Compact In-Memory Models for Compression for Large Text Databases

Justin Zobel Hugh E. Williams
Department of Computer Science, RMIT University
GPO Box 2476V, Melbourne 3001, Victoria, Australia
{jz,hugh}@cs.rmit.edu.au

Abstract

For compression of text databases, semi-static word-based models are a pragmatic choice. They provide good compression with a model of moderate size, and allow independent decompression of stored documents. Previous experiments have shown that, where there is not sufficient memory to store a full word-based model, encoding rare words as sequences of characters can still allow good compression, while a pure character-based model is poor. In addition, there are other kinds of semi-static model that can be used for text, such as word pairs. We propose a further kind of model that reduces main memory costs of a word-based model: approximate models, in which rare words are represented by similarly-spelt common words and a sequence of edits. We investigate the compression available with different memory efficient models, including characters, words, word pairs, and edits, and with combinations of these approaches. We show experimentally that carefully chosen combinations of models can significantly improve the compression available in limited memory and greatly reduce overall memory requirements.

1 Introduction

Compression has several benefits for text retrieval systems. The space required for storage of text and index is reduced, and less time is required for both index processing and text retrieval. These gains are possible because, in the time required to access a small block of data, a typical CPU can execute around 2–10 million instructions, more than twenty times that of a decade ago: CPU speeds are increasing much more rapidly than are disk access rates. With compression, average seek times and transferred volumes of data are reduced, and the time saved can be used for decompression, yielding a net reduction in time overall.

Both of the major components of text retrieval systems, the index and the stored text, can be effectively compressed. Index compression techniques are typically based on efficient representation of integers by variable-bit codes [6, 10, 12], allowing space reduction by a factor of three to six. These compression algorithms must compete for resources with other components of the retrieval system, such as the query evaluator. In practical systems it is essential that the compression algorithms can operate in limited memory: databases may be hundreds of gigabytes; there may be hundreds of users; and memory is also required for structures such as the accumulators needed for fast query evaluation [12] and for caches for frequently accessed inverted lists.

For compression of the stored text, the topic of this paper, the most attractive technique has been Huffman coding with a zero-order semi-static word-based model [12], in which words and non-words are modelled separately and coding alternates between the two models. Semi-static Huffman coding has two potential disadvantages, that new words have no code in the model and therefore cannot be represented, and that the model size may become prohibitive, in particular for large corpora on machines with large numbers of users. Moffat, Zobel, and Sharman described two techniques for addressing the "new word" problem: extensible codes that allow representation of new words as they arrive, without changing the codes of other words; and an escape code method, in which new words are coded as an escape followed by a series of Huffman-coded characters [7]. These results showed that within limits, both approaches have only a moderate impact on compression efficiency. The escape method also provides a solution to the problem of model size; by imposing a limit on memory allocated to the model, words can be chosen on the basis of frequency until the model is full, then all remaining words coded as escapes.

In this paper, we investigate whether the compression efficiency attainable within limited memory, that is, with the restriction of a fixed model size, can be improved. We quantify the effect of moving from a word model to a word-pair model, and show that, with judicious selection of pairs, some gain in compression efficiency can be achieved. We also explore two techniques for maintaining compression efficiency given

strict memory limits for models, the escape model and a generalisation, an escape model based on n-grams rather than individual characters. A new approach that we propose is an approximate model based on edits: a rare word omitted from the model can be represented by a similar, more common word, with the differences encoded as the edits needed to transform the common word to the rare.

Our experiments show that edit models and escape models used in conjunction are slightly more efficient than escape models alone; and that limited use of word pairs allows significant improvement in compression efficiency within a fixed model size. These gains are of benefit in limited-memory applications: given a fixed memory limit our techniques can reduce, for example, a 260 Mb compressed file by a further 10 Mb. Compared to previous techniques, our approaches achieve the same compression efficiency using models that are smaller by a factor of ten.

Test data

We have used several test collections in the experiments reported in this paper. Most are drawn from the data accumulated by the TREC experiments [3]. In this paper we report on the experiments with two of these collections, for which the results were representative of all the collections used. The first is PARTWSJ, the first 63.3 Mb (1,000,000 lines of text) of the "Wall Street Journal" component of TREC disk 1, which contains approximately 10,000,000 word occurrences and 110,000 distinct words; this shows performance on a smaller database. The second, showing performance on a larger database, is DISK2, the complete "Associated Press", "Federal Register", "Wall Street Journal", and "Ziff Publishing" collections from TREC disk 2, amounting to 863.8 Mb, which together contain about 138,000,000 word occurrences and 460,000 distinct words.

Throughout this paper, we report compression results as a percentage of original file size; thus a technique that compresses a 63.3 Mb file to 21.6 Mb has a compression efficiency of 34.1%. For a file compressed with Huffman coding, this figure includes both the compressed text and a compressed representation of the model. We also report the approximate size that the model for words might occupy in memory during decoding, estimated as one byte for each character of each distinct token in the model plus four bytes per string for housekeeping such a terminator and pointer. (We do not include the non-word model in this in-memory estimate: we have not applied the same kinds of optimisations to it, although it would be straightforward to do so in a production system, and for a given collection the size of the non-word model is the same for all the techniques described in this paper other than the uninteresting case of a character model.) In memory, the model is rather larger than when stored on disk, the magnitude of the change depending on factors such as the amenability of the lexicon to front-coding. Thus the size of the in-memory model is only a broad indicator of the space required to store the model on disk.

2 Text database compression

Semi-static Huffman coding with a zero-order model based on words is an effective compression technique for text databases. A *zero-order* model, in which the probability of occurrence of each symbol is independent of the neighbouring symbols, is used because practicalities such as memory limitations prohibit higher-order word-based models. A *semi-static* model—in which one pass through the data is used to accumulate statistics with which to build a model and a second pass is used to compress the data with respect to the model—is desirable for databases because it allows individual records to be independently decoded, regardless of the order in which they are fetched. In contrast to a semi-static approach, in the adaptive models favoured for general-purpose compression, records must be either decoded in sequence or gathered into indivisible large blocks, defeating any potential savings in retrieval time.

A *word-based* model for semi-static Huffman coding provides a trade-off between model size and compression efficiency. With a model based on smaller tokens, such as characters, compressed file size typically exceeds 60% of the original data, whereas 28% is typical for a model based on words [12]. These figures include the model itself: a couple of kilobytes only for a character model, and around 1% of the original data size for a word model. For models based on word pairs, as we show in the next section, model size overwhelms the gains in compression efficiency.

Compression with semi-static Huffman coding not only saves space, but can reduce query evaluation costs. Some years ago, Zobel and Moffat observed experimentally that, for sequential retrieval of data, compression could lead to a net penalty in retrieval time, but for random access patterns typical of text database systems compression allowed savings [13]. However, because of the changes in hardware since those experiments, we would expect that compression would today always lead to savings in retrieval time, and have observed better throughput due to compression in the context of index processing [10] and genomic retrieval [9].

For standard applications of compression, performance is typically measured by the size of the compressed data and by coding and decoding speed. For Huffman-coded text databases, memory constraints place an upper bound on the size of the in-memory model during decoding; for a large database it may

simply be impractical to hold the ideal model in memory. (If the model is not memory-resident, decoding of each symbol potentially requires a disk access, almost certainly leading to unacceptable performance.) Thus measurement of a compression technique for databases must also consider memory usage. Also, encoding cost is relatively unimportant, since in most contexts data is accessed far more often than it is stored—investing extra CPU cycles in achieving better compression can yield a net payoff through gains in retrieval efficiency.

The compression efficiency of Huffman coding depends on how the data is parsed into symbols. Zero-order character models yield poor compression (and slow decoding), because the variable-bit codes are inefficient—the fractional-bit differences between the theoretical minimum code-length [8] and the actual code length are significant because most of the codes are short—and because the tendency of characters to occur in particular groupings is not represented. However, character models have very small memory requirements. The performance of character-based Huffman coding is shown in the first line of Table 1.

A word model yields much better compression, at the cost of requiring more memory. The performance of the word model is shown in the second line of Table 1 (and in Figure 1). Word models are seen as providing good compression efficiency for text databases [12, 13], and Huffman coding provides compression within 1% of the optimum because the probabilities are relatively small. Other compression schemes, such as predictive modelling, can yield somewhat better compression, but are inherently slow, rely on large models [1], and are at their best when adaptive modelling can be used.

However, as collection size increases, so does model size. Text databases do not appear to have lexicons with a more or less constant set of words. As new documents appear so do new words: chemical names, place names, and in particular typographic errors. For example, in addition to the lexicon of disk 1 of TREC, on average disk 2 contains a new word every 740 word occurrences; and in addition to the lexicon of disks 1–4 of TREC, on average disk 5 contains a new word every 763 word occurrences. That is, the rate of appearance of new words has barely slowed as more text is observed. In some retrieval environments the memory required for a word model will not be available.

One solution to this problem was described by Moffat, Zobel, and Sharman [7], in which the size of the model was reduced by omitting some words. (New approaches to the problem of limiting model size are described in Sections 4 and 5.) Words in the model are represented by a Huffman code. Other words are omitted and are each represented by an *escape* code, whose length is determined by the number of times it is used, followed by a character count and the sequence of characters comprising the word. There is some loss of compression efficiency, because a single word code is generally much shorter than the combined length of an escape, count, and sequence of characters, and rare words tend to be long. However, distinct rare words can comprise the bulk of the model, while the total number of occurrences of rare words is quite small. Because storing rare words in the model is inefficient, the escape method can be used to greatly reduce model size without any significant detrimental change in overall compression efficiency.

Moffat, Zobel, and Sharman tested different methods for choosing which words to omit [7]. They showed that a reasonable method—their "method B"—is to simply discard words on the basis of frequency of occurrence, in effect discarding all words whose frequency is less than some constant threshold E; varying E yields different lexicon sizes. However, this is not necessarily optimal: it may be desirable to keep a long rare word in the model (because of the cost of coding it as individual characters) and to discard a shorter word that is slightly more common. Their results showed that an insignificant improvement is obtainable by iteratively choosing words to discard—their "method C"—on the basis of their net impact on the model and on compressed file size. In this paper we have used method B, because the very slight gain given by method C is at substantial computational cost.

The performance of the escape model is illustrated in the third and fourth lines of Table 1 (and in Figure 1); in the first case only the rarest words are omitted from the model, while in the second most words are omitted. As can be seen, model size can be reduced by a factor of at least four with negligible impact on compression efficiency. However, if model size is reduced too far compressed file size begins to grow.

An extension of the escape method—which to our knowledge has not previously been evaluated—is to code n-grams, or sequences of n characters, rather than code characters individually. In comparison to the character-based escape model, counts will be smaller and codes should be more efficient, but for larger n model size for the n-grams may become significant. The use of n-grams is shown in the fifth and sixth lines of Table 1 (and in Figure 1). These results show that with $n = 4$, the size of the model of n-grams outweighs any potential savings—compared to the escape model with a threshold of 10, compression is much poorer while model size has increased. Even with $n = 2$, compression efficiency for a given volume of model memory is not as good as under the escape model.

The results reported in this section confirm that a word model provides good compression efficiency, and that the escape model allows this efficiency to be nearly maintained while model size is substantially reduced. In the remainder of this paper we show how these results can be further improved.

Table 1: *Compression efficiency (%), file size (megabytes), and in-memory model size (megabytes) for different standard models.*

	PARTWSJ			DISK2		
	Eff.	Size	Model	Eff.	Size	Model
Character model	61.2	38.76	<0.01	63.3	546.74	<0.01
Word model	28.4	17.97	1.20	29.9	258.48	5.26
Escape model, $E = 10$	28.6	18.12	0.31	30.0	258.90	1.28
Escape model, $E = 1000$	39.2	24.84	0.01	33.5	288.95	0.10
Escape, 2-grams, $E = 1000$	38.8	24.54	0.03	33.3	287.86	0.13
Escape, 4-grams, $E = 1000$	34.9	22.12	0.39	32.2	277.78	1.32

3 Word pairs

A zero-order word model yields better compression efficiency than a zero-order character model, because the probability of a given combination of characters (that is, a word) is not simply a function of the probabilities of the individual characters. It is therefore reasonable to ask whether a model based on longer symbols might be even more efficient.

A possibility is to combine each word with its following non-word, thus yielding a larger symbol set. We have not explored this option in detail, but do not believe that it would be valuable. Such a model is in effect an arbitrary combination of a semantic property, that is, words, with formatting that is not dependent on the semantics. There may be some small saving (in particular because a single space, by far the most common non-word, is represented inefficiently in an alternating model), but at the cost of a much larger model overall. On DISK2 we have found that compression efficiency falls slightly while model size roughly triples.

Another possibility is to use word pairs. Compression with word pairs would begin by tokenising the text into word pairs and non-words (or non-word pairs), then proceed as for a word model.

Words are easy to define—a sequence of alphabetic characters, as in this paper, or a sequence of alphanumerics and perhaps characters such as apostrophe and hyphen—but definition of word-pairs is less straightforward, because they overlap. Counting occurrences of all overlapping word pairs does not yield an accurate model. It is therefore necessary to allocate each word to be either the start or end of a pair. If the allocation is on the basis of order of occurrence, phrases will be broken in different ways in different contexts; if the allocation is on the basis of frequency of co-occurrence, individual words may be isolated between occurrences of pairs. Identifying the best pairings is a topic for further research; in this paper we have used what is probably the simplest approach, pairing words in order of occurrence, so that the phrase

> and the quick brown fox jumped over the lazy dog

is paired as

> and the / quick brown / fox jumped / over the / lazy dog

As a further simplification we only considered words separated by white space, on the basis that most other co-occurrences—such as between the word at the end of one sentence with the word at the start of the next sentence—would be more or less random.

Not surprisingly, a model based on word pairs is not efficient. Even using our fairly restrictive definition of pairs—alphabetic strings separated by white space—for PARTWSJ model size is almost as great as the total file size for a word model, as can be seen from the second line of Table 2. (The first line is repeated from Table 1, for reference.) The vast majority of these word pairs occur only rarely; even in DISK2, over 60% of the pairs occur once only and over 90% occur less than ten times each.

However, we contend that pairs are nonetheless of value. It is reasonable to suppose that, just as judicious selection of words allows good compression in reasonable memory with the escape model, so too judicious selection of pairs may aid compression. A straightforward way of making use of pairs is to include common pairs of words, including their linking white space, in the word model. Defining a common pair as one that occurs more than P times in the collection, varying P allows the increase in model size to be traded against compression efficiency.

Results are shown in the last two lines of Table 2 and in Figure 1. In the figure, the effect of the escape models is at the left-hand end of the graph: varying threshold E degrades compression efficiency as model size is decreased. The pair model is at the right-hand end: by increasing model size through inclusion of pairs, compression efficiency can be improved. There is an improvement of over 1% of original file size, for moderate values of P. (We have omitted points corresponding to small values of P, because they require a graph of

different scale; these points show both model size and compressed file size increasing rapidly.)

An alternative approach is to have a separate model of pairs, and switch between the pair and word models through escape characters. This approach has the advantage that a pair of words can be represented with the same code regardless of the intervening non-word. However, the need for an escape preceding each word or each pair is likely to outweigh any savings, given that the commonest white space, a single space character, contributes around 90% of non-word occurrences.

The method described above for choosing pairs is rather simplistic, and a better method may improve compression. A possibility is to choose pairs that occur more often than is indicated by their words' frequencies, a technique that is used to identify likely words in Chinese text segmentation. In preliminary experiments with such a pair selection approach we have not been able to use it to achieve additional improvements in compression but believe that it is worth exploring in further work. Another option would be to choose pairs estimated to have the most effect on overall compression, considering impact on model size and code lengths [4, 7]; we have not explored this option, but note that it is difficult to make such estimations reliably.

These results show that a model of word pairs is not efficient. However, the use of a limited number of word pairs allows a significant improvement in compression efficiency. Even when the number of pairs is tiny, compared to the number of words, the improvement is marked.

4 Edit models

In the escape model, each rare word is represented by an escape code, the word length n, and codes for the n characters. We observed that many of the rarest words are close in spelling to more common words. In particular, most new words are typographic errors, often differing in spelling from a previous word by only a single character.

We therefore suggest that an edit-based or approximate model may be an efficient way to represent these words. An edit model can be used for compression as follows. Each document is transformed by replacing each rare word by a similarly-spelt common word. For example, after replacing all rare words (less than 1000 occurrences in WSJ) that were of similar spelling to a common word, the text fragment

```
"I don't want to suggest that we can use
infinitely high-dose TPA,
or even a modest dose for an extended
period, without the risks of
bleeding," said Burton Lobel of
Washington University, St. Louis.
```

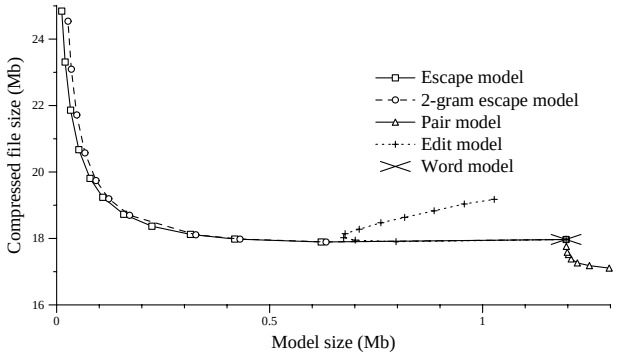

Figure 1: *Compressed file size versus in-memory model size for a range of model management schemes, for* PARTWSJ.

is transformed to[1]

```
"I don't want to suggest that we can use
definitely high-done TPA,
or even a modest done for an extend
period, without the risk of
bleeding," said Boston Nobel of
Washington University, St. Louis.
```

In this example, among other changes characters 42 and 43 have been substituted, with "in" replaced by "de", and characters 102 and 103, "ed", have been deleted. Rebuilding the original requires reversal of the replacement and insertion of "ed" after character 101.

Once the document has been approximated, an edit table is created for each document, storing each position at which an edit is required and the edit itself. An edit table can be organised in several ways. One way is a nested structure based on characters:

> For each character in the alphabet,
> Store a list of positions at which
> that character should occur,
> and for each position
> Store a bit indicating whether
> the character should be a
> replacement or an insertion.

This scheme is similar to one developed by Williams and Zobel for compression of genomic nucleotide data [9]. In that case, only four characters (A, C, G, and T) occur with frequency greater than 1%; these can be represented with 2-bit codes. The other eleven, wildcard characters are stored as ⟨character, position-list⟩ pairs, indicating the positions at which each wildcard

[1]The resemblance between this technique and lossy text compression [11] is purely superficial.

Table 2: *Compression efficiency (%), file size (megabytes), and in-memory model size (megabytes) for different pair models.*

	PARTWSJ			DISK2		
	Eff.	Size	Model	Eff.	Size	Model
Word model	28.4	17.97	1.20	29.9	258.48	5.26
All pairs	37.0	23.42	17.37	32.9	284.21	110.90
Pair model, $P = 10$	27.1	17.14	1.88	28.6	246.86	12.26
Pair model, $P = 1000$	27.8	17.59	1.20	28.8	248.66	5.33

can be substituted. (A randomly-chosen nucleotide is used as a placeholder. Pairs with an empty list are omitted.) The particular attraction of this scheme for nucleotide data is that the wildcard information can often be neglected during matching, which is based on edit distances—that is, the lossy representation can be used as is.

Another approach is a nested structure based on positions:

> For each position at which a change is required,
> Store the character to be used,
> and store a bit indicating whether
> the character should be a
> replacement or an insertion.

or a re-organisation based on edit-type:

> For each edit type, that is, for insertions and for replacements,
> Store a list of positions at which
> that edit type is required,
> and for each position,
> store the character to be used.

Continuing the example above, with this method the edit structure would have two lists,

> 42 i 43 n 60 s 86 s 148 u 149 s 154 L

for replacements and, for insertions,

> 101 e 101 d 126 s

This final approach is the one used in our experiments because hand estimation indicated that it was likely to be the most compact. In each of the two lists, the positions are sorted, allowing differences to be taken and representation with a variable-bit Golomb code [2]; this representation of positions is the same as is widely used for index compression [6, 10, 12]. However, in this context the positions tend to cluster, and alternative codes may yield better compression [5]. The characters are represented with a Huffman code.

Compared to the escape method, for some words the edit method can potentially yield considerable savings. Consider (in rough figures) a rare word of ten characters that differs from a more common word in one character only, in a collection with around one billion word occurrences. In a word model, the rare word requires a code of around 25 bits, corresponding to approximately 2^5 occurrences. In the escape method, the rare word requires an escape code, say 14 bits, a count, say 5 bits,[2] and ten characters, in say 50 bits, giving 69 bits altogether. In the edit model, the rare word requires the code for the more common word, say 18 bits, a position, say 10 bits, and a character, say 5 bits, giving 33 bits altogether—less than half that of the escape model and only a little more than the code length of the original word. The longer the word, the greater the improvement yielded by the edit model compared to the escape method.

In small collections, it is reasonable to suppose that most of the rare words will not have a homologue that requires only a few edits, and the edit model would thus yield little advantage. In larger collections, as the number of typographic errors accrues, it should provide more benefit. Whether the edit model becomes more efficient with increasing collection size is one of the hypotheses tested by our experiments.

Results for the edit model on PARTWSJ are shown in the third and fourth lines of Table 3 (in which the first two lines are repeated from Table 1) and Figure 1. In these results, words were defined to be of similar spelling if they had two-thirds or more of their characters in common. As for the escape model, a threshold A was varied: one word was replaced by another if the former had frequency less than A, the latter had frequency greater than A, and they were similarly spelt.

[2]There are many heuristics that can be used to make slight improvements in compression efficiency. One that applies here is to use a fixed length of 5 bits to represent word length. Words of greater than 32 characters must then be represented as a pair of words separated by a null non-word. If words of this length are rare, this heuristic improves compression overall efficiency. If they are more common, as may be the case in special-purpose databases such as of chemical names, this heuristic may not not effective, and our estimate of number of bits for a count must be increased.

We considered only insertions and replacements, and in preliminary tests deletion edits appeared to be rare. These results were disappointing; while up to half the lexicon could be represented by an edit, these words are such a tiny proportion of all word occurrences that the saving compared to the escape model is negligible. The explanation for the poor performance was somewhat subtle. First, the intuition that most of the rare words—of say 100 occurrences or less—would be close to a more common word was incorrect: for these collections, only up to half of the words were suitable for an approximate representation. Second, as A was increased, the number of words that could be used for approximation dropped, so that the edit model became steadily less efficient, as illustrated by the top half of the curve.

A modification that may allow the edit model to be more efficient is to cluster together rare words of similar spelling and represent them by a centroid, either an actual word or an artificial word. We have observed that many, perhaps most, rare words are close in spelling to other words, but these words also tend to be rare. By creating such centroids, we believe that the edit method's efficiency could be greatly improved. However, it is not clear how such centroids can be identified efficiently.

5 Combined models

The models described in the previous sections have differing effects. The word model, which is the basis of the other models, provides good compression efficiency and reasonable model size. The escape and edit models allow compression efficiency to be nearly maintained, while reducing memory requirements. The pair model improves compression efficiency at the cost of increasing in-memory model size.

However, the pair model, edit model, and escape model can readily be combined. As shown in Table 2, for large values of P the impact of including pairs on model size is small, but yields a good improvement in compression efficiency. For a database compressed with a word model that includes a small number of pairs, the lexicon is virtually identical to that of a model without pairs, and the edit and escape models can be applied without modification. By first applying the edit model, to remove the words that can be represented by approximations, then applying the escape model, it should be possible to achieve better compression efficiency for a given model size than is possible with any of the individual techniques.

As illustrated in Figure 2 for DISK2, combining the techniques does indeed improve performance. The top line is the escape model for a range of E values. The bottom line is the combination of the pair model with

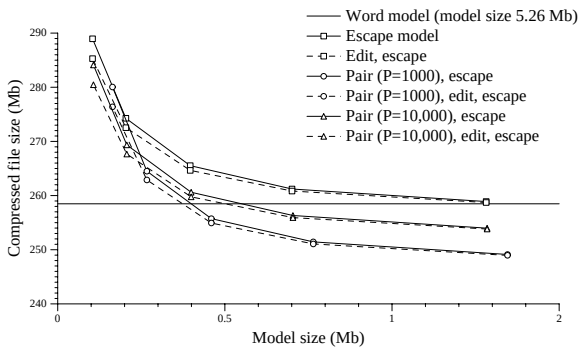

Figure 2: *Compressed file size versus in-memory model size for a range of model management schemes, for* DISK2.

edits and escapes, with fixed P and varying $E = A$. (Use of fixed $A = 10$ and varying E produced slightly worse results, somewhat surprisingly given the results reported in Section 4, where $A = 10$ showed the best results for the edit model.) Compared to the escape model alone, use of pairs gives a saving of around 10 Mb, or 1.2% of original file size, across a range of model sizes. For the smaller models, the edit model yields a further saving of around 0.1%–0.3%. However, these results also indicate that the value of the edit model does not increase with collection size; in experiments not reported here, we observed that it led to substantially greater savings on PARTWSJ.

Note that each of the parameters can be chosen automatically at compression time: pairs can be accrued up to a memory limit, words discarded down to a memory limit, and so on. While our presentation of the algorithms has been in terms of frequency thresholds, memory-oriented techniques should be used in practice.

Use of schemes such as escape or pair models not only reduces the memory required during decompression, but ensures that, for a given degree of compression, the memory requirement only grows slowly with increasing data size. For example, DISK2 is over thirteen times larger than PARTWSJ, and word model size is greater by a factor of four. However, taking the compression efficiency of a word model as the base case, Figure 3 shows that, for a combined model, model size only doubles from PARTWSJ to DISK2 for a compression efficiency identical to that of a word model, and that the growth is even less when some loss of compression efficiency is allowed.

Note that the impact on decompression speed of combined models, in particular the combination of pair and escape modelling, without edit modelling, is likely to be small. (To allow full experimentation with options and combinations, our modular prototype system is built of multiple, separate components and does not allow us to conduct representative timings; however,

Table 3: *Compression efficiency (%), file size (megabytes), and in-memory model size (megabytes) for edit and escape models.*

	PARTWSJ			DISK2		
	Eff.	Size	Model	Eff.	Size	Model
Escape model, $E = 10$	28.6	18.12	0.31	30.0	258.90	1.28
Escape model, $E = 1000$	39.2	24.84	0.01	33.5	288.95	0.10
Edit model, $A = 10$	28.5	18.02	0.67	29.7	256.86	2.76
Edit model, $A = 1000$	30.3	19.17	1.08	61.2	528.24	4.08

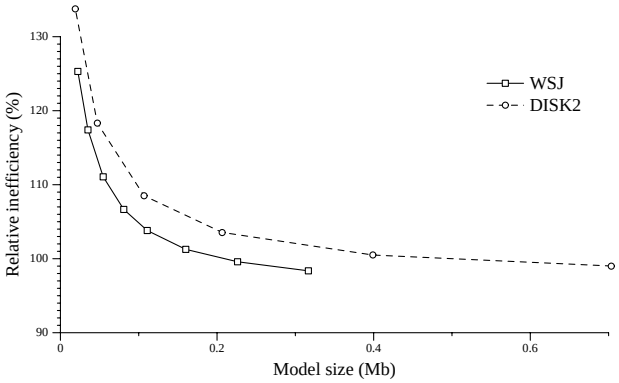

Figure 3: *Relative compression inefficiency versus in-memory model size for combined pair, edit, and escape models, on* PARTWSJ *($P = 1000$) and* DISK2 *($P = 10{,}000$). Relative compression inefficiency is with respect to compression acheived by a word model (of 100%).*

the components are simple and we foresee no difficulties developing a production implementation.) Moffat, Zobel, and Sharman showed that even reducing model size to only 10 Kb for a 500 Mb file only degraded speed by 22%, with marginal degradation for larger models [7]. Introducing pairs increases the speed, since more bytes are output per code. The edit model is likely to have similar impact to the escape model; extrapolating from the impact of edits on decode speed for genomic nucleotide data, we would expect speed to degrade by 10%–20%.

6 Conclusion

We have investigated a range of models for text compression in the context of document databases, evaluating both compression efficiency—the ability to produce a small compressed file—and the size of the model that must be resident in memory during query evaluation. Previous work has focused on word models and Huffman coding. We chose not to investigate alternative coding methods, since Huffman coding provides compression efficiency close to the theoretical minimum in this context, and allows fast decompression. Our investigation of models indicates that word models are an appropriate choice: other models provide poor compression efficiency, and our techniques for refining the word models (through pairs, edits, and escapes) led to only moderate reductions in compressed file size.

Our suggestion that common pairs be included in the word model provided the greatest improvement, increasing memory requirements but improving compressed file size by around 1.3%, in a typical case from 28.4% to 27.1% of original data size. We plan to investigate alternative techniques for choosing pairs to include in the model, which we expect to lead to further improvements in compression efficiency.

By omitting rare words from the model and coding them in the compressed text as sequences of characters, it has previously been shown that model size can be dramatically reduced with only moderate impact on compression efficiency. As an alternative to this escape model we proposed an edit model, in which rare words are approximated by a similarly-spelt word and a correcting edit instructions stored with the compressed text. This edit model had little benefit. However, we believe that with further work—generation of centroids and better representation of the edits—it may prove to be of value.

These models can be combined, yielding better compression than any of the models independently. For example, on our 860 Mb data file, combining words, pairs, edits, and escapes allows a model of 0.5 Mb to yield the same compression efficiency as a pure word model that is ten times larger. Together, these techniques significantly reduce both in-memory requirements for the model and compressed file size.

References

[1] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

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

[3] D. Harman. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management*, 31(3):271–289, 1995.

[4] U. Manber. A text compression scheme that allows fast searching directly on the compressed file. *ACM Transactions on Information Systems*, 15(2):124–136, 1997.

[5] A. Moffat and L. Stuiver. Exploiting clustering in inverted file compression. In J. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 82–91, Snowbird, Utah, 1996.

[6] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[7] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):302–313, 1997.

[8] C.E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:55, 1951.

[9] H. Williams and J. Zobel. Compression of nucleotide databases for fast searching. *Computer Applications in the Biosciences*, 13(5):549–554, October 1997.

[10] H.E. Williams and J. Zobel. Compressing integers for fast file access, 1999. (To appear).

[11] I.H. Witten, T.C. Bell, A. Moffat, C.G. Nevill-Manning, T.C. Smith, and H. Thimbleby. Semantic and generative models for lossy text compression. *Computer Journal*, 37(2):83–87, 1994.

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

[13] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software—Practice and Experience*, 25(8):891–903, 1995.