

Text Compression for Dynamic Document Databases*

Alistair Moffat[†] Justin Zobel[‡] Neil Sharman[§]

March 1994

Abstract

For compression of text databases, semi-static word-based methods provide good performance in terms of both speed and disk space, but two problems arise. First, the memory requirements for the compression model during decoding can be unacceptably high. Second, the need to handle document insertions means that the collection must be periodically recompressed, if compression efficiency is to be maintained on dynamic collections. Here we show that with careful management the impact of both of these drawbacks can be kept small. Experiments with a word-based model and 500 Mb of text show that excellent compression rates can be retained even in the presence of severe memory limitations on the decoder, and after significant expansion in the amount of stored text.

Index Terms Document databases, text compression, dynamic databases, word-based compression, Huffman coding.

1 Introduction

Modern document databases contain vast quantities of text. It is generated endlessly by newspaper reporters, academics, lawyers, and government agencies; and comes in packages ranging from 10-line sonnets to multi-megabyte judicial findings. The challenge to designers of document databases is to provide mechanisms that not only store such text, but do so in an efficient manner, as well as allow users to selectively retrieve documents based upon their content. There are thus two problems to be addressed when text databases are designed. First, mechanisms for indexing and accessing text must be considered, since without an index, query processing is intractable. There have been many different strategies proposed

*This paper includes material presented in preliminary form at the 1994 IEEE Data Compression Conference.

[†]Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia; Telephone +61 3 3449168; Facsimile +61 3 3481184; Internet alistair@cs.mu.oz.au

[‡]Department of Computer Science, RMIT, GPO Box 2476V, Melbourne 3001, Australia.

[§]Department of Computer Science, The University of Melbourne, Australia 3052.

for indexing text [7, 8]. Second—and this is the problem considered in this paper—there is the need to efficiently represent the documents in the first instance. This is the problem of text compression.

There are good reasons to compress the text stored in a document database system. Not only are the disk space requirements dramatically reduced, but, with choice of an appropriate compression scheme, the CPU cost of decoding can be partially or entirely compensated for by the reduction in time required to fetch the documents from disk. One suitable regime is the use of a semi-static word-based model [2, 11, 14, 23] coupled with canonical Huffman coding [10, 12, 16, 24], a combination which has all of the necessary properties for application to text databases: excellent compression, fast decoding, and individual documents are independently decodable [1]. Augmented-alphabet character models have also been used in text database applications [3], but do not obtain the same compression rates as the word-based model.

In *semi-static* modelling a preliminary pass over the text is used to gather statistics about the frequency of occurrence of each token. In a word-based model the tokens are organised in two lexicons: the *words*, or sequences of alphanumeric characters, and *non-words*, or sequences of non-alphanumeric characters. The statistics accumulated in the first pass are used to build two probability distributions, one for each lexicon. For Huffman coding these probabilities—derived from the symbol occurrence counts—are used to generate an assignment of distinct bitstrings, or *codewords*, one for each token in the lexicon. The length of each code is inversely governed by the frequency of the corresponding token, so that common words have short codes and *vice versa*. In general, if the probability of the i 'th symbol is p_i then it should be assigned a code of $-\log_2 p_i$ bits. If this assignment can be achieved then the average cost of storing the information, measured in bits per symbol and averaged over an alphabet of n symbols, is given by the entropy

$$\sum_{i=1}^n -p_i \cdot \log_2 p_i.$$

By Shannon's source coding theorem, the compression is then optimal [18].

After the models have been computed, a second pass is used to encode the data with respect to the models, by replacing each token with its code. Words and non-words are strictly alternated, and so the compressed representation can be unambiguously decoded to construct an exact replica of the original text—*lossless* compression. Use of a word-based model typically reduces size to around 25–30% of the original text, or roughly 2.2 bits per input byte. Moreover, use of a semi-static model allows random access into the compressed collection. This is in contrast to the problems presented by random access into texts compressed with an adaptive model [1].

If integral-length codewords are to be assigned and the probability distribution is not dyadic then some compression inefficiency must be tolerated, since it is not possible to assign a codeword of exactly $-\log_2 p_i$ bits unless p_i is an integral power of $1/2$. However Huffman's algorithm minimises this inefficiency for fixed codewords, and in many applications—

including the word-based model assumed here—comes remarkably close to the entropy [13]. Moreover, when two passes over the source data can be made and a static assignment of codewords employed, Huffman coding provides extremely quick decoding, requiring little more than a single “shift and test” operation per input bit.

There are two drawbacks to this word-based Huffman-coded compression technique. The first is that a great deal of decode-time memory space might be required to store the lexicon of the model of words, that is, to store all of the distinct words* occurring in the database. In our experience the number of distinct words in a text grows as an almost linear function of its size, without the tailing-off effect often predicted [26]. These new words are often acronyms and place names, but it is also worth noting that new misspelt words occur at a reasonably constant rate, and all are regarded as novel by the compression system. As part of the international *TREC* information retrieval experiment we have been dealing with several corpora of English text [9]. One of the collections is several years of articles from the *Wall Street Journal*, and this 508.15 Mb *wsj* database uses 289,101 distinct words totalling 2,159,044 characters; and 8,912 distinct non-words requiring in total 77,882 bytes. Allowing a 4-byte string pointer for each word, and ignoring for the moment the possibility of storing the words compressed in memory, the total requirement during decompression is about 3.3 Mb, a non-trivial amount even by workstation standards. Moreover, there is only limited overlap when these parts are combined to make the 2 Gb *trec* collection. Using the same method of calculating space, to decode *trec* more than 11 Mb of memory is required.

The second drawback of this compression regime is that use of a semi-static model assumes the complete text is known in advance. In full-text applications this will often be the case, for example, when databases are being mastered onto CD-ROM. However there are other situations in which new documents are to be appended to the collection. One obvious example of this is a newspaper archive, to which articles are added almost continuously. In this case the compression scheme must permit the text to be dynamic, since it is clearly unreasonable to suppose that the entire collection should be recompressed after each insertion, or even once a day. But without such recompression, new words will not have codes in the model, and compression performance will degrade as lexicon statistics become inaccurate.

We have examined both of these difficulties. To reduce the memory required to store the model, use is made of a subsidiary character-level model to code words deliberately omitted from the word-level model. We give details of one simple selection algorithm that allows the decoder memory requirement to be held to a few hundred kilobytes with almost no impact on compression rates.

To solve the second difficulty, the need to be able to handle new words, we permit the model some small and controlled amount of leeway to extend itself during document insertion. The method is best described as taking “one and a bit” passes, since it is neither

*For simplicity, we refer only to the “words” of the compression lexicon. This should be taken to refer to both the words and non-words. All of the compression methods described in this paper are lossless, and any action applied to the lexicon of words is also applied to the lexicon of non-words.

one-pass nor two-pass. We describe a suitable strategy for maximising compression, and show that the “bit” can be as small as one part in a thousand with almost no loss of efficiency. That is, we demonstrate that a compression model developed on some text can be usefully employed to compress a text 1,000 times larger.

Note that the difficulties considered in this paper would apply to any compression scheme being used for a database, since it is the database context—the need for random access and the volume the data involved—that is problematic, not the choice of compression scheme. Other compression regimes would also benefit from application of methods similar to those we describe for the word-based model.

The remainder of the paper is organised as follows. Section 2 describes methods for reducing the decode-time memory requirements of the word-based model. Dynamic collections are considered in Section 3. In Section 4 the same methods are employed to allow compression of one text based upon source statistics from another. It is demonstrated that this can give very good compression, but fails in some cases. Section 5 examines a number of related issues, and compares the performance of the word-based model with a variety of other compression methods. Section 6 concludes the paper with a brief description of the retrieval system in which our experiments were carried out.

All compression figures listed in this paper include both the words and the non-words, account for all lexicon and other auxiliary files, and are for lossless compression of the source text. They are expressed as a percentage remaining of the original source text. For example, if a 500 Mb text is reduced to 100 Mb of compressed text and a 10 Mb lexicon file we will say that the compression efficiency is 22%. All experiments were run on an otherwise idle Sun SPARC 512 Model 10.

2 Reducing memory requirements

In this section we assume a static text collection, and consider construction of a model occupying a fixed amount of memory space during decoding. This is the situation that would apply, for example, if a text collection is being prepared on a well-configured machine for read-only access on a computer of limited resources.

If a word-based compression model is being used, a simple way to reduce the decode-time memory requirement is to omit some of the words from the lexicon. When such words are encountered during the coding pass, they must be represented in a different way. Our proposal is that a subsidiary character-based model—with only modest demands on main memory—should be used to explicitly spell these out.

Since a full first pass is being made over the text, statistics for the character model can be accumulated based upon knowledge of the words omitted from the lexicon and their frequencies, and the question to be considered is methods for choosing which words should be dropped from the lexicon. We explored three different methods for performing this pruning operation, and these are described below. In all cases an *escape* code must be provided,

appearance of which in the compressed text signals the decoder to receive the next word character by character rather than as a single token. To actually spell the word, a Huffman-coded length is issued, and then a sequence of Huffman-coded characters. That is, two small additional models are maintained, one storing the lengths of rejected words and one storing the distribution of characters occurring in rejected words. In a semi-static situation both of these models, and the escape code itself, can be based upon the actual probabilities of occurrence.

2.1 Method A

The first method considered for choosing which words to accept and which to reject is to prohibit addition of words to the lexicon after the memory limit is reached. That is, novel symbols are added to the lexicon during the first pass only if there is space to accommodate them, and, once the lexicon limit is reached, no more words are added. The remainder of the first pass continues to accumulate frequencies for words that did make it into the lexicon, but novel words are treated only as sequences of characters, and no attempt is made to determine whether they might warrant the allocation of space in the lexicon and the assignment of word codes.

Although simplistic, there is one important reason why this approach might be useful—it means that the amount of memory required in the encoder is also bounded. This is important if encoding as well as decoding is to be performed on a machine of limited capacity. Furthermore, despite its simplicity, it can also be expected to give reasonable compression, since intuitively one expects that frequent words will benefit most by being included in the lexicon, and the first appearance of a frequent word should be early in the text.

2.2 Method B

If the encoder is permitted enough memory to retain all words, then a more disciplined approach is to record all words and their frequencies, and at the end of the statistics-gathering pass select into the lexicon those words with the highest probability of appearance. This ensures that only rare words are spelt out in the relatively inefficient character model, and so overall compression should be better than that achieved by Method A. This method supposes that encoding is performed on a better endowed machine than is decoding, a situation that will often be the case for text retrieval with static collections. Even if the same machine is to be used for both processes, it may be that the one-off database creation task can be allocated more resources than are appropriate during querying.

2.3 Method C

Both Method A and Method B are approximations of the “optimal” selection of words into the bounded amount of memory space. Ideally, the selection of words should be such

that no exchange between the accepted list and the rejected list decreases the length of the compressed text. To identify an almost optimal selection, the following technique is used.

First, the accepted list is seeded with words using Method B. Then the Huffman code on words and the Huffman code on characters are established. Each accepted word occupies $l + 4$ bytes in the decode-time lexicon, where l is its length in bytes and a string pointer requires 4 bytes; and it contributes some known number of bits to the compressed output stream, calculable from its frequency and the length of the corresponding codeword. On the other hand, if it were to be moved to the rejected list, then, according to the character codes, it would contribute some greater number of bits to the output file. Thus the bytes of lexicon currently occupied by this word can be *priced* at the difference between these two quantities, divided by $l + 4$ to give the cost per byte of lexicon.

Similarly, each currently rejected word would save some number of output bits were it to be transferred into the lexicon, and so it can be regarded as *bidding* for entry at a certain price in terms of bits in the output per byte of lexicon occupied. In this case the length of the word Huffman code can only be estimated, since it does not currently have a codeword assigned. However, a reliable approximation of the code length is to use $-\log_2 p_i$, where p_i is the probability of the word in question.

At each iteration of the selection process, all of the bids and prices are evaluated, and any rejected word that, per byte, bids higher than any currently accepted price is swapped into the lexicon, until the least price in the lexicon is greater than the highest bid. Then all of the Huffman codes are re-evaluated, and the prices and bids recalculated. The process is continued until, immediately after the code recalculation, there are still no bids greater than any of the prices. This establishes a lexicon where no words can be exchanged from the rejected to the accepted state without increasing the total output bitlength.

2.4 Results

All three methods have been implemented and tested against *wsj*. Figure 1 shows the effect each of these three strategies has upon compression rate, plotted as a function of decoding model size. Throughout this paper compression rates are given as a percentage remaining of the original input text size, and include all auxiliary lexicon files necessary for decoding. The latter are, by and large, stored using a simple zero-order character-based model, and account for less than half a percentage point in all of the compression figures listed. The memory requirements shown on the horizontal axis are exclusive of the space required by the subsidiary character-level model, which adds about 1 Kb to the memory requirements. There are a number of other small in-memory tables (totalling less than 1 Kb) that have not been included, so that the horizontal axis in Figure 1 is purely the memory required by the decode-time word model. This is calculated based upon the estimate of one byte for each character of each accepted token, plus one four byte string pointer per token; this is generous because front coding (also known as prefix omission) can be used to reduce the space required by the bytes of each string, and the strings can be indexed in blocks, with

one pointer per block. These techniques are discussed in Section 5, and the saving accruing through their use is in addition to the memory reductions shown in Figure 1.

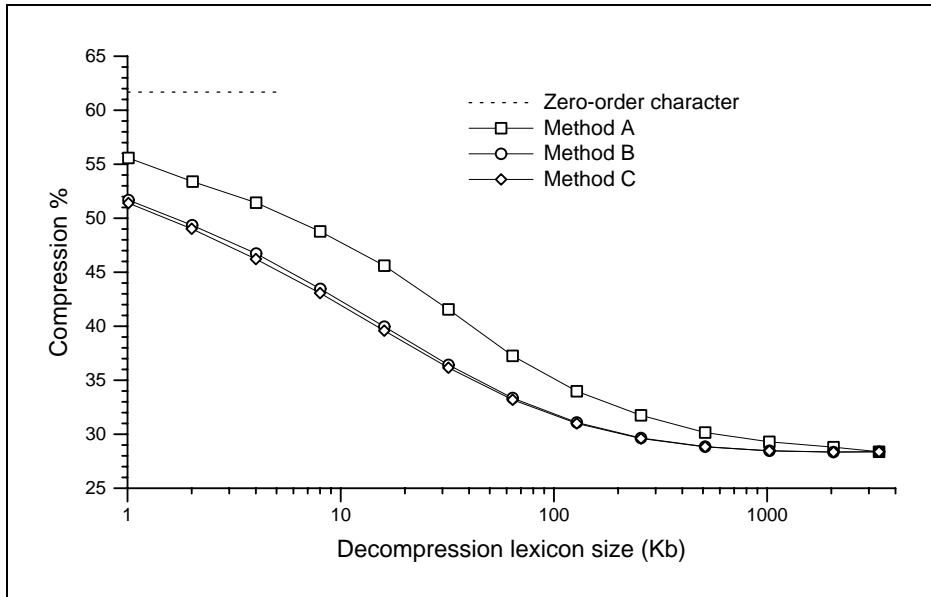


Figure 1: Effect on compression for *wsj* of reduced model size

As expected, Method A is outperformed by Methods B and C. More surprising is the excellent performance of Method B, and it is only marginally inferior to Method C. Choosing the most frequently appearing terms is an excellent heuristic. With either Method B or Method C the decode-time memory requirement can be reduced to well under 1 Mb with only negligible degradation of compression. Figure 1 also shows the compression achieved by a zero-order character model, which, by method of calculation described above, requires 0 Kb.

Table 1 shows in detail some of the points on the curve for Method C. When no memory is allocated for word storage the model is effectively a zero-order character model. The first 10 Kb of word storage improves the compression rate by nearly 20 percentage points, whereas the last 2,000 Kb gain only a final 0.1 percentage points.

Memory Allowed	Words		Non-words		Hit rate %	Compression %
	#	Kb	#	Kb		
0.0	0	0.0	0	0.0	0.0	61.7
10.0	983	9.3	80	0.7	78.0	42.0
100.0	9,207	97.7	255	2.3	95.0	31.7
1,024.0	91,028	1,002.7	1,891	21.3	99.7	28.5
3,348.6	289,102	3,237.7	8,913	110.9	100.0	28.4

Table 1: Compression with restricted model size, Method C

Using Methods A and B, a compression model can be determined very quickly, requiring just a few seconds at the end of the first pass. Method C is more computationally demanding. It typically took 3–5 iterations before the accepted and rejected words stabilised after seeding with the Method B selections, but occasionally took much longer. This is because there is some risk of instability, with the accepted list oscillating between two slightly different states. As a safeguard, the iterative process was always halted after at most 100 iterations. The computation is performed once only at the completion of the first pass, and when the size of the text is considered, 100 recalculations (around 5 CPU-minutes) is only a small overhead on the encoding time (about 1 CPU-hour to perform two passes over *wsj*). If even this overhead is deemed excessive, then Method B should be used. The corresponding Method B compression rates at 10, 100, and 1024 Kb were 42.3%, 31.8%, and 28.5% respectively, so the difference is quite insignificant.

One of the advantages of the word-based model is its fast decoding, resulting from the use of multi-byte tokens. Use of a restricted lexicon means that some fraction of the text is coded in a character model, and decompression speed suffers. In practice the loss is small, because most of the frequently appearing symbols are still allocated word codes. For example, with a 100 Kb model just 5% of the tokens are coded in the auxiliary character model, and even with a 10 Kb model nearly 80% of the tokens were lexicon words—the “hit rate” in Table 1. Exact values for decoding speed are presented in Section 5.

3 Dynamic databases

The second problem we consider is that of providing extensibility. Some text archives are intrinsically static, but many are dynamic, with new documents being added and a collection growing by perhaps many orders of magnitude during its lifetime. In this case it is not at all clear that a semi-static model should be relied upon for text compression, since, if applied strictly, the entire text should be completely recompressed after every document insertion. In this section we consider three methods for extending a collection without recourse to recompression, assuming that some amount of seed text is available. This latter is a reasonable requirement. If nothing else, the document stream to be stored can be sampled prior to creation of the database, or an initial crude system could be used during a bootstrap interval and then the system reinitialised based upon some accumulated text.

3.1 Method A

Given the discussion in Section 2, one obvious way in which the model can be made open-ended is to supply an escape code and a subsidiary character model, so that novel words in new text can still be encoded. The only difference is that the probabilities of the escape symbol and in the character level model must be estimates rather than exact values, since the text they are called upon to represent is, at time of model creation, still unknown. The character model can be assumed to be similar to the character probabilities already observed

in the text available, provided only that every symbol is allocated a code whether or not it has occurred. The escape probability is harder to estimate, but based upon previous experiments [15], method XC of Witten and Bell [22] provides a good approximation. This technique assigns the escape symbol a “frequency” of the number of symbols that have occurred once. That is, the proportion of tokens of frequency zero is approximated by the proportion of symbols of frequency one.

Figure 2 shows the “instantaneous” compression achieved on *wsj* for three models. The data points are the compression rate achieved for each 4 Mb chunk of input text, and so the overall compression rate for the file is the area under the curve plus the cost of storing the lexicon. No limitations on decoding memory were assumed.

The darkest line shows the compression assuming that a complete first pass over *wsj* is made, and that the model used during the second compression pass truly reflects the text being compressed. The compression rate is uniform, although the second part of the text does appear to be a little less compressible than the first. The other two lines show the instantaneous compression rates achieved when models built assuming knowledge of the first 25% of *wsj* (dark grey) and 6.25% of *wsj* (light grey) are used to compress the whole text. That is, they represent the compression that would be achieved if an initial collection of about 125 Mb was expanded to 500 Mb by the insertion of new text; and the compression that would result if an initial text of 35 Mb was available at time of model creation, and then the database grew to 500 Mb.

Note how the two partial models give *better* compression on the initial text they have seen, but are worse on the text they have no foreknowledge of. This is because word usage is not even throughout the text, and there are some words that appear commonly in the second half of *wsj* that do not appear at all in the first. In particular, the changes at about 230 Mb and then again at 370 Mb catch the two partial models unawares, and compression suffers. (The reasons for the change are considered further in Section 4 below.) Nevertheless, the degradation is small. The compression rates for the two partial models, including the lexicon, are 29.9% and 30.5%, only fractionally worse than the compression rate reported in the last line of Table 1.

3.2 Method B

The problem with Method A is that non-lexicon words are spelt out every time they appear in the text, and so compression worsens as more and more novel words—some of which are common, but only in a limited part of the collection—enter the vocabulary. For example, news reports prior to 1985 rarely made use of the word “Chernobyl”, but it has certainly been frequent since then. Another topical example is the word “Clinton”.

If one is restricted to a semi-static model and only a certain amount of known seed text, it might seem that little more can be done. However, in contrast to other applications of compression, in the context of document databases there are *two* channels of communication between encoder and decoder. The first is the compressed text, stored in the database, ready

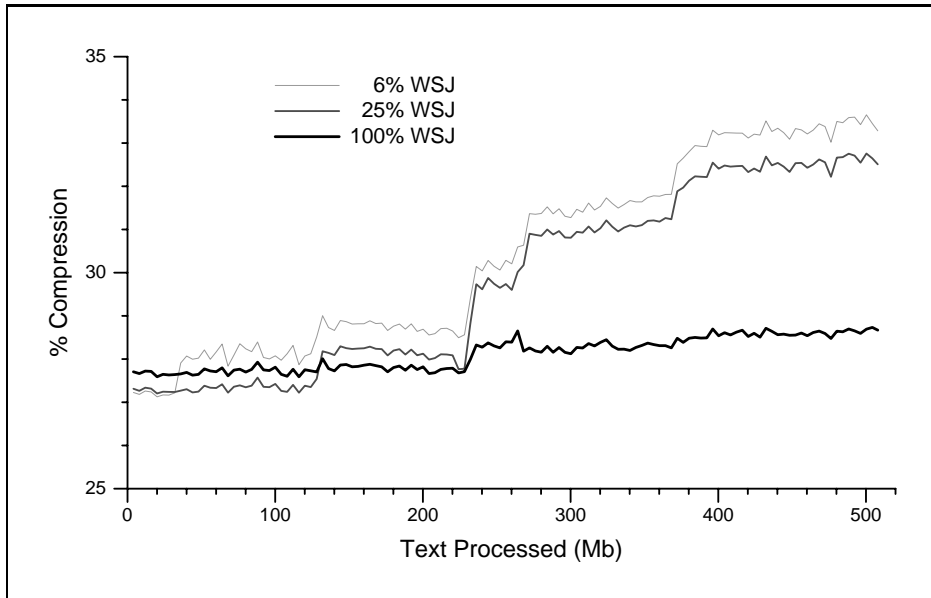


Figure 2: Instantaneous compression for *wsj*

to be retrieved and decoded at any time. The second channel is the lexicon of words used to control the compression. It is much smaller than the text, and, provided that the codes for existing words are not changed, new words can be appended to it without rendering undecodable previously compressed documents. If the encoder installs words into the lexicon as they are discovered during document insertion operations, they will certainly be there before the decoder can ever be called upon to emit them. All that is required is a set of codewords to indicate arbitrary positions in an auxiliary lexicon of “escaped” words. Then, rather than escaping to a subsidiary character model when it encounters a non-lexicon word, the compressor escapes to an auxiliary list of words, and, if the word is not in that list either, the compressor is free to add it and emit the corresponding code. This arrangement is shown in Figure 3.

In Figure 3a, a collection of documents has been processed to generate an initial compression model. An escape code is included in that model, so that the decoder can be told that a word is not part of the regular model. Suppose then that some document D is to be added to the collection. As far as possible, D is coded using the existing codes of the compression model. But D will also contain some new words that do not appear in the compression model. These are represented as an escape, followed by an index into the list of words stored in the auxiliary lexicon. If such a word already appears in the auxiliary lexicon—because it was in some previous document appended to the database—then it is represented by its index. And even if the word is new to the auxiliary lexicon, it can be coded as an index, since the encoder is free to add it into the next vacant location in the auxiliary lexicon. During the course of encoding document D , the auxiliary lexicon might

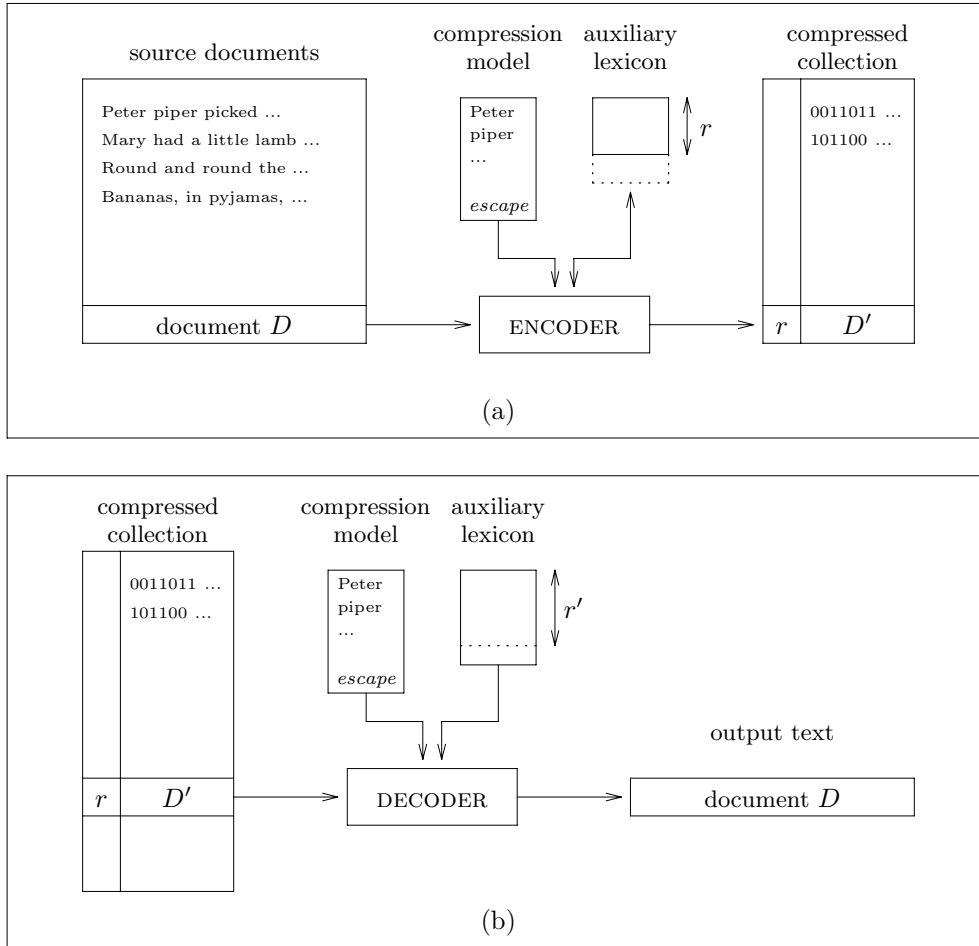


Figure 3: Inserting a document: (a) adding a document to the collection; and (b) decoding that document at some later time

thus grow from r entries to r' entries.

Figure 3b then shows that same document being decoded, but after further insertions have taken place. Because r , the relevant size of the auxiliary lexicon, is coded as part of the compressed record D' , the decoder can know exactly how to decode index entries into the auxiliary lexicon; and since each word not in the main compression model is prefixed by a known escape code, the decoder knows exactly when the auxiliary lexicon must be consulted. Words at locations greater than r' will simply not be examined, since none of the codes embedded in D' refer to such positions.

To make this scheme work, locations in the auxiliary lexicon must be coded. The next three sections discuss suitable coding methods.

Number	Binary ($r + 1 = 10$)	Elias's C_γ	Elias's C_δ	Modified C_δ ($r_{\text{lex}} = 10$)
1	000	0	0	0 000
2	001	100	1000	0001
3	010	101	1001	0010
4	011	11000	10100	0011
5	100	11001	10101	0100
6	101	11010	10110	0101
7	1100	11011	10111	01100
8	1101	1110000	11000000	01101
9	1110	1110001	11000001	01110
10	1111	1110010	11000010	01111
11	—	1110011	11000011	1000000
100	—	1111110100100	11011100100	11000011101
1000	—	111111111011101000	1110010111101000	11011101110001

Table 2: Codes for positive integers

Method B1 One possibility is to use Elias's C_δ code [6] for positive integers. This is a static code that uses $1 + 2\lceil \log_2 \log_2 2n \rceil + \lceil \log_2 n \rceil$ bits to encode n , on the assumption that small numbers will occur more frequently and should therefore be encoded in fewer bits; in particular, the number 1 is encoded in 1 bit. Some example codewords are shown in the third column of Table 2. A description of how these codes are constructed appears in Witten *et al.* [24]. Each word in the auxiliary lexicon is assigned a C_δ code based on its ordinal number. The lexicon grows each time a novel word is encountered, but this is a simple append operation, and in the compressed text this first appearance and all subsequent occurrences of this word are coded as its ordinal number in the auxiliary lexicon.

For example, suppose the escape code uses ten bits. Using Method B1, the first new word will require eleven bits, ten for the escape and one for its ordinal number. The thousandth new word will require 26 bits, 10 for the escape and 16 (see Table 2) for the ordinal number.

Method B2 The C_δ code is biased heavily in favour of small values—for example, in employing a one bit code, it assigns an implicit probability of 0.5 to the number 1. A flatter distribution of codes might better match the actual distribution of novel words, and so a binary code is also a possibility. Because binary is not an infinite code, the number of novel words encountered prior to the encoding of the current document must be prefixed to the document's representation in the text file. This is the arrangement that was illustrated in Figure 3. The open-ended nature of the C_δ code means that storage of r is in fact not required in method B1.

During encoding, if r of the novel words have been encountered to date, then binary codes in the range 1 to $r + 1$ are emitted, with $r + 1$ indicating that the next novel word is now in play and that r should be incremented. Each code requires either $\lceil \log_2(r + 1) \rceil$ or

$\lceil \log_2(r+1) \rceil$ bits, depending upon the exact value of $r+1$ and the value being coded. The second column of Table 2 shows the binary codewords that would be assigned with $r+1=10$; the codewords are either 3 or 4 bits long. Symbols one to nine have been seen, and so the tenth code brings symbol ten into play and indicates that r should be incremented.

This strategy is effective because the novel words are installed in the lexicon in appearance order by the encoder. The per-document prefix was coded in our experiments using C_γ , and was a very small overhead on each compressed record. Some example values of C_γ are shown in the third column of Table 2.

Method B3 As a third possibility, a hybrid code was implemented. The implicit probabilities used in the binary code of method B2 are now perhaps too even, and it makes sense to allow some variation in codeword length, even if not the dramatic differences allowed by the C_δ code. We still imagine that words that appear early in the appended text are more likely to reappear than words that appear late in the appended text.

The C_δ code in effect assigns symbols to *buckets*, with 1 symbol in the first bucket; 2 in the second; 4 in the third; and so on, as shown in Table 2. The code for a symbol in this scheme is generated as a C_γ -coded bucket number; and a binary “position-within-bucket” value; C_γ is another Elias code, requiring $1 + 2\lceil \log_2 n \rceil$ bits to encode n , so that C_γ has a stronger bias than does C_δ towards small numbers (see Table 2). To flatten the probability distribution, we placed r_{lex} symbols in the first bucket rather than 1, where r_{lex} is the number of words in the compression lexicon that are assigned proper Huffman codes. The second bucket is given $2r_{\text{lex}}$ symbols, the third $4r_{\text{lex}}$, and so on.

For example, if at the end of the seed text there are 10,000 symbols, then the first 10,000 novel words that appear thereafter are in bucket 1 with a 1-bit bucket code and a 13- or 14-bit binary component (as is shown in Table 2 for $r+1=10$, when binary numbers between 1 and $r+1=10,000$ are to be assigned codewords, numbers 1–6,384 are allocated 13-bit codes, and numbers 6,385–10,000 given 14-bit codes); the next 20,000 novel words are in the second bucket, which has a 3-bit bucket code followed by either 14- or 15-bit suffix; and the next 40,000 also have a 3-bit bucket code, but with a 15- or 16-bit binary component. Some example codewords using a more modest value of $r_{\text{lex}}=10$ are illustrated in the last column of Table 2; even using this small value the codewords for large values are shorter than those employed by C_δ .

This hybrid code is somewhat skewed in favour of low values but is not as biased as C_δ . Like C_δ , it is an infinite code, and so r need not be stored in the compressed records.

3.3 Method C

The model can be even more flexible than is allowed in Method B. The requirement for synchronisation is only that at the *start* of each document the model be in some consistent and known state, and therefore it can be adaptive *within* each document.

One way this flexibility could be exploited is to make the Huffman code for the lexicon words adaptive. But the resultant complex juggling of codewords substantially slows decoding [15], and one of the key virtues of the word-based scheme is lost. On the other hand the words in the auxiliary lexicon of Method B are coded using very simple codes, and there is also some advantage to be gained by reorganising those codewords on the fly. This is the avenue we chose to pursue. Method B3 was selected as a basis for further experiments, for two reasons: first, because it gave the best compression of the approaches described so far; and second because, in contrast to the binary code of Method B2, it offers codewords of varying length.

Bentley *et al.* [2] describe a “Move-to-Front” (MTF) heuristic for assigning codes to symbols. We considered the use of this strategy for reorganising the list of words in the auxiliary lexicon, but discarded it because of the relatively high cost of tracking list positions. Instead a simpler rule was used, which we call “Swap-to-Near-Front” (SNF). At the commencement of decoding of each document the list of auxiliary terms is assigned codewords in order of their first appearance in the collection. This satisfies the synchronisation requirement. A pointer p , initialised to zero, is used to partition the auxiliary lexicon into words seen and words not seen in the current document; and at any given time during the processing of each document, words with codes of p or less have occurred at least once.

Then the document is encoded. When each non-lexicon word is encountered it is encoded as a location within the auxiliary lexicon, using the hybrid B3 code. Immediately after being encoded, its current location x is checked against the pointer p . If $x > p$ then it has not yet been swapped, and so p is incremented, and words x and p exchanged. On the other hand, if $x \leq p$ no action is taken, since word x has already been moved forward at least once, and to move it again would only displace some other word that is also known to have already occurred within this document. Pseudo-code showing the action of the encoder to encode and append a set of documents is shown in Figure 4, assuming that each document is a sequence of words. (In practice the processing of the non-words must be interleaved.) The decoder has a similar structure, except that the auxiliary lexicon always contains every word necessary, and so it does not need to insert words or write a revised file.

At the end of the document, all of the swaps are undone in the reverse order to that they were applied in so that the model returns to the “standard” state ready to commence the processing of the next document. This is easily accomplished by maintaining a list S of values x for which swaps have occurred, and applying them in reverse order, decrementing p after each such swap. Swapping two strings by pointer exchange is a substantially faster operation than reorganising a dynamic search structure such as the splay tree required by the MTF policy, and the SNF approach has no effect on encoding and decoding rates.

This then is Method C: a Huffman coded lexicon based upon whatever seed text is available; an escape code assigned a probability using method XC; an open-ended auxiliary lexicon containing all of the non-lexicon words in order of first appearance; a modified C_δ code with which locations in the auxiliary lexicon are represented, parameterised in terms

```

1. Read the parameter  $r_{\text{lex}}$ .
   Read the main lexicon into array  $L$ .

2. Read the current auxiliary lexicon into array  $A$ .

3. For each document  $d$  to be encoded do
   (a) Set  $p \leftarrow 0$ .
   (b) For each word  $w$  in  $d$  do
       i. Search  $L$  for  $w$ .
       ii. If  $w \in L$  then emit  $HuffmanCode(w)$ .
       iii. Otherwise,
           A. Emit  $HuffmanCode(escape)$ .
           B. Search  $A$  for  $w$ .
           C. If  $w \notin A$  then append  $w$  to  $A$ .
           D. Set  $x$  to the location in  $A$  of  $w$ .
           E. Emit  $ModifiedDeltaCode(x)$  using parameter  $r_{\text{lex}}$ .
           F. If  $x > p$  then
               Set  $p \leftarrow p + 1$ ,
               swap  $A[p]$  and  $A[x]$ ,
               set  $S[p] \leftarrow x$ .
   (c) /* Return the auxiliary lexicon to its original ordering */
       While  $p > 0$  do
           swap  $A[p]$  and  $A[S[p]]$ ,
           set  $p \leftarrow p - 1$ .

4. Write the modified auxiliary lexicon  $A$ .

```

Figure 4: Encoding documents using Method C

of r_{lex} , the number of words assigned Huffman codes; and the SNF strategy to allow the codewords within the auxiliary lexicon to be self-modifying and, hopefully, locally adaptive. Although complex in description, all of the essential features of the word-based model are retained, and this mechanism provides both synchronisation and fast decoding. Moreover, as demonstrated below, it handles quite extraordinary text expansions with exemplary compression ratios.

Method C does have one disadvantage. Because it modifies the auxiliary lexicon whilst decoding documents, it cannot be used in environments where multiple users are concurrently sharing a single copy of the decoding model, as will be the case in large commercial text retrieval systems. In such applications either extra memory must be allocated for a set of pointers into the auxiliary lexicon, or a non-adaptive method such as B3 should be used.

3.4 Results

Figure 5 shows the application of these methods to the *wsj* collection. The horizontal axis shows the fraction of the original text used to build the model. For example, at an expansion

factor of 100, the first 1%—a little over 5 Mb—of the text of *wsj* is assumed to be available to the encoder to be used to establish a compression model, and then this plus the other 99% is compressed according to the model constructed at that time. The minimum amount of seed text used was a little under 32 Kb, to obtain an expansion ratio of 16,384.

The uppermost line shows the result of using Method A, which has a subsidiary character-level model and every appearance of each novel word spelt out in full. The first, third, and fifth points on this curve correspond to the three lines plotted in Figure 2. This method provides reasonable compression for expansion ratios of up to about 10, but compression steadily degrades thereafter. Note the sudden change in compression rate between expansion factors 1 and 2 common to all of the methods; this is caused by the slightly different nature of the two halves of the *wsj* collection, demonstrated in Figure 2. Despite its poor compression performance, Method A does have one clear advantage over all of the other approaches: the decode-time memory requirement is fixed, regardless of expansion in the collection.

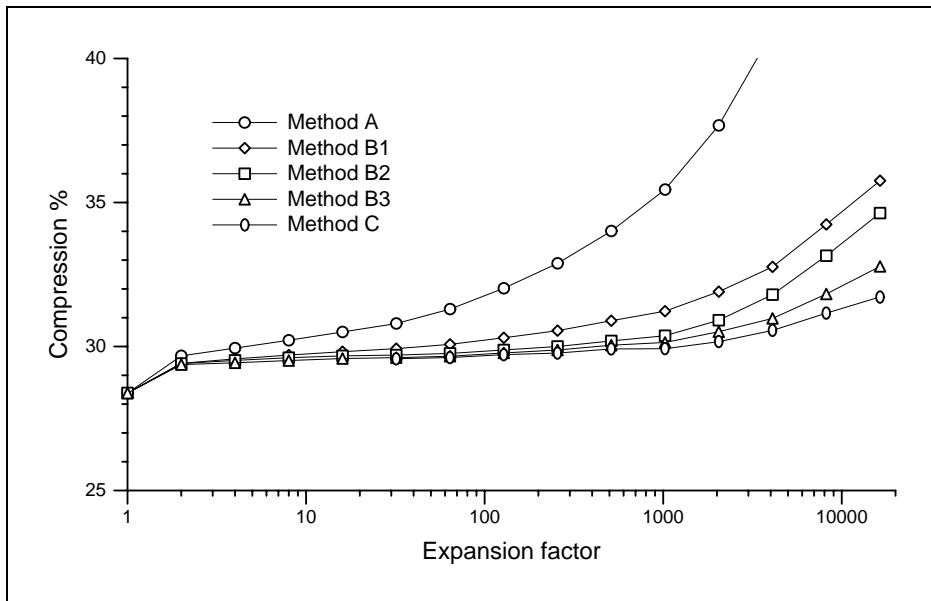


Figure 5: Simulated expansion of *wsj*

The next three lines show the effect of allowing novel words to be inserted into the lexicon and assigned codes using Methods B1, B2, and B3. As expected, all give markedly better compression than Method A. Method C then provides further improvement. Quite remarkably, this final method allows the compression degradation to be restricted to less than 2 percentage points, even in the face of a 4,000-fold expansion in the size of the collection. For Methods B2, B3, and C, a collection that starts at 1 Mb will reach 1 Gb before the compression gets even slightly worse; recompression at that point to establish a new model sets the scene for expansion beyond 1 *terabyte*.

4 Cross-compression

In the previous section it was assumed that the seed text was of a similar nature to the text being added to the collection. However, Figure 2 and the first two data points of Figure 5 have already illustrated the penalty of using a static model to compress text of a different nature to that expected. We were interested to assess the effectiveness of the techniques described in Section 3 for coping with dynamic collections when the seed text may *not* be representative of the text being inserted.

The 2 Gb *trec* collection consists of several parts: *wsj*, already described; *ap*, about 500 Mb of articles drawn from the Associated Press news service; *doe*, roughly 200 Mb of abstracts and other material drawn from the US Department of Energy; *fr*, a collection of Government regulations; and *ziff*, 400 Mb of articles covering a wide range of subjects in the areas of science and technology. Each of these is large enough and homogeneous enough to be considered a database in its own right; and we have also been combining them to make the single 2 Gb *trec* database.

File	Size (Mb)	Model Text						
		Self			<i>wsj</i>			<i>trec</i>
		100%	6.3%	0.4%	100%	6.3%	0.4%	100%
<i>wsj</i>	508.15	28.4	29.6	29.8	28.4	29.6	29.8	30.0
<i>ap</i>	493.14	28.3	28.5	28.9	31.7	31.7	31.5	30.1
<i>doe</i>	183.79	26.6	26.9	27.3	32.6	32.5	31.7	30.7
<i>fr</i>	451.87	28.0	28.3	28.8	38.4	37.6	36.1	31.0
<i>ziff</i>	418.27	28.4	28.7	29.2	36.5	37.0	35.9	30.7
<i>trec</i>	2055.22	28.1	28.6	29.0	33.4	33.6	33.0	29.7

Table 3: Cross-compression results, Method C

Table 3 shows the compression rates achieved by Method C for these various databases. Several different models were used to compress each collection. To set a baseline, each collection was first compressed relative to itself, with no restriction on lexicon size. These values appear in the third column. The value in the last row for *trec* is the weighted sum of the other five values, supposing that they were independently compressed. Note the very uniform compression rate obtained by the word-based model over substantially different styles of text.

To see how they would cope if expanded, each was compressed using as a seed text the first 1/16th (6.25%) and then the first 1/256'th (approximately 0.4%). Again, the value in the last row assumes a multi-database collection. The anomaly within *wsj* can be clearly seen—the other four collections suffer only slight compression degradation, about 0.3 percentage points for 16-fold expansion, and less than one percentage point for 256-fold expansion. These results are further verification of the usefulness of Method C.

The sixth, seventh, and eighth columns show the compression achieved when various

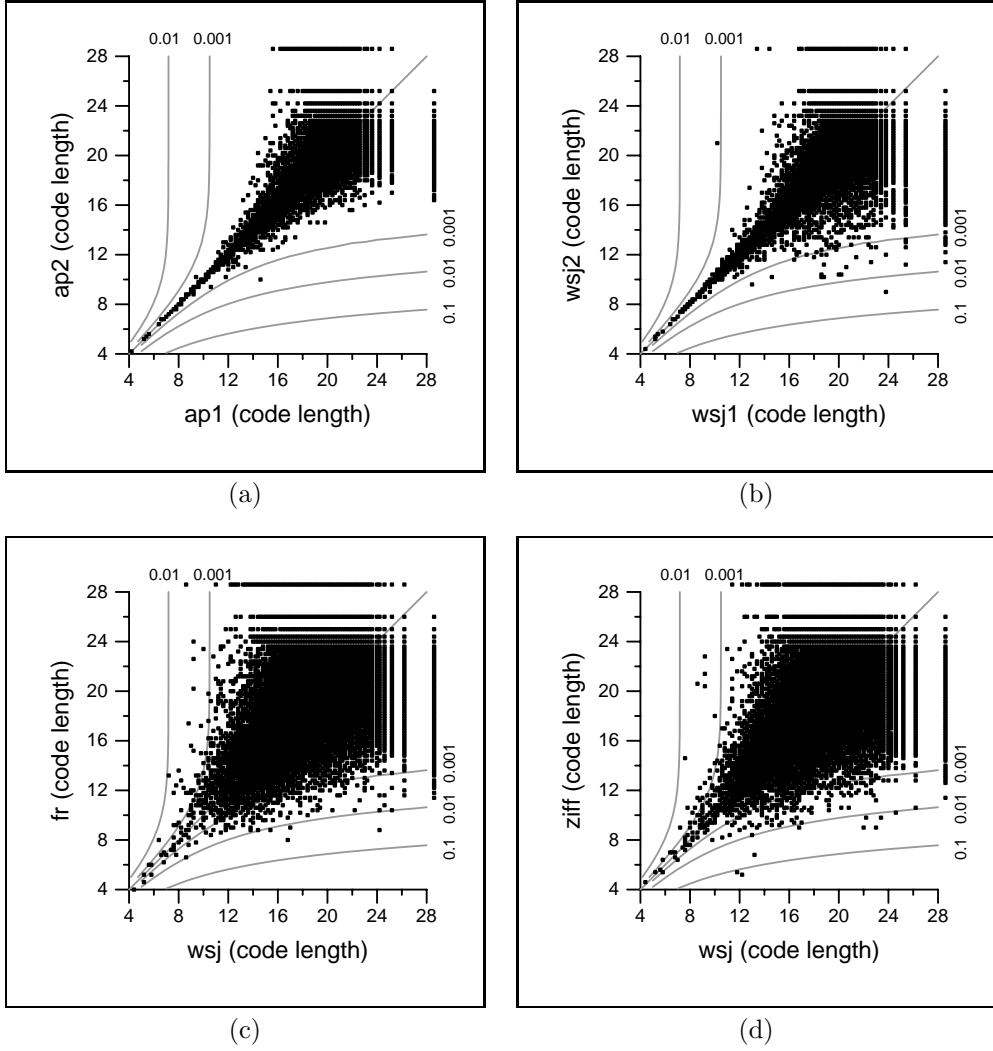


Figure 6: Word frequencies in different texts: (a) *ap1* vs. *ap2*; (b) *wsj1* vs. *wsj2*; (c) *wsj* vs. *fr*; and (d) *wsj* vs. *ziff*

fractions of *wsj* are used as the seed text for each of the other collections. It is clear that the text of *wsj* is quite different from *fr* and *ziff*, and compression worsens by as much as nine percentage points, almost irrespective of the amount of *wsj* used. Indeed, when the collection is dissimilar to the seed text, better compression is obtained when *less* seed text is used.

The relationship between word frequencies in different texts is shown graphically in Figure 6. The *ap* and *wsj* collections are distributed in two parts, each roughly half of the total. Each black dot in Figure 6a corresponds to one word that appears somewhere in *ap*, and shows the length of the code (i.e., $-\log_2 p_i$) the word should have to be optimal for that part of the collection. The horizontal axis represents the codes allocated for *ap1*, the first half of the collection, and the vertical axis shows codelengths in *ap2*, the second half of the

collection. For plotting purposes, words that appear in one sub-collection and not the other were arbitrarily assigned codes of “length” 28.5; and correspond to the row and column of black dots outside the axes of the plot.

If the two sub-collections were ideally matched, all of the black dots would lie on the $x = y$ diagonal, indicating that a compression model built for one half of ap can be used to optimally compress the other half too. Variations from the diagonal line correspond to cross-compression inefficiency, since inaccurate probabilities are being used. The greater the variation, the larger the inefficiency. The gray dotted lines show regions of “badness”. Points below the lowest gray line (of which there are none in Figure 6a) correspond to more than 0.1 bit per symbol of inefficiency. The entropy of the word distribution is about 10 bits per symbol, and so this corresponds to a 1% loss of compression relative to the compressed size. The next two gray lines represent inefficiencies of 0.01 bits per symbol and 0.001 bits per symbol. Similar error lines are plotted above the $x = y$ diagonal, and show the inefficiency that arises when a symbol appears less often than is predicted.

These lines are calculated by supposing that a symbol estimated to appear with probability p and thus has an assigned code of $-\log_2 p$ bits actually appears with probability p' , and then determining the value of p' that leads to a given amount x of excess code by finding the roots of

$$x = (-p' \log_2 p - (1 - p') \log_2(1 - p)) - (-p' \log_2 p' - (1 - p') \log_2(1 - p')).$$

This relationship defines p' as a function of p and x , assuming a binary alphabet (that is, whether each symbol is the word in question with probability p , or some other word with probability $1 - p$), and is what is plotted as the gray lines, one pair of lines for each value of x in $\{0.1, 0.01, 0.001\}$. In Figure 6a there are almost no words of significantly different frequency, and so the compression loss when $ap2$ is compressed based upon $ap1$ as seed text is very small. The ap results in Table 3 show that this self-similarity remains for even very small amounts of seed text.

On the other hand, when the two halves of wsj are compared (Figure 6b) there are several words that have probabilities sufficiently far from their correct values so as to introduce compression inefficiency, and the two halves of the text are less similar than observed for the ap collection. Table 4a shows some of the words in the wsj collection that introduce large losses in compression when $wsj2$ is compressed using $wsj1$ statistics. The table shows the value of $-\log_2 p$ for the word in each of the two sub-collections, and is ranked by decreasing excess code. Notice how it is much more expensive for a frequent word to be assigned a long code based upon an erroneous low probability estimate than it is for a rare word to be assigned a short code. The first few words listed correspond to the outlying dots in Figure 6b. The single biggest offender is the word “PAGE”; in $wsj1$ it appears only a few times altogether, but in $wsj2$ it appears about every 500 words.

A number of other words are also listed, together with their rank order. Token “HL” (part of the SGML markup, indicating a headline) is used frequently in the first part but

Rank	Word	<i>wsj1</i> (codelength, bits)	<i>wsj2</i> (codelength, bits)	Compression loss (bits per symbol)
1	PAGE	23.74	8.93	0.027343
2	NME	20.07	10.30	0.006597
3	STATES	20.15	10.34	0.006466
4	AMERICA	18.71	10.27	0.005688
5	NORTH	18.57	10.27	0.005536
6	UNITED	18.64	10.33	0.005323
21	HL	10.11	21.03	0.001298
49	Saddam	18.98	13.04	0.000537
76	Reagan	11.30	14.04	0.000324
99	Iraq	13.96	11.86	0.000267
100	Iraqi	15.78	12.72	0.000266

(a)

Rank	Word	<i>wsj2</i> (codelength, bits)	<i>wsj1</i> (codelength, bits)	Compression loss (bits per symbol)
1	HL	21.23	9.91	0.010304
2	the	4.66	4.16	0.004426
3	PAGE	9.14	23.53	0.002561
4	of	5.44	4.98	0.002110
5	a	5.79	5.28	0.002089
6	to	5.53	5.06	0.002061
7	and	5.99	5.50	0.001629
8	in	6.04	5.61	0.001209
9	said	7.55	6.90	0.001076
10	AMERICA	10.47	18.50	0.000991

(b)

Table 4: Codelengths in *wsj1* and *wsj2*: (a) error when *wsj1* used to predict *wsj2*; and (b) error when *wsj2* used to predict *wsj1*

sparingly in the second, and is the first word in the rank listing that decreases in probability. The words “Saddam”, “Reagan”, “Iraq”, and “Iraqi” are the only words in the top 100 that contain any lower case letters.[†]

Much of the discrepancy can be attributed to a change in style—notice how “UNITED STATES [of, presumably] AMERICA” is much more common in the second half than the first, perhaps because in the first part it was, by convention, written as “United States of America”. Indeed, the predominance of uppercase words in the top 100 indicates that the compression loss between *wsj1* and *wsj2* is more due to an abrupt shift—a much heavier use of capitalisation in *wsj2*—than to an evolutionary drift of content.

Table 4b shows the compression loss to be expected if the reverse cross-compression was performed, with *wsj2* used to establish a model for the compression of *wsj1*. Except for the words at the very top of the two lists, the words appear in quite a different order. Most of the top one hundred words in the arrangement summarised in Table 4b are common all-lowercase terms—they are the words that appear more often than expected when *wsj1*

[†]The first half of *wsj* covers the period December 1986 to November 1989; the second half covers April 1990 to March 1992, which includes the Gulf war.

is compressed using the *wsj2* statistics, because a very large number of all-uppercase terms appear less frequently than expected. This change in style is why Figure 2 shows a jump in instantaneous compression rate after 240 Mb. One can imagine a different editor taking control, or a different style guide being introduced, or perhaps even a different computerised formatting system being installed.

Figure 6c and Figure 6d compare the entire *wsj* collection with *fr* and *ziff* respectively. As can be seen, there are many words of quite different probabilities, and the cross-compression results are correspondingly poor. The *wsj* collection is not a good seed text for cross-compression of either *fr* or *ziff*.

In the final column of Table 3 the entire *trec* collection is used as seed text. That is, the seed text is actually a superset of the text being compressed, rather than the subset approach that has been discussed so far, and so for the *trec* value in the final row only one copy of the model is required, and the compression rate is a little less than the weighted sum of the 5 values above it in the same column. However, for each of the databases compression worsens, and this is true both individually and when they are combined. Five separate models are better than one combined lexicon, and the cost of multiple storage for common words is easily recouped by the use of more accurate probabilities within each collection. This suggests that for the large collections better compression might be achieved if the texts are broken into (say) 100 Mb chunks, assuming there is sufficient memory for the resultant large number of models. Certainly, Figures 2 and 6b show that *wsj* would benefit from being considered as two components.

The conclusion to be drawn from these results is clear—to compress a dynamic collection, a small amount of representative seed text drawn from inside that collection is a much better starting point than any amount of unrepresentative seed text drawn from outside the collection. But, given a suitable seed text, good compression rates can be maintained through very large expansion ratios.

A final advantage of the two-lexicon scheme proposed here is that the auxiliary lexicon can, if necessary, be handled separately to the main lexicon. For example, it is inconceivable that the main lexicon could be stored on disk, since a disk access per decoded word would slow decoding by a factor of 1,000 or more. However, the auxiliary lexicon is accessed relatively rarely, and it may be reasonable to retain it on disk rather than in memory. Alternatively, if the SNF strategy is not being used, the auxiliary lexicon can be held compressed in main memory using a character level code—really just another form of slow device.

5 Other considerations

In this section a number of other issues are discussed. The first subsection details the result of experiments into front coding, a technique whereby a sorted list of items can be stored in less space by prefix omission. Additional space savings result, but decoding time is affected.

The second subsection examines the process of rebuilding the database. Finally, the

third subsection compares the methods employed here with several other standard data compression programs.

5.1 Front compression

Further memory savings in the model can be achieved by the use of front-coding (sometimes known as prefix-omission) and blocking. If the words are grouped in blocks of l and each word is prefixed by a byte to indicate its length, then $l + 4$ overhead bytes are sufficient for indexing purposes rather than the $4l$ counted above. Moreover, within each block common prefixes can be omitted, typically saving a further 3–5 characters per word [19]. The cost of this saving is slower decoding, since each word must be reconstituted before it is emitted. Both blocking and front coding can be applied to the main lexicon of Huffman coded words, provided that the Huffman codes are assigned in a systematic way for each different code length. (Because the words in the auxiliary lexicon are not sorted, and, in Method C, subject to reordering, these techniques are applicable only to the main lexicon.) Table 5 shows the memory savings achieved by various values of l for the three restricted-memory *wsj* lexicons described in Table 1. The second column in each of the three groups shows the resulting decoding speed. As can be seen, the use of blocking and front coding saves up to half of the lexicon memory space, but also halves the decoding speed. There is, of course, no effect on compression ratio caused by these techniques, and the output file is identical within each column of the table.

Block size l	Target Lexicon Size					
	10 Kb		100 Kb		1 Mb	
	Memory (Kb)	Speed (Mb/min)	Memory (Kb)	Speed (Mb/min)	Memory (Kb)	Speed (Mb/min)
1	10.0	51	100.0	63	1024.0	65
2	8.3	49	78.9	57	786.8	60
4	6.9	45	63.8	51	623.2	52
8	6.2	41	56.2	43	541.2	43
16	5.9	31	52.3	33	500.2	32

Table 5: Effect of blocking and front coding on *wsj* lexicons

5.2 Recompression

When it finally becomes necessary, complete recompression of the collection must be performed. This should be carried out when the opportunity presents itself; for example when new disks are installed and not yet full of data, or during holiday periods. To expedite this, accurate frequency counts should be maintained in both the main lexicon and the auxiliary lexicon, a detail omitted from Figure 4. Then all of the information required to recalculate

Method	Compression (% of input)	Encoding (Mb/min)	Decoding (Mb/min)	Description
<i>lzrw1</i>	63.5	70	190	LZ77 method, adaptive, tuned for speed [21]
<i>pack</i>	61.8	60	30	Zero-order character-based, semi-static, Huffman coding
<i>cacm</i>	61.2	6	5	Zero-order character-based, adaptive, arithmetic coding [25]
<i>compress</i>	43.2	25	55	LZ78 implementation, adaptive [20]
<i>gzip-f</i>	38.6	15	95	LZ77 method, adaptive, fast option
<i>gzip-b</i>	36.8	8	100	LZ77 method, adaptive, best option
<i>dmc</i>	28.7	1	1	Variable-order bit-based, adaptive, arithmetic coding [5]
<i>huffword</i>	28.4	10	65	Zero-order word-based, semi-static, Huffman coding [16]
<i>ppmc</i>	24.1	4	4	Fifth-order character-based, adaptive, arithmetic coding [4]

Table 6: Compression and throughput on *wsj*

the assignment of codewords is already to hand, and the recompression can be carried out as a single-pass “read-decompress-compress-write” process.

5.3 Performance Comparison

We have also experimented with a variety of other compression methods, both semi-static and adaptive. One-pass methods do not allow random-access decoding, and so are not appropriate for full-text applications; nevertheless, they provide useful reference points for speed and compression performance. The results of these experiments are listed in Table 6. Of the methods considered, only *pack* is directly suitable for document databases, as all of the others are general-purpose compressors and employ adaptive models. However, if implemented in a semi-static manner they would give similar compression rates, and so the table provides a reliable guide to the efficacy of the underlying compression models. Moreover, the speed of the methods that make use of arithmetic coding would be largely unchanged by the change from an adaptive to a static model, a factor of two improvement at most [15], and so the execution speeds listed are also indicative.

The second to last row shows the *huffword* method that we have assumed throughout

this paper: a zero-order word-based model using canonical Huffman coding [16]. Although it is relatively slow during compression, it provides the best compromise between compression efficiency and decode speed.

6 Summary

The focus throughout this paper has been upon text compression, and we have not discussed document indexing at all. It is useful, however, to stand back and describe the larger system in which these results were obtained. We have implemented a text management system that is capable of efficiently manipulating multi-gigabyte text collections. Our testbed has been the 2 Gb *trec* collection, and we have been using the text compression regime described here to store the documents. Compression reduces the requirement from 2,055 Mb to about 607 Mb, a substantial saving. To locate documents in the collection an index is necessary. We use a compressed inverted index which notes, for each word and number in the text, all of the documents it appears in. The index also includes the number of times each word appears in each document, so that ranked queries can be carried out. Use of index compression allows the 142,000,000 document pointers to be represented in about one byte each, and the index requires under 7% of the space of the text being indexed. Including all other auxiliary files, the complete retrieval system requires less than 38% of the source data. This is in sharp contrast to conventional document database systems, for which the total space required for text plus index can be as large as 200% of the original text. Database construction—both text compression and index creation—takes place at a combined rate of about 250 Mb per hour.

Despite the fact that both text and index are compressed, query execution is fast. The system supports both Boolean “exact match” and ranked queries based upon the cosine similarity measure. Typical Boolean queries of 3–5 terms returning a handful of documents operate with sub-second response, and even ranked queries of 30–50 terms and returning a megabyte or more of text operate in just tens of seconds. A detailed description of the indexing method employed may be found elsewhere [17]; and an overview of all components of the system is provided by Witten *et al.* [24].

This paper has concentrated on the text compression methods suitable for such a document database. Word-based models have been advocated by several researchers. In large information retrieval systems of the type we have described, retrieval is non-sequential, and new text is continually being appended. We have shown that the word-based model can be adapted to cope well both with dynamic environments, and with situations in which decode-time memory is limited. In the latter case as little as 100 Kb of main memory is sufficient to achieve excellent compression, provided a suitable choice of tokens is used as the compression lexicon. To solve the former problem a new paradigm of compression has been introduced, in which some components of the compression model are required to remain static to ensure that all parts of the text can be decoded, and some parts are extensible,

so that new text can also influence the assignment of codewords. An additional heuristic—Swap-to-Near-Front—allows collections to be seeded with as little as 1/1000 of their final text with minimal loss of compression efficiency. The resulting compression method is ideal for large dynamic document collections.

Acknowledgements

This work was supported by the Australian Research Council and the Collaborative Information Technology Research Institute.

References

- [1] T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531, October 1993.
- [2] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.
- [3] A. Bookstein, S.T. Klein, and D.A. Ziff. A systematic approach to compressing a full-text retrieval system. *Information Processing & Management*, 28(6):795–806, 1992.
- [4] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32:396–402, April 1984.
- [5] G.V. Cormack and R.N. Horspool. Data compression using dynamic Markov modeling. *The Computer Journal*, 30(6):541–550, December 1987.
- [6] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.
- [7] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, 1985.
- [8] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
- [9] D.K. Harman. Overview of the first text retrieval conference. In D.K. Harman, editor, *Proc. TREC Text Retrieval Conference*, pages 1–20, Gaithersburg, Maryland, November 1992. National Institute of Standards Special Publication 500-207.
- [10] D. Hirschberg and D. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, April 1990.
- [11] R.N. Horspool and G.V. Cormack. Constructing word-based text compression algorithms. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 62–81, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.

- [12] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, September 1952.
- [13] D. Manstetten. Tight upper bounds on the redundancy of huffman codes. *IEEE Transactions on Information Theory*, 38:144–151, January 1992.
- [14] A. Moffat. Word based text compression. *Software—Practice and Experience*, 19(2):185–198, February 1989.
- [15] A. Moffat, N.B. Sharman, I.H. Witten, and T.C. Bell. An empirical evaluation of coding methods for multi-symbol alphabets. *Information Processing & Management*. To appear.
- [16] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 72–81, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.
- [17] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. Technical Report 94/2, Collaborative Information Technology Research Institute, RMIT and The University of Melbourne, February 1994.
- [18] C.E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 623–656, 1948.
- [19] R. Simion and H.S. Wilf. The distribution of prefix overlap in consecutive dictionary entries. *SIAM Journal on Applied and Discrete Methods*, 7:470–475, July 1986.
- [20] T.A. Welch. A technique for high performance data compression. *IEEE Computer*, 17:8–20, June 1984.
- [21] R.N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In J.A. Storer and J.H. Reif, editors, *Proc. IEEE Data Compression Conference*, pages 362–371, Snowbird, Utah, April 1991. IEEE Computer Society Press, Los Alamitos, California.
- [22] I.H. Witten and T.C. Bell. The zero frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, July 1991.
- [23] 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.
- [24] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.
- [25] I.H. Witten, R. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1987.

- [26] J. Zobel, R. Wilkinson, R. Sacks-Davis, and A. Moffat. Effective retrieval of partial documents. *Information Processing & Management*. To appear.