Memory Efficient Ranking

Alistair Moffat
Department of Computer Science,
The University of Melbourne,
Parkville, Victoria 3052,
Australia

Justin Zobel
Department of Computer Science,
Royal Melbourne Institute of Technology,
GPO Box 2476V, Melbourne, Victoria 3001,
Australia

Ron Sacks-Davis
Collaborative Information Technology Research Institute,
723 Swanston St., Carlton, Victoria 3053,
Australia

June 27, 2002

Correspondence should be addressed to the first author.
Electronic mail: alistair@cs.mu.oz.au.
Telephone: +61 3 3445230. Facsimile: +61 3 3481184.

Abstract

Fast and effective ranking of a collection of documents with respect to a query requires several structures, including a vocabulary, inverted file entries, arrays of term weights and document lengths, an array of partial similarity accumulators, and address tables for inverted file entries and documents. Of all of these structures, the array of document lengths and the array of accumulators are the components accessed most frequently in a ranked query, and it is crucial to acceptable performance that they be held in main memory. Here we describe an approximate ranking process that makes use of a compact array of in-memory low precision approximations for the lengths. Combined with another simple rule for reducing the memory required by the partial similarity accumulators, the approximation heuristic allows the ranking of large document collections using less than one byte of memory per document, an eight-fold reduction compared with the space required by conventional techniques. Moreover, in our experiments retrieval effectiveness was unaffected by the use of these heuristics.

1 Introduction

Ranking techniques are used to find the documents in a document collection that are most likely to be the answers to an informally-phrased query [15, 17]. When the documents are stored in a database that is indexed by an inverted file [4, 8, 19], several structures must be used if ranked query evaluation is to be acceptably fast. These structures include the vocabulary of the document collection; the inverted file entries that record, for each term, which documents contain that term; the word weights; the document lengths; and address tables into the inverted file and the text itself. It is also usual for a set of accumulators to be required, normally containing one accumulator for each document in the collection.

In this paper we briefly survey methods that can be used to reduce the memory space required by the various search and index structures, then focus on the cost of storing the document lengths. We show that using a relatively small number of bits to represent lengths does not, up to a point, significantly affect ranking effectiveness, so long as an appropriate compaction technique is chosen. We discuss two techniques, one model-based and one frequency-based, and demonstrate their relative effectiveness on test databases. These techniques make only a small difference to retrieval speed but can reduce the memory space required for document lengths by a factor of between four and eight. We also describe an alternative use of the compacted document lengths, in which they are employed to restrict the number of disk accesses required by a full and 'exact' ranking of the collection.

The accumulators, storing partial similarity values for the documents of the collection as the ranking is carried out, are the second major demand on random access memory, and we also consider methods by which the memory allocated to these can be reduced. By effectively switching from an 'or' query to an 'and' query at some predetermined point, the space required by accumulators can be cut to just one or two bits per document. In our experiments this strategy resulted in a negligible deterioration in retrieval effectiveness even when the number of permitted non-zero accumulators was just a small percentage of the number of documents.

Together, these mechanisms allow ranking for even large document collections to be carried out in limited memory space. For example, using the techniques we describe, it would be possible for ranking on a collection of one million documents to be carried out within one megabyte of main memory. That is, we might consider a situation in which large document collections are distributed on CD-ROM and accessed via informal queries on low-powered personal computers.

In the description of our techniques we assume that the database is static, which allows parameters such as the minimum and maximum document length—required before compaction takes place—to be determined. For this reason our results will be most directly applicable to situations involving read-only media. However, we also believe that the methods we describe will readily generalise to dynamic collections.

The ranking technique we use to evaluate our techniques is the cosine measure, introduced in Section 2. This is just one of the possible mechanisms that can be used to perform ranking, and there are many others—see, for example, Salton [15] for a description of alternatives. The cosine measure suits our purposes because, if anything, it is one of the more demanding similarity measures, in that the similarity value assigned to each document depends not just upon that document, but also upon all of the other documents in the collection. Certainly, our techniques will in full or in part also be applicable to other similarity measures.

The structures used by document databases and the query evaluation technique are described in Section 3, and in Sections 4 and 5 we show how the array of document lengths can be reduced in size, while still retaining either fast performance or exact ranking. Section 6 then discusses methods by which the space required for partial similarity measures can be correspondingly restricted. Finally, in Section 7 we briefly discuss how dynamic databases might be handled.

2 The cosine measure

The ranking technique we will use in this paper is the cosine measure [15, 17], one of the most effective ranking techniques [1]. The cosine measure evaluates the relevance of a document to a query via the function

$$cosine(q,d) = \frac{\sum_t w_{q,t} \cdot w_{d,t}}{\sqrt{\sum_t w_{q,t}^2} \cdot \sqrt{\sum_t w_{d,t}^2}} \ ,$$

where q is the query, d is the document, and $w_{x,t}$ is the weight of word t in document or query x. The expression $\sqrt{\sum_t w_{x,t}^2}$ is a measure of the total weight or *length* of document or query x in terms of the weight and number of words in x. We will refer to the length of document d as W_d, and the length of the query as W_q.

One commonly used function for assigning weights to words in document or query x is the frequency-modified inverse document frequency, described by

$$w_{x,t} = f_{x,t} \cdot \log_e(N/f_t) \ ,$$

where $f_{x,t}$ is the number of occurrences of word t in x, N is the number of documents in the collection, and f_t is the number of documents containing t. This is commonly called the $tf \cdot idf$ weighting. Note that this function includes both global information—the word weight $\log_e(N/f_t)$—and local information—the 'within document' term frequency $f_{x,t}$.

Ranking techniques are tested by applying them to standard databases and query sets, in which the queries have been manually compared to the documents to determine relevance. The databases we use in this paper are *Cacm*, abstracts from Communications of the ACM; *Cisi*, abstracts of library science articles; *Time*, articles from Time magazine; and, most importantly, *TREC*, a large collection of articles on finance, science, and technology, from several sources. The parameters of these collections are shown in Table 1. Before deriving the figures given in this table, the words in the collections were stemmed—that is, all letters folded to lowercase and variant endings removed—using Lovin's algorithm [10]. The queries were preprocessed to remove a small set of common stop-words, and were similarly case-folded and stemmed.

	Cacm	*Cisi*	*Time*	*TREC*
Number of records	3,204	1,460	425	742,985
Average words per record	33	72	349	474
Distinct words in collection	5,510	6,278	12,750	538,497
Number of queries	64	112	83	47
Retrieval effectiveness (%)	28.5	26.0	62.8	17.4

Table 1: Parameters of document collections

Throughout this paper, for *Cacm*, *Cisi*, and *Time* the retrieval effectiveness is measured using an eleven point recall-precision average based upon a total ordering of the entire collection. For *TREC*, the recall-precision average was also calculated as an eleven point average, but, because of the size of the collection, was based only upon the top 200 retrieved documents rather than a total ranking. This was the methodology suggested during the TREC experiment, and we have chosen to continue with this convention. Precision was taken to be zero at recall values that would have required that more than 200 documents be accessed, and so the retrieval effectiveness cited can be regarded as maximally pessimal. For example, if more documents are retrieved the efficiency increases, since more non-zero ‘recall’ points are brought into the average. With 1,000 documents retrieved for each *TREC* query, the effectiveness increased to 24.6%; if only the top 100 are accessed, the effectiveness drops to 13.6%.

3 Document databases

In an inverted file document database, each distinct word in the database is held in a *vocabulary* [4, 8, 11, 19]. The vocabulary might be an array, or a search structure such as a B-tree that can be efficiently updated. The vocabulary entry for each word contains an address pointer to an *index entry*, a list of the documents containing the word. To support the cosine measure, the vocabulary should also hold, for each word t, either f_t, the number of documents that contain t, or, equivalently, the value $\log(N/f_t)$. Knowledge of the value f_t also allows the useful heuristic of ‘process the shortest inverted file entry first’ to be exploited. This is particularly important for boolean queries—that is, queries for which the answers are exactly those documents that contain all of the terms specified in the query—as it allows the space and time needed to find the intersection of the inverted file entries to be minimised. For ranked queries, where the list operation is effectively an ‘or’ rather than an ‘and’, knowledge of the number of items in the inverted file entries is also useful, and allows several time-saving heuristics to be employed [4, 11].

The index entry for each word stores a list of the documents that contain that word, and, for ranked queries, the ‘within document frequency’ $f_{d,t}$ must be held with each document number. This allows the weight of each word in each document to be computed. The storage of these ‘document number, within document frequency’ pairs is a major component of the overall cost of storing the information retrieval system, and index compression has long been exploited to reduce the need for secondary disk space [3]. In the absence of compression, and with a naive implementation, we might choose to expend a four byte integer to store each of the two values, or a total of 64 bits for each term-document pair. Using parameterised compression techniques such as those described by Moffat and Zobel [12, 13], it is possible for the space required to be reduced to under eight bits per pointer. On the two gigabyte *TREC* collection these techniques compress the inverted file from 1000 megabytes to 135 megabytes, a dramatic saving. For this reason, if the information retrieval system is to be available on CD-ROM, and if we wish to maximise the amount of information stored on each disk, we should employ compression of both the index and also the stored text [2, 3, 9]. This is the environment that we consider here.

The vocabulary, the f_t values, and the inverted file address table can be stored either on disk or in memory. If they are stored in memory, a typical query of 25 terms will require that 25 inverted file entries be accessed from disk before the query can be processed. If, to conserve memory space, the vocabulary and associated information are merged on disk with the inverted file entries and some form of hashing (such as extensible hashing [14]) is used, the expected number of disk accesses can kept to about 1.2 on average per query term, but at the cost of a 20%–30% expansion in the size of the inverted file. A more practical solution is to allow two accesses per query—the first into the index file containing the vocabulary and term information, including the inverted file entry address, and the second to actually retrieve the inverted file entry. In this case the 20% space wastage caused by hashing is restricted to the vocabulary alone, which is normally an order of magnitude smaller than the inverted file. Fox et al. have considered the use of minimal perfect hashing [6, 7]; this allows the vocabulary to remain compact (indeed, the words need not be stored at all), but a non-trivial amount of memory space is required for the description of the hash function, approximately 2–4 bits per key, and two disk accesses will still be required per query term if the address table is stored on disk.

The document address table can also be stored either on disk or in memory. In a typical application perhaps the top 25 ranked documents would be displayed in response to a query; and even if the address table is held on disk, display of answers will require about 50 disk operations, assuming that the top 25 documents can be identified without access to their text.

To make this whole inverted file based ranking process possible, the final data structure required is an array storing the length of each document—that is, the vector of values W_d. These document lengths are query invariant, and for efficiency should be computed at database creation time.

There are several methods for determining which of the documents in a collection will have a high cosine measure with respect to a query. In small databases, it is feasible to compute *cosine* for each document and so form a total ranking. In this case the document lengths can be stored on disk with the corresponding documents. However, for most databases this approach yields unacceptable response times because of the volume of text to be processed, and so techniques have been developed to reduce the set of candidate documents examined. One effective technique of this kind is to inspect the vocabulary to determine which words in the query are the most important, construct a boolean query out of these words, and rank the documents retrieved by this query [16].

Another method for determining the ranking is to directly use the inverted file structure and document lengths [4, 8, 11], as we choose to do. In this method, the index entry for each word in the query is retrieved in turn. An accumulator variable is created for each document containing any of these words, in which the result of the expression $\sum_t w_{q,t} \cdot w_{d,t}$ is accrued. In some cases, it is not necessary to access the index entries for words of low weight because they are unlikely to affect the ranking, and so a possible heuristic is to access the entries in decreasing order of weight [18], as is done to minimise processing time for boolean queries. Buckley and Lewit [4] and Lucarella [11] have further explored this possibility.

Once all of the index entries have been inspected, for each document with a non-zero accumulator value the document length W_d is accessed and the final ranking is formed. A further heuristic that can be applied at this stage is to access the document lengths in the order given by their $\sum_t (w_{q,t} \cdot w_{d,t})$ values for the query under consideration, since when this value is small it is unlikely that the document will be ranked highly enough to be judged worthy of presentation to the user. We will expand upon this heuristic below.

The effect of applying the document lengths is to reorder the ranking, sometimes significantly. Because of this reordering, to determine the final ranking and identify the top (say) 25 documents, the lengths of 200 or more documents might need to be consulted. Certainly, the number of accesses to the document lengths will usually exceed, often by an order of magnitude, the number of answers; and if, because of pressure of memory space, the document lengths are stored on disk, these accesses could become the dominant cost of answering queries.

One method that has been suggested for avoiding this bottleneck is to store, in the inverted file entries, not the raw $f_{d,t}$ values described above, but instead scaled values $f_{d,t}/W_d$ [4, 11]. Then, as the partial similarities are built up in the accumulators, they will be 'self-normalising', and no explicit access to the document lengths will be required.

Unfortunately, this approach cannot be reconciled with the need to compress the inverted file entries. Consider a single 'document number, within document frequency' pair from an inverted file entry. After compression of each such entry the $f_{d,t}$ component is typically represented in 1–2 bits [13], and storing a normalised value in the same restricted space is simply impossible. Indeed, in normalising every $f_{d,t}$ value, the length W_d of document d is effectively being stored once for every term that appears in the document, and, from a compression perspective, the inverted file *must* expand, since more information is being represented. Hence, this 'self-normalising' technique will only work if the pairs comprising the inverted file are to be stored as four byte quantities. If we are interested in minimising the disk space required by the inverted file, we must seek an alternative solution.

To further illustrate this argument, let us start with a hypothetical system that retains in main memory all data structures except the compressed inverted file entries and the indexed documents themselves. In this system a query of 25 terms and 25 displayed answers would require a total of 50 disk accesses. If, to conserve main memory, the vocabulary and inverted file address table are moved to disk, an additional 25 disk operations might be required. If the document address table also migrates to disk, 25 more disk accesses will be necessary, and the total might reach 100, a roughly twofold increase in response time over the original system. But if the array W_d of document lengths—which must be accessed because the inverted file is compressed—is moved to disk, query response time might degrade by a further factor of four or more. For this reason it is important that the document lengths, or, as we shall describe, some approximation of them, be held in memory, even if there is insufficient space for any of the other access structures.

4 Approximating document lengths

Since ranking is an approximate process it is reasonable to suppose that computation of *cosine* with approximate numbers will be sufficiently accurate. One obvious method to reduce the space required for the document lengths is to simply use fewer bits to represent each number. However, if the greatest advantage is to be taken of the few bits available for each number, it is not sufficient to simply take the most significant digits; rather, a mapping is needed that reflects the distribution of the numbers being represented. In this section we outline two mappings, one model-based, and one based on the frequency of each of the values to be represented. Both mappings generate b-bit approximate values, where b is fixed in advance. It would also be possible to employ some variable width encoding, such as a Huffman code, but for fast access a fixed width code is preferable, and we have not explored this further possibility.

4.1 Model-based mapping

One technique for coding a b-bit approximation of a set of numbers is as follows. We assume that each number x that we wish to represent is such that $L \leq x < U$, for some strictly positive lower bound L and upper bound U. Since U must be strictly larger than the largest value to be represented, we would normally take U to be the largest length plus ϵ, for some small value ϵ.

For a base $B = (U/L)^{2^{-b}}$, chosen so that $\log_B(U/L) = 2^b$, the value $f(x) = \lfloor \log_B(x/L) \rfloor$ will then be integral in the range $0 \leq f(x) < 2^b$ and will require only b bits as a binary code. The use of logarithms in this approximation allows a wide range of magnitudes to be represented while still being reasonably accurate for small values of x.

If x is represented by code c, that is, $f(x) = c$, an approximation $\hat{x}$ to x can be computed as $\hat{x} = g(c + 0.5)$, where g is the inverse function $g(c) = L \times B^c$ and 0.5 is added because truncation rather than rounding was used to compute $f(x)$. In effect, each code value c corresponds to a range of values x that are all assigned the same approximation, that is, the x values in the range $g(c) \leq x < g(c + 1)$. In fact, at the implementation level, there is no need to add the 0.5, since the rank order will be the same if we take $\hat{x} = g(c)$. The calculation of $g(c)$ would then normally be implemented by table look-up.

Let us calculate some example codes. Suppose for some collection that the smallest document length is 20.47, that the largest document length is 347.12, and that we wish to use 3-bit approximate lengths, that is, we desire $b = 3$. Taking $\epsilon = 0.01$, we then have lower bound $L = 20.47$, upper bound $U = 347.13$, and base $B = (347.13/20.47)^{0.125} = 1.4245$. Using this code the value $x = 87.14$ would be coded as $\lfloor \log_B(87.14/L) \rfloor = \lfloor 4.09 \rfloor = 4$, and, when decoded, would be approximated by $\hat{x} = L \times B^{4+0.5} = 100.61$. Table 2 shows the complete code assignment that would be created by this choice of parameters.

We tested recall effectiveness when the document lengths were represented using this scheme, taking L to be the smallest length observed in each collection and U to be a little greater than the largest length. The retrieval efficiency and average degradation in retrieval efficiency caused by the use of approximate document lengths are shown in Table 3.

Compaction of the document lengths has the expected effect: for very small values of b the effectiveness is worse than if the document lengths were simply

Code c	Code value	Corresponding range for x	$\hat{x} = g(c + 0.5)$
000	0	[20.47, 29.16)	24.43
001	1	[29.16, 41.54)	34.80
010	2	[41.54, 59.17)	49.58
011	3	[59.17, 84.30)	70.63
100	4	[84.30, 120.08)	100.61
101	5	[120.08, 171.06)	143.32
110	6	[171.06, 243.68)	204.17
111	7	[243.68, 347.13)	290.84

Table 2: Example code assignments with $L = 20.47$, $U = 347.13$, and $b = 3$

b	*Cacm*	*Cisi*	*Time*	*TREC*	Degradation (%)
exact	28.5	26.0	62.8	17.4	0.0
8	28.5	26.0	62.9	17.5	0.0
6	28.3	26.0	64.1	17.7	−0.8
4	27.8	26.0	64.6	16.0	+1.9
3	27.8	25.9	65.4	14.5	+3.8
2	24.6	25.2	62.4	7.9	+18.0
0	28.7	23.4	53.5	1.0	+29.6

Table 3: Retrieval effectiveness with reduced document length precision (%)

ignored. For example, with $b = 2$ the documents are being grouped into just four categories, 'very short', 'short', 'long', and 'very long', and the calculation of $\hat{x}$ means that the ranking process will strongly favour 'short' documents over 'long', even when a 'long' document has a much larger value for the accumulator and is only slightly longer in reality. This result was expected. More surprising was the small improvement in retrieval effectiveness—that is, negative degradation—that was obtained for $b = 8$, $b = 6$, and, in one case, $b = 4$. While we would hesitate to claim this improvement as a 'feature' of the compaction technique, it gives us some confidence in claiming that 8-bit or 6-bit approximations to the document lengths are perfectly acceptable and appear to have no measurable effect on retrieval effectiveness. For *TREC*, changing from 32-bit to 8-bit lengths for the W_d array released over two megabytes of main memory, and, as is shown in Table 3, there seems little reason to use a larger value of b.

It is interesting to note that with the small collections, use of no scaling at all has little effect on retrieval effectiveness, but for *TREC* the use of $b = 0$ gives both poor retrieval effectiveness and slow execution. The poor retrieval effectiveness is caused by the very large variation in document lengths in *TREC*—the shortest document, just a few hundred bytes, had a length of 7.6; while the longest document was over 2.5 megabytes, and had a length of 64,000. Given this large range, and the 'no scaling by document length' implied by $b = 0$, performing a ranked query is tantamount to retrieving all of the longest documents, since the common words in long documents will outweigh the information-bearing words in short documents by sheer weight of numbers. On the other hand, scaling by the document length, as required by the exact cosine method, appeared to discriminate against the long documents, and they were never retrieved in the course of normal processing. Based upon these observations, one possibility for increased retrieval effectiveness might be to use a weaker normalisation process, dividing perhaps by the square root of the length; or to produce a greater number of more uniform length documents by indexing at a paragraph level, and retrieving documents in which any of the paragraphs are judged to be strongly relevant.

The slow execution with $b = 0$ arises because, with no normalisation at all, each query resulted in between 80 and 200 megabytes being extracted from the database. Moreover, our system compresses the text of the collection as well as the index, and all of this data needed to be decompressed. As a result, the processing run with $b = 0$ took more than 24 hours, compared with about 25 minutes for the large values of b.

4.2 Frequency-based mapping

The compaction technique described in Section 4.1 provides a mapping from a bag of numbers (that is, a set with duplicates) onto integers between 0 and 2^{b-1}. The mapping was defined by a function f, which broke the input bag into 2^b groups; it could, however, have been defined by a sorted array of length $2^b - 1$ specifying the boundaries between the groups. So long as the boundaries are such that the groups of numbers are of roughly equal size, one might expect that the resultant encoding would be a good approximation to the actual values.

A pragmatic way of computing the mapping is to order the n values; find the numbers occurring at $n/2^b$, $2 \cdot n/2^b$, $3 \cdot n/2^b$, $\ldots$, $(2^b - 1) \cdot n/2^b$; and make these numbers the boundaries of the 2^b codewords. To ensure that no two boundaries are identical, and that no codewords are unused, it is necessary to search up or down from $k \cdot n/2^b$ until adjacent numbers have different values. For example, if the values in the array

2	9	9	12	12	12	32	36

are to be represented with a 2-bit code, the mapping might be ≤ 2, ≤ 9, ≤ 12, and ≤ 36 for codes 0, 1, 2, and 3 respectively.

Intuition suggests that the mapping derived by this pragmatic technique should lead to better performance than the mapping defined by f, since it will be a closer fit to the skew of the distribution of numbers. However, in practice this method performed poorly on our test collections. Table 4 shows, for the three small collections, the retrieval effectiveness achieved when document lengths are approximated by this technique. These results were too poor to justify our undertaking the substantial programming effort required to test this technique on *TREC*, which was responsible for most of the decline in performance in Table 3.

b	*Cacm*	*Cisi*	*Time*	Degradation (%)
exact	28.5	26.0	62.8	0.0
8	28.5	25.9	62.7	+0.2
6	27.4	25.9	62.9	+1.4
4	27.1	25.3	61.7	+3.1
3	24.9	24.9	61.7	+6.2
2	21.9	23.0	59.9	+13.1
0	28.7	23.4	53.5	+8.0

Table 4: Effectiveness of frequency-based compaction (%)

The reason for the poorer performance of the frequency-based mapping is interesting. We had expected that an approximate code based upon the actual values to be coded would, de facto, result in better approximations, and that ranking effectiveness would improve. However, this was not the case. With the model-based technique the ratio $x/\hat{x}$ is bounded below by $1/\sqrt{B}$ and above by $\sqrt{B}$. For the frequency-based approximation the average error can also be bounded, but there is no such bound for individual values of x. Experiments showed the distribution of $\log x$ values for these document collections to be bell-shaped, and the frequency-based code at its worst, that is, with $x/\hat{x}$ furthest from unity, for x either very small or very large. These two situations are exactly the cases when the document length is crucial to the ranking process.

On the other hand, at the extremes of the range of x the model-based code was creating buckets with very few x values, but still with reasonably accurate approximations $\hat{x}$.

Given that the model-based approach also requires no storage of decoding tables, it is the method of choice for this application.

5 Exact ranking

In the previous section we described a technique for approximating document lengths in terms of saving memory space at the price of possible degradation of retrieval effectiveness. We can also use the approximate document lengths as a guide to document retrieval when 'exact' ranking must be performed, at the cost of slightly increased query time.

Suppose that we maintain b-bit model-based approximate lengths in memory, but also store the full 32-bit length as a field in the document address table on disk. The approximate length gives a guide as to the final value of the cosine measure, since if the coded length for some document is c, the actual length x is such that $g(c) \leq x$. Hence, once all the inverted file entries have been scanned and the values of the query dependent components of $\mathit{cosine}(q, d)$ accumulated, it is possible to use the approximate length to calculate, for each document d, an upper bound on the value of $\mathit{cosine}(q, d)$.

To generate an exact ranking of the first k documents we must be sure, for each document that is not considered, that document's value of *cosine* is less than the kth largest value of *cosine*. This can be achieved by the following method. Suppose that x_i is the length of document i; that c_i is the b-bit code assigned to document i; that $g(c)$ is the inverse function described in Section 4.1; and that A_i is the value of the accumulator for document i,

$$A_i = \frac{\sum_t (w_{q,t} \cdot w_{i,t})}{W_q},$$

where W_q is, as previously defined, the length of the query. The assumption we make is that c_i and A_i are available without consulting the set of exact document lengths x_i, but that the x_i values are available on disk if required.

First, for every document i with a non-zero accumulator A_i we calculate an upper bound $\hat{C}_i$ on the value of $\mathit{cosine}(q, i)$ using

$$\hat{C}_i = \frac{A_i}{g(c_i)}.$$

Next, we order this list of $\hat{C}_i$ values from largest to smallest. The k documents required might be amongst the first k on this list, and so we access the exact lengths x_i for all of the first k documents on this ordered list and calculate the exact values C_i of $cosine(q, i)$ for these documents.

We must now check any other documents for which $\hat{C}_i$ is larger than the smallest of the k exact values. We do this by accessing the exact value x_i for such documents, still in decreasing $\hat{C}_i$ order, and, after each access, calculating the exact value C_i and updating the list of the k highest values of C established so far. As we do this, the kth smallest cosine value will be non-decreasing, while the $\hat{C}_i$ values are non-increasing, and so we can expect, after some small number of values x_i have been retrieved, that it will be possible to say that the kth smallest of the known C_i values is larger than $\hat{C}_i$ for all remaining documents, and that no further exact lengths need to be fetched.

Note that we do not in fact require a total ordering on the $\hat{C}_i$ values, only that the top k of the $\hat{C}_i$ be initially identified, and, thereafter, that the 'next largest' be repeatedly available. The natural structure to support this sequence of queries is the heap [5], requiring $O(N + k' \log N)$ time to find the k' largest of N items. A total ordering would require $O(N \log N)$ time.

Finally, the actual documents are required. If the exact lengths of documents are interleaved on disk with the document addresses, a further k disk operations will suffice to fetch the documents selected, since the addresses of the required records will already be available in main memory.

Table 5 shows the number of exact document lengths that were required to guarantee exact ranking, for various combinations of k and b, averaged over the queries comprising each of our test collections. For example, with $b = 4$—that is, assuming that 4-bit approximate lengths using the model-based code were available in main memory—and $k = 25$, an average of 29.2 lengths were required before the top 25 documents could be unambiguously identified for the *Cacm* queries. Coupled with the 25 disk accesses required to actually fetch the documents, we have the choice of either spending 50 disk accesses to perform an approximate ranking of the type described in Section 4.1, or of spending (on average) 54.2 disk accesses to ensure that the ranking performance is not degraded by the use of the approximate code. In either case, just four bits per document of main memory is sufficient to give fast, effective ranking. Including the the 50 or so disk operations already required to access the inverted file for a 25 term query, the overhead cost of guaranteeing an exact ranking is perhaps as little as 5%.

The overheads on the *TREC* queries are, not surprisingly, much greater than for the three small collections, simply because *TREC* is more than an

k	b	*Cacm*	*Cisi*	*Time*	*TREC*
1	8	1.0	1.0	1.1	1.3
	6	1.0	1.1	1.2	3.9
	4	1.6	1.6	1.1	56.3
	3	3.1	2.6	2.6	390
	2	10.0	7.3	5.0	4,150
	0	320	210	56.2	64,100
5	8	4.3	5.2	5.2	6.0
	6	4.9	5.5	5.4	13.9
	4	7.1	7.1	6.7	112
	3	11.8	11.0	8.9	610
	2	29.0	23.8	14.5	6,300
	0	450	360	108	80,300
25	8	20.9	25.5	25.4	30.0
	6	22.4	26.9	26.4	48.5
	4	29.2	33.2	30.3	240
	3	40.4	44.4	37.5	1,090
	2	73.5	77.8	54.1	9,800
	0	580	580	200	108,000

Table 5: Number of document lengths required for exact ranking

order of magnitude larger. Even so, with $b = 8$, the top 25 documents out of nearly 750,000 can be determined with fewer than 30 disk accesses to retrieve lengths. Similarly, with $b = 8$ the top 200 documents can be determined using an average of 219 exact lengths, and the top 1,000 documents can be identified using an average of just 1077 exact lengths.

Note that with $b = 0$ a large number of lengths are required to effect the ranking, thereby justifying our claim of Section 3 that fast access to document lengths or some approximation thereof is crucial to efficient ranking. Note also that there are values in the table less than k, caused by the presence of queries in which there were fewer than k documents with non-zero accumulators.

6 Bounding the accumulator space

In the previous sections we described mechanisms by which the space required for document lengths can be reduced to four or six bits per document. Throughout that description we supposed that it was possible to store, for each document in the system, an accumulator variable in which to build up the cosine contributions. The simplest data structure we might use to support these accumulators is an array of (say) four byte floating point values, with each entry initialised to zero.

Alternatively, if we are prepared to assume that the number of non-zero accumulators is small compared to the number of documents stored, we might consider using a dynamic data structure such as a balanced search tree or a hash table [5]. In this case we require storage of the document number for each non-zero accumulator, so that the set can be searched, and also some pointers or other structural information. In total, we might consume sixteen to twenty bytes of memory for each non-zero accumulator.

For each of the *TREC* queries there were on average more than 600,000 non-zero accumulators, and in this case the array is the more economical structure. However, at four bytes per document, the array of accumulators is no less expensive than the array of document lengths—three megabytes for the *TREC* collection, for example—and methods whereby this space might be reduced are also of practical importance. The inverted file techniques of Buckley and Lewit [4]; Harman and Candela [8]; and Lucarella [11] similarly assume that it is possible to perform random access upon a set of accumulators.

To guarantee that the space required by the accumulators could not become large, we placed an *a priori* bound on the number that would be permitted to be non-zero, and used a dynamic search structure to store these accumulators and their corresponding document identifiers. Two strategies for processing the inverted file whilst observing this bound were then considered. The first possibility is to process inverted file entries in increasing order of f_t, and simply 'quit' processing when the number of accumulators exceeded the bound, going on at that point to the normalisation and ordering steps of the ranking algorithm. This has the advantage of providing a 'short-circuit' to the processing of inverted file entries and hence faster ranking, but at the possible expense of poor retrieval performance.

The second heuristic we tested continued the processing of inverted file entries, but allowed no new documents into the accumulator set. In this case, we effectively perform an 'or' query until the bound on the number of accumulators is reached, and then switch to an 'and' query, continuing to process inverted file entries but searching only for document numbers that already appear in the set of accumulators. Both heuristics generate the same set of candidate documents, but in a different permutation, and so when the top k documents are extracted from this set and returned, different retrieval effectiveness can be expected.

A third possibility would be to incorporate some replacement rule, so that, whilst observing the bound on accumulators, new documents might be allowed to replace poorly performing documents in the pool of candidates if it seemed likely that they would eventually have a higher cosine value. We are currently experimenting with several different variations on this third possibility. To date we have carried out preliminary experiments exploring the first two alternatives: quitting all processing at the bound; and continuing to add cosine contributions, but allowing no new accumulators.

Number of	11pt effectiveness (%)	
accumulators	Quitting	Continuing
742,985	17.4	17.4
74,298	13.5	17.8
7,429	9.5	17.7
742	7.9	14.7

Table 6: Retrieval effectiveness for *TREC* with a bounded number of non-zero accumulators, 200 answers, 47 queries

Table 6 shows the results of these experiments. The first row shows the retrieval effectiveness when every document is permitted an accumulator; the second when 10% of the documents are permitted accumulators; and so on. The 'quit' strategy is clearly inferior, and retrieval effectiveness drops quickly. On the other hand, the 'continue' strategy appears quite insensitive to the number of accumulators allowed. Since the two strategies extract the same set of candidates from the database, just in a different order, we conclude that the primary terms—those processed while the set of accumulators is growing—act only as a crude filter on the database to extract a set of documents for further consideration, and that it is the secondary terms, processed after the transition point is reached, that are responsible for the good behaviour of the cosine measure. For this reason, our results support the strategy proposed by Salton, Fox and Wu [16]. Even when as few as 1% of the documents are permitted accumulators, the 'continue' strategy gives good retrieval effectiveness.

Noting from Table 3 that approximate ranking with $b = 6$ also gave slightly improved retrieval effectiveness, we ran a further $b = 6$ experiment, allowing 74,298 accumulators, and found that effectiveness again slightly increased, to 18.0%. That is, the two heuristics we suggest appear to be independent of each other.

A dynamic search structure will require about 16 bytes—128 bits—for each candidate. If 1% of documents are permitted accumulators, then about 1.3 bits per document must be allowed. Including the 6 bits previously allocated for the approximate length, a little under one byte per document is sufficient to allow in-memory ranking, with no measurable degradation in retrieval effectiveness.

Buckley and Lewit [4] and Lucarella [11] describe techniques whereby, given that all accumulators can be held in memory, the number of inverted file entries processed can be minimised whilst still guaranteeing that the top k ranked documents are retrieved, even if not in exactly the 'correct' order. An interesting research problem is to consider whether the 'bounded accumulator' approach we describe here can be married with those heuristics to allow short-circuiting of the processing of inverted file entries. Our results here should, however, be considered also as a caution—the 'short circuit' heuristics attempt to stop processing inverted file entries as soon as the set of k documents has been identified, yet Table 6 clearly shows that for large values of k the ordering within this set of answers is important. One should not perform a retrieval of this kind to obtain the top k documents, and then only look at a few of them. Either all should be examined, or the retrieval repeated with a smaller k.

7 Dynamic databases

The techniques we have described in this paper are best suited to static document collections, for two main reasons. First, with the cosine measure, document insertion can change all word weights and hence all document lengths. Second, the model used to represent values in a small number of bits requires the range of values in advance.

These problems are, however, relatively easy to address. Although the cosine measure is poor with regard to update, other ranking measures do not have this limitation [4, 8], and moreover, as we have seen, only crude approximations to length are needed for effective ranking. It would probably be sufficient to recompute lengths each time the database doubled in size, requiring perhaps ten full passes over the data in the lifetime of the database. If a new document had a length outside the given range, the approximation to the length could simply be the minimum or the maximum in the current range. If the database already contains a sufficiently representative set of documents, the error introduced by this approximation would be slight.

Alternatively, the original code might be designed to allow for values in the range $L/2 \leq x < 2U$, where L and U are the bounds for the current set of documents; this would bring a small degradation in resolution compared with the more exact code, but would permit expansion of the collection without distortion of the code.

8 Conclusions

We have shown that the memory required for ranking of static document collections held on disk can be substantially reduced. This saving derives from the observations that very little precision is required to specify the numbers used for document ranking, and that the number of non-zero accumulators can be safely held at a small percentage of the number of documents. Our techniques are of particular importance when large, static, collections are being compressed for distribution on relatively slow read-only media such as CD-ROM. In these situations, when database access will be on a low-powered machine, it is of paramount importance that the text and index be compressed; that the number of disk accesses be kept low; and that only moderate demands be placed upon main memory.

We have presented two techniques for representing ranges of values in small numbers of bits: a model-based method that maps numbers into small integers, and a frequency-based method that groups numbers into a small number of equal-sized bags. Of these, the model-based method appears to be the more effective.

If 'exact' retrieval efficiency is to be maintained, the approximate lengths can be used to guide the ranking process, with accurate lengths kept on disk. In our experiments this expanded query processing time by at most about 20%, an acceptable price for the four- to eight-fold reduction in memory usage that results.

More generally, the exact ranking technique allows retrieval effectiveness to be held high, with the tradeoff between increased space or increased time controlled by the parameter b. Similarly, the approximate ranking technique allows time to be held low, with the choice of b controlling the tradeoff between increased space and reduced precision.

We have also described a simple rule that allows the memory required by the document accumulators—the partial similarities—to be bounded. Even when as little as two bits per document is available, it is possible to perform the ranking with no measurable degradation in retrieval effectiveness. When coupled with the use of six bit approximate document lengths, collections can be ranked using as little as one byte per document. Given that even personal computers are likely to have two or more megabytes of memory, collections of up to one million documents can be handled in this way.

Acknowledgements

We would like to thank James Thom for a number of helpful suggestions, and Neil Sharman for his assistance with the implementation. This work was supported by the Australian Research Council.

References

[1] S. Al-Hawamdeh and P. Willett. Comparison of index term weighting schemes for the ranking of paragraphs in full-text documents. *Int. J. of Inf. and Lib. Research*, pages 116–130, 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*. To appear.

[3] A. Bookstein, S.T. Klein, and D.A. Ziff. A systematic approach to compressing a full-text retrieval system. *Information Processing & Management*, 28(5), 1992.

[4] 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.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Massachusetts, 1990.

[6] E.A. Fox, Q.F. Chen, A.M. Daoud, and L.S. Heath. Order-preserving minimal hash functions and information retrieval. *ACM Transactions on Office Information Systems*, 9(3):281–308, 1991.

[7] E.A. Fox, L.S. Heath, Q. Chen, and A.M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, January 1992.

[8] D. 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] S.T. Klein, A. Bookstein, and S. Deerwester. Storing text retrieval systems on CD-ROM: Compression and encryption considerations. *ACM Transactions on Office Information Systems*, 7(3):230–245, January 1989.

[10] J.B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computation*, 11(1-2):22–31, 1968.

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

[12] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 72–81, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.

[13] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In N. Belkin, P. Ingwersen, and A.M. Pejtersen, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 274–285, Copenhagen, June 1992. ACM Press, New York.

[14] R. Sacks-Davis and K. Ramamohanarao. Recursive linear hashing. *ACM Transactions on Database Systems*, 9(3):369–391, 1984.

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

[16] G. Salton, E.A. Fox, and H. Wu. Extended Boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, 1983.

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

[18] A.F. Smeaton and C.J. van Rijsbergen. The nearest neighbour problem in information retrieval. *ACM SIGIR Forum*, 16:83–87, 1981.

[19] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In L.-Y. Yuan, editor, *Proc. International Conference on Very Large Databases*, pages 352–362, Vancouver, Canada, August 1992.