Information Retrieval Systems for Large Document Collections

Alistair Moffat*   Justin Zobel†

Abstract

Practical information retrieval systems must manage large volumes of data, often divided into several collections that may be held on separate machines. Techniques for locating matches to queries must therefore consider identification of probable collections as well as identification of documents that are probable answers. Furthermore, the large amounts of data involved motivates the use of compression, but in a dynamic environment compression is problematic, because as new text is added the compression model slowly becomes inappropriate. In this paper we describe solutions to both of these problems. We show that use of centralised blocked indexes can reduce overall query processing costs in a multi-collection environment, and that careful application of text compression techniques allow collections to grow by several orders of magnitude without recompression becoming necessary.

1 Introduction

Practical information systems are required to store many gigabytes of data while supporting rapid query evaluation. It is common for a single site to be used as a repository for many separate collections, either centrally managed or stored on separate machines; as indeed is evidenced by the diverse collections in the TREC data. In such environments queries may be posed to the set of collections as well as to individuals in the set.

Queries to the joint collection could be resolved by having a single centralised index of all documents. But such an index inevitably duplicates information held in the index of the individual collections, and, even with the low space requirements possible when the index is stored compressed,[8] may become onerously large if there are many collections. Moreover, if the centralised index is large, access to it may become a bottleneck in query processing.

Another approach is to provide a centralised index that does no more than identify which collections are likely to contain matches, so that queries are resolved by dispatching them to the hosts of likely collections. Such an approach has the advantage of distributing query processing costs, but raises the question of how a collection can be identified as likely to contain answers. A possible method would be to index the vocabulary of each collection—equivalently, index each collection as if it were a single document—with the expectation that a collection with a highly ranked vocabulary is likely to contain answers. However our initial experiments indicated that there was almost no correlation between vocabulary rank and value of a collection. Similar problems are presented by querying of information over Internet, for which a comprehensive search would be prohibitively expensive.[3,11]

Intermediate between these extremes is the possibility of the centralised index being to blocks of fixed numbers of documents, with the expectation that a highly ranked block is more likely to contain highly ranked documents than is a lowly ranked block. We explore such blocking in this paper, with, for the TREC methodology,[10] mixed results. It does allow reduction in the size of the centralised index, but only at the cost of identifying large numbers of false blocks (that is, blocks that rank highly yet contain no highly ranked documents). But the TREC experiments are designed to be high recall, and to some extent the

*Department of Computer Science, The University of Melbourne, Parkville 3052, Victoria, Australia. Email: alistair@cs.mu.oz.au

†Department of Computer Science, RMIT, GPO Box 2476V, Melbourne 3001, Victoria, Australia. Email: jz@cs.rmit.oz.au

problems arise because we are attempting to identify 1,000 highly ranked documents. When only a few documents are required, as is typically the case in ad hoc queries, performance improves.

These experiments have led to some interesting observations. One is that there appears to be no problem with, for a query, combining similarity values from separate collections, even though the weights for the query terms vary. Another is that tiny perturbations in weights can substantially perturb rankings, since even the 1,000'th document in a ranking often has a similarity value that, numerically, is greater than 50% of the score achieved by the top ranked document.

The other problem we have investigated in the last twelve months is the design of efficient compression methods for the text stored in dynamic collections. Although modifications are not as likely in text databases as they are in traditional database systems, it is relatively common for text collections to be continuously extended by addition of new data. Compression techniques for static text databases[8] use semi-static models, computed by initial inspection of the data and thereafter fixed. Semi-static models are required because queries retrieve documents randomly from within collections, and decompression is only practical if all documents have the same model. New text in a dynamic collection can also be coded with a semi-static model computed for the previous data, but compression performance slowly degrades, and if compression efficiency is to be maintained the model should be periodically recomputed and the collection recompressed. This may only be necessary each time the database doubles in size, but for a large collection this might still be an onerous requirement.

To reduce the frequency with which recompression must take place the constraint that all documents have the same model can be slightly relaxed. It is mandatory that the codes allocated to words not change. However if space is left in the model for new codes, to be allocated to new words as they appear, then existing codes can be retained. We have experimented with such a tandem model of document compression, in which known words are allocated codewords based upon their frequency in some initial seed text, and novel words are also stored in the lexicon and are allocated ordinal codes. Introduction of a "recency" technique then allows collections to expand by a factor of 1,000 before compression degradation is evident. For example, a collection of 100 Mb might be safely allowed to grow to 100 Gb before any significant compression degradation occurs. The details of this system are described in full elsewhere;[9] below we provide an overview of the technique and summarise the results obtained.

2 Indexing of multiple collections

Diverse collections of data can be stored by a single information retrieval system, and some information needs can span multiple collections. This is exemplified by the TREC data, the first 2 Gb of which contains nine distinct collections. Some queries would properly be directed only a single collection, such as the more recent set of articles from the Wall Street Journal. But many information needs are dealt with in all of the collections, as is shown by the relevance judgements for the TREC topics.

We were interested to explore how multi-collection queries might be processed if the nine collections were stored separately. The approach we considered was use of a centralised index that could be used to dispatch each query to the sub-collections most likely to contain relevant answers. The returned similarity values would then be compiled into a single ranked list. The combined ranking would then be used as a basis for selecting documents for presentation as answers, which would be requested from the separate collections. Such an approach has obvious application to a distributed system, in which it is desirable to minimise processing costs, and might also be appropriate for more anarchic environments such as collections at separate nodes on Internet.

The experimental regime was as follows. Each collection had a separate index, listing for each word the documents containing the word and the in-document frequencies. The indexes also stored word frequencies, to allow computation of inverse document frequencies in each collection. Thus, for each collection, similarity values were computed only on the information within each collection, and global information was not available. It was therefore quite possible for a query term to have high weight in one collection and low weight in another. The ranking measure used was the cosine measure with logarithmic in-document frequency,[5] that is, similarity of query q and document d is given by

$$C(q,d) = \frac{\sum_{t\in q\wedge d}(w_{q,t}\cdot w_{d,t})}{\sqrt{\left(\sum_{t\in q} w_{q,t}^2 \cdot \sum_{t\in d} w_{d,t}^2\right)}}$$

where $w_{x,t} = \log(f_{x,t}+1)\cdot\log(N/f_t+1)$, $f_{x,t}$ is the frequency of t in x, N is the number of documents in the collection, and f_t is the number of documents containing t.

There was also a centralised index of blocks of documents, where the number B of documents per block was fixed. That is, the words of the kth document of a collection were recorded as occurring in block $\lceil k/B\rceil$ of the collection, and overall word frequencies were based on the number of blocks containing each word rather than the number of containing documents. In such a regime it is quite possible for a block to contain a combination of words but for no document in the block to contain that combination, a problem that becomes more acute as B is increased. We refer to a block as a *false match* if it is highly ranked but contains no highly ranked documents.

Query evaluation with centralised indexes can proceed as follows. The centralised index is used to find the blocks that are highly ranked with respect to the query. If r answers are required, at least $\lceil r/B\rceil$ blocks must be identified; but almost certainly highly ranked blocks will contain some irrelevant documents, so more than $\lceil r/B\rceil$ blocks should be fetched. Even fetching r blocks does not guarantee that the top-ranked documents will be retrieved, particularly when B is large, since a highly similar record might have its strength diluted by occurring in the same block as many records containing no query terms. Exploration of the number of blocks that should be fetched is an important theme of these experiments.

Once the blocks have been ranked, the query is dispatched to each collection with matching blocks. The whole of each such collection could be ranked against the query, so that the purpose of the centralised index is to heuristically select a subset of the collections for further query evaluation. We instead chose to evaluate a mechanism in which only the documents in matching blocks were considered, to reduce processing effort. Each collection c has r_c documents in its matching blocks; of these the top r similarity values must be returned to the central processor if $r \le r_c$, or r_c similarity values otherwise. At the central processor the similarity values from each collection are merged to produce a single ranking of as many as $\sum r_c$ answers; of these the top r are then retrieved from the collections, collated into rank order, and presented to the user. This four-stage model of query processing is illustrated in Figure 1.

Consider the behaviour of the centrally indexed system as B is varied. With $B = 1$, the central index is ranking documents directly, but this ranking is not the same as would be given by querying the individual collections, as the $\log(N/f_t+1)$ values will vary between collections. Simply requesting the top r answers may omit documents that would be ranked highly in their collection, so fetching $r' > r$ blocks (each of one document) may yield better retrieval. Even if $r' = r$ blocks are ranked, the similarity values returned by the collections will permute the ranking generated by the central index. Hence, this is already different from our previous approach to TREC, in which the nine subcollections were treated as a single monolithic whole.

Now suppose B is increased, to say 10. The size of the central index diminishes, as the number of blocks containing each word decreases as does the size of the interblock gaps, which in turn allows better index compression. But the new size will still be considerably greater than 10% of the old, because the blocks of 10 documents contains many more distinct terms than the old blocks of one document each. The increase in B also increases the possibility of false block matches, but potentially at least the number of blocks that needs to be fetched to find r answers falls, since a matching block can contain more than one answer. Again, varying r', the number of blocks considered, controls retrieval performance. However, since the number of similarity values returned from the collections to the index will be largely unchanged, overall processing costs at the central index should be substan-

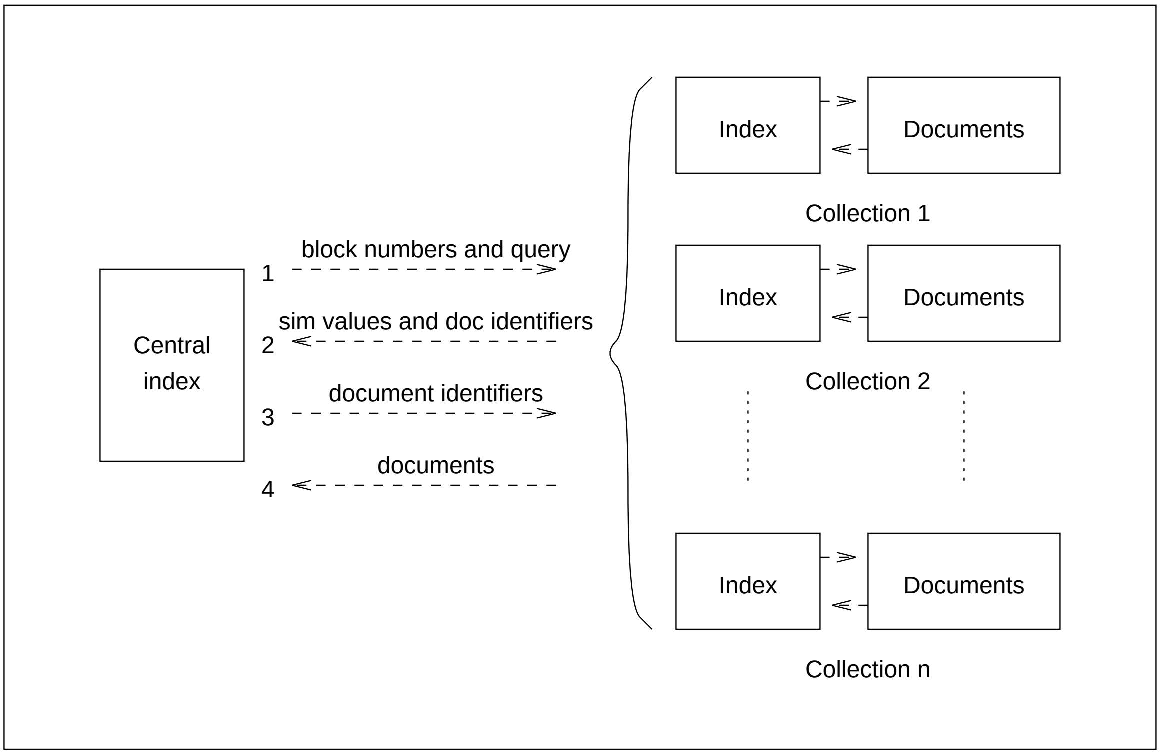

Figure 1: Architecture of a centrally indexed information retrieval system

tially reduced.

As B is increased further central index size and central processing costs continue to fall, but with diminishing returns. Larger blocks imply higher probability of false block match, so that, even if the number r' of blocks considered is decreased, the overall number of documents that must be ranked increases. We would expect that, for sufficiently large B, effectively the whole of each collection would have to be ranked to produce good retrieval effectiveness, somewhat defeating the purpose of a central index.

Following this thought experiment to the limit, when B is huge each collection is a single block. In this case the only benefit of the central index is that some collections might not need to be considered, if for example the rankings of the collections are widely separated. But the likelihood of false collection match seems very high, and we would expect there to be little benefit to such coarse level indexing.

Note that we have not in any way supposed that blocks are constructed according to relatedness of topic. If blocks consisted of similar documents then retrieval performance for a given r' should improve.

3 Experimental results

We have experimented with a centralised index by regarding the TREC data as consisting of nine separate collections, AP1, AP2, DOE, FR1, FR2, WSJ1, WSJ2, ZIFF1, and ZIFF2. The centralised index was implemented by using `mg`[8,13] to index the TREC data, but with document separators deleted to mimic aggregation of documents into blocks. Thus reindexing was required for each value of B tested. Topics 51–150 were transformed into `mg` input by removal of stopwords and SGML markup. The output of each query was a list of block numbers, which were transformed into lists of TREC document identifiers by Unix utilities. We had already precomputed for each collection a 10,000-document list of similarity values for each query; and so final lists of answers were computed by joining the query output to

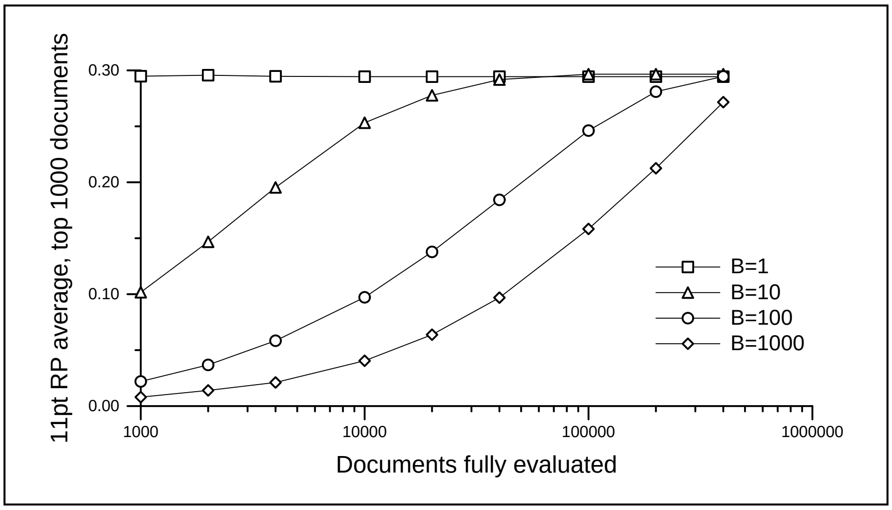

Figure 2: Recall-precision at different block sizes, r = 1,000, topics 51–150

these values. That is, the process of selecting answers from collections was simulated, and while this gives exact values for retrieval effectiveness, we can only estimate the CPU time required by this phase of query evaluation.

As a baseline, we also indexed the full TREC collection in the normal way—that is, as one monolithic collection—and computed retrieval effectiveness as an 11-point average assuming that 1,000 documents are returned as answers for each query. The result (queries 51–150) was an effectiveness of 29.4%.

We then experimented with the retrieval effectiveness achieved for retrieval of 1,000 documents per query and various values of B, as shown in Figure 2. The horizontal axis is the number Br' of documents considered, that is, B times the number of blocks identified as highly ranked. For $B = 1$, the central index is identical to that of our baseline, and as can be seen performance is almost independent of the number of documents fetched. There is a slight peak around $Br' =$ 2,000, indicating that there may be a small advantage to ranking documents as members of individual (presumably homogeneous) collections compared to ranking them as members of one heterogeneous collection. We would expect such an effect to be more pronounced over larger numbers of collections, or sets of collections that are more diverse.

For higher B, false block match rapidly becomes a significant problem. For blocks of 10 records, even fetching 1,000 blocks (that is, $Br' =$ 10,000) results in a loss of retrieval effectiveness. For $B =$ 1,000, the bulk of the collection must be processed to achieve good performance.

These problems are to some extent due to the experimental methodology. The TREC procedure of fetching 1,000 records is used to ensure high recall, so that there is some degree of certainty that most of the relevant documents have been located. But for ad hoc queries a much smaller number of answers are typically requested. To test the performance of a central index in this context, we have rerun the above experiments but assuming that only 10 answers are returned for each query. Results are shown in Figure 3. In this case the vertical axis records average precision after 10 documents are retrieved. As can be seen, there is a similar pattern of performance: a disproportionately large number of documents must be processed to achieve good performance when B is large. However in this case the total amount of effort required is reduced. For example, if a $B = 100$ index is being used, around 100 blocks and 10,000 documents must be ranked to achieve performance comparable to that attained by the $B = 1$ index examining just 10 blocks. But, if we equate volume of index inspected with ranking time, in the latter case all of the central index must be examined, plus 10/742,000'ths of the document indexes; while in the former case,

Blocking factor B	Index Size (Mbyte)
1	127.6
10	63.5
100	26.9
1,000	9.7

Table 1: Size of central inverted index

a central index one fifth the size must be examined, plus 10,000/742,000'ths of the component indexes, a net saving in index processing time.

Table 1 shows the sizes of the central indexes used in these experiments. The nine subcollection indexes totalled 118.4 Mb.

4 The 1994 experiments

This section summarises the two experimental runs that were submitted for assessment in 1994. For both of these runs topics 151–200 were transformed into queries by removal of SGML tags, removal of stopwords, and removal of all non-alphabetic characters. Detailed performance analysis of these two runs appears elsewhere in these proceedings.

Run citri1

This was our control run, and used a blocksize of $B = 1$ and extracted the top $r = 1{,}000$ documents according to the central index and then permuted them in the final ranking according to their similarity to the query using the collection weights. Based upon the performance when used with topics 51–150, we expected this to give almost identical behaviour to a ranking based solely upon the central index, using the global term weights.

Run citri2

In this run a blocksize of $B = 100$ was used, and the top 1,000 blocks were expanded in the collections for each query. Hence, the final ranking was the top 1,000 documents—according to the subcollection weights—out of the 100,000 documents contained in the 1,000 most highly ranked blocks according to the central index. If each block contains on average just one good candidate, overall retrieval performance should be good.

5 Dynamic text compression

It is attractive to compress the contents of an information retrieval system.[4,12] With an appropriate choice of coding scheme, such as a semi-static word-based model with Huffman coding,[1,2,6,7] not only can the space required to store the data be reduced to under 30% of the original, but the time required to retrieve data can actually be reduced because of lower data transfer times and smaller seek distances on disk. The disadvantage of this approach is that if the collection is dynamic and documents are to be appended the new text must be compressed with the existing model, so that the model parameters slowly become inappropriate and compression performance degrades.

To allow coding of any new words at all, it is necessary to leave probability space in the model. In the case of Huffman coding this is equivalent to explicitly reserving a family of codes for new words. A straightforward approach is to allocate a single escape code; a new word is then coded as an escape followed by some simple representation such as a length followed by ASCII characters. However, as the stored text grows such an approach quickly leads to progressively worse compression; for example, in a newspaper database, appended text represents a chronological progression of events, in which new words come into frequent use ("Cherynobyl" is an example in the Wall Street Journal) and the use of others declines ("Reagan", for example, is now relatively infrequent).

But given that code space has been reserved, an arbitrary number of new codes can be allocated, so long as each new code in turn leaves space for further codes to be added, and provided that, once assigned, no codeword is ever changed. Thus new words can, without risk of ambiguity, be inserted into the lexicon of known words and allocated a new code. In a sense, two channels of communication from encoder to decoder can be used. The first is the conventional stream of encoded text, as for all compression applications. The second channel is where we benefit from the

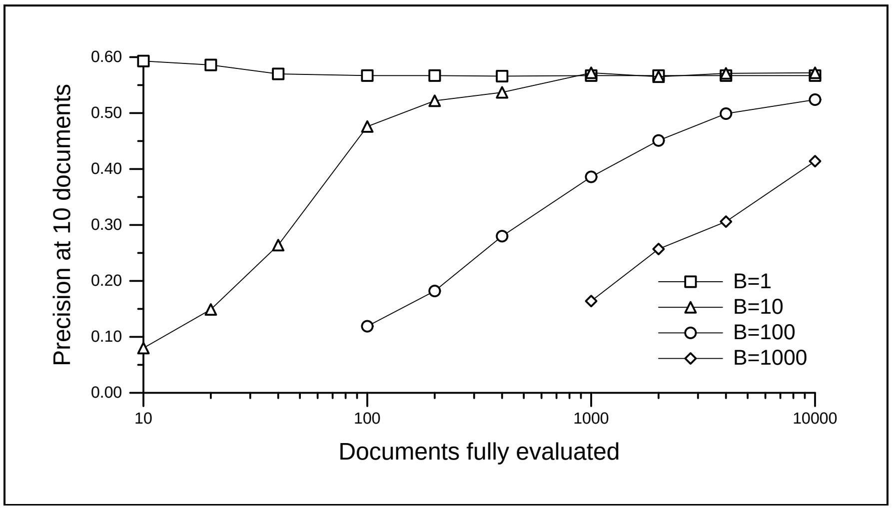

Figure 3: Precision at different block sizes, $r = 10$, topics 51–150

specialised nature of this application: it is the lexicon of terms and codewords assigned, and because no document can be accessed (and hence decoded) prior to its insertion, the scheme is robust.

We have had success with a tandem scheme on the above principles. In this scheme, a pass is made over all of the text that is initially available, a word model computed, and canonical Huffman codes allocated to the words. The code set also includes an escape code, calculated upon an estimate of the rate at which new words can be expected to appear. Then for new words an auxiliary lexicon is used, in which words are listed in order of first occurrence an assigned ordinal codes based upon some unambiguous method for representing infinitely large integers. During the compression process words in the primary lexicon are coded using their pre-calculated Huffman codes, while words in the auxiliary lexicon are coded as the escape code followed by their ordinal number in the auxiliary lexicon. Words novel even to the auxiliary lexicon are simply assigned the next ordinal code and installed, and that ordinal code then emitted. In particular, there is no need for novel words to be "spelt out" to the decoder as would be the case in a more conventional one-channel adaptive compression scheme.

Since any fixed code disadvantages words that are frequent in the appended documents (which might far outnumber the seed documents) but rare in the original seed text, we have added a further heuristic to allow the list of auxiliary words to be self-adjusting within any particular document. The codeword assignment must, of necessity, be fixed at the commencement of each document. However, once an auxiliary word has been seen once in a document it is likely that it will appear again a second time in that document. To exploit this recency effect we adjusted the ordering within the auxiliary lexicon after each use of it, swapping the word used to the next position at the front of the lexicon. This gives improved compression because any infinite encoding of the positive integers must assign arbitrarily long codewords to arbitrarily large integers, and so small integers must be assigned codewords of bounded length. Hence, novel words that appear more than once in any document are usually coded with a shorter codeword at their second and subsequent appearances than at their first.

The question then becomes one of selecting a coding scheme for new words. After experimenting with a variety of codes, we used a variable-length code parameterised by the size of the initial lexicon. The combination of all of these components allows the text to grow by a factor of 1,000 before compression performance significantly degrades, as is illustrated in Table 2, which shows compression performance on the 508 Mb

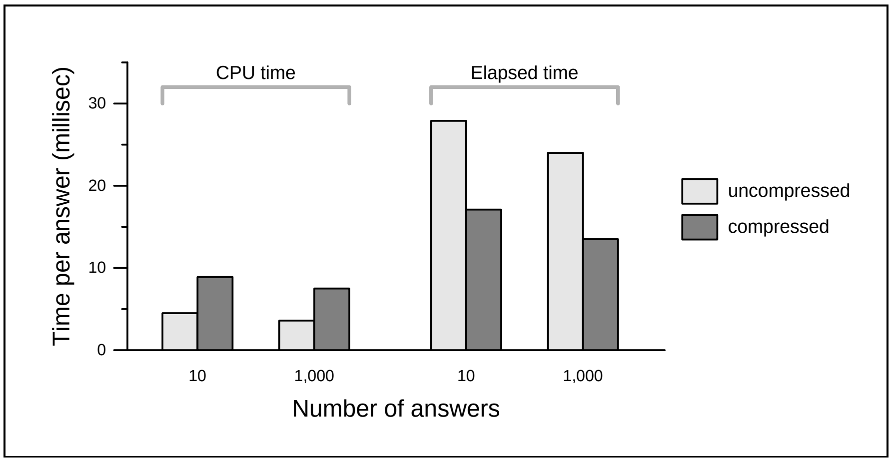

Figure 4: Speed of compressed and uncompressed retrieval systems, milliseconds per answer

Wall Street Journal (collections WSJ1 and WSJ2). In this table, the lefthand column is the factor by which the text has expanded since the model was computed and the codes allocated; thus, in the bottom line, the initial text is only about 32 Kb. The second column is the compressed size achieved, as a percentage of the original size, using a fixed Huffman code on characters to "spell" new words at each of their appearances; while the third column is the compression attained by the tandem model advocated here. The base compression, corresponding to an expansion factor of 1, is 28.4%.

Expansion factor	Character Huffman (%)	Tandem Huffman (%)
4	29.9	29.4
64	31.3	29.6
1,024	35.5	29.9
16,384	49.2	31.7

Table 2: Compression performance for the Wall Street Journal for varying initial text size

Decoding using the improved model is still fast, and it takes around 3–5 milliseconds to decompress the typical TREC document of around 3 Kb. This is only a small component of retrieval time, which is still dominated by disk access and transfer costs. Indeed, the 70% space saving brought about by compression means that not only are transfer times less when the data is compressed, but also that seek times are less. Figure 4 shows the time, in milliseconds per answer, to randomly retrieve 10 and 1,000 documents from a compressed and an uncompressed WSJ database respectively using the `mg` software. In each experiment a random list of ordinal document identifiers was generated and the system asked to fetch and present those documents in document number order, mimicking the effect of a Boolean query. Each of the points plotted is the average over 500,000 document retrievals, run on an otherwise idle machine and against a retrieval system stored on local disks. As can be seen, the CPU time required per answer is greater when the database is compressed, but the elapsed time—which includes disk seek and transfer costs—is less.

Further details of the compression scheme proposed for dynamic collections are described in the full paper.[9]

6 Conclusions

We have explored two hypotheses during the last twelve months. The first is that large agglomerations of data such as the TREC corpus can be usefully regarded as a collection of subcollections without substantial loss of retrieval performance. As an extension to this, we have also considered how the separate collections might be managed in a distributed environment. Or results are mixed: effective information retrieval can be carried out by using a central index to refer queries on to the appropriate subcollections, but in the TREC framework only at the expense of inspecting a non-trivial amount of each subcollection. On the other hand, ad hoc queries involving a small number of answers can be efficiently partitioned, with a net saving in work and a useful distribution of workload.

The second hypothesis explored in 1994 is that compression can be effectively applied to the text of dynamic collections just as well as it can to static. Our results here show that, seeded with as little as 0.1% of the final WSJ collection, compression results comparable to those attained for the complete collection can be attained. That is, compression can be applied to dynamic collections with little need for periodic subsequent rebuilding of the database.

Acknowledgements

We would like to thank Tim Shimmin for programming support. This work was supported by the Australian Research Council, the Key Centre for Knowledge-Based Systems, the Centre for Intelligent Decision Systems, and the Collaborative Information Technology Research Institute.

References

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

[2] 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.

[3] C.M. Bowman, P.B Danzig, and M.F. Schwartz. Research problems for scaleable Internet resource discovery. Technical Report CU-CS-643-93, Computer Science Department, University of Colorado at Boulder, 1993.

[4] G.V. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, December 1985.

[5] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[6] 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.

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

[8] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[9] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. Technical Report TR-94–4, Collaborative Information Technology Research Institute, RMIT and The University of Melbourne, 1994.

[10] National Institute of Standards and Technology. *Proc. Text Retrieval Conference (TREC)*, Washington, November 1992. Special Publication 500-207.

[11] M.F. Schwartz, A. Emtage, B. Kahle, and B.C. Neuman. A comparison of Internet resource discovery approaches. *Computing Systems*, 5(4):461–493, 1992.

[12] D.G. Severance. A practitioner's guide to data base compression. *Information Systems*, 8(1):51–62, 1983.

[13] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and indexing documents and images.* Van Nostrand Reinhold, New York, 1994.