over 250 megabytes, to locate and return all documents containing both of the words *managing* and *gigabytes* took MG a tenth of a second. Moreover, the compressed text, compressed index, plus all other

[1]The MG system is available by anonymous ftp from `munnari.oz.au` in the directory `/pub/mg`.

auxiliary files, occupy less than 36% of the space required for the unindexed source text.

A more thorough treatment of such systems, which are capable of storing and indexing text, images, and other data, can be found in [8]. In this paper we concentrate on how compression can be used to decrease the space required by a document collection and its index. That is, we are essentially considering the full-text retrieval situation, albeit it in the context of an integrated system.

2. Data compression

Data compression allows secondary storage consumption to be reduced by recoding information into a more economical representation. By carefully processing data that is being stored, its size can be reduced, yet the original data can be restored before being displayed to a user. The normal cost of this space saving is the expense of extra computation. However, CPU performance continues to increase at a faster rate than disk performance, and so it is becoming increasingly attractive to pay this price. In some cases—such as when data is distributed on CD-ROM—the storage savings that arise with compression make it possible to manipulate volumes of data that would otherwise be impossible.

Compression systems use many different strategies. One important distinction is whether a system is *lossless* or *lossy*. In a lossless system, the original data is reconstructed exactly as it was originally. This contrasts with a *lossy* system, where the data is "similar" to the original, but not necessarily identical. Lossy systems are appropriate for data such as images and audio, where the stored data is already an approximation to the original source. Lossless compression is essential for text and similar discrete data, where even the smallest of changes is unacceptable and may result in complete reversal of meaning.

Several standards have been developed for compression. To date these have primarily been in the area of image and video compression, and often describe lossy techniques: examples include the gray-scale JPEG standard, the motion picture MPEG standard, and the (lossless) JBIG binary image standard. In this paper we focus on lossless compression methods for *textual* data, an area in which there are presently no official standards.

The most popular text compression methods for conventional applications are *adaptive*, which means that they change the method of coding to suit the type of data being compressed [8]. Such techniques make only one pass through the data, and adaptivity is obtained by basing the coding strategy on the text already processed and hence known to the decoder as well as the encoder. The main advantage of such methods is that the text need only be processed once, and so encoding can be done as a "filter" process in applications such as communications channels. However, the adaptive approach is not suitable for full-text retrieval situations, because decoding from a specified point in a file requires that all data prior to that point be decoded first. The adaptive compressor could be reset at the beginning of each indexed point in the data, but these are usually sufficiently frequent that the compressor has little time to adapt to the text before being reset, and compression effectiveness is poor.

Fortunately it is not so important to use adaptive compression in this situation. The data is usually archival in nature, which means that it is relatively static, and so a suitable coding strategy can be pre-determined by a prior scan of the data. For this reason, *semi-static* compression methods are likely to be more appropriate than adaptive ones in information retrieval applications. Two passes must be made over the source text, but the expense of the extra pass is incurred only once, at database creation time, while the benefits—fast decoding and random access—are exploited every time the retrieval system is accessed.

Also important to effective data compression is the distinction between *modeling* and *coding*, an idea that was articulated in the 1980s [5]. In a data compression system the *model* provides *predictions* for symbols in the text, in the form of a probability distribution. For example, after the characters *digital libra*, the model might "predict" that the next character will be a letter *r* with high probability, and that all other characters are unlikely. The *coding* task involves taking this probability and converting it into bits to represent the symbol. Shannon's work [7] has established that the appropriate number of bits to encode a symbol of probability p is $–\log_2 p$.

Huffman's algorithm is a well-known mechanism for performing the coding task [4]. Huffman's method does not necessarily achieve Shannon's bound, but it has been shown that it cannot exceed the bound by an average of more than $\Pr[s_1] + 0.086$ bits per symbol, where $\Pr[s_1]$ is the probability of the most likely symbol to be coded [2]. The value of $\Pr[s_1]$ will be high when the probability distribution is skew and accurate predictions are being made, and in this case Huffman coding can be very inefficient. In such cases, the more recent technique of *arithmetic coding* (see [8] for a description and references) is appropriate. Arithmetic coding achieves compression arbitrarily close to Shannon's bound, and is particularly well suited for use with adaptive compression models. Huffman's method is better suited if the probability distribution does not contain any high-probability symbols, if the model is static, and if speed is important.

Given that good tools are available for coding, the main question in compression is how best to model the data being compressed. The best sort of model is one that makes good "predictions," that is, one that estimates a high probability for the symbols that actually occur. In the next two sections we shall examine models that are suitable for predicting the structure of a body of text, and for predicting the entries that occur in an index.

3. Compressing the main text

Choosing a compression method for an application invariably involves a tradeoff between the amount of compression achieved and the speed at which compression (or, more importantly in the present application, decompression) is performed. In a full-text retrieval situation, the data being compressed is relatively static, so it is possible to construct a coding scheme that is specific to a particular document collection.

We have experimented with several strategies for compressing text in this situation, and have found that *word-based* compression is particularly suitable. In a word-based compression system, a "dictionary" is constructed that contains all the "words" in the collection. A word might be defined, for example, to be a maximal contiguous sequence of alphanumeric characters. The text is then coded by replacing each word in the text with a reference to its position in the dictionary. The punctuation and white space that separate words can also be treated in the same way; a dictionary of such strings or "non-words" is constructed, and a single code used to represent all of the characters that separate each pair of words. The compressor alternates between the two dictionaries.

Word-based compression is not suitable as a general-purpose data compression method because it assumes that the input can be broken up into words. However, it is suitable for text, and results in effective compression. It also provides fast decoding—several characters are coded onto each codeword, and so only a few operations are required per output byte.

Table 1 shows some of the "words" that would be used for word-based coding of a sample text we shall refer to as *WSJ*, approximately 100,000 articles from the *Wall Street Journal* totaling 266 megabytes of text. The first column shows words parsed from the text, the second shows their frequency, and the third shows their estimated probability, which is simply the words' relative frequencies. In total there are 212,000 distinct words, 5,800 distinct non-words, and about 46,000,000 tokens of each sort are actually coded to make the compressed text.

The are several methods that might be used to code words in this dictionary. A simple one is to use a fixed-length number—of $\lceil \log_2 212{,}000 \rceil = 18$ bits, in this case—to identify a word. Better compression can be obtained by basing the coding on the probability distribution, using Huffman or arithmetic coding. In this semi-static situation, Huffman coding is ideal because there are no high-frequency symbols (in the example the most frequent word, *the*, has a probability of just 4.42%); and because a particularly efficient variant known as *canonical* Huffman coding can be used. The fifth column of Table 1 shows a canonical Huffman code for the words. Like a conventional Huffman code, the codewords for more probable symbols are shorter than those for less probable ones. However, in the canonical code the codewords are chosen so that when the codewords are sorted in lexical order, they also run from longest to shortest. This means that codes of the same length are simply a sequence of consecutive binary numbers, so it is not necessary to store a complete decoding tree, only the lexicographically first codeword

Word	Frequency	Estimated probability	Length (bits)	Codeword
the	2,024,105	4.42%	5	11111
of	1,147,721	2.51%	5	11110
to	1,083,451	2.37%	5	11101
a	931,603	2.03%	6	111001
and	798,442	1.74%	6	111000
in	739,764	1.62%	6	110111
s	473,111	1.03%	7	1101101
that	408,296	0.89%	7	1101100
for	401,440	0.88%	7	1101011
The	355,042	0.78%	7	1101010
is	321,292	0.70%	7	1101001
said	303,628	0.66%	7	1101000
on	248,643	0.54%	8	11001111
it	231,415	0.51%	8	11001110
...	...	...	...	...
zurui	1	0.000002%	25	0000000000000000000000100
zwei	1	0.000002%	25	0000000000000000000000011
zygal	1	0.000002%	25	0000000000000000000000010
zymurgical	1	0.000002%	25	0000000000000000000000001
zz	1	0.000002%	25	0000000000000000000000000

Table 1 Word-based coding example

of each length. Thus, to record the code in Table 1, the only 7-bit code that is stored is the one for *said*, along with a pointer to the position of *said* in the wordlist of column 1. To calculate the code for *for*, we simply observe that it comes three places before *said* in the wordlist, and obtain its code by adding 3 to the code for *said*. Exactly the same principle guides decoding. For example, the code 1101100 is four greater than the first 7-bit codeword (1101000, for *said*), and so the corresponding symbol must be *that*. This arrangement of codewords allows fast decoding with a minimum of decode-time memory.

Using two Huffman codes, one for the words and one for the non-words, the 266 megabyte *WSJ* collection is reduced to 73 megabytes, or just 27.5% of the original size. On average, each word is coded into 11.1 bits, and each non-word in 2.3 bits. These compare favorably with the 18 and 13 bits that would be required by a fixed binary encoding.

4. Compressing indexes

An index is used to locate documents that contain a given term or terms specified in a query. *Boolean* queries involve a combination of terms and operations such as AND and OR that are effected by calculating the intersection and union respectively of the sets of documents containing each term. Alternatively, *ranked* queries might be provided, which score documents according to some similarity measure, and the set of "most similar" documents is returned as an answer to the query [6].

There are several data structures that can be used to support Boolean queries. One of the simplest is a "bitmap," which is a two-dimensional array of bits in which each row corresponds to a term that appears in the collection, and each column corresponds to one document. An entry in the bitmap is set to *true* if the corresponding term occurs in the indicated document. Documents that contain a Boolean combination of terms can be located by performing the corresponding bitwise operation on rows of the bitmap.

The bitmap is an excellent tool for thinking abstractly about the indexing process, but in practice it is usually far too large to be stored explicitly. Most of the entries in a bitmap are *false* (i.e., zero), and so methods for storing sparse matrices can save considerable space. Two general strategies that are suitable are referred to as *inverted files* and *signature files*. These two strategies are not normally regarded as compression techniques, but do in fact perform the function of bitmap compression. In an inverted file, each row of the bitmap is replaced by a list of document numbers that indicate where the 1-bits occur. In other words, it simply stores a list of locations for each term. Signature files decrease the storage requirements by retaining the "one bit per column" of the bitmap, but allowing multiple terms to be stored in any given row. The effect of the ambiguity so caused is minimized by storing each term in several of the rows, so that the probability of any two terms corresponding to exactly the same set of rows is small. The resulting structure is probabilistic, in that it may now indicate that a term is present in a document even when it is not, and any text retrieved must be checked for false matches before being presented. However, if a sufficient number of rows are used for each term, the probability of false matches can be kept low.

Signature files are generally more efficient than simple inverted files, in terms of both speed and storage requirements. However, compression techniques can be applied to inverted files to make them much smaller than an equivalent signature file, and yet they need not be much slower. The improved storage is obtained by modeling the data stored in the inverted file. Moreover, signature files are ineffective if the documents vary widely in length, or if ranked queries must be supported. The latter difficulty is a result of the need, when evaluating ranked queries, to know not only whether or not a term appears in a document, but also the number of times it appears. In these cases inverted files are the indexing method of choice.

Compressing an inverted file requires constructing a model to predict where a particular word is likely to occur in a text. One effective way to predict word occurrences is to assume a simple statistical model, and assign a probability to each of the different possible gaps between occurrences of a word. The size of each gap that actually occurs is coded with respect to this model. The frequency of occurrence of a word has a significant influence on the expected size of a gap. For example, there will be many words that occur only once in the entire text, for which only one position is encoded. This position is essentially random. At the other extreme, common words such as *the* occur regularly throughout the text, and small gap sizes can be expected. In order to capture all of these possibilities, the model for occurrences of words needs to be parameterized by the frequency with which the word occurs.

The probabilities of gap sizes implied by this sort of model are conveniently coded by representing the sequence of actual gap sizes using a *Golomb* code [3]. The length of a Golomb code increases as the value being encoded increases, but the exact relationship between the value and its coded length is determined by a parameter. The underlying model for this code is that the words are randomly distributed in the index list, and that the distribution of gap sizes follows a geometric distribution. To compress an inverted file entry, the parameter depends in turn on the frequency of the term corresponding to the entry—the probability that a randomly selected document contains the word in question. For rare terms the parameter is selected to favor large gaps; while frequently occurring terms use an assignment that generates short codewords for small gaps and relatively long codewords for the supposedly unlikely long gaps. Golomb codes can also be encoded and decoded quickly, which means that little time is

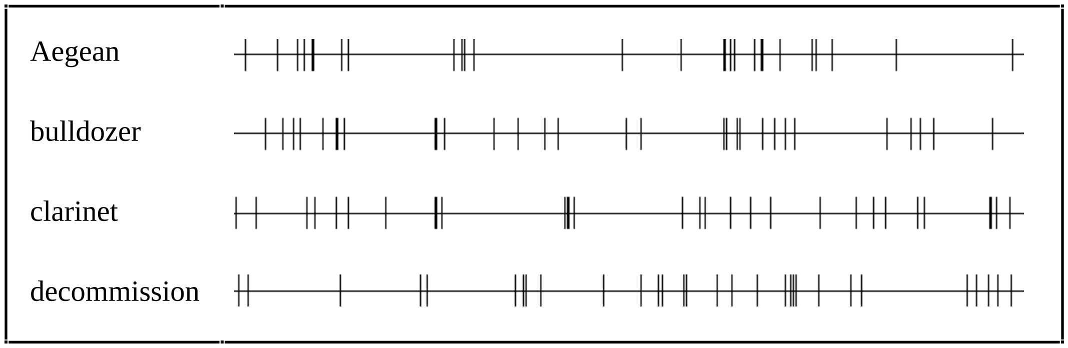

Figure 1 Positions of words in the *WSJ* collection

required to decode the inverted lists needed to answer any query.

The above model assumes that words are equally likely to occur anywhere in the text, which is not the case in practice. For example, Figure 1 shows diagrammatically the locations of four different words in the *WSJ* collection. Each of the words appears in exactly 30 documents, and each appearance is indicated by a vertical tick. Notice that the words tend to appear in clusters, even quite generic terms such as *bulldozer*, *clarinet* and *decommission*. In this case the order of the documents in the collection is chronological, and so it is not surprising that there are often small gaps between articles on the same or related subjects—an item of news is topical, and appears in several consecutive issues, but then drops away. Clustering can be modeled by predicting small gaps between words once they have been observed once, and longer gaps once the cluster appears to have finished. A model based on this idea can give slightly better compression than one that does not take account of clustering, although at the disadvantage of requiring considerably more memory space.

When compressed, the index is small. For the 266 megabyte *WSJ* text a Golomb coded index requires an average of 5.2 bits per document number and 2.0 bits per within-document frequency, a total of just 18 megabytes, roughly 7% of the size of the input text. Moreover, this index stores the document number and frequency of *every* word and number, including common terms such as *the, and,* and *1,* and so permits Boolean or ranked queries on any combination of terms extracted from the text.

5. Creating indexes

Before the index can be compressed, it must be constructed. In the large-scale applications we have been considering here, generating the index is a non-trivial task. Indeed, constructing an index for a large text using elementary methods may take many months of computer time—an impossible requirement, particularly if the index is to be revised or updated frequently.

One method of index construction operates as follows. First, the text is processed. As each document is considered, a list of terms that appear in it is generated. This list is written to a temporary file of $<t, d, f_{d,t}>$ triples, indicating that term t appears in document d with frequency $f_{d,t}$. The file is generated in increasing order of d, and random order of t. When the input text has been completely processed, the temporary file is sorted into order of increasing t, and within equal values of t, of increasing values of d. Once this has been done, the inverted index can be generated directly by sequentially reading the temporary file, and constructing the compressed inverted lists. This method operates within modest amounts of time, but has the drawback of requiring considerable extra disk space for the temporary file. Indeed, if conventional sorting methods are used, *two* copies of the temporary file are required.

Compression can be used to reduce this requirement. Within the temporary file the d field in each pair can represented as a d-gap, or difference, using a fixed code for integers such as Elias's δ code [1]. Similarly, provided that the temporary file is written as a sequence of sorted runs, Elias's γ code can be used to represent t-gaps economically; and γ is also appropriate for the $f_{d,t}$ values. Use of these methods allow the temporary file to be stored in only about 30% more disk space than is required by the final compressed inverted file. An in-place merging algorithm is then the last phase of the operation. This converts the temporary file into the desired inverted file without requiring any additional disk space at all. Hence the entire process can be executed in only a little more disk space than is eventually occupied by the compressed index that is generated. Details of this scheme can be found in [8].

For the same 266 megabyte *WSJ* text, the index can be constructed using this algorithm in less than 30 minutes on a mid-range workstation [2] using just a few megabytes of main memory. Compression of the text of the collection requires a further 30 minutes. In total, it takes about one hour to build a complete retrieval system for this text. We have also experimented with a two gigabyte collection, of which the *WSJ* database is one

[2]Sun SPARC 10 Model 512, with programs written in C.

small part; and all of the results mentioned here scale up linearly to this large corpus.

6. Query processing

Despite the compression applied to both text and index, query processing is fast. Only the required portions of the index are decoded at each query, and this is accomplished at a rate of about 400,000 document numbers per second. The canonical Huffman code used for the text decompresses at a rate of approximately one megabyte per second. Queries usually return a few tens or hundreds of kilobytes of text (for example, a typical novel is only a few hundred kilobytes), and so unless a query involves a pathological number of terms or a very large volume of output text, response is completed within tenths or hundredths of a second. For example, it took just 0.1 seconds to process the query "managing AND gigabytes" against the *WSJ* collection, including the decoding of 13,000 document numbers from the index, and the decompression of 15 kilobytes of answer text. Eight disk accesses were required.

Ranked queries are slower, but principally because they usually contain more terms than because of any intrinsic expense. For example, to extract the top twenty ranked documents for the query "managing gigabytes compressing indexing documents images" from the *WSJ* collection took about one second (0.4 seconds of CPU time) and produced 50 kilobytes of output. More than 45,000 pointers were decoded to answer this query, and a total of 53 disk accesses performed.

These rates are comparable with those that would be obtainable with a system that did not use compression. In the absence of compression, less CPU time would be required, but overall query response time would remain about the same because of the need for 4–6 times as much data to be retrieved from the disk.

7. Conclusion

Compression is important for taming the resource-hungry digital library. In fact, using compression it is possible to store fully indexed data in less than half the space required by the original text. Moreover, retrieval speed need not be significantly worse, because compression methods are available that require relatively little computation, and lower volumes of data are handled by secondary storage devices. Compression effectively provides a means for exploiting the increasingly powerful CPU available in modern computers.

We have also examined the demanding process of creating an index, and demonstrated how compression can be used to balance the use of resources so that the indexing process does not require unusually large quantities of disk space or CPU time.

Working with very large libraries of information requires a delicate balance between the use of primary memory, secondary memory, and CPU power. Compression is an important tool both for facilitating the storage of large amounts of data, and for generating and storing the indexes that are required to access it.

References

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

[2] Gallagher, R.G. (1978) "Variations on a theme by Huffman." *IEEE Trans. on Information Theory*, IT-24:668–674.

[3] Golomb, S.W. (1966) "Run-length encodings." *IEEE Trans. on Information Theory*, IT-12(3):399–401.

[4] Huffman, D.A. (1952) "A method for the construction of minimum redundancy codes." *Proc. IRE*, 40(9):1098–1101.

[5] Rissanen, J. and Langdon, G.G. (1981) "Universal modeling and coding." *IEEE Trans. on Information Theory*, IT-27:12–23.

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

[7] Shannon, C.E. and Weaver, W. (1949) *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, Ill.

[8] Witten, I.H., Moffat, A. and Bell, T.C. (1994) *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York.