Filtered Document Retrieval with Frequency-Sorted Indexes[1]

Michael Persin

Department of Computer Science, RMIT,
723 Swanston St., Carlton 3053, Australia
email mp@kbs.citri.edu.au

Justin Zobel[2]

Department of Computer Science, RMIT,
GPO Box 2476V, Melbourne 3001, Australia
Phone +61 (3) 9282-2486; fax 9282-2490; email jz@cs.rmit.edu.au

Ron Sacks-Davis

Faculty of Applied Science, RMIT,
GPO Box 2476V, Melbourne 3001, Australia
Phone +61 (3) 9282-2420; fax 9282-2490; email rsd@cs.rmit.edu.au

July 1995

[1]Some of the material in this paper appeared in preliminary form in the 1994 ACM SIGIR Conference and the 1994 International Conference on Applications Of Databases.

This research was supported by the Key Centre for Knowledge-Based Systems, the Australian Research Council, and the Collaborative Information Technology Research Institute CITRI.

[2]Dr. Zobel is the contact author for all correspondence.

Abstract

Ranking techniques are effective at finding answers in document collections but can be expensive to evaluate. We propose an evaluation technique that uses early recognition of which documents are likely to be highly ranked to reduce costs; for our test data, queries are evaluated in 2% of the memory of the standard implementation without degradation in retrieval effectiveness. cpu time and disk traffic can also be dramatically reduced by designing inverted indexes explicitly to support the technique. The principle of the index design is that inverted lists are sorted by decreasing within-document frequency rather than by document number, and this method experimentally reduces cpu time and disk traffic to around one third of the original requirement. We also show that frequency sorting can lead to a net reduction in index size, regardless of whether the index is compressed.

1 Introduction

Ranking is used to retrieve documents from a database and present them in order of estimated relevance to the user's query [13, 14]. For the multi-gigabyte databases now available, ranking is considered the best option for data access: Boolean queries require expert formulation, and techniques such as browsing are ineffective for the initial location of answers from among large numbers of documents. The need for ranking has led to efforts such as the international TREC project, a cooperative experiment involving a two gigabyte text database and manual checking of over 300,000 documents for relevance to a test query set [7].

In comparison to Boolean queries, which retrieve exactly those documents that contain the specified query terms, ranked queries are statistically compared to the documents. The statistical *similarity* of a document to a query is assumed to correspond to the likely relevance of the document to the query, so the answers to the query are the documents with the highest similarity values. Many functions have been proposed for computation of similarities. One of the most successful functions—in terms of *retrieval effectiveness*, or ability to locate answers that humans judge to be correct—is the cosine measure [13, 14]. In a straightforward implementation of a similarity measure such as the cosine measure, the document database has an inverted index that contains, for each term in the database, an *inverted list* of the identifiers of the documents

containing that term. The costs of ranked query evaluation on such an index are: memory, to store the similarity values, usually requiring one *accumulator* per document in the database; disk traffic, to transfer inverted lists for each query term from disk to memory for processing; and cpu time, to process this index information.

For a large document database, the cost of evaluation of the cosine measure can be prohibitively high, because ranked queries are usually expressed in natural language and can therefore contain a large number of terms, some of which will occur in a high proportion of the database's documents, and because ranking techniques assign a similarity value to every document containing any of the query terms. As a consequence, typically most of the documents in the database will have non-zero similarity, and will hence be candidates for presentation to the user. For this reason, only the top-ranked documents are retrieved—most of the candidate documents are discarded.

We propose a technique for filtering documents during ranking, allowing a significant reduction in the volume of main memory required. The effect of the filter is that a document's accumulator is updated only if the combination of the frequency of the term in the document and the term's importance is large enough to be likely to have an impact on the final ordering of documents. Thus the inverted list of even a common term may be processed, but only for those documents in which the term is frequent will the accumulator be updated. Our experiments in applying this technique to the cosine measure show that it allows evaluation of the queries on a large document collection in approximately in 2% of memory of previous techniques, and without deterioration in retrieval effectiveness.

We also show how to re-organise inverted files to support the filtering heuristic. Inverted lists are generally *document-sorted*, that is, sorted by document identifier, but for the filter this implies that the whole of each list has to be processed, even when there are only a few documents in which the term is frequent. By sorting inverted lists by decreasing within-document frequency, so that they are *frequency-sorted*, the identifiers of the interesting documents are brought to the start of the list, also yielding a reduction in disk traffic because only part of each inverted list must be retrieved. Frequency-sorting can potentially have an adverse impact on index size, because index compression techniques rely on the small differences between adjacent documents in longer inverted lists to achieve size reductions [1, 10]. We show, however, that it is possible to use frequency-

sorting to achieve a net reduction in index size, regardless of whether the index is compressed. Together, these improvements make information retrieval possible for small machines such as PCs, and for large multi-user document systems such as library systems, which can have thousands of simultaneous users.

Document databases and the cosine measure are described in Section 2. The technique of document filtering is described in Section 3, together with experimental results. In Section 4 we show how to structure inverted lists to support filtering, and give experimental results for both compressed and uncompressed inverted files. Conclusions are presented in Section 5.

2 Ranked query evaluation

The ranking technique we use to demonstrate our techniques is the *cosine measure* [13, 14]. For this measure, the similarity of document d and query q is for practical purposes computed by

$$C_{q,d} = \frac{\sum_t sim_{q,d,t}}{W_d} ,$$

where W_d is the length of document d and $sim_{q,d,t}$ is the *partial similarity* of q and d with respect to term t, defined by

$$sim_{q,d,t} = w_{q,t} \cdot w_{d,t}$$

where $w_{x,t}$ is the weight of t in document or query x. The accumulators are used to hold the running totals for the expression $\sum_t sim_{q,d,t}$; the information for these totals is extracted from the inverted lists. The W_d values are precomputed with the expression

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

and stored elsewhere.

Several term weighting systems have been proposed and explored [4, 12, 14]. We assign the weight to a term in a query or a document using the frequency-modified inverse document frequency, described by

$$w_{x,t} = \log_2 f_{x,t} \cdot \log_2 \frac{N}{f_t} ,$$

where $f_{x,t}$ is the number of occurrences (or *within-document frequency*) of term t in x, N is the number of documents in the collection, and f_t is the number of documents containing t. The expression $w_t = \log_2(N/f_t)$ is the weight or

importance of t in the collection. This function assigns a high weight to terms which are encountered in only a small number of documents in a collection. It is supposed that rare terms have high discrimination value and the presence of such a term in both a document and a query is a good indication that the document is relevant to the query.

Database structure

We use inverted files to index documents [13, 14, 15]. An inverted index for a document database typically has two components: a *vocabulary* and a set of *inverted lists*. The vocabulary contains each term t in the database and the number f_t of documents containing t. Knowledge of f_t allows the terms in a query to be processed in order of decreasing weight [2, 8], as is necessary for the technique we shall describe. There is one inverted list for each t, consisting of the identifiers of the documents containing the term and, with each identifier d, the within-document frequency $f_{d,t}$ of t in d. Thus inverted lists consist of *document entries*, that is, pairs of $\langle d, f_{d,t} \rangle$ values.

Inverted lists are usually sorted by document identifier, not only for convenience of processing but because such sorting allows index compression—once sorted, the differences (or *run-lengths*) between adjacent identifiers can be computed, yielding small integers that are suitable for compression. For example, consider the list consisting of the following $\langle d, f_{d,t} \rangle$ pairs

$$\langle 5, 3 \rangle \langle 9, 2 \rangle \langle 12, 2 \rangle \langle 16, 5 \rangle \langle 21, 1 \rangle \langle 25, 2 \rangle \langle 32, 4 \rangle \, ,$$

which represents the fact that the term being indexed occurs three times in document 5, twice in document 9, and so on. This list can be converted into the sequence of run-lengths

$$\langle 5, 3 \rangle \langle 4, 2 \rangle \langle 3, 2 \rangle \langle 4, 5 \rangle \langle 5, 1 \rangle \langle 3, 2 \rangle \langle 7, 4 \rangle \, .$$

Given that the number of documents containing a given term can be used to compute the average run-length, using a parameterised code the run-lengths can be efficiently compressed, as the run-lengths will conform to a known distribution with a known mean. For high-frequency terms, often only 1 or 2 bits are required to represent a run-length if coded using integer coding schemes such as those of Elias [3] or Golomb [5]. The $f_{d,t}$ values are already a skew distribution of small integers, and can be effectively represented in unary or

1. For each document d in the collection, set accumulator $A_d \leftarrow 0$.
2. For each term t in the query,
   (a) Retrieve the inverted list for t from disk.
   (b) For each term entry $\langle d, f_{d,t} \rangle$ in the inverted list, set $A_d \leftarrow A_d + sim_{q,d,t}$.
3. Divide each non-zero accumulator A_d by the document length W_d.
4. Identify the k highest accumulator values (where k is the number of documents to be presented to the user) and retrieve the corresponding documents.

Figure 1: Basic algorithm for computing a cosine measure

in an Elias code such as the gamma code [3]. Overall, such inverted index compression techniques can reduce index size by a factor of six or more [1, 10].

For a large document database indexed by an inverted file, the index can be used to simultaneously compute the cosine correlation between each document in a collection and the query as follows [4, 10, 13, 14]. An accumulator is created for each document, either by initially allocating an accumulator for every document in the database or by dynamically adding an accumulator for a document when it is allocated non-zero similarity. The similarity of each document to the query q are then computed by retrieving the inverted list for each query term and adding $sim_{q,d,t}$ to the accumulator for every document d in the term's inverted list. Then each accumulator is divided by the appropriate W_d value and the k documents with the highest cosine values are chosen. A version of this algorithm, as given by Moffat and Zobel [10], is shown in Figure 1.

Evaluation of the cosine measure also requires a file containing the length W_d for each document. These values are query independent and need to be computed only once, at database creation time; and can be effectively compacted and stored in a few bits each [11]. The reason they are stored separately is to allow effective compression of the inverted file. Storage of the within-document frequencies normalised by the document lengths would imply storage of floating point numbers rather than small integers that can be effectively compressed and, hence, a substantial increase in the size of the inverted file.

Thus the main costs of query evaluation are memory space, for the accumulators; disk traffic, to retrieve inverted lists; and cpu time, to decode inverted

lists. Reducing all of these costs to levels suitable for a small machine is the subject of this paper.

3 Reducing the number of accumulators

As we have described above, the usual approach to the evaluation of ranked queries is consecutive processing of every term in a query and of the whole inverted list for each term. This technique computes, for each query term and each document containing the term, a partial similarity of the document and the query; each document requires an accumulator.

Thus a particular shortcoming of this technique is the memory required for the accumulators. The most common terms in a typical query are contained in a large proportion of the documents in a collection. Processing of all identifiers in these inverted lists leads to a large number of accumulators. Moreover, most of the partial similarities are given by common terms and thus have very low weight. Processing of these values produces little increase in accuracy and is expensive, particularly in systems that use compression for inverted lists, since, to evaluate queries, large volumes of data have to be decompressed.

There have been many attempts to improve the efficiency of ranked query evaluation [2, 4, 6, 8, 10]. Elimination of *stop-words*—that is, of very frequent words or closed-class words such as "and" and "of"—is often used to reduce the number of uninformative terms processed. But it is often difficult to determine the list of stop-words. For example, in our test database the word "text", which is not especially common in English, is encountered in every document in the collection and hence does not have any discrimination value. Another word, "Washington", is also common in the collection, but does seem to provide useful discrimination.

More sophisticated algorithms implement some dynamic stopping condition. The typical approach taken by these algorithms is to order terms in a query by decreasing weight, and then process terms in this order until some stopping condition is met [2, 6, 8, 10]. Moffat and Zobel [10] implemented the stopping condition by limiting the number of accumulators. They tested two versions of the algorithm. In the first version, processing of a query was stopped as soon as the number of accumulators exceeded a certain limit. In the second, processing of query was continued after reaching the limit number of accumulators but no new documents were inserted into the set of candidates. The first version of

this algorithm showed dramatic improvement in response time but at the cost of significant deterioration in retrieval effectiveness. The second version gave the same retrieval effectiveness as a basic version that processed all inverted lists, and in conjunction with a modification to the index structure discussed below approximately halved processing time.

Harman and Candela [6] experimented with another pruning algorithm. They accumulated partial similarities given by all documents in all inverted lists (like the second algorithm by Moffat and Zobel) but limited the number of accumulators by setting a condition for the insertion of new documents into the set of relevant documents: their algorithm only considered those documents which contained terms with inverse document frequency more than a certain fraction of the maximum inverse document frequency of any term in the database. An overview of pruning algorithms and some additional references are given by Salton [13] and Frakes and Baeza-Yates [4].

These techniques have the effect of saving time, by neither retrieving nor processing some inverted lists, and of saving space, by having fewer accumulators. However, there is often a penalty in retrieval effectiveness. The property common to all of these techniques is that they may process the inverted list for a term even if it is not particularly important in any document, or not process the inverted list for a discriminating term simply because it is fairly frequent; and that they abruptly switch from free addition of accumulators to allowing no addition of accumulators at all. They yield a reduction in the number of processed term entries but usually lead to deterioration in retrieval effectiveness, because the decision to stop is based only on global parameters of the data set. These algorithms select, for processing or rejection, whole inverted lists rather than separate document entries within these lists, and as a consequence these algorithms cannot provide a gradual transition from acceptance of terms to rejection of terms.

Filtering documents

Accumulator values cannot be effectively compressed because they are unpatterned real numbers, so the only way of reducing the space requirement is to reduce the number of documents for which an accumulator is required. We propose use of a *filtering* technique that provides a gradual transition from inclusion to omission of documents, by taking into consideration both the global parameter of term importance across the collection and the local parameter

of the number of occurrences of a term in each document. We modify the algorithm of Figure 1 in the following way.

As in the basic algorithm, query terms are sorted by decreasing w_t, so that important terms are processed first. Then, before each term t is processed, two thresholds are computed, an insertion threshold s_{ins} and an addition threshold s_{add}, where $s_{add} \leq s_{ins}$. As we process the inverted list for t, the partial similarity $sim_{q,d,t}$ of query q and each document d in the list is compared to the thresholds. If $s_{ins} \leq sim_{q,d,t}$, document d is important enough to be one of the candidates: if necessary an accumulator is created, then $sim_{q,d,t}$ is added to d's accumulator's value. If $s_{add} \leq sim_{q,d,t} < s_{ins}$, document d is not important enough to be interesting to the user by itself but $sim_{q,d,t}$ is likely to affect the final order of documents; so if d already has an accumulator then $sim_{q,d,t}$ is added to its value, but if not no action is taken. And finally, if $sim_{q,d,t} < s_{add}$, the information is unimportant and therefore discarded.

The rationale for the use of thresholds is that, if there are a large enough number of candidate documents with high values of similarity to the query, it is not profitable to consider small partial similarities—they are unlikely to significantly change the final ranking. For example, in the test database we used for our experiments (described later in this section), a typical less common query term had $w_t \approx 8$, whereas a typical common query term had $w_t \approx 1$. After the first few query terms were processed, the highest accumulator values were on the order of 500 to 5,000, with differences between adjacent accumulator values of from 10 to 100 or more. In this context, the $sim_{q,d,t}$ values of from 1 to 10 typically given by common terms do not have much effect on the final ordering.

Using the threshold s_{add} we can ignore inverted list entries that yield small partial similarities, thus saving cpu time. Likewise, the threshold s_{ins} allows us to ignore some documents, thus saving memory space. In other words, the thresholds provide a mechanism for tuning system load. Thresholds have previously been used to decide whether to process or reject whole inverted lists [6], but not to decide whether to process or reject individual documents.

The values of both thresholds for a term t are determined as a function of the accumulated partial similarity of the currently most relevant document S_{max}. This heuristic supposes that if the current most relevant document has a high weight then we do not need to process a document that has a small value of similarity to a query, as it is unlikely to change the final ranking or identify important document that is not yet included in the set of relevant documents.

The values of the thresholds are determined as

$$s_{ins} = c_{ins} \cdot S_{max} \text{ and}$$
$$s_{add} = c_{add} \cdot S_{max} \ ,$$

where $0 \leq c_{add} \leq c_{ins}$ are constants; choice of values for these constants is discussed below. The effect is that, as query terms are processed and the value of accumulated similarity of documents in the set of answers grows, it becomes increasingly difficult to update or add new accumulators.

We process term entry $\langle d, f_{d,t} \rangle$ in the inverted list of t only if the partial similarity $sim_{q,d,t}$ of d and query q is greater than the current value of threshold s, where s is either s_{ins} or s_{add}. Substituting the definitions of $w_{d,t}$ and $w_{q,t}$ into the definition of $sim_{q,d,t}$, we obtain

$$s \leq f_{d,t} \cdot w_t \cdot f_{q,t} \cdot w_t \ .$$

The final condition is

$$\frac{s}{f_{q,t} \cdot w_t^2} \leq f_{d,t} \ ,$$

thus expressing the decision of whether to process a term entry $\langle d, f_{d,t} \rangle$ as a condition on $f_{d,t}$. The thresholds can now be directly expressed in terms of frequencies:

$$f_{ins} = \frac{c_{ins} \cdot S_{max}}{f_{q,t} \cdot w_t^2} \text{ and}$$
$$f_{add} = \frac{c_{add} \cdot S_{max}}{f_{q,t} \cdot w_t^2}$$

These threshold values are constant during processing of an inverted list, so that the decision of whether to use a term entry requires only a single integer comparison.

The use of thresholds provides a smooth transition from acceptance to rejection of term entries in inverted lists, as it is progressively more difficult for accumulators to be added or updated. For the first terms processed the value of S_{max} is small and the value of w_t is large, so that most identifiers are considered. As S_{max} rises and w_t falls, the thresholds rise, until, in the limit, all $f_{d,t}$ values are less than f_{add}, so that processing an inverted list has no effect on accumulator values. The filtering algorithm for computing the cosine measure is shown in Figure 2.

The constants c_{ins} and c_{add} are used to control the resources required by the algorithm. By increasing the constant c_{add} we reduce the number of term entries (and, correspondingly, reduce the number of partial similarities of documents

1. Create an empty structure of accumulators.
2. Sort the query terms by decreasing weight.
3. Set S_{max} to 0.
4. For each term t in the query,
   (a) Compute the values of the thresholds f_{ins} and f_{add}.
   (b) Retrieve the inverted list for t from the disk.
   (c) For each term entry $\langle d, f_{d,t} \rangle$ in the inverted list,
      i. If $f_{d,t} \geq f_{ins}$, create an accumulator for A_d if necessary, and set $A_d \leftarrow A_d + sim_{q,d,t}$.
      ii. Otherwise, if $f_{d,t} \geq f_{add}$ and A_d is present in the set of accumulators, set $A_d \leftarrow A_d + sim_{q,d,t}$.
      iii. Set $S_{max} \leftarrow \max(S_{max}, A_d)$.
5. Divide each non-zero accumulator A_d by W_d.
6. Identify the k highest accumulator values and retrieve the corresponding documents.

Figure 2: Filtering algorithm for computing the cosine measure

and a query) inspected and accumulated by the algorithm, and hence decrease cpu time. By increasing the constant c_{ins} we reduce the number of documents that can be candidates, and hence decrease memory usage. The constants should be chosen so that the discarded information would, if included, have minimal impact on the final ordering. In a production system, the constant values could simply be adjusted at each query based on observation of system load, or occasional queries could be run for several values of each constant, and best values chosen according to the distortion introduced into the answer set.

A potential weak point of the filtering technique is its vulnerability to presence of documents with a large number of occurrences of a rare term. Such documents have very large weight and can theoretically make the values of the filters so large that no more documents will be able to meet filtering conditions and be taken into consideration. If this document contains the first (rarest) term in a query, then the set of answers to the query will consist only of the documents containing that term. To the test robustness of the method of filtering, we have tried another way of calculating the thresholds, in which S_{max}

is replaced by S_Q, defined by

$$S_Q = \sum_{t' \in Q} \left(\log_2 \frac{N}{f_{t'}} \right)^2 ,$$

where Q is the set of query terms that have already been processed. However, experimentally we have found that the difference in performance of the two versions of the filtering algorithm is insignificant, and we have used the S_{max} approach in the experiments described below. Another possibility is to use the average similarity of several top documents instead of the highest one, but this version would be more expensive.

Document filtering sharply reduces the volume of main memory needed for evaluation of ranked queries. However, the filtering technique as it stands does not yield substantial savings in either disk traffic or cpu time. To perform a ranking we still have to fetch and process the whole inverted list for every query term, comparing $f_{d,t}$ for every document to the current threshold values. For the long inverted lists only a few $f_{d,t}$ values pass the thresholds, so that most of the time spent processing these lists has no effect on the final ranking. Section 4 describes techniques for avoiding these problems.

Experimental results

The database we have used in our experiments is a collection of *Wall Street Journal* articles, extracted from the TREC data [7]. The value of this database is that it has a set of queries with manual relevance judgements that can be used to determine retrieval effectiveness. The database contains 173,000 documents, totalling 508 Mb; average document size is 510 term occurrences; the longest document consists of 22,200 terms. We have used queries 51–150 from the TREC experiment, after stemming and removing SGML markup; the length of the queries ranges from 66 to 313 terms. We measured the retrieval effectiveness of algorithms—their ability to retrieve answers a human judges to be relevant—from the recall (proportion of relevant documents retrieved) and precision (proportion of retrieved documents that are relevant), by averaging precision at 0%, 10%, ..., 100% recall. For consistency with the TREC experiments, we retrieved only the top 200 documents for each query, and pessimistically assumed all recall values outside the top 200 to be zero. All results shown are average values over all 100 queries.

Retrieval effectiveness is shown as a function of the addition threshold in Figure 3. We depict two parameters on the horizontal axis: the value of the

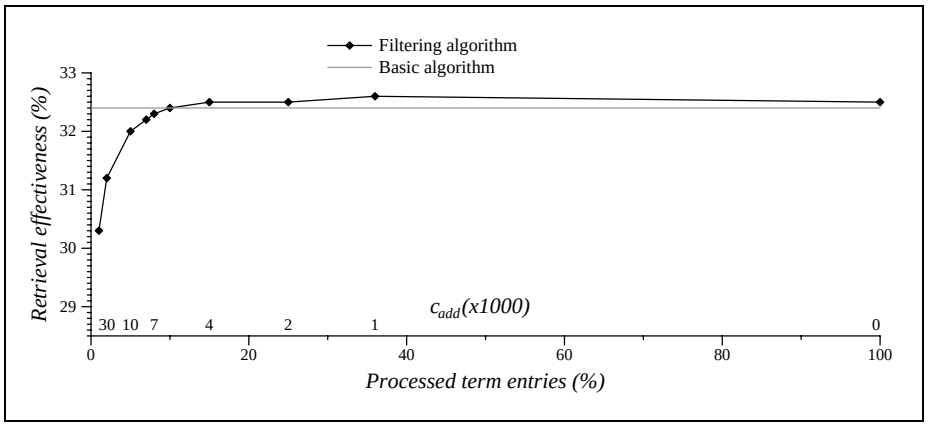

Figure 3: Retrieval effectiveness for different values of c_{add} ($c_{ins} = 0.12$)

constant c_{add} and the percentage of term entries processed by the algorithm for this value of c_{add}. For comparison, we also show as a horizontal line the performance with $c_{ins} = c_{add} = 0$, that is, for the algorithm shown in Figure 1.

The value of the insertion threshold was fixed in this experiment. Prior to these experiments we measured retrieval effectiveness for different values of c_{ins}, and chose 0.12 because it gave good retrieval effectiveness using a small number of accumulators. For this value of c_{ins}, we can obtain an answer to a ranked query with the same retrieval effectiveness as the basic algorithm (32.4%) having processed only 10% of all term entries. Interestingly, processing 15% of all term entries we obtain even better retrieval effectiveness in comparison to the standard algorithm. We believe that this is because of the pruning of common terms, which are encountered in almost every document and create informational noise rather than help discriminate between documents. Note that it is only necessary to process a very small number of term entries to obtain a decent level of retrieval effectiveness. For example, while processing only 1% of the term entries, the deterioration in retrieval effectiveness is only 2.1%.

Figure 4 shows the dependency of retrieval effectiveness on the number of accumulators. The number of accumulators was varied by changing the insertion threshold. Both the value of the constant c_{ins} and the corresponding number of accumulators is depicted on the horizontal axis. We used $c_{add} = 0$ in this experiment to prevent the skipping of common terms; that is, for each of document included in the set of candidate documents we accumulated all partial similarities given by all terms. Note that using a relatively small number of candidate documents we obtain better retrieval effectiveness than does the basic

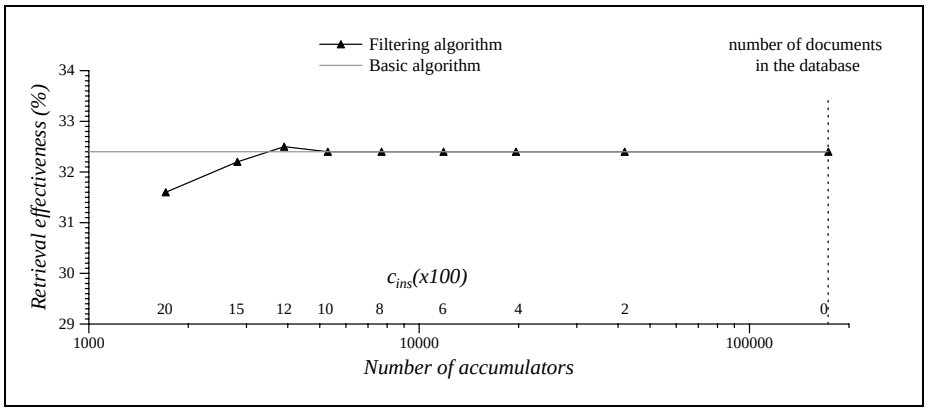

Figure 4: Number of accumulators for different values of c_{ins} (c_{add}=0)

algorithm. Interestingly, this phenomenon is consistent for different techniques and different document collections; for example, similar results were obtained by Moffat and Zobel in their experiments with an explicit limit on the number of accumulators [10], and in our own experiments with a different version of the cosine measure.

The main saving yielded by this technique is a sharp reduction in the number of accumulators. This is illustrated in Figure 4. On the horizontal axis we vary c_{ins}, which affects the number of accumulators; for example, $c_{ins} = 0.12$ results in roughly 4,000 accumulators, whereas $c_{ins} = 0$ results in almost every document having an accumulator, or around 173,000 accumulators in total. The vertical axis is retrieval effectiveness, which remains high even when the number of accumulators is small; until the number of accumulators drops below 4,000 retrieval effectiveness is constant and is equal to that given by the basic algorithm. The technique also yields a small saving of cpu time, as we do not have to compute the $sim_{q,d,t}$ values for document identifiers that are filtered out.

The filtering algorithm is reasonably insensitive to both c_{ins} and c_{add}, providing good performance across a wide range of values. Moreover, per query the largest number of accumulators used in our experiments was no more than 3 times the average value, so that performance does not greatly depend on characteristics of individual queries. Thus the major effect of the thresholds is on system performance, with c_{ins} affecting memory usage and c_{add} affecting response time and disk traffic.

To confirm these results we applied the filtering method to another subset of the TREC data, the *Associated Press* subcollection. We observed almost identical behaviour: excellent performance with only a few thousand accumulators

and little impact on cpu time.

The queries used in these experiments are quite long. It might be argued that short queries of only a few terms would be adversely affected by the information discarded during filtering, but we believe that this would not be the case. Filtering discards contributions that are small compared to values accumulated so far, so that less information is discarded for the first few query terms, with typically no information discarded for the first one to five terms processed. While the performance gains for short queries (which have modest resource requirements) would be less spectacular than for long queries, we would not expect effectiveness to degrade.

Other term weighting systems

The cosine measure as described in Section 2 is not the only similarity measure. There are other similarity measures, for example those described by Lucarella [8] and Harman and Candela [6]. We tested the robustness of document filtering by applying it to these similarity measures.

Lucarella determined the similarity of a document and the query using the formula

$$\frac{\sum_t w_{q,t} \cdot w_{d,t}}{\sqrt{\sum_t w_{q,t}^2} \cdot \sqrt{\sum_t w_{d,t}^2}} \ ,$$

where q is the query, d is the document, and $w_{x,t}$ is the weight of the term t in a document or query x. The weight of a term is determined as

$$w_{x,t} = (0.5 + 0.5 \cdot f_{x,t}/f_x^{max}) \cdot w_t \ ,$$

$$w_t = \log_2 \frac{N}{f_t} \ ,$$

where $f_{x,t}$ is the number of occurrences of the term t in x, f_x^{max} is the maximum occurrence frequency among the terms associated with the document or query x, N is the number of documents in the collection, and f_t is the number of documents containing t. This measure is similar to our form of the cosine measure, but the importance of the within-document frequency of a term in a document is smaller in Lucarella's measure since it is normalised.

Harman and Candela employed the similarity measure

$$\sum_t \frac{\log_2 (f_{d,t} + 1) \cdot (w_t + 1)}{\log_2 M_d} \ ,$$

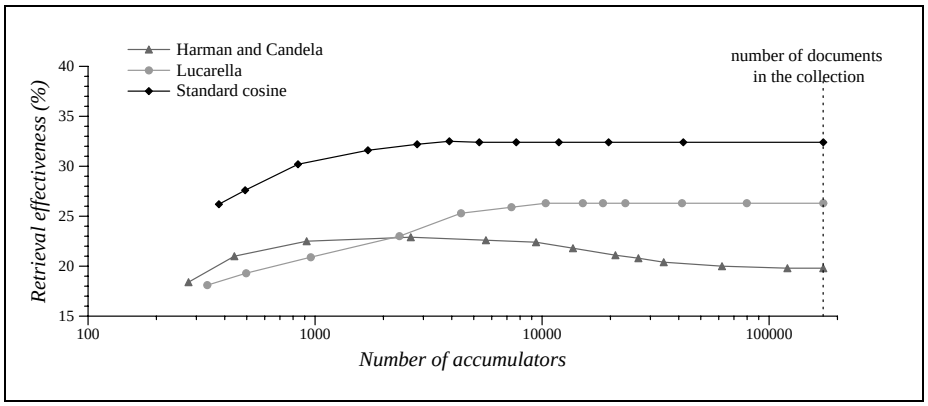

Figure 5: Retrieval effectiveness for different number of accumulators

where M_d is the total number of significant terms (including duplicates) in the document d. This similarity measure considers only a frequency of a term in documents, not taking into account the number of term occurrences in a query.

For these measures we examined the reduction in the number of accumulators. (We were not able measure time savings that our technique would yield for these similarity measures, as this would have required a reimplementation of the inverted index.) As the volume of computation required for evaluation of the similarity of the query and documents is approximately the same both for our similarity measure and for the measures used by Harman and Candela and by Lucarella, we expect that the time savings should be the same as for the similarity measure used in our system.

Figure 5 shows, for these similarity measures, retrieval effectiveness as a function of the number of accumulators. The number of accumulators was varied by changing the constant c_{ins}. As for the standard cosine measure document filtering allows queries to be evaluated without deterioration in retrieval effectiveness, using only about 1% and 6% of the previous memory requirement for Harman and Candela's and Lucarella's algorithms respectively.

4 Inverted file structures for filtering

For our ranking technique, the decision about whether to process or reject a term entry depends on the within-document frequency $f_{d,t}$. For the usual structure of inverted lists, where term entries are sorted by document identifier, we have to process the whole list, comparing $f_{d,t}$ in every term entry to the current value of the threshold. We propose that inverted lists instead be

frequency-sorted, that is, sorted by decreasing $f_{d,t}$, so that the time wasted processing small $f_{d,t}$ values can be entirely avoided. First, once an $f_{d,t}$ value is encountered that is below the threshold, processing of the inverted list can stop. Second, if the inverted list is longer than a disk block, only one block of the list needs to be retrieved at a time: since the tail of a long inverted list will contain only small $f_{d,t}$ values, it is unlikely to be required, and there is little cost associated with leaving it on disk until requested.

It is also useful to store in the vocabulary the maximal within-document frequency f_t^{max} for each term t, to allow skipping of inverted lists. Before commencing the processing of each term in a query we compute the threshold frequencies f_{add} and f_{ins} and compare them to the maximal within-document frequency of the term f_t^{max}. If f_t^{max} is less than f_{add} then no document containing this term will be processed and we can proceed to the next term in the query without retrieving the inverted list from disk.

Unfortunately, frequency sorting is incompatible with compression of inverted lists. If the document identifiers are unsorted, run-lengths cannot be taken and the index size will dramatically increase. Besides the impact on space requirements, an immediate effect of this increase is in the real time required to compute a ranking: inverted lists become more expensive to retrieve from disk. For some queries this penalty will outweigh the gain of re-ordering.

Thus it is crucial that we find some way of maintaining compression performance. A simple way of having some compression within frequency-sorted inverted files is to, for the term entries with the same $f_{d,t}$ value, sort by document identifier. Inverted lists then consist of a series of *sequences*, where each sequence is a triple

$$(f, p_f, (d_1, \ldots, d_{p_f}))$$

where f is the $f_{d,t}$ value of the documents $d_1, \ldots, d_{p_f}$ in the sequence and p_f is the number of documents. For a sequence of several documents with the same frequency there is a potential space saving, as the frequency only has to be stored once. The identifiers in a sequence are sorted, allowing run-lengths to be taken and hence allowing compression. For example, the inverted list illustrated in Section 2 would under this scheme be represented as

5, 1, (16)	4, 1, (32)	3, 1, (5)	2, 3, (9,3,13)	1, 1, (21)

in which each box is a sequence, the first number is the frequency, the second is the number of documents in the sequence, and the expression in brackets

is the documents in that sequence. The expression $(9, 3, 13)$ represents the document numbers 9, 12, and 25 after run-lengths have been taken—these are the documents that contain the term with frequency 2.

However, the sequence method might not yield as good compression as for document-sorted inverted lists. One reason for possible poorer compression is the pattern of document identifiers within sequences. A run-length of k can typically be compressed to a little over $\log_2 k$ bits; since the average run-length between identifiers in a sequence is larger than the average run-length in the sorted inverted list, compression performance degrades. Another reason for possible increase in size is that, although many sequences are only one or two documents long, the per-sequence parameters still have to be stored.

In a database of N documents, the size of a document-sorted inverted list of p identifiers can be estimated as follows. The number of bits required to store the document identifiers is approximately [9]

$$B_G(p) = p\left(1.5 + \log_2 \frac{N}{p}\right) .$$

In addition an $f_{d,t}$ value must be stored for each document. The space required for these values will depend on the distribution of frequencies. We assume that the distribution is given by a integral function $I(p, f)$ for which $\sum_{f=1}^{r} I(p, f) = p$, where r is the largest $f_{d,t}$ value in the distribution. We also assume that each $f_{d,t}$ value is represented by a gamma code [3]; the number of bits required to represent frequency f using gamma is

$$B_\gamma(f) = 1 + 2\lfloor \log_2 f \rfloor .$$

Thus the space required for the $f_{d,t}$ values is

$$\sum_{f=1}^{r} (I(p, f) \cdot B_\gamma(f))$$

and the total space for a document-sorted inverted list is approximately

$$B_{DS}(p) = B_\gamma(p) + B_G(p) + \sum_{f=1}^{r} (I(p, f) \cdot B_\gamma(f)) ,$$

where $B_\gamma(p)$ bits are needed to represent the length of the list.

Based on the same assumptions, the size of a frequency-sorted list can be determined as follows. In the sequence for frequency f there are $I(p, f)$ identifiers, so each sequence requires $B_G(I(p, f))$ bits for identifiers. In addition each sequence requires approximately 1 bit for the frequency (the frequencies

are ordered so differences can be taken, and usually the difference will be 1) and $B_\gamma(I(p,f))$ bits to store the number of identifiers in the sequence. In total, the space required for a frequency-sorted inverted list of p identifiers is approximately

$$B_{FS}(p) = B_\gamma(r) + \sum_{f=1}^{r} \left(B_G(I(p,f)) + 1 + B_\gamma(I(p,f))\right) .$$

where r is again the largest $f_{d,t}$ value in the inverted list and $B_\gamma(r)$ bits are needed to represent the number of sequences.

Whether B_{FS} or B_{DS} is larger depends on the distribution of frequencies. One extreme is that all documents have $f_{d,t} = 1$, that is,

$$I(p,f) = \begin{cases} p & \text{if } f = 1 \\ 0 & \text{otherwise} \end{cases}$$

for which we have

$$B_{DS}(p) = B_\gamma(p) + B_G(p) + p \cdot B_\gamma(1) = B_\gamma(p) + B_G(p) + p$$

and

$$B_{FS}(p) = B_\gamma(1) + B_G(p) + 1 + B_\gamma(p) \,. = B_\gamma(p) + B_G(p) + 2$$

In the case of inverted lists in which all $f_{d,t}$ values are 1, therefore, frequency-sorting results in slightly better compression. Another extreme is when $p = r$ and each document has a different $f_{d,t}$ value, that is,

$$I(p,f) = \begin{cases} 1 & \text{if } f \le p \\ 0 & \text{otherwise} \end{cases}$$

for which we have

$$B_{DS}(p) = B_\gamma(p) + B_G(p) + \sum_{f=1}^{p} B_\gamma(f)$$

and

$$B_{FS}(p) = B_\gamma(p) + p \cdot \left(B_G(1) + 1 + B_\gamma(1)\right) .$$

In this case, of each $f_{d,t}$ value occurring once, which is better will depend on p, but the sizes will be similar.

For the *Wall Street Journal* database, we have observed that most of the $f_{d,t}$ values in most inverted lists are 1, most of the remainder are 2, and so on—there is a strong skew towards low frequencies. This distribution can be modelled as

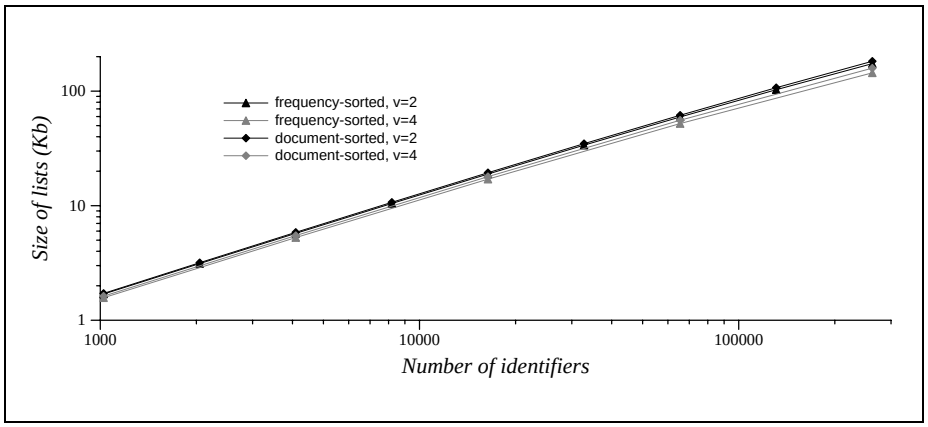

Figure 6: Estimated size of compressed inverted lists

follows. Suppose that for some integer v, the distribution of frequencies is such that $(v-1)/v$ of the identifiers in each inverted list have $f_{d,t} = 1$, of the remainder $(v-1)/v$ have $f_{d,t} = 2$ (that is, $(v-1)/v^2$ of the total), and so on. That is, the number of identifiers with $f_{d,t} = f$ is given by

$$I(p,f) = \begin{cases} p(v-1)/v^f & \text{if } f \leq \log_v p \\ 1 & \text{if } f = \log_v p + 1 \\ 0 & \text{otherwise.} \end{cases}$$

for p such that $\log_v p$ is integral. Estimated sizes for compressed inverted lists are plotted in Figure 6 for $v = 2$ and $v = 4$ for a database of 1,000,000 records. As can be seen, the sizes are almost identical, with the frequency-sorted index very slightly smaller.

It is straightforward to extend the model developed above to predict the volume of index data retrieved in response to a query, but the result depends on several estimates—the function $I(p,f)$, the distribution of p values for query terms, and the likely S_{max} value—so the predictions made by such a model are at best a broad indicator of possible performance. What is clear is that use of filtering reduces, and has the potential to drastically reduce, disk traffic. The scale of reduction is best determined experimentally, as we do for the *Wall Street Journal* later in this section.

A possible drawback of frequency-sorting of inverted lists is the impact on update. The costs of update for an inverted index are: locating and fetching the list; identifying the part of the list to be modified; modifying the list; and writing the list to disk, making any reorganisation necessary to minimise space fragmentation if the list's length has changed. Of these costs, only the

second—searching the list—is affected by the change from document sorting to frequency sorting; typically the searching cost might double, while the other costs are unchanged. We therefore believe that frequency sorting has only a minor impact on update. However, as for most indexing methods for text databases, update is expensive, requiring disk accesses for every indexed term in each modified or inserted document.

Our method of filtering and re-ordering inverted lists into sequences of documents of the same frequency is not the only possible solution to the problem of ignoring the majority of document identifiers. Moffat and Zobel have proposed that inverted lists be ordered by identifier, but in addition contain pointers into the inverted list at evenly-spaced intervals, to allow the search to "skip" sections of the list without decompression [10]. Such skipping provides the benefit of random access (usually impossible in the context of compression) while maintaining reasonable compression performance. In conjunction with their scheme of a small, fixed number of accumulators, the skipping reduces cpu time without degrading retrieval effectiveness; however, this scheme slightly increases disk costs, and does not support filtering. As we show below, the gain they achieve is limited compared to that given by the scheme we describe here.

Other representations of sequences

The analysis above indicates that the sequence method for representing inverted lists should yield reasonable compression, but better compression may be possible, particularly for the sequences of higher frequency terms—a typical long inverted list will contain many term entries for which $f_{d,t}$ is 1 or 2, and a small number of term entries for which $f_{d,t}$ is large. That is, for the high frequencies, many sequences will have only one or two documents and the overheads of representing a short sequence (the need to store the number of documents and the loss of compression due to the large run-lengths) are high.

These problems can be overcome by selective application of the idea of sequences. As we have seen, there are advantages to the long sequences of low frequencies, but short sequences are inefficient. It follows that an efficient form of inverted list is an initial sequence of $\langle d, f_{d,t} \rangle$ pairs, for the high frequencies that would lead to short sequences, followed by a series of sequences, one for each of the low frequencies. We therefore propose the following structure for representing an inverted list. Each list is split into n sequences (the problem of choice of n is discussed later). The *leading sequence* is of $\langle d, f_{d,t} \rangle$ pairs, for

all documents with $f_{d,t} \geq n$. Each remaining sequence is of documents of some frequency $f_{d,t} < n$, and the sequences are ordered by decreasing frequency. Within each sequence, the entries are sorted by document identifier. Within the leading sequence, rather than storing $f_{d,t}$ values we store $f_{d,t} - n + 1$. The minimum value of n is 1, in which case the whole list is stored in one sequence. The final filtering algorithm, using sequences, is shown in Figure 7. Such a scheme should be effective because, for even the longer inverted lists of more common terms, the distribution of $f_{d,t}$ values is highly skew. Thus, in the above scheme, each of the low $f_{d,t}$ values would have its own sequence, which would be long; whereas the high $f_{d,t}$ values would share a sequence.

At the start of each inverted list that has been grouped into sequences we store the number of sequences; each sequences starts with the number of entries in it. The frequency of a sequence is determined by its ordinal number. This method means that, for all but the leading sequence, frequencies are not explicitly stored, and also means that we have to store zero as the number of documents for an empty sequence. An example of this method of storing inverted lists, using $n = 3$, is as follows.

3	3|$\langle 5,1\rangle\langle 11,3\rangle\langle 16,2\rangle$	3|9,3,13	1|21

This example corresponds to the inverted list shown above. The first box is the number of sequences in the list. The second box is the leading sequence and the third and fourth boxes are the sequences for frequencies 2 and 1 respectively.

We now examine, for inverted lists compressed with the sequence method, optimisation for index size and query evaluation time. Consider the effect of having the same n for all inverted lists, and of the inverted file that results from varying this n. As we increase n, we increase the number of sequences in each inverted list. On the one hand, this allows storage of more document identifiers without their corresponding frequencies. On the other hand, we have to store a sequence length for each sequence, including zeros for sequences that do not contain documents. Sequence lengths are a significant overhead on the size of the inverted file, and as n increases they quickly become unacceptably large. Also, decrease in the length of each sequence implies an increase in the average run-length and, hence, a worse rate of compression.

Small n implies a small inverted file. But now consider the problem of proper choice of n for fast query evaluation, in which case we wish to stop processing term entries (ordered by decreasing $f_{d,t}$) as soon as $f_{d,t} < f_{add}$ is found. If

the value of threshold f_{add} is less than the minimal frequency n of documents in the leading sequence, we process the whole leading sequence, and possibly some subsequent sequences, and for all documents processed we update their accumulators; thus no decoding time is wasted. But if f_{add} is more than n we must process the whole of the leading sequence, even though some of the documents in the sequence will be ignored. So, using one value of n for all inverted lists, to achieve fast query evaluation we have to increase the size of n, which will however increase the size of the inverted file.

The other possibility to allow n to vary between lists. A simple method would be to, for each list, set n to 1, compress the list; then increment n and compress it again; and so on until a minimum is found. The existence of a minimum is guaranteed, as the size of the sequence lengths will, in the limit, be dominant. (Note that, in a scheme with varying n, in addition to the sequence lengths the value of n must be stored in each list.) However, such a scheme is impractical.

The heuristic scheme we chose for selection of n is based on the observation that using a separate sequence for each $f_{d,t}$ value when $f_{d,t}$ is high (and the length of the sequence is low) is expensive because of the per-sequence overheads. Let us call the number of identifiers at which overall compression gains outweigh overheads the *sequence threshold* T. (In fact T is a function of, not just sequence length, but of the $f_{d,t}$ for the sequence; but since in our test collection almost all inverted lists have only a few frequencies with sequences of any length, this approach is a reasonable approximation.) To achieve good compression, we should avoid sequences of a length less than T. We determine the size of n for the inverted list for a term t using the following procedure. Initially, for each distinct value of $f_{d,t}$, we find the number of documents that contain t this number of times. Then we find the highest $f_{d,t}$ for which the number of documents is at least T. Let us denote this frequency as f_T. We then create the inverted list by having per-frequency sequences for frequencies from 1 to f_T and a leading sequence that contains documents with all remaining frequencies. The value of n for such an inverted list is $f_T + 1$.

On the one hand, if T is 1 then every frequency in every inverted list will have its own sequence, and the value of n for an inverted list will be the highest $f_{d,t}$ value in that list. On the other hand, for (say) a database of a million documents, if T is 100,000 then most inverted lists will have n of 1, and thus have only one sequence; but the inverted lists for the most common terms

will probably have several sequences, because these terms would in a typical database occur in almost every document.

Having leading sequences of mixed $f_{d,t}$ allows us to achieve two aims simultaneously. On one hand, we avoid creation of inverted lists containing many short sequences that cannot be effectively compressed, and similarly avoid storing many sequence lengths. On the other hand, we are able to keep the leading sequences short and, hence, have fast query evaluation.

Experimental results

Using the *Wall Street Journal* database we built a document-sorted inverted file and a frequency-sorted inverted file and evaluated the TREC queries described above. In all of these experiments we used filter values $c_{ins} = 0.12$ and $c_{add} = 0.007$, as these gave good retrieval effectiveness while requiring only a small number of accumulators. All times and volumes of disk traffic are per query, averaged over the 100 TREC queries, on a Sun SPARC 10 model 512, using local disks.

The size of the document-sorted compressed inverted file is 35.4 Mb; that of the frequency-sorted inverted file is 33.4 Mb, or only 6.6% of the size of the original data. Overall, therefore, the cost of storing the per-sequence parameters is more than offset by the saving of not storing duplicate $f_{d,t}$ values. On the document-sorted index, average query evaluation is 3.18 cpu seconds for stopped queries (from which closed-class words have been removed, on the grounds that they have little impact on retrieval effectiveness) and 10.18 cpu seconds for unstopped queries; on the frequency-sorted index, the comparable times are 1.20 cpu seconds and 1.73 cpu seconds respectively. These times are very similar, demonstrating that the filtering method almost completely excludes stop-words from consideration. That is, our method obviates the need to manually select a list of stop-words.

Frequency-sorted indexes require far less data to be fetched from disk than do document-sorted indexes, since we usually have to read only the first block of each inverted list. For document-sorted inverted files and stopped queries, the volume of data fetched was 532 Kb; for unstopped queries, it was 2,108 Kb. In contrast, using our technique the volume of index fetched was just 157 Kb and 249 Kb respectively. The number of disk accesses is also reduced, since deciding whether to reject a term does not require a disk access.

These results compare well to those of the "skipping" scheme of Moffat and

Zobel [10], who on a larger database are only able to halve cpu time, and actually increase disk traffic slightly. However, their scheme is also applicable to Boolean queries, for which they achieve much greater performance gains. The idea of their scheme is to break usual identifier sorted indexes into blocks and to store some additional information allowing decoding algorithm to skip a block if necessary without decoding its contents. The same scheme can be applied to the frequency sorted index. Inverted lists in such an index consist of sequences and store documents ordered by their numbers inside of each sequence, as in Moffat's and Zobel's scheme. Skipping information can be inserted into each sequence, thus allowing efficient searching for documents by their numbers during evaluation of boolean queries. We believe that this approach should provide good performance of evaluating Boolean queries at the cost of slightly decreased efficiency of processing ranked queries due to necessity of decompressing additional skipping information.

We also built a series of indexes using different values of the sequence threshold T, to experiment with the effect of T on performance. The size of an inverted file is shown as a function of the sequence threshold T in Figure 8. At one extreme, assigning T to 1 forces creation of a separate sequence for every frequency with at least one document. Inverted lists in such a file do not have leading sequences. This leads to an increase in index size because of the shortness of the sequences and because the number of sequence lengths to be stored is larger. Large values of T also lead to a gradual increase in inverted file size, as the leading sequences becomes long and we have to represent many large frequencies in these sequences.

We also examined query evaluation time for different values of T. The time is almost constant for small values of T (up to $T = 100$ or so) since the difference in size of leading sequences is small. Performance deteriorates for large values of T, because, during processing of common terms, we have to process long leading sequences, searching for the documents that pass the filter and ignoring the rest. In the limit, of huge T, we have a document-sorted index. Note that there are processing overheads that are independent on the number of processed documents; hence the decrease in the time of query evaluation is not a linear function of the quantity of index processed.

The volume of inverted lists fetched and decompressed during query evaluation is shown in Figure 9, again for both stopped and unstopped queries. Frequency-sorted inverted files built with small values of T provide an almost

constant amount of decompressed data. This is because, on the one hand, the smaller the value of T the smaller the leading sequence and, hence, the smaller the number of documents which have to be decompressed but ignored; on the other hand, small values of T give rise to inverted lists consisting of many small sequences, so that the overheads for storing sequence parameters increase and the same number of compressed identifiers occupy more space. For small values of T these phenomena are almost in balance, producing a plateau in the graph. On the other hand, inverted files built with large values of T have long leading sequences, leading to increase in the amount of data that is fetched and processed.

Overall, performance is excellent across a wide range of T values, and for all T values performance is better than for document-sorted compressed inverted files. Retrieval effectiveness is maintained; index size is reduced; and cpu time and disk traffic are much reduced. We expect that relative performance would improve further with growth in the database size. Since performance depends only marginally on T, we conclude that $T = 1$ can be used in a production system. Note, however, that the majority of documents in the *Wall Street Journal* database are short and average within-document frequency is small. For databases of longer records, a higher T value may be preferable.

Uncompressed inverted files

Our structure for inverted files, where documents in inverted lists are ordered by decreasing $f_{d,t}$, would also be effective in systems that use uncompressed inverted files. Using this structure yields significant reduction in the size of inverted files. Typically, a $\langle d, f_{d,t} \rangle$ pair occupies 6 bytes, consisting of 4 bytes for storage of the document number and 2 bytes for storage of the term frequency. Using our structure of an inverted file allows decrease in the size of the inverted file from 238 Mb for the basic structure to 160 Mb; that is, we can almost completely avoid storing $f_{d,t}$ values. The size of the uncompressed inverted file for different values of the sequence threshold is shown in Figure 10.

5 Conclusions

We have shown how to make dramatic reductions in the major costs of ranking a query on a large document database—disk traffic, cpu time, and memory usage—without degrading retrieval effectiveness. The basis of these reductions

is the filtering method, in which only the documents with high within-document frequency are considered as candidate answers; it is this technique that reduces memory usage, as having fewer candidates means that fewer accumulators are required to store information about these candidates. Despite the reduction in memory usage, there is no deterioration (and even with some improvement) in retrieval effectiveness.

The reductions in disk traffic and cpu time are based on the simple observation that, by ordering inverted lists by decreasing within-document frequency, only the first part of each list will contain high frequencies, and so the rest can be ignored. Frequency-sorted inverted lists can be effectively compressed by splitting inverted lists into sequences of documents of the same frequency and applying the existing compression techniques within each sequence. Both modelling and experiment have shown that change to frequency sorting has no negative impact on index size.

For our test database, these techniques maintain retrieval effectiveness; reduce memory requirements from 173,000 to 4,000 accumulators; reduce the quantity of data requested from disk from 532 Kb to 157 Kb; and reduce cpu time from 3.18 to 1.20 seconds. The gains for unstopped queries are even greater. The time saving is most noticeable for systems that use compression for storage of data, since the cost of decompression of long inverted lists is the major component of processing time. There is also a slight reduction in index size, from 35.4 Mb to 33.9 Mb, already a massive saving on the 238 Mb required for an uncompressed index. Together, these dramatic improvements allow ranking to be performed much faster, and on much smaller machines, than was previously possible.

Acknowledgement

We would like to thank Alistair Moffat.

References

[1] T.C. Bell, A. Moffat, C.G. Nevill-Manning, I.H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531, October 1993.

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

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

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

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

[6] 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.

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

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

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

[10] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, in press.

[11] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Information Processing & Management*, 30(6):733–744, November 1994.

[12] S.A. Perry and P. Willett. A review of the use of inverted files for best match searching in information retrieval systems. *Journal of Information Science*, 6:59–66, 1983.

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

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

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

Symbols used

Symbol	Meaning	Symbol	Meaning
A	set of accumulators	N	number of documents
A_d	accumulator for document d	q	query
$C_{q,d}$	cosine for document d	S_Q	current similarity threshold
c_{ins}, c_{add}	threshold constants	s_{ins}, s_{add}	partial similarity thresholds
d	document identifier	$sim_{q,d,t}$	partial similarity of q and d with respect to t
f_{ins}, f_{add}	within-document frequency thresholds	T	sequence threshold
$f_{d,t}$	frequency of t in d	t	term
$f_{q,t}$	frequency of t in q	$w_{d,t}$	weight of t in d
f_T	sequence frequency threshold	$w_{q,t}$	weight of t in q
f_t	number of documents containing term t	w_t	weight of t in the collection
k	number of answers	W_d	weight or length of d

1. Create an empty structure of accumulators.
2. Sort the query terms by decreasing weight.
3. Set $S_{max} \leftarrow 0$.
4. For each term t in the query,
   (a) Compute the values of the filters f_{ins} and f_{add}.
   (b) If $f_t^{max} < f_{add}$ go to step 4.
   (c) For the leading sequence in t's inverted list and each document d in the sequence,
      i. If $f_{d,t} \geq f_{ins}$, create an accumulator for A_d if necessary, and set $A_d \leftarrow A_d + sim_{q,d,t}$.
      ii. Otherwise, if $f_{d,t} \geq f_{add}$ and A_d is present in the set of accumulators, set $A_d \leftarrow A_d + sim_{q,d,t}$.
      iii. If A_d was updated, set $S_{max} \leftarrow \max(S_{max}, A_d)$.
   (d) For each remaining sequence in t's inverted list with $f_{d,t} \geq f_{add}$ and each document d in the sequence,
      i. If $f_{d,t} \geq f_{ins}$, create an accumulator for A_d if necessary, and set $A_d \leftarrow A_d + sim_{q,d,t}$.
      ii. Otherwise, $f_{d,t} \geq f_{add}$; if A_d is present in the set of accumulators, set $A_d \leftarrow A_d + sim_{q,d,t}$.
      iii. If A_d was updated, set $S_{max} \leftarrow \max(S_{max}, A_d)$.
5. Divide each non-zero accumulator A_d by the document length W_d.
6. Identify the k highest values of accumulators (k is the number of documents to be presented to the user) and retrieve the corresponding documents.

Figure 7: Filtering algorithm using sequences to compute the cosine measure

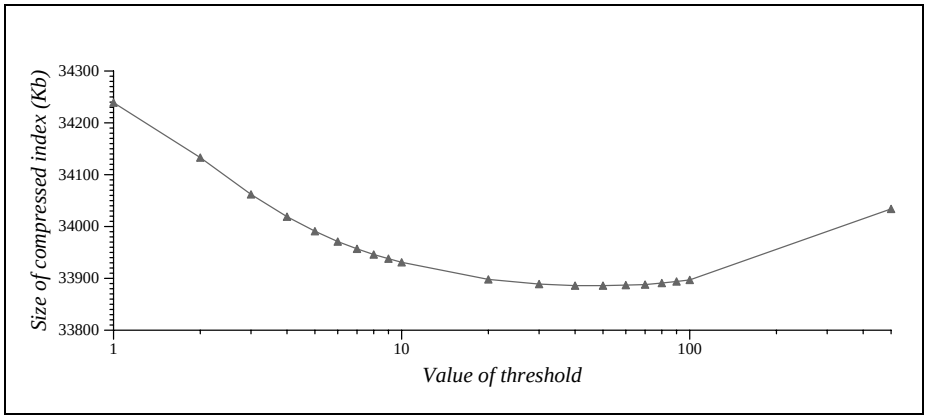

Figure 8: Size of compressed index for different values of sequence threshold

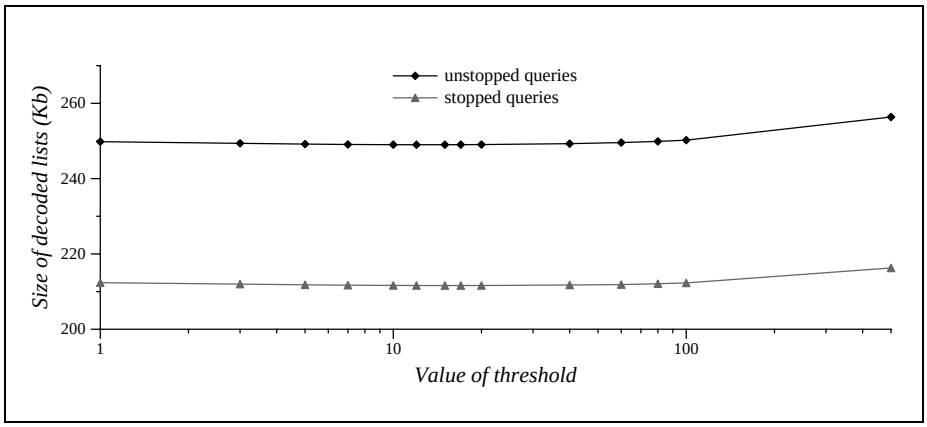

Figure 9: Volume of inverted lists decoded during query evaluation

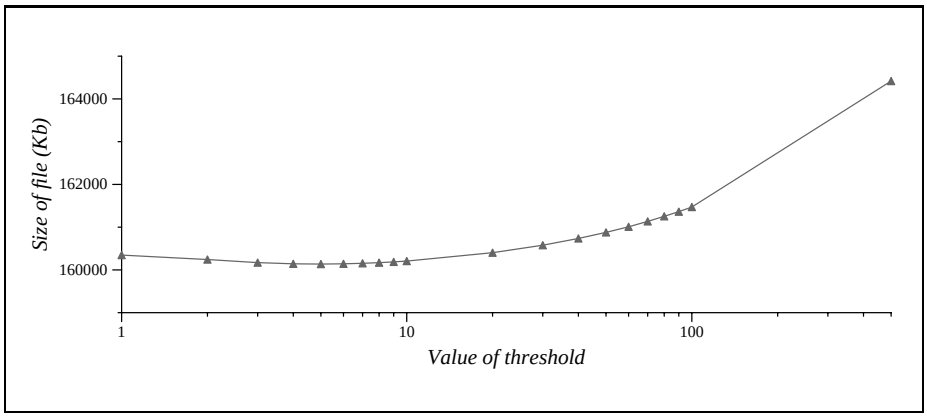

Figure 10: Size of uncompressed index for different values of sequence threshold