Efficient Retrieval of Partial Documents

Justin Zobel* Alistair Moffat† Ross Wilkinson‡ Ron Sacks-Davis§

June 1994

Abstract Management and retrieval of large volumes of text can be expensive in both space and time. Moreover, the range of document sizes in a large collection such as TREC presents difficulties for both the retrieval mechanism and the user. We consider division of documents into parts as a solution to the problem of the range of document sizes and show that, for databases with long documents, use of document parts can improve the quality of the information presented to the user. We also describe the compressed text database system we use to manage the TREC collection; the compressed inverted files with which it is indexed; and the techniques we use to evaluate the TREC queries, both on whole documents and on document parts.

1 Introduction

Management of a large volume of text such as the TREC document collection requires a text database system. Not only do the documents have to be stored and subsequently located, but indexes are required if ad hoc queries are to answered in a reasonable amount of time; and, to answer queries effectively, some form of ranking technique is required. In contrast to Boolean queries—for which the question of whether a document is a match to a query is a binary value—ranking techniques assign a similarity value to every document in a collection with respect to a given query, and order the documents by these values.

The TREC collection presents several problems for text database management systems. First, there is the sheer volume of data to be stored. We address this by use of text compression techniques—a word-based compression model supported by Huffman coding. In this application Huffman coding yields near-optimal compression and fast decoding, the CPU cost of which is offset by savings in retrieval time; and the word-based model reduces the stored text to under 30% of its original size. Likewise, we show that there is no reason to regard inverted files as unduly space intensive. Using simple integer coding techniques, the document-level inverted indexes required for ranking can be stored in about 10% of the original text size, again with only minimal impact on retrieval time. Taken together, these compression techniques allow the storage of the complete retrieval system for the TREC

*Department of Computer Science, RMIT, GPO Box 2476V, Melbourne 3001, Australia. jz@cs.rmit.oz.au
†Department of Computer Science, The University of Melbourne, Parkville 3052, Australia.
‡Department of Computer Science, RMIT, GPO Box 2476V, Melbourne 3001, Australia.
§Faculty of Applied Science, RMIT, GPO Box 2476V, Melbourne 3001, Australia.

collection (disks one and two) in about 820 Mb, a substantial saving compared to the initial 2,055 Mb.

Second, the memory required to evaluate ranked queries can be large: space is required for accumulators for document similarity values, and for the document weights that are used to normalise the similarities with respect to each document's length. We have shown that using only a few bits of precision for the document weights need degrade neither retrieval effectiveness nor retrieval speed. Similarly, limiting the number of accumulators does not, within reasonable bounds, degrade retrieval effectiveness. Moreover, the fact that most documents are not allocated an accumulator allows improved query response time, since the bulk of the index information does not have to be processed.

Third, the mix of document lengths in the TREC collection presents difficulties for the retrieval mechanism—they vary in size by a factor of over 10,000. For ranking techniques such as the inner product that do not normalise by document length, long documents are favoured simply because they contain more words. It is difficult, however, to define a fair normalisation technique. For example, the standard cosine measure [13] favours short documents, and a popular variant [3, 9] is even more biased.

We suggest that the problem of length diversity be addressed by dividing long documents into parts that can be retrieved separately. Such division has other advantages, not least of which are lower retrieval cost and the greater ease with which an interface can present a short piece of text. What is not clear, however, is how to divide documents. We experimented with the simplistic measure of dividing documents into roughly equal-sized "pages", and with a more sophisticated approach based on the documents' sections. Surprisingly, page-based retrieval outperformed both document-level and section-based retrieval when applied to the FR subcollection of TREC, which has many long documents. Moreover, the use of document parts improved the presentation of answers, by eliminating irrelevant material. The division of documents was also useful in that it provided a stress test of our system, substantially increasing both the number of objects to be stored and the complexity of query evaluation.

The organisation of our text database system is described in Section 2. The text and index compression methods employed are summarised in Section 3. Section 4 describes methods for rapidly evaluating queries in limited memory. In Section 5 we discuss possible methods for partitioning documents, and how ranking might be performed in this context. Conclusions are presented in Section 6.

The software used for our experiments is available free of charge by anonymous ftp from the directory /pub/mg on munnari.oz.au [128.250.1.21].

2 Organisation of text databases

A simple text database system designed for ranked query evaluation consists of a lexicon, inverted lists, text records (that is, documents), and a table of document data, as shown in Figure 1. Each document contains a series of words, some of which may occur several times. The lexicon contains all of the distinct words that occur in the documents, and for each word has a pointer to an inverted list, which is a list of identifiers d of the documents containing that word. The lexicon also stores, for each term t, the number f_t of documents

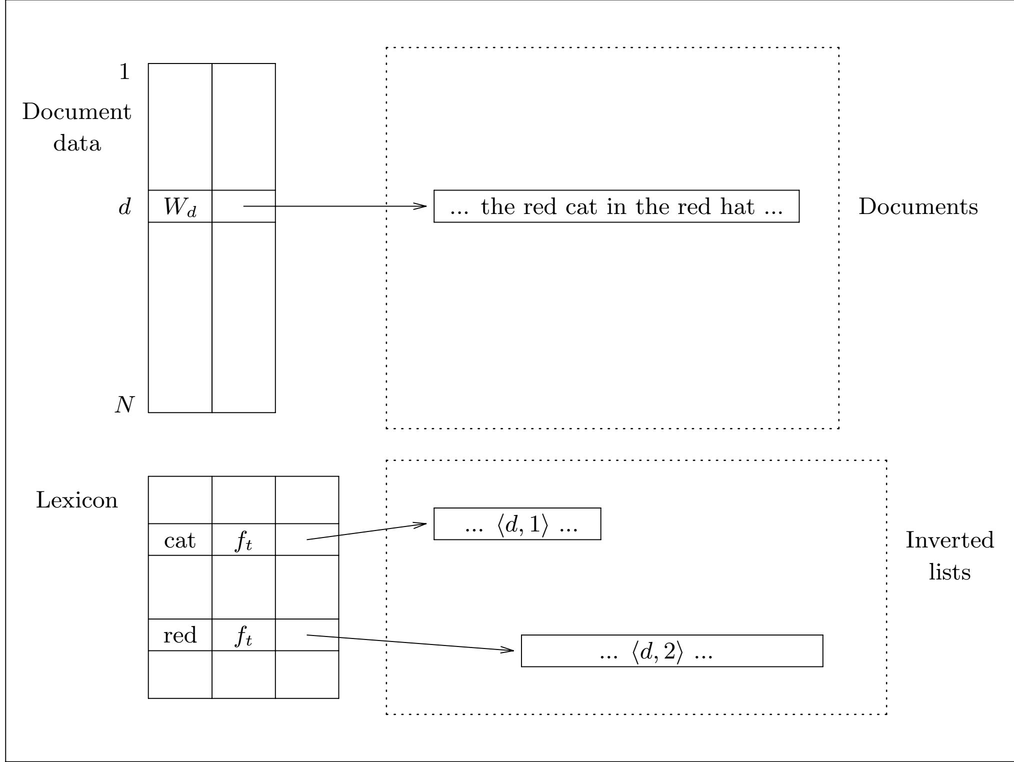

Figure 1: Organisation of a text database system

in which t appears, and w_t, the weight or "importance" of term t (which may be derived from f_t). With each identifier d in the inverted list for term t is stored the word's within-document frequency $f_{d,t}$ for that document. We refer to each such pointer as a $\langle d, f_{d,t} \rangle$ pair. Finally, the table of document data is an array that maps each document identifier d to the document's location on disk, and also contains the document weights W_d used for normalisation. For the document shown in Figure 1, there are two occurrences of "red" and one of "cat"; hence the inverted lists for both of these words include d as a document, together with the respective numbers of occurrences.

Such a text database can be used to evaluate ranked queries as follows. To rank documents against a query, a function is required for computing the similarity value of each document d with respect to query q. One such function is the standard cosine measure, defined for evaluation purposes by

$$C_{q,d} = \frac{\sum_t w_{q,t} \times w_{d,t}}{W_d} = \frac{1}{W_d} \sum_{t \in q} w_{q,t} \times w_{d,t}$$

where $w_{x,t}$ is the weight of term t in query or document x and W_d is the weight of document d. In the cosine measure it is usual to take W_d to be the length in n-space of the document

vector $\langle w_{d,t} \rangle$:

$$W_d = \sqrt{\sum_t w_{d,t}^2}.$$

The term-document weight $w_{x,t}$ is usually defined by $f_{x,t} \times w_t$, where $f_{x,t}$ is the within-document frequency of t in x and w_t is the weight or importance of term t. Finally, one common method for assigning term weights is to take $w_t = \log(N/f_t)$, where N is the number of documents stored—the inverse document frequency (IDF) rule. With these assignments, the cosine similarity measure can be calculated as

$$C_{q,d} = \frac{1}{W_d} \sum_{t \in q} f_{d,t} \left(\log \frac{N}{f_t} \right)^2 \tag{1}$$

There are many alternative mechanisms for assigning term and term-document weights [7], but most of them can be calculated using the same framework. Throughout our experiments we assumed the formulation given in equation (1).

A simple way to compute the cosine measure is as follows. First, for each document in the database an accumulator is created in which the term-dependent components of (1) are summed. The initial value of each accumulator is zero. Then, for each of the terms in the query, the corresponding inverted list is retrieved, and the document identifiers and within-document frequencies in the list are used to update the accumulator for each referenced document, by adding in $w_{q,t} \times w_{d,t}$ terms. Finally, when all of the inverted lists have been processed, the accumulator values are divided by the document weights W_d to give the cosine values. A partial sort using a tree structure should then be used to extract the top documents. The top r values of N can be selected in $O(N + r \log r \log(N/r))$ steps on average [15, pp. 164–169], and when $r \ll N$ this is linear in N.

This simple ranked query evaluation strategy is effective at locating relevant documents, but the performance is far from ideal: large volumes of disk space can be needed for data and index; excessive main memory may be consumed by the accumulators and W_d values; and megabytes of inverted lists might need to be retrieved, processed, and, in our case, decompressed. How we addressed these problems in the context of TREC is the subject of the next two sections.

In addition to the performance problems, the cosine measure tends to be biased against long documents; moreover, it does not provide any indication as to which part of a long document is relevant. Approaches to solving these problems are discussed in Section 5.

3 Text database compression

One of the biggest problems posed by the TREC collection is that of size. In all of the experiments reported here the data on disks one and two was used, and in raw form there was 2,055 Mb of text to be manipulated. The index is also potentially very large. As part of our investigation into partitioning of documents (described in Section 5), the TREC collection has been mechanically broken into over 1,700,000 "pages" of about 1,200 bytes (200 words) each. The index for the paged TREC contains 196,000,000 pointer pairs $\langle d, f_{d,t} \rangle$, and stored without compression as six-byte records, would consume 1,120 Mb, an overhead of more

than 50%. In this section we survey the compression techniques used to reduce the space required by the text and index. In combination, they allow both the text *and* index to be stored in under 40% of the space occupied by the text alone—an almost fourfold reduction in space.

3.1 Compression of stored documents

The largest single component of a text database system is the stored documents. The obvious way to reduce the space the documents occupy is to apply some text compression algorithm, but most of the methods in common use are inappropriate in this application. In particular, most file compression techniques make use of an adaptive model to produce a continuous bitstream that must be sequentially accessed, and do not provide any facility by which the bitstream can be decoded starting at an arbitrary location [1]. Such behaviour conflicts with the need to return individual documents in a text database. Applying the same compression techniques to the individual documents rather than the entire collection is not a solution either. Adaptive compression techniques give good compression only after tens of kilobytes of text have been processed, and give relatively poor compression during an initial "learning" period. Given that the indexing of short pages of text extracted from the TREC collection is one of our aims, there is no place for conventional adaptive compression methods. Although desirable from the point of view of compression performance, adaptivity conflicts with the need to access documents in an order different to that of the original encoding.

A further potential problem with compression of large collections such as TREC is the memory space required by the compression regime. The process of compression is generally regarded as having two components, a model that assigns a probability to each symbol, and a coder that determines a representation for each symbol given the set of probabilities [1]. To decode compressed data requires random access into the model to identify the encoded symbols, hence the need to hold the model in memory.

The model size depends on the choice of what to regard as a symbol; in general, the greater the number of distinct symbols, the better the compression, but the larger the model. In the context of text compression, using whole words as symbols yields compression comparable to the best-known multi-order character-based compression schemes [15]. The only drawback with this choice is the amount of memory required to store the lexicon of the collection.

The largest document collection available to us prior to TREC was 132 Mb and had a lexicon of about 75,000 words and non-words. (Non-words—the tokens interposed between consecutive words—must also be stored if the compression is to be reversible and lossless.) For that collection, choosing words as symbols led to a decompression model requiring around 820 Kb of memory during query processing. For TREC we also chose words as symbols. Following conventional wisdom, we expected to observe that, as more documents were inserted into the database, the occurrence of a new word (that is, a new symbol to be included in the model) would become increasingly rare, and that by the end of the collection the size of the model would be effectively constant. To our surprise new words continued to appear, at a rate of around one word per one thousand word occurrences. Figure 2 shows

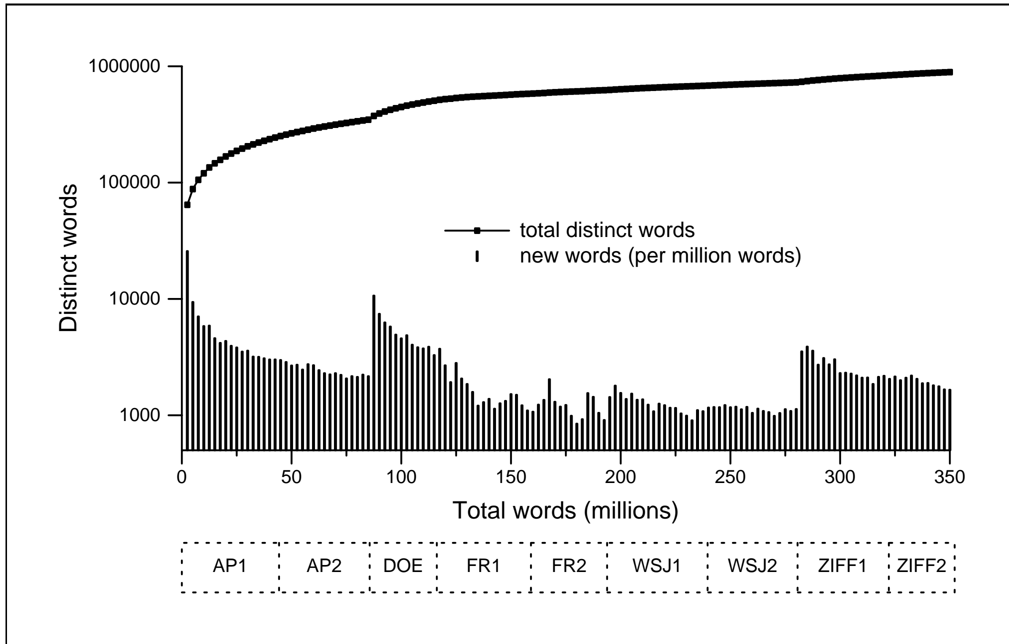

Figure 2: Compression lexicon size for disks one and two of TREC

the number of distinct words in the TREC collection, and the rate at which they appear in the text. Note the sudden increases when the nature of the text changes. For example, the first jump at about 85 million words corresponds to the end of the two sections of the AP collection and the start of the DOE collection. Conversely, the lack of discontinuity at the commencement of the WSJ collection indicates that one of the previous components (in fact, AP) contained text of a similar nature to that stored in WSJ.

The new words are primarily place names, technical terms, spelling errors, unusual proper names, acronyms, and abbreviations. Despite their scarcity—many appear only once—they are all stored and indexed. The resulting compression model required 11 Mb of memory; and although well within the capacity of a modern workstation, would be difficult to support on a PC. Different techniques are needed for collections that are larger or more diverse than TREC, and mechanisms for trading a small loss in compression for reduced model space are part of our current work [10]. The index vocabulary, which also becomes large, can be stored on disk with little impact, since compared to the compression model it is accessed only lightly during query processing.

To avoid the problems of adaptive modelling we used a semi-static model [1]. Hence, the compression process requires two passes through the data: one to accumulate the statistics with which to drive the model and a second to actually encode the text. A two-pass approach is perfectly acceptable for static collections such as TREC. In other work we have shown that the same methods can be extended with minimal degradation to dynamic collections and other situations in which it is not possible to make two passes [10].

For text data with a word-based model, even the most common symbol—the word "the"—constitutes less than 5% of the symbols to be encoded, and so Huffman coding is near optimal [5]. Huffman coding also avoids the throughput problems of the only prac-

tical alternative, arithmetic coding. In our implementation, which makes use of a canonical code [15], decoding is very fast, requiring only three or four machine instructions per output byte. On a SPARC 10 Model 512 compressed text can be decoded at an output rate of about 1 Mb per second.[1]

3.2 Index compression

The bulk of the space required for an index is to store the inverted lists. For a collection as large as TREC, each pair of $\langle d, f_{d,t} \rangle$ values would require six bytes uncompressed, four bytes for d and two for $f_{d,t}$. In this section we show how this requirement can be reduced to about one byte per $\langle d, f_{d,t} \rangle$ pair. The key to index compression is the observation that each inverted list is an ordered sequence of unique d values interleaved with $f_{d,t}$ values that are generally small.

Since the d and $f_{d,t}$ values are effectively independent they can be considered separately. The d values in each inverted list are a strictly increasing sequence of integers in the range 1 to N, the number of stored documents. A typical sequence might be

$$8, 9, 21, 22, 32, 34, \ldots$$

In this form, there is no pattern or repetition of values to be exploited for compression, but if we take differences (or run-lengths) between adjacent numbers, denoted as d-gaps, to yield the sequence

$$8, 1, 12, 1, 10, 2, \ldots,$$

then a pattern begins to emerge—a series of d-gaps whose average is around 6. Using this average, for each inverted list a number b is computed, and each run-length is encoded as a pair of values q and r, where the d-gap equals $bq + r + 1$, with $0 \leq r < b$. These codes are known as Golomb codes [6]. The log of b determines the number of bits required to store r as a binary integer; and q, which will usually be 0 or 1, should be represented in unary. For the example inverted list above, the first d-gap is 8. If this was coded with $b = 4$, which is the optimal value of b when the average gap is of length 6, then $q = 1$ and $r = 3$. These in turn are represented by the bitstring "10,11", where the first component is unary for $q = 1$ and the second is a 2-bit binary code (2-bits since $b = 4$) for $r = 3$. Similarly, the second d-gap of 1 is coded with $q = 0$ and $r = 0$ as the bitstring "0,00"; and the third d-gap, with $q = 2$ and $r = 3$, is represented by "110,11". The $f_{d,t}$ values can also be represented using an encoding for small integers. One suitable method is Elias's γ code, with which integer x is represented by a codeword of $1 + 2\lfloor \log_2 x \rfloor$ bits [4].

Using the above coding scheme, a pair of $\langle d, f_{d,t} \rangle$ values can be represented in an average of around eight bits each, yielding an index of less than 10% of the size of the original text [2]. Decoding is again fast, requiring around fifty machine instructions to decompress a $\langle d, f_{d,t} \rangle$ pair. This corresponds to a decompression rate of about 400,000 inverted list entries per second.

[1] All execution rates and processing times given in this paper are for this machine, with experiments carried out using files on local disks and when the machine was otherwise idle.

If word locations within documents are also to be stored—for example, if word-adjacency queries must be answered without recourse to a post-retrieval scan—then further information must be stored with each pointer. Similar methods can be used to store any desired level of location hierarchy, such as word position within sentence within paragraph within document. In this case the total index becomes larger, both because of the more detailed information being stored, and also because multiple appearances of a term in a document require multiple pointers. Using the same compression techniques a full word-level index requires approximately 25–30% of the text being indexed. Indeed, it is hard to see how it could require any less than this amount, since the best compression methods reduce English text to around 25–30%, and—albeit with some considerable effort—the source text can be reproduced almost exactly from a word-level index.

A particular advantage of compression is that the inclusion of common words in a document-level index is not particularly expensive. This means that the choice of which words to stop can—indeed, should—be deferred to query evaluation time, and different stopwords can be used by different query evaluation techniques on the same database and index. All of the results below are for an index that records the page location of all words and numbers that appear in the text, where, as explained previously, a page is about 1,200 bytes of text.

3.3 Index construction

As part of our investigation of large text databases, new algorithms have been developed for index construction [15, pp. 201–203]. Using these algorithms, the paged index was built in under 4 hours, using a peak of 40 Mb of main memory and less than 50 Mb of temporary disk space above and beyond the final size of the inverted file. These indexing methods also rely heavily upon the use of the compression techniques described earlier. Two passes are made over the text, and so the methods are again most suited to static collections.

3.4 Compression performance on TREC

Table 1 summarises the result of applying these compression methods to the paged TREC collection built from the data on disks one and two. All of the components of the retrieval system are listed, and no other files are required to evaluate queries. That is, a total of 820 Mb of disk storage is sufficient to allow Boolean and ranked queries on the paged TREC data. The percentages listed in the final column are relative to the original 2,055 Mb of unindexed data. The complete retrieval system, including all index and vocabulary files, occupies less than 40% of this size. The equivalent document-level system, in which the index stores document numbers rather than page numbers, has an inverted index of 129 Mb.

Including both indexing and compression times—two passes for each, but implemented as two filters so that each pair of passes shared the data and only two passes in total were required—it took about 8 hours to build the TREC retrieval system described in Table 1.

File	Content	Size	
		(Mb)	(%)
Text	1,743,848 pages of text (742,358 documents)	604.91	29.5
Index	195,935,531 $\langle d, f_{d,t} \rangle$ pairs	184.36	9.0
Compression model	933,379 $\langle \text{word}, \text{code} \rangle$ pairs	4.19	0.2
Lexicon	538,244 $\langle t, f_t, \text{address} \rangle$ triples	9.42	0.5
Document information	1,743,848 $\langle W_d, \text{address} \rangle$ pairs	13.30	0.6
Approximate weights	1,743,848 six-bit W_d approximations	1.25	0.1
Total		817.43	39.8

Table 1: Components of the TREC document database (disks one and two, paged)

4 Query evaluation in limited memory

The query evaluation technique described in Section 2 is both time-intensive and greedy of main memory. In this section we describe how we address these problems. The two major consumers of memory are the W_d values used for normalisation, and the document accumulators. The requirements of these two structures are examined in the next two subsections. Then, in the third subsection, the cost of processing index lists is considered. The final subsection describes the result of applying these three techniques to the TREC collection, and details the performance levels that were achieved on both the original document-level collection and on our paged collection.

4.1 Memory for document weights

In the query evaluation strategy described earlier thousands or tens of thousands of W_d values need to be accessed to finalise the ranking. These values must be held in memory if excessive disk traffic is to be avoided. They are typically floating point values, stored in four or eight bytes, and are generally unique and hence not amenable to lossless compression. Ranking, however, is an approximate process, and there seems little reason to use high precision to represent W_d values. We have investigated representing them lossily in a small number of bits, using a geometric assignment of codewords so that short lengths are represented with the same relative error as longer ones [12]. This is important because they are eventually used in a multiplicative manner.

For example, if the lengths vary from 1 to 10,000 and two bits are to be used for each W_d value (that is, four distinct approximations), then a linear representation would imply that the range be broken into four subranges: 1–2,500, 2,501–5,000, 5,001–7,500, and 7,501–10,000. The values in each subrange are represented by one number from within that range. This, however, is unsatisfactory for any collection in which most of the W_d values are small, in which case most of the W_d values will fall into the bottom range, so that there is no way of distinguishing between the lengths of most documents. With a geometric representation, the subranges would be 1–10, 11–100, 101–1,000, and 1,001–10,000, allowing both smaller and larger document lengths to be distinguished. (Throughout this example it has been assumed that the lengths are integral. In practice they are not, and the buckets must be

assigned fractional boundaries.)

Experiments with this scheme have shown that with as few as six bits per W_d value there is little adverse effect on retrieval performance, leading to a roughly sixfold saving in memory [12].

4.2 Memory for accumulators

Neither lossy compaction nor lossless compression are viable options for the accumulators. Compaction would lead to cumulative errors; and a compression-decompression cycle (whether lossless or lossy) at each accumulator access would be unacceptably slow. The only remaining way to reduce the space occupied by accumulators is to have fewer of them.

A simple way to reduce the number of accumulators is to impose an arbitrary limit K on their number, and prevent the ranking process from allocating more once this limit is reached. With such a bound, the cosine measure is evaluated as follows. Initially there are no accumulators. Terms are processed in decreasing order of w_t (that is, in increasing frequency order, so that important terms are considered first), in two phases. In the first phase, for each document referenced in an inverted list, an accumulator is added if necessary and the value of the accumulator for each referenced document is incremented. This phase continues until, at the completion of some inverted list, there are at least K accumulators. In the second phase the remaining inverted lists are processed as before, but no new accumulators are added; the effect is to reorder the pool of documents selected during the first phase. We denote this approach the *continue* strategy, since the processing of inverted lists continues even after the accumulator limit is reached. A similar "pruning" strategy is described by Harman and Candela [8].

There is also another possibility, and that is to simply discontinue processing inverted lists once the accumulator limit is reached. We call this method *quit*. It returns the same pool of approximately K answers, but in a different order, and so must also provide an approximation to the full ranking. Furthermore, it is the longest inverted lists that are left untouched, so execution is very fast. Detailed descriptions and pseudo-code for these two strategies are given elsewhere [11].

For TREC, the number of accumulators increases quickly as terms are processed. Even the rarer terms in the TREC queries occur in thousands of documents. For example, when processing topics 51–100 a limit of 10,000 accumulators is on average reached after the inverted lists of only six terms are processed.

The two dark lines in Figure 3 show the performance achieved on the TREC data by the *quit* and *continue* strategies. (The gray line will be discussed below.) Although both the time taken to perform the ranking and the retrieval effectiveness achieved are independently controlled by K, the number of accumulators permitted, a user of the system is far more interested in the tradeoff between these two than in how many accumulators were used to achieve any particular result. With this in mind, Figure 3 plots retrieval effectiveness as a function of the time taken to achieve that effectiveness. Details relating to query formation and measurement of retrieval effectiveness are described in Section 4.4 below; the times shown are the average over three runs of the time required to process topics 51–100 relative to the paged TREC collection described in Table 1.

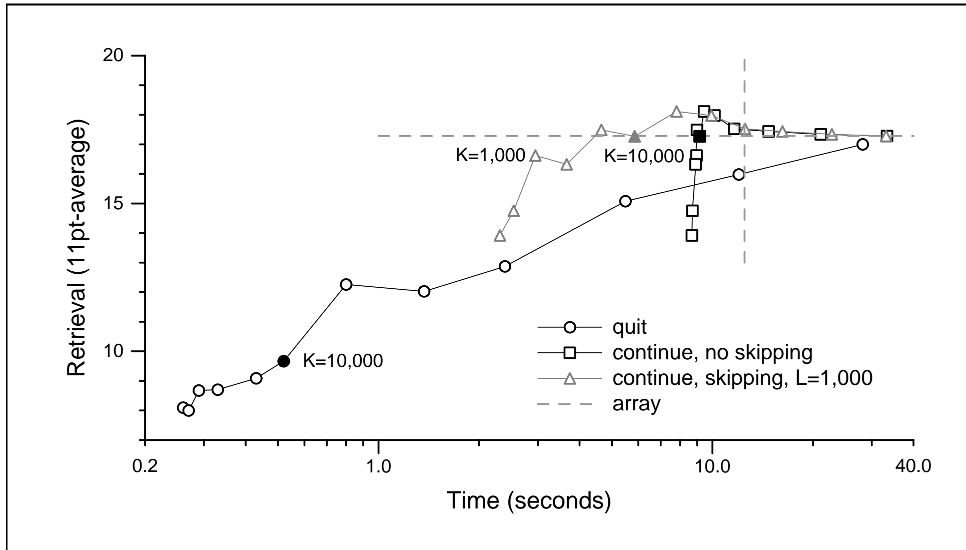

Figure 3: Tradeoffs in query processing on TREC (disks one and two, paged)

Each point on each curve corresponds to a different value for the accumulator limit K, ranging from 215 on the left (because an integral number of terms is processed, 1,500 actual accumulators were used at this point) to one million accumulators at the righthand end of each curve. The $K = 10{,}000$ point on each curve is filled in and labelled to provide a reference. Also plotted (as the intersection of the two gray dashed lines) is the time taken and retrieval performance obtained by the straightforward cosine implementation described in Section 2, in which an array is used to maintain an accumulator for every document, and every inverted list is fully processed.

The line labelled *quit* shows that query processing can be fast, but the corresponding retrieval performance is relatively poor. As the inverted lists for more terms are processed, retrieval improves, but the query takes longer. On the other hand, the *continue* strategy gives good retrieval performance—in some cases better than the full cosine method—but the time taken is much greater than the *quit* method, and is almost independent of K when K is small. If retrieval effectiveness is to be held high, restricting the number of accumulators appears to save space, but not time.

One interesting conclusion from these experiments is that rare terms are good at selecting relevant documents, but the more common terms are better at ordering them. Indeed, allowing the more common terms to add new documents appears to degrade retrieval performance, and almost all relevant documents contain at least one of the rare terms in the query. For TREC, a limit of $K = 10{,}000$ accumulators gave excellent retrieval effectiveness, and required a memory allocation of less than one bit per stored document. Even use of $K = 1{,}000$ gave retrieval effectiveness only slightly worse than the full cosine measure.

4.3 Fast ranking

With the limited accumulator scheme and the *continue* strategy, much of the decoding in the second phase is of information that is immediately discarded: the d and $f_{d,t}$ values for

documents that do not have an accumulator. Mechanisms for avoiding this unnecessary decoding would greatly reduce the time required to compute a ranking. One method to reclaim some of this waste is to insert into each inverted list a simple index, consisting of pointers at intervals that in effect divide the inverted list into a sequence of regular-sized blocks chained together by the pointers [11]. Each pointer or *skip* stores the bit address of the start of the next skip as well as the first d-value in the block. This arrangement is illustrated in Figure 4.

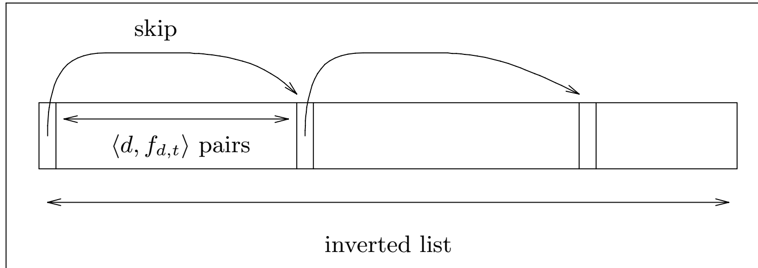

Figure 4: Adding skips

The skips can be used to decide whether any of the $\langle d, f_{d,t} \rangle$ pairs within the current block are for a document that has been allocated an accumulator; if not, the block can be skipped and decoding time saved. The size of the blocks—the distance covered by a skip—in each inverted list can be optimised for the total list length, given some assumed value L for the accumulator limit. The actual value K used in any particular query cannot, of course, be known at the time the index is created, and so L must be chosen prior to use of the database, matching some anticipated value of K. Use at query time of values of K other than the value L for which the index is optimised slows response, but does not otherwise affect the querying process.

Suppose that it costs one unit of time to decode each $\langle d, f_{d,t} \rangle$ pair and two units of time to decode each skip, that there are L accumulators to be checked, that on average half of each of L blocks must be decoded, and that there are p_1 skips and they must all be decoded. Then the total decoding effort for a term with frequency f_t is proportional to

$$\left(2p_1 + \frac{Lf_t}{2p_1}\right)$$

which has minimal value

$$2\sqrt{Lf_t}$$

when

$$p_1 = \frac{\sqrt{Lf_t}}{2}.$$

In the absence of skipping a total of f_t decoding steps would be performed, and so when $L \ll f_t$, the use of skipping provides processing of an inverted list in time sublinear in its length. For example, typical of the TREC queries processed is $p = 75{,}000$, and we have seen from Figure 3 that $K = 1{,}000$ yields acceptably good retrieval. In this case the analysis

	Index size (Mb)		Retrieval Time (CPU sec)		Non-zero accumulators	
	Doc.	Page	Doc.	Page	Doc.	Page
$L = K =$ 1,000	145.8	205.4	2.36	2.95	2,330	2,701
$L = K =$ 10,000	156.7	220.3	4.46	5.57	12,855	13,238
$L = K =$ 100,000	161.6	230.2	9.80	12.00	107,737	109,370
All (array)	128.6	184.4	7.66	12.48	582,783	1,307,115

(a) Resources required

	Terms processed		Precision at 200		Pessimal 11pt effectiveness	
	Doc.	Page	Doc.	Page	Doc.	Page
$L = K =$ 1,000	3.08	2.84	0.271–0.612	0.260–0.639	0.164	0.166
$L = K =$ 10,000	6.80	6.06	0.346–0.530	0.319–0.554	0.185	0.173
$L = K =$ 100,000	16.84	14.26	0.333–0.446	0.326–0.504	0.175	0.175
All (array)	42.44	42.44	0.331–0.444	0.321–0.502	0.174	0.173

(b) Retrieval effectiveness

Table 2: Document vs. page retrieval: (a) resources required, and (b) retrieval effectiveness

predicts that if an index is constructed with L = 1,000 and a query is processed with K = 1,000 then the number of decoding steps can be reduced from 75,000 to 17,000, a factor of about four. This saving is based upon the insertion of 4,300 skips, each spanning a block of about 17 base-level $\langle d, f_{d,t} \rangle$ pointers. At query time values of K other than the particular value L for which the index was constructed can also be used, but in such cases the speedup, while still substantial, will be suboptimal.

The third curve in Figure 3 (marked in gray) shows the time savings achieved by running the *continue* strategy against a skipped inverted index. The index was constructed with L = 1,000 and then used with a variety of values of K; retrieval effectiveness is the same as for the corresponding point on the *continue* curve, but when K is small and close to L the ranking is significantly faster. The third point marks the performance observed with K = 1,000, and, as predicted, performance is almost four times faster than the unskipped index. Table 2 lists results for indexes constructed for other values of L, assuming that the database designer clairvoyantly knows the value of K that will be used during query processing.

The inclusion of skipping pointers enlarges the index slightly, by 11% for TREC with L = 1,000. This is an insignificant overhead given the already small size of the compressed index. Moreover, with skipping the time taken to process a query is limited more by the cost of transferring inverted lists from disk than it is by decoding time. Given that the compressed inverted files can be retrieved more quickly than uncompressed inverted files (both transfer costs and seek costs are less, the latter because the file is smaller), it can be seen that compression improves both space and time requirements.

4.4 Documents vs. pages

The implementation techniques described above, skipping and limitation of the number of accumulators, can be applied to document retrieval as well as to page retrieval. Table 2 summarises the resources required and performance achieved by both document and page retrieval, for various values of K, the nominal number of accumulators.

The following points should be noted in connection with the data in Table 2.

Choice of L In each case the experiments were performed with an index constructed with $L = K$, and so the processing times reflect the best that can be expected. It is, however, worth noting that the conversion process from one value of L to another is relatively fast. All that is required is for one index to be read sequentially, the skips extracted, and then a different set of skips inserted as the index is rewritten. At the observed coding rate of about 400,000 pointer pairs per second, a complete "re-skipping" of the paged TREC index takes under thirty minutes. It is not necessary to rebuild the index from scratch, a process that takes about four hours.

More generally, Figure 3 shows the situation in which a static index is constructed with a single choice of L and then used with differing values of K. Query processing time is relatively insensitive to the exact value of L used during index construction.

Query construction A simple approach was adopted to produce query lists from the TREC topics. All punctuation and the SGML markup was removed, and a set of 601 stop-words was eliminated. (Although all words were all indexed and could be queried if so desired.) When applied to topics 51–100 this process produced queries containing an average of 70.4 terms and (after stemming) 42.4 distinct terms, each of which appeared on average in about 75,000 pages of TREC. Term multiplicity in the query was allowed to influence the ranking.

Index size Stemming was applied during inverted file construction, but no words were stopped. All alphanumeric strings were indexed, totalling 333,000,000 term occurrences of 538,244 distinct terms. The document index contained 136,000,000 $\langle d, f_{d,t} \rangle$ pointers, the paged index 196,000,000 such pairs. Each pair is compressed to occupy less than one byte. The variation in index size is due to the insertion of skips, or, in the case of the "All" row, their absence.

Both document and paged systems used the same compressed text, the size of which is listed in Table 1. In all cases the source data was the nine files contained on disks one and two, totalling 2,055 Mb.

Retrieval time The times listed in Table 2 and plotted in Figure 3 cover all activity from issue of query until a ranked list of 200 document numbers is calculated, but do not include the cost of retrieving and presenting answer documents. They are CPU processing times, and do not include delays due to disk seeks and data transfer.

Elapsed times (including all disk activity) during the ranking process are generally about 1.6 seconds greater than CPU time, in the course of which an average of 42 disk accesses are made to the lexicon; 42 disk accesses made to the inverted file itself (which

for the document-level index fetched a total of 1.65 Mb of data, containing about two million compressed $\langle d, f_{d,t} \rangle$ pairs); and 275 accesses to the combined file of document weights and document addresses. All of these files except the inverted file are small, and likely to have been buffered into main memory, hence the small overhead.

Fetching of the 200 answers, which occupy about 230 Kb compressed, adds a uniform 1.9 CPU seconds per query including the cost of decompression. Decompressed, there is an average of approximately 830 Kb of output text per query. Elapsed time for the presentation of documents was about 5.0 seconds per query greater than this CPU time, caused by the need to perform 200 seeks into the file containing the compressed text. This file is too large for there to be any significant buffering effect.

Non-zero accumulators When skipping is employed (the first three rows in each section of the table), memory space during query processing is proportional to the number of non-zero accumulators. In these experiments a hash table was used, and an average of 14 bytes per accumulator required. For the "All" experiments an array of accumulators was used, occupying 2.8 Mb for document retrieval and 6.6 Mb for page retrieval. The corresponding $K = 10{,}000$ values are 176 Kb and 181 Kb.

An array of 6-bit approximate document lengths was used to guide the retrieval process [12], requiring 0.5 Mb for document retrieval and 1.2 Mb for page retrieval.

Terms processed The values for "Terms processed" indicate the average number of terms processed before processing switched from the first to the second phase of *continue*. For each query a whole number of inverted file entries were processed, and this is why the "non-zero accumulators" average is greater than the target value K. Not surprisingly, with page retrieval fewer terms were required, on average, to consume the available accumulators.

Precision at 200 This column lists the fraction of the top 200 documents retrieved that were judged to be relevant to the topic by the TREC information assessors. The relevance judgements are non-exhaustive, and for each topic only a small fraction of the document collection has been inspected. The assessing process was designed to ensure that the documents most likely to be relevant were inspected first, but the magnitude of the task prohibited a complete examination, and in our system all of the test topics return some documents that have not been judged. The range of precision values in this column shows the fraction of unjudged documents that were accessed by our implementation of the cosine rule. The lower number of each pair assumes that all unjudged documents are irrelevant; the upper is calculated assuming that all are relevant. The true precision value lies somewhere between these two extremes.

Pessimal 11pt effectiveness These values are calculated based solely upon the top 200 ranked documents, assuming pessimistically that all remaining relevant documents are ranked last, and that all unjudged documents are not relevant. This was the methodology employed during the first round of the TREC experiment. In the second round, the top 1,000 documents were taken into account, and so 11pt averages were higher.

The decomposition of documents into pages provided an excellent stress test of our system. The number of text records indexed rose from 742,358 to 1,743,848, the average number of records containing at least one query term rose from 582,783 to 1,307,115, and index size grew by 50%. As a consequence the space required for accumulators and W_d values more than doubled and substantially longer inverted lists had to be processed to answer each query. For simple compressed indexes (without internal indexing) the CPU time taken to evaluate queries rose by 63%, from 7.66 seconds to 12.48 seconds. But with internally indexed compressed indexes and $L = 10{,}000$, the time taken rose by only 25%, from 4.46 seconds to 5.57 seconds. Based upon these experiments we conclude that:

- use of a limited number of accumulators does not appear to impact retrieval effectiveness, and so is an extremely attractive heuristic for ranking large collections because of the dramatic savings in retrieval-time memory usage that result;
- introduction of skips to the compressed inverted file entries significantly reduces processing time in this restricted-accumulators environment;
- index compression can, in this way, become "free": if only partial decoding is required, the input time saved by compression can be more than enough to pay the CPU cost of fractional decoding;
- the *quit* strategy is markedly inferior to *continue*, and all non-stopped terms should be allowed to contribute to the ranking; and
- even relatively simple pagination gives retrieval not measurably inferior to document retrieval.

When dealing with pages, the retrieval system was implemented to display the entire document, but highlight the page or pages that had caused the document to be selected. This decision made the paged collection particularly easy to use, and when we browsed the answers to the queries it was very satisfying to be able to find the exact passage that had triggered the match.

On the other hand, a problem that was brought out by these experiments was the difficulty of comparing retrieval effectiveness in the face of non-exhaustive relevance judgements. When precision rates are around 30%, and a further (in the $K = 1{,}000$ case) 30% of documents are unjudged, there can be little significance whatsoever attached to the difference between even 30% precision and 40% precision. Thus, the retrieval effectiveness figures of Table 2 are sufficiently imprecise that no conclusion can be drawn about the appropriate value of K that should be used. There is clearly scope for research into other methodologies for comparing retrieval mechanisms.

5 Partitioning documents

A particular challenge for information retrieval presented by TREC is the inclusion of long documents: the largest is over 2.5 Mb and contains 400,000 words. Documents such as this one are difficult from several aspects. Simply retrieving it is expensive, even compressed, and

buffering it in memory would be out of the question for a typical small machine. Presented as a monolithic block of text, a user would have great difficulty in discovering why it is relevant, nor would reading it be an option—it is almost as long as *War and Peace*. Finally, it is not at all clear that, in a mix with documents of only a few hundred words, the cosine measure and similar techniques treat long documents fairly, despite the use of normalisation. But if such documents cannot be retrieved—or, once retrieved, cannot be used—there seems little point in storing them at all.

We propose as a solution to all of these problems that longer documents be broken into parts. Parts of documents can be used in several ways. If they are of roughly even length, the problem of bias against longer documents is eliminated. Parts can be ranked individually, either as if they are whole documents or in the context of the document from which they are drawn. Once ranked, either the parts can be retrieved or the knowledge of their individual ranks can be used to construct a rank for whole documents. Finally, a part of a document can be presented to the user more easily than the whole, or if a whole document is presented then the highly ranked parts can be flagged as such, making the returned text easier to comprehend.

We considered two ways of partitioning documents: use of the embedded SGML markup to divide documents into logical sections; and breaking documents into parts of similar size on paragraph boundaries. Both mechanisms effectively replace the database of whole documents by a database of document parts. For the SGML-driven division of documents into logical parts, the following categories of document component were regarded as separate sections: ⟨purpose⟩, ⟨abstract⟩, ⟨start⟩, ⟨summary⟩, ⟨title⟩, ⟨supplementary⟩, and a general category ⟨misc⟩ that included all remaining categories. Sections were also indicated by the ⟨T2⟩ and ⟨T3⟩ tags.

For the paragraph-driven division of documents into similar-sized parts, the strategy used was to define a part to be a series of paragraphs and ensure that each part was at least some given minimum length. This is the "paged" decomposition assumed in the discussion of Sections 3 and 4, for which the given minimum was 1,000 bytes, resulting in an average page length of about 1,200 bytes. With this strategy, there is no implication that a part is a logical unit of text—it is simply a chunk of a document of manageable size.

With documents broken into parts by either mechanism, there are several ways in which parts could be used for ranking.

- Compute similarities for document parts then retrieve whole documents according to their highest-weighted part.

- Compute similarities for document parts then retrieve whole documents according to a rank given by combining the ranks for the parts. This combination might be a sum, an average, or could take into account the type of the part—whether it was an introduction, for example.

The above methods return whole documents. In some cases it would be preferable to return document parts; again, there are several possible approaches.

- Rank and return document parts, ignoring the fact that the parts are drawn from documents.

- Rank whole documents (using one of the above methods), then rank the parts within the documents.

The full TREC collection is not well suited to testing the latter two possibilities because there are no relevance judgements for the document parts. Moreover, the vast majority of documents in TREC are short, and use of the full collection would obscure the changes in performance that we might expect to result from document partitioning. To circumvent these problems we considered only the FR subcollection of TREC, which contains most of the long documents as well as a large number of short documents. The average number of words in the FR documents was 1,530, compared to 449 words per document for the full TREC collection, and altogether FR contains about 470 Mb of text. Having restricted the experiment to FR, there were 21 queries (of queries 51–100) for which there was at least one relevant document; and a total of 502 "relevant" judgements distributed over the 21 queries.

We then carried out a series of experiments to test the viability of section-based and paged-based retrieval.

Experiment 1: Rank each document using the cosine measure $C_{q,d}$. This provides a baseline against which part-based retrieval can be judged.

Experiment 2: Measure the similarity of each section using the section-based cosine measure

$$C_{q,s} = \frac{\sum_t w_{q,t} \times w_{s,t}}{W_s}$$

where $w_{s,t}$ is the weight of term t in section s and W_s is the weight of s. Then rank documents by the similarity of their highest-ranked section.

Experiment 3: Measure the similarity of each page using the page-based cosine measure:

$$C_{q,p} = \frac{\sum_t w_{q,t} \times w_{p,t}}{W_p}$$

where $w_{p,t}$ is the weight of term t in page p and W_p is the weight of p. Then rank documents by the similarity of their highest-ranked page. We ran this experiment for several minimum page sizes, ranging from 250 bytes to 4,000 bytes.

Results are shown in Table 3. The second column shows the number of indexed fragments; while the subsequent columns show the precision at the indicated volume of answers. Because the number of known relevant documents for these queries is as little as one, it is impossible to achieve 100% precision even at $k = 5$; and the final row of Table 3 shows the maximum average precision value that could be achieved for each value of k. The entries in the table should be considered relative to this maximum.

As can be seen, the naive strategy of breaking documents into parts of similar length is substantially better than either breaking the documents into an equivalent number of parts according to their SGML markup or applying the ranking to the documents directly. We believe that this difference is because the document-level database has too many long fragments, which the cosine measure handles poorly; while the section-based partition creates a large number of tiny fragments, many of only a few words, for which only one occurrence of

	Experiment	# Parts	Precision at k documents for $k =$					
			5	10	20	30	50	200
1	Documents	45,820	0.171	0.152	0.105	0.090	0.088	0.050
2	Sections	340,288	0.095	0.110	0.081	0.070	0.062	0.036
3	Pages (250)	959,222	0.190	0.157	0.131	0.121	0.105	0.053
	Pages (500)	621,724	0.219	0.186	0.157	0.140	0.116	0.055
	Pages (1,000)	367,896	0.267	0.200	0.155	0.144	0.133	0.057
	Pages (2,000)	206,224	0.238	0.190	0.164	0.148	0.130	0.061
	Pages (4,000)	118,697	0.248	0.195	0.145	0.127	0.108	0.059
Maximum			0.714	0.581	0.486	0.432	0.354	0.120

Table 3: Document-based vs. page-based vs. section-based retrieval (precision at fixed number of documents returned)

one relevant term can give the fragment a high rank—even though, statistically speaking, such occurrences should not be regarded as a good indicator of relevance. In contrast, the paging strategy yields parts of similar, reasonable length, for which absurd rankings are much less likely.

It should be noted that the relevance judgements were, to a certain extent, demand driven. There are relatively few judgements—either "yes" or "no"—against the FR collection, which implies that documents from FR were only sparingly retrieved in the formal experiments undertaken by the various research groups participating in the TREC project. Because of this, the majority of the documents retrieved in these experiments were unjudged. Nevertheless it is quite clear that, of the known answers within FR, the page-based indexing method is returning them earlier.

For the complete TREC collection, although there was no marked change in retrieval performance given by using paragraph-based parts (Table 2), there was a promising indicator. For queries 101–150, ranking and retrieval of whole documents fetched 129 documents from FR, of which only 16 (or 12.4%) were relevant. However, ranking of pages (of nominal size 1,000 bytes) and retrieval of whole documents based on their highest-ranked page fetched 250 documents from FR, of which 67 (or 26.8%) were relevant. We interpret this as indicating that retrieval based on parts—in this case pages—at least partially removes the bias against long documents, allowing them to be judged more fairly. The lack of significant change overall in Table 2 is probably because only a small proportion of the TREC documents are long enough to break into parts, and many of those that were partitioned have not been judged and are de facto assumed to be not relevant.

In an attempt to provide more accurate section-based ranking, Wilkinson has described methods based on combining the weights of individual sections to yield weights for complete documents [14], the second alternative outlined earlier. Wilkinson has also described strategies for retrieving individual sections based both on their characteristics and the characteristics of the documents from which they are drawn. These methods may be worthwhile as a post-retrieval filter for long documents, as they can be used to highlight relevant parts and guide the user through the document. In the presence of long documents, ranking doc-

uments and simply presenting them to the user is not satisfactory. It is necessary to rank the sections within documents to remove uninteresting material, or to have ranked the parts in the first instance so that relevant passages can be highlighted as they are presented.

6 Conclusions

By making available data and relevance judgements on a scale previously unthinkable, the TREC project has provided a unique and stimulating challenge. To date our efforts in meeting this challenge have been primarily algorithmic rather than in the information retrieval area. We have investigated text compression methods for both static and dynamic document collections that significantly reduce the space required by the stored text of such databases; we have developed novel index compression methods that allow economical indexing of every word and number in this data; we have designed index construction algorithms that allow creation of the inverted index in practical amounts of memory, disk space, and processor time; and we have examined the space and time requirements of the ranking process when applied to a large collection of documents. All of these methods have been implemented and tested against the 2 Gb of data distributed as part of the TREC project.

As a result of this work, we can conclude that it is both possible and practical to manipulate large volumes of textual data on a moderately configured workstation. Large machines are not necessarily required to support large databases.

Our experiments with division of documents into parts that can be individually indexed have shown that improved retrieval performance can be achieved on collections with highly disparate document lengths. Moreover, this division has not presented any difficulties algorithmically, and queries can be evaluated in only a little more time than is required for a database of whole documents. On the other hand a similar SGML-based decomposition into parts was not especially helpful—the results were worse than for document retrieval. In the context of a database containing long documents, some treatment of document parts is essential, as the elimination of irrelevant material from within documents can greatly improve the quality of information presented to the user; we propose pagination as one suitable technique.

7 Acknowledgements

We would like to thank Neil Sharman, Daniel Lam, and Lachlan Andrew for their work on the implementation. We also thank Donna Harman for her vision and effort in coordinating the TREC project and for welcoming us as participants. This work was supported by Melbourne's Collaborative Information Technology Research Institute and Centre for Intelligent Decision Systems, by the Australian Research Council, and by the TREC project itself.

References

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

[2] 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.

[3] C. Buckley and A.F. Lewit. Optimization of inverted vector searches. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 97–110, Montreal, Canada, June 1985. ACM Press, New York.

[4] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

[5] R.G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24(6):668–674, November 1978.

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

[7] D.K. Harman. Ranking algorithms. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 14, pages 363–392. Prentice-Hall, 1992.

[8] D.K. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

[9] D. Lucarella. A document retrieval system based upon nearest neighbour searching. *Journal of Information Science*, 14:25–33, 1988.

[10] A. Moffat, N.B. Sharman, and J. Zobel. Static compression for dynamic texts. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 126–135, Snowbird, Utah, March 1994. IEEE Computer Society Press, Los Alamitos, California.

[11] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*. To appear.

[12] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Information Processing & Management*. To appear.

[13] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Massachusetts, 1989.

[14] R. Wilkinson. Effective retrieval of structured documents. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, Dublin, Ireland, 1994. To appear.

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