Self-Indexing Inverted Files for Fast Text Retrieval*

Alistair Moffat† Justin Zobel‡

February 1994

Abstract

Query processing costs on large text databases are dominated by the need to retrieve and scan the inverted list of each query term. Here we show that query response time for conjunctive Boolean queries and for informal ranked queries can be dramatically reduced, at little cost in terms of storage, by the inclusion of an internal index in each inverted list. This method has been applied in a retrieval system for a collection of nearly two million short documents. Our experimental results show that the self-indexing strategy adds less than 20% to the size of the inverted file, but, for Boolean queries of 5–10 terms, can reduce processing time to under one fifth of the previous cost. Similarly, ranked queries of 40–50 terms can be evaluated in as little as 25% of the previous time, with little or no loss of retrieval effectiveness.

CR Categories: E.4 [Coding and Information Theory]: *data compaction and compression*; H.3.1 [Information Storage and Retrieval] Content Analysis and Indexing—*indexing methods*; H.3.2 [Information Storage and Retrieval] Information Storage—*file organisation*; H.3.3 [Information Storage and Retrieval] Information Search and retrieval—*search process*;

Keywords: full-text retrieval, information retrieval, index compression, inverted file, query processing.

1 Introduction

Text databases are widely used as information repositories and can contain vast quantities of data. Two main mechanisms for retrieving documents from these databases are in general use: Boolean queries and informal ranked queries. A Boolean query—a set of query terms connected by the logical operators AND, OR, and NOT—can be used to identify the documents containing a given combination of terms, and is similar to the kind of query used on relational tables [30]. Ranking, on the other hand, is a process of matching an informal query to the

*This paper includes material presented in preliminary form at the 1994 Australian Database Conference and at the 1994 IEEE Conference on Data Engineering.

†Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia; alistair@cs.mu.oz.au

‡Department of Computer Science, RMIT, GPO Box 2476V, Melbourne, Victoria 3001, Australia.

documents and allocating scores to documents according to their degree of similarity to the query [29, 30].

A standard mechanism for supporting Boolean queries is an inverted file [7, 11]. An inverted file contains, for each term that appears anywhere in the database, a list of the numbers of the documents containing that term. To process a query, a vocabulary is used to map each query term to the address of its inverted list; the inverted lists are read from disk; and the lists are merged, taking the intersection of the sets of document numbers for AND operations, the union for OR, and the complement for NOT. For example, if the inverted lists for the three terms "index", "compression", and "algorithm" are

$$\begin{aligned} I_{\text{“index”}} &= \langle 5, 8, 12, 13, 15, 18, 23, 28, 29, 40, 60 \rangle \\ I_{\text{“compression”}} &= \langle 10, 11, 12, 13, 28, 29, 30, 36, 60, 62, 70 \rangle \\ I_{\text{“algorithm”}} &= \langle 13, 44, 48, 51, 55, 60, 93 \rangle , \end{aligned}$$

then the answers to their conjunction are documents 13 and 60. Note that conjunctive queries—those in which the terms are connected by AND operators—are, in general, much more useful than disjunctive OR queries. This is a natural consequence of the fact that the database is large: a typical term occurs in thousands of documents, and conjunction means that the set of answers is smaller than any of the inverted lists for the query terms whereas disjunction means that documents containing any of the query terms are answers.

Ranking techniques can also be supported by inverted files. When the documents are stored in a database that is indexed by an inverted file several additional structures must be used if evaluation is to be fast [3, 18, 31]. These include a weight for each word in the vocabulary; a weight for each document; and a set of accumulators, usually one for each document in the collection.

Compared with Boolean query evaluation, the principal costs of ranking are the space in random access memory, and the time required to process inverted lists. More memory is required because in a ranked query there are usually many candidates—that is, documents about which information must be kept because they are potential answers. In a conjunctive Boolean query, the number of candidates need never be greater than the frequency of the least common query term; whereas, in a ranked query, every document in which any of the query terms appears is normally regarded as a candidate, and is allocated an accumulator in which its score is accrued. More time is required because conjunctive Boolean queries typically have a small number of terms, perhaps 3–10, whereas ranked queries usually have far more. In a conjunctive Boolean query the answers lie in the intersection of the inverted lists, but in a ranked query, they lie in the union, and so adding more terms to a ranked query broadens the search rather than narrowing it. Adding terms also means that more disk accesses into the inverted file are required, and more time must be spent merging. Moreover, the larger number of terms in a ranked query, and the fact that ranked queries are often English text, means that long inverted lists must be scanned, since it is likely that at least some of the terms in a ranked query occur in many of the documents.

The high cost of processing inverted lists is exacerbated if, for space efficiency, the inverted lists are stored compressed. Without compression, an inverted file can easily be as large or larger than the text it indexes. Compression results in a net space reduction of as much as 80% of the inverted file size [1], but even with fast decompression—decoding at approximately 400,000 numbers per second on a Sun Sparc 10—it involves a substantial overhead on processing time.

Here we consider how to reduce these space and time costs, with particular emphasis on environments in which index compression has been used. We describe a mechanism for adding a small amount of information into each inverted list so that merging operations can, in most cases, be performed in time sublinear in the length of the lists being processed. This *self-indexing* strategy has been tested on a database containing almost two million "pages" of text totalling 2 Gb. For typical conjunctive Boolean queries of five to ten terms the query processing time is reduced by a factor of about five. Furthermore, the overhead in terms of storage space is small, typically under 25% of the inverted file, or less than 5% of the complete stored retrieval system.

For ranked queries, we show that by effectively switching from a disjunctive query to a conjunctive query at some predetermined point in the processing of terms, the number of candidates can be dramatically cut without adversely affecting retrieval effectiveness. We then show that self-indexing inverted files allow the time required to process ranked queries to be reduced by a factor of between two and four.

Section 2 describes the structures used by document databases, and describes the Boolean and ranked retrieval paradigms examined here. Our test data is described in Section 3. Section 4 introduces the notion of a self-indexing inverted file, and analyses the performance improvement produced. Experiments are given that show the efficacy of the method on Boolean queries. Section 5 discusses methods by which the space required for accumulators can be restricted, and shows how this restriction can, together with internal indexing in each inverted list, be used to improve query evaluation time for ranked queries. Conclusions are presented in Section 6. A table of mathematical symbols is provided at the end of the paper.

2 Document Databases

In an inverted file document database, each distinct word in the database is held in a *vocabulary* [3, 11, 18, 24, 25, 31]. The vocabulary entry for each word contains an address pointer to an *inverted list* (also known as a *postings list*), a contiguous list of the documents containing the word. Each document is known by a unique *identifier*, which we assume to be its ordinal number. To support efficient query processing the vocabulary should also hold for each term t the value f_t, the number of documents that contain t. Knowledge of the value f_t allows inverted lists to be processed in order of increasing frequency, which is crucial for the algorithms below.

The inverted list for each word stores a list of the documents d that contain that word, and this is sufficient to allow evaluation of Boolean queries. To support ranking, the "within-document frequency" $f_{d,t}$ is also stored with each document number d in each inverted list [18]. This allows the weight of each word in each document to be computed. In the absence of compression four bytes and two bytes respectively might be allocated for the d and $f_{d,t}$ values, that is, six bytes for each $\langle d, f_{d,t} \rangle$ pair. Using compression the space required can be reduced to about one byte per pair [1]. On the 2 Gb TREC collection, described below, these methods compress the inverted file from 1100 Mb to 184 Mb, an irresistible saving.

2.1 Compressing inverted files

Techniques for compressing inverted lists, or equivalently bitmaps, have been described by many authors, including Bell *et al.* [1], Bookstein, Klein, and Raita [2], Choueka, Fraenkel, and Klein [4], Fraenkel and Klein [12], Klein, Bookstein, and Deerwester [21], and Linoff and Stanfill [22]. Faloutsos described the application of similar techniques to the compression of sparse signatures [8, 9].

Our presentation is based on that of Moffat and Zobel [26], who compare a variety of index compression methods. To represent each inverted list, the series of differences between successive numbers is stored as a list of *run-lengths* or *d-gaps*. For example, the list

$$5, 8, 12, 13, 15, 18, 23, 28, 29, 40, 60$$

of document numbers d can be equally well stored as a list of d-gaps:

$$5, 3, 4, 1, 2, 3, 5, 5, 1, 11, 20.$$

One consequence of this representation is that small gaps are common, since frequent words must of necessity give rise to many small gaps. Hence, a variable-length encoding of the integers in which small values are stored more succinctly than long values can achieve a more economical overall representation than the more usual flat binary encoding.

Elias [6] described a family of "universal" codes for the positive integers that are at most a constant factor inefficient for any non-increasing probability distribution. His γ code represents integer x as $\lfloor \log_2 x \rfloor + 1$ in unary (that is, $\lfloor \log_2 x \rfloor$ 1-bits followed by a 0-bit) followed by $x - 2^{\lfloor \log_2 x \rfloor}$ in binary (that is, x less its most significant bit); the δ code uses γ to code $\lfloor \log_2 x \rfloor + 1$, followed by the same suffix. Some sample values of codes γ and δ are shown in Table 1; both codes use short codewords for small integers, and longer codewords for large numbers. In Table 1, commas have been used to separate the suffixes and prefixes; these are are indicative only, and are not part of the compressed string. The δ code is longer than the γ code for most values of x smaller than 15, but thereafter δ is never worse, and for integer x requires $\lfloor \log_2 x \rfloor + O(\log \log x)$ bits. The codes are both *prefix-free*—no codeword is a prefix of another—and so unambiguous decoding without backtracking is possible.

x	Coding method		
	Elias, γ	Elias, δ	Golomb, $b = 3$
1	0,	0,	0,0
2	10,0	100,0	0,10
3	10,1	100,1	0,11
4	110,00	101,00	10,0
5	110,01	101,01	10,10
6	110,10	101,10	10,11
7	110,11	101,11	110,0
8	1110,000	11000,000	110,10

Table 1: Examples of Codes

The γ and δ codes are instances of a more general coding paradigm as follows [12, 26]. Let V be a (possibly infinite) vector of positive integers v_i, where $\sum v_i \geq N$, the number of documents in the collection. To code integer $x \geq 1$ relative to V, find $k \geq 1$ such that

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^{k} v_j$$

and code k in some representation followed by the difference

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

in binary, using either $\lfloor \log_2 v_k \rfloor$ bits if $d < 2^{\lceil \log_2 v_k \rceil} - v_k$ or $\lceil \log_2 v_k \rceil$ bits otherwise. In this framework the γ code is an encoding relative to the vector $(1, 2, 4, 8, 16, \ldots)$, with k coded in unary.

Consider another example. Suppose that the coding vector is (for some reason) chosen to be $(9, 27, 81, \ldots)$. Then if k is coded in unary, the values 1 through to 7 would have codes "0,000" through to "0,110", with 8 and 9 as "0,1110" and "0,1111" respectively, where again the comma is purely indicative. Similarly, run-lengths of 10 through to 36 would be assigned codes with a "10" prefix and either a 4-bit or a 5-bit suffix: "0000" for 10 through to "0100" for 14, then "01010" for 15 through to "11111" for 36.

Golomb [15] and Gallager and Van Voorhis [14] also considered prefix-free encodings of the integers. They showed that coding relative to the vector

$$V_G = (b, b, b, b, \ldots)$$

for

$$b = \left\lceil \frac{\log(2-p)}{-\log(1-p)} \right\rceil,$$

generates an optimal set of prefix-free codes for the geometric distribution with parameter p. That is, if a term appears in each document independently with probability p, the probability

of a d-gap of length x is given by $(1-p)^{x-1}p$, and the Golomb code with parameter b is, in effect, a Huffman code for this infinite distribution. The final column of Table 1 shows a subset of the Golomb codes generated when $b = 3$. This is an optimal assignment of codewords when $0.1809 < p < 0.2451$ (approximately).

The effectiveness of compression for an inverted list varies with the choice of coding vector, and depends upon the extent to which the probability distribution implied by the vector differs from the "actual" distribution. In practice, term distribution is not random amongst documents, and so the Golomb code can be improved upon. Details of some alternative methods and experimental results may be found elsewhere [1, 26]. In most cases the improvement is relatively small, and in the remainder of this paper we assume that d-gaps in inverted lists are represented using a Golomb code, with the parameter b chosen appropriately for each inverted list.

The within-document frequencies $f_{d,t}$ stored in the inverted lists must also be coded; and Elias's γ code is a suitable method [1, 26].

2.2 Boolean query evaluation

Suppose that a conjunctive Boolean query is being processed. Each query term is located in the vocabulary, which might be resident in memory, if space is available, or might be on disk. In the latter case, one disk access per term is required. The next step is to sort the terms by increasing frequency, and all subsequent processing is carried out in this order. The inverted list for the least frequent term is then read into memory. This list establishes a set of *candidates*, documents that have not yet been eliminated and might be answers to the query.

The other terms are then processed. As each inverted list is read, each candidate remaining is checked off against that list. Unless a candidate appears in all lists it cannot be an answer, and so any candidate absent from any list can be eliminated. The set of candidates is thus non-increasing. When all inverted lists have been processed, the set of remaining candidates (if any) are the desired answers. This strategy is summarised in Figure 1.

There are two points to note about this evaluation method. The first point concerns the selection of the least frequent term at step 2. This is for efficiency. Suppose that term l is the least frequent, and that it appears in f_l documents. Then for a conjunctive query the set of candidates will never contain more than f_l entries and space usage can be minimised. Processing the remaining terms in increasing frequency order is a heuristic intended to quickly reduce the number of candidates to zero, at which point no further terms at all need be considered. A query for which the set of candidates reaches zero is not particularly informative; nevertheless, a surprising fraction of actual queries have exactly this result.

The second point concerns the process carried out at step 4b. In this step, each of a relatively small set of candidates is tested for membership in a comparatively long inverted list. Suppose at first that the inverted list is uncompressed. When $|C| \approx |I_t|$, the most efficient strategy is a linear merge, taking $O(|C| + |I_t|) = O(|I_t|)$ time. This is, however, the

1. For each query term t,
   (a) Search the vocabulary for t.
   (b) Record f_t and the address of I_t, the inverted list for t.
2. Identify the query term t with the smallest f_t.
3. Read the corresponding inverted list.
   Use it to initialise C, the list of candidates.
4. For each remaining term t,
   (a) Read the inverted list, I_t.
   (b) For each $d \in C$,
       if $d \notin I_t$ then
           $C \leftarrow C - \{d\}$.
   (c) If $|C| = 0$,
       return, since there are no answers.
5. For each $d \in C$,
   (a) Look up the address of document d.
   (b) Retrieve document d and present it to the user.

Figure 1: Evaluation of conjunctive Boolean queries

exceptional case. More normally, $|C| \ll |I_t|$, and it is far more efficient to perform a sequence of binary searches taking $O(|C| \log |I_t|)$ time, or even a sequence of fingered exponential and binary searches [19] taking time $O(|C| \log(|I_t|/|C|))$. Each inverted list must still be read from disk, and so the overall time to process term t is $O(|I_t| + |C| \log(|I_t|/|C|)) = O(|I_t|)$ and the cost of processing the entire query is $O(\sum_t |I_t|)$; nevertheless, substantial CPU savings can be achieved.

Binary search is only possible if the document numbers are in sorted order within list I_t, and if they can be accessed in an array-like manner. Unfortunately, the use of compression destroys random access capabilities, since the resulting non-uniform lengths in bits make it impossible to jump into the middle of a compressed inverted list and decode a document number. This means that if the inverted file is compressed, not only must a linear merge be used irrespective of the length of the inverted list, but each inverted list must be fully decompressed in order to do so. The cost is still $O(\sum_t |I_t|)$, but the constant factor is large. At face value, then, the use of compression saves a great deal of space in the inverted file, but imposes a substantial time penalty during conjunctive query processing. Reducing this overhead is the problem addressed in this paper.

2.3 The cosine measure

Another important retrieval paradigm is *ranking*, in which each document is assigned a numeric score indicating similarity with the query, and then the documents that score the highest are displayed as answers. The ranking technique we use in this paper is the cosine

measure [29, 30]. It estimates 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* (or "importance") of word t in document or query x. The expression

$$W_x = \sqrt{\sum_t w_{x,t}^2}$$

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. The answers to a ranked query q are the r documents with the highest $C_d = cosine(q,d)$ values, for some predetermined bound r.

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

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

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 function allots high weights to rare words, on the assumption that these words are more discriminating than common words; that is, the presence of a rare word in both document and query is assumed to be a good indicator of relevance.

The cosine measure is just one method that can be used to perform ranking, and there are many others—see, for example, Harman [17] or Salton [29] for descriptions 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.

2.4 Ranked query evaluation

The usual method for determining which of the documents in a collection have a high cosine measure with respect to a query is to compute *cosine* from the inverted file structure and document lengths [3, 17, 18, 24]. In this method, an *accumulator* variable A_d is created for each document d containing any of the words in the query, in which the result of the expression $\sum_t w_{q,t} \cdot w_{d,t}$ is accrued as inverted lists are processed. A simple form of this query evaluation algorithm is shown in Figure 2. Note that the partial ordering required by step 4 can be performed efficiently using a priority queue data structure such as a heap [5], and there is no need for the set of C_d values to be completely ordered.

The evaluation technique in Figure 2 supposes that the document lengths—the values W_d—have been precalculated. They are query invariant, and so for efficiency should be computed at database creation time. The effect of applying them is to reorder the ranking, sometimes significantly. Moreover, there are usually many documents for which $A_d > 0$ at

1. For each document d in the collection, set accumulator A_d to zero.
2. For each term t in the query,
   (a) Retrieve I_t, the inverted list for t.
   (b) For each $\langle$document number d, word frequency $f_{d,t}\rangle$ pair in I_t, set $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
3. For each document d, calculate $C_d \leftarrow A_d/W_d$, where W_d is the length of document d and C_d the final value of $cosine(q, d)$.
4. Identify the r highest values of C_d, where r is the number of records to be presented to the user.
5. For each document d so selected,
   (a) Look up the address of document d.
   (b) Retrieve document d and present it to the user.

Figure 2: Algorithm for computing *cosine* and returning r answers

the completion of step 2, since any documents listed in any of the inverted lists have this property. This means that the number of accesses to the document lengths is usually many times greater than the number of answers; and if the document lengths are stored on disk, these accesses could become the dominant cost of answering queries.

One technique that has been suggested for avoiding this bottleneck is to store, in the inverted lists, not the raw $f_{d,t}$ values described above, but instead scaled values $f_{d,t}/W_d$ [3, 24]. Such scaling is, however, incompatible with index compression: it does reduce memory requirements, but this reduction comes at the cost of a substantial growth in the size of the inverted file.

A better method is to use low-precision approximations to the document weights, which can reduce each document length to around six bits without significantly affecting retrieval effectiveness or retrieval time [27]. Furthermore, in a multi-user environment the cost of storing the weights can be amortised over all active processes, since the weights are static and can be stored in shared memory.

Use of these techniques leaves the accumulators A_d as the dominant demand on main memory. They cannot be moved to disk, since they are built up in a random-access manner; they cannot conveniently be compacted into fewer bits, because of the processor time required by the necessary transformations and the large number of times the transformation must be carried out; and they cannot be shared, since they are query specific. We shall return to this problem—how best to represent the accumulators—in Section 5 below.

3 Test Data

The database used for the experiments reported in this paper is TREC, a large collection of articles on finance, science, and technology that were selected as test data for an ongoing international experiment on information retrieval techniques for large text collections [16].

These articles vary in length from around a hundred bytes to over two megabytes; as part of other current research, we have broken the longer documents into *pages* of around 1,000 bytes, to ensure that retrieved text is always of a size that can be digested by the user. It is the problems encountered with the large number of resulting records that prompted the research in this paper. In the paged form of TREC there are 1,743,848 records totalling 2054.5 Mb; an average of 191.4 words per record; and 538,244 distinct words, after folding all letters to lowercase and removal of variant endings using Lovin's stemming algorithm [23]. The index comprises 195,935,531 stored $\langle d, f_{d,t} \rangle$ pairs.

3.1 Boolean queries

To measure the time taken by Boolean operations, a set of 25 lists of terms was constructed, each list containing 50 words. To construct each list, a page of the collection was selected at random. The words of the page were then case-folded and duplicates were removed, as were a set of 601 frequently occurring *stopwords*—that is, words such as "also" and "because", which can in most contexts be ignored because they have low information content. The remaining list of words was then counted, and lists containing fewer than 50 terms were replaced.

embattled systems vendor prime computer natick mass purchase debts wholly owned subsidiary cad cam vision bedford principal amount debentures 110 million presently convertible 333 33 bonds officials made time open market represents attractive investment current prices fending hostile takeover bid mai basic tustin calif week lost battle district court boston

expanded memory equipment suppose computer 640k ram runs finish building worksheet solve problem install board lets work data fit dos limit ideal solution boards expensive intel corp 800 538 3373 oreg 503 629 7369 2 megabytes costs 1 445 configured software buy 3 releases 01 growing number programs release os

Figure 3: Sample Boolean queries

This mechanism generated queries with at least one guaranteed answer each—namely, the document from which the text was selected—and ensured that query processing could not terminate before all terms in any particular list had been inspected. Figure 3 shows the 50 words comprising the first two lists, in the order in which they appeared in the text. We wanted 50-term lists, not because we believe that 50-term queries are typical, but because any subset of these lists would have at least one answer; that is, they allowed us to test performance on queries with any number of terms from 1 to 50. For example, the 4-term queries corresponding to Figure 3 were "embattled AND systems AND vendor AND prime", and "expanded AND memory AND equipment AND suppose".

Table 2 shows the number of answers and the number of inverted file pointers considered during the processing of the queries, for some of the query sizes used in the experiments. Queries of perhaps 3–10 terms are the norm for general purpose retrieval systems. Note

that we regarded an answer to be a document rather than a page of the document and that any particular document might have more than one matching page; this is the reason, for example, why the 72,000 pointers considered in the single term queries resulted in only 51,000 answers.

Number of terms	Average answers	Average $\sum_t \lvert I_t \rvert$
1	51,450	72,323
2	4,780	168,257
4	58.7	293,256
8	1.16	510,361
16	1.00	942,837
32	1.00	1,857,513

Table 2: Processing of sample Boolean queries

On average each term used in these queries appeared in about 60,000 pages of the collection. Thus, by the time four terms of any query have been considered, on average $|C| \approx 60$ and $|I_t| \approx 60{,}000$ at each execution of step 4b in Figure 1. These typical values will be used in the time estimates made in Section 4 below.

3.2 Ranked queries

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 TREC data is suitable for experiments with ranking both because of its size and because test queries and relevance judgements are available. The test queries are 50 "topics", or statements of interest. For each of these topics, some thousands of likely documents have been manually inspected to yield relevance judgements. A sample topic is shown in Figure 4.

Retrieval effectiveness, or measurement of ranking performance, is usually based on *recall* (the proportion of relevant documents that have been retrieved) and *precision* (the proportion of retrieved documents that are relevant) [30]. For example, if for some query there are known to be 76 relevant documents, and some query evaluation mechanism has retrieved 100 documents of which 26 are relevant, the precision is $26/100 = 26\%$ and the recall is $26/76 = 34\%$. In this paper we have calculated retrieval effectiveness as an eleven point average, in which the precision is averaged at 0%, 10%, ..., 100% recall; because of the size of the TREC collection, the recall-precision is based upon the top 200 retrieved documents rather than on a total ranking. This was the methodology employed during the initial TREC experiment [16], and we have chosen to continue with this convention.

From each of the TREC topics we extracted two sets of query terms. To create the first set we removed all non-alphabetic characters, and case-folded and stemmed the resulting words. This gave a set of 50 queries containing, on average, 124.2 terms, 64.6 distinct terms,

Domain: International Economics

Topic: Rail Strikes

Description: Document will predict or anticipate a rail strike or report an ongoing rail strike.

Narrative: A relevant document will either report an impending rail strike, describing the conditions which may lead to a strike, or will provide an update on an ongoing strike. To be relevant, the document will identify the location of the strike or potential strike. For an impending strike, the document will report the status of negotiations, contract talks, etc. to enable an assessment of the probability of a strike. For an ongoing strike, the document will report the length of the strike to the current date and the status of negotiations or mediation.

Concept(s):

1. rail strike, picket, stoppage, lockout, walkout, wildcat
2. rail union, negotiator, railroad, federal conciliator, brotherhood
3. union proposal, talks, settlement, featherbedding, cost cutting
4. working without a contract, expired contract, cooling off period

Figure 4: Sample test topic

and involving, on average, 21,600,000 of the $\langle d, f_{d,t} \rangle$ pointer pairs, or 330,000 pairs per term per query.

The second set of query terms was constructed from the first by eliminating stopwords. This query set had, on average, 42.4 distinct terms per query; 3,221,000 pointer pairs processed per query; and 76,000 pairs per term per query. The "stopped" query generated from the topic shown in Figure 4 is shown in Figure 5. The superscripts indicate multiplicity; because the queries are small documents in themselves, we allowed multiple appearances of terms to influence the weighting given to that term.

document^5 predict^1 anticipate^1 rail^5 strike^{12} report^4 ongoing^3 relevant^2 impending^2 describing^1 conditions^1 lead^1 provide^1 update^1 identify^1 location^1 potential^1 status^2 negotiations^2 contract^3 talks^2 enable^1 assessment^1 probability^1 length^1 current^1 date^1 mediation^1 picket^1 stoppage^1 lockout^1 walkout^1 wildcat^1 union^2 negotiator^1 railroad^1 federal^1 conciliator^1 brotherhood^1 proposal^1 settlement^1 featherbedding^1 cost^1 cutting^1 working^1 expired^1 cooling^1 period^1

Figure 5: Sample ranked query

As is demonstrated below, the two query sets yield similar retrieval performance, and for the bulk of the results presented here we chose to use the second query set, because a stoplist would normally be applied to queries in production systems to minimise processing costs.

However we also experimented with the unstopped queries. There were two reasons for this. First, we have been interested in mechanisms for a smooth transition from words that are included to words that are excluded, rather than the abrupt transition given by stopping. For example, each of the words "american", "computer", "journal", and "washington" occur

in over 100,000 TREC pages; they could be stopped on the basis of frequency, but do provide a little discrimination and may be the key to effective retrieval for some queries. Second, using the unstopped queries to some extent mimics the performance of a database an order of magnitude larger in which stopwords are applied—the most frequent unstopped term in a twenty gigabyte collection would have about the same frequency as words such as "the" in the two gigabyte TREC collection. That is, we seek to demonstrate that our techniques are scalable.

For this reason we did not apply a stoplist while constructing the index; the sizes reported below are for an index that records every word and every number. The decision as to whether or not a stoplist is applied is made at query time, and in general should not be preempted by decisions made when the index is created.

4 Fast Inverted File Processing

Let us now consider the cost of evaluating conjunctive Boolean queries. The strategy described in Figure 1 is potentially slow because the inverted list of every query term is completely decoded.

Suppose that $k = |C|$ candidates are to be checked against an inverted list containing p pairs $\langle d, f_{d,t} \rangle$. Suppose further that it costs t_d seconds to decode one pair. Then, because of the cost of the compression and because random access binary searching is not possible, the total decoding cost T_d in processing this one inverted list is approximated by

$$T_d = t_d \, p,$$

assuming, pessimistically, that the pairs are randomly distributed in the list and that one pair occurs close to the end. The whole list must be decoded to access this last pair because, as described in Section 2, it is not possible to randomly access points in a compressed inverted list. That is, the first bit of the compressed list is, conventionally, the only point at which decoding can commence.

However, while every inverted list must be processed, not every $\langle d, f_{d,t} \rangle$ pair is required—in a conjunctive query all that is necessary is for each candidate to be checked for membership in the current inverted list. This observation allows processing time to be reduced in the following manner.

4.1 Skipping

When $k \ll p$, faster performance is possible if *synchronisation points*—additional locations at which decoding can commence—are introduced into the compressed inverted list. For example, suppose that p_1 synchronisation points are allowed. Then the *index* into the inverted list contains p_1 "document number, bit address" pairs, and can itself be stored as a compressed sequence of "difference in document number, difference in bit address"

runlengths. If these compressed values are interleaved with the runlengths of the list as a sequence of *skips*, a single self-indexing inverted list is created.

For example, consider the set of $\langle d, f_{d,t} \rangle$ pairs

$$\langle 5,1\rangle\langle 8,1\rangle\langle 12,2\rangle\langle 13,3\rangle\langle 15,1\rangle\langle 18,1\rangle\langle 23,2\rangle\langle 28,1\rangle\langle 29,1\rangle \ldots \quad .$$

Stored as d-gaps, these are represented as

$$\langle 5,1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,1\rangle\langle 5,2\rangle\langle 5,1\rangle\langle 1,1\rangle \ldots \quad .$$

With skips over (say) every three pointers, the inverted list becomes a sequence of *blocks* of three pairs each, with skips separating the blocks. The example list corresponds to

$$\langle\!\langle 5,a_2\rangle\!\rangle\langle 5,1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle\!\langle 13,a_3\rangle\!\rangle\langle 1,3\rangle\langle 2,1\rangle\langle 3,1\rangle\langle\!\langle 23,a_4\rangle\!\rangle\langle 5,2\rangle\langle 5,1\rangle\langle 1,1\rangle\langle\!\langle 40,a_5\rangle\!\rangle \ldots \quad ,$$

where a_2 is the address of the first bit of the second skip pair, a_3 is the address of the first bit of the third skip, and so on. This format still contains redundancy, in that both the list of document numbers in the skips and the list of bit addresses can be coded as differences, and the first document number in each set of three $\langle d, f_{d,t} \rangle$ values is now unnecessary. Incorporating these changes, the final inverted list becomes

$$\langle\!\langle 5,a_2\rangle\!\rangle\langle 1\rangle\langle 3,1\rangle\langle 4,2\rangle\langle\!\langle 8,a_3-a_2\rangle\!\rangle\langle 3\rangle\langle 2,1\rangle\langle 3,1\rangle\langle\!\langle 10,a_4-a_3\rangle\!\rangle\langle 2\rangle\langle 5,1\rangle\langle 1,1\rangle\langle\!\langle 17,a_5-a_4\rangle\!\rangle \ldots \quad .$$

To access the compressed list to see if document d appears, the first skip is decoded to obtain the address a_2 of the second skip, which is also decoded. If the document numbers d_1 and d_2 implied by these skips are such that $d_1 \leq d < d_2$, then if d appears it is in this first block, and only that block need be decoded. If $d_2 \leq d$, the second skip is traced to locate the third, and enough information is on hand to decide whether d lies in the second block. In this case the first block need never be decoded, and some processing time has been saved, at the expense of increasing the size of inverted list.

Section 2 describes several methods by which the two values coded into each skip can be represented. The Golomb code is particularly well suited for coding the skips, as all of the blocks in the compressed inverted list are roughly the same length. In the results given below, a Golomb code is used for both the inverted lists and the two components of the skips inserted into each.

4.2 Analysis

The benefits of skipping are estimated as follows. If there are $k = |C|$ document numbers to be checked against an inverted list of $p = f_t = |I_t|$ pointers, then on average half of each of k blocks will need to be decoded, one half-block for each candidate document d. It is also likely that almost all of the skips will need to be decoded, since they themselves constitute a file of variable length compressed records, a miniature of the original problem. Allowing two units of decoding time for each skip processed (each of the two values stored

in a skip occupies about the same number of bits as a $\langle d, f_{d,t} \rangle$ pair), and using the notation introduced above,

$$T_d = t_d \left(2p_1 + \frac{kp}{2p_1} \right) .$$

This is minimised when

$$p_1 = \frac{\sqrt{kp}}{2} ,$$

resulting in

$$T_d = 2t_d\sqrt{kp} = O\left(\sqrt{|C| \cdot |I_t|} \right) .$$

Taking sample values of $k = 60$, $p = 60{,}000$, and $t_d = 2.5 \times 10^{-6}$ seconds (the last being the measured rate of decoding on a Sun Sparc 10 Model 512), the use of indexing is estimated to reduce the decoding and searching time from 0.150 seconds without any skipping to 0.009 seconds with skipping.

The saving in processing time does not, however, come without cost. In the same example, $p/p_1 \approx 63$, and so the inverted list grows by roughly two pointers for every 63 pairs, a 3% overhead. (If anything, this accounting is pessimistic, since one of the pointers is partially compensated for by the document number that is saved in the block; it is, however, a convenient approximation.) Before claiming the usefulness of the technique, we should confirm that the cost of reading this extra data into memory does not outweigh the CPU saving. Let t_r denote the cost of reading one $\langle d, f_{d,t} \rangle$ pair into memory as part of a bulk read. Then the total elapsed time T required to search one inverted list is given by

$$T = t_d \left(2p_1 + \frac{kp}{2p_1} \right) + t_r \left(p + 2p_1 \right) ,$$

which is minimised at

$$p_1 = \frac{\sqrt{kp/(1 + t_r/t_d)}}{2} .$$

Assuming that $t_r = 0.5 \times 10^{-6}$ seconds, which at one byte per pointer corresponds to a transfer rate of about 2 Mb per second, and including the cost of reading the inverted list into memory, the time taken to process the 60 candidates can be reduced from 0.180 seconds to approximately 0.040 seconds.

The time to process a self-indexing compressed inverted list also compares well with the time required to perform the same operations on an uncompressed inverted list. In this case T_d is effectively zero, since the inverted list can be binary searched very quickly for document numbers. However, at six bytes per pointer pair (four for the document number d, and two for $f_{d,t}$, the within-document frequency), $t_r = 3 \times 10^{-6}$ seconds, so that just reading an inverted list with $p = 60{,}000$ entries requires $T_r = 0.180$ seconds. This can be reduced to 0.120 seconds if the within-document frequencies $f_{d,t}$ are dispensed with and only Boolean queries supported, but even so is greater than the time calculated above. Inserting skips and compressing inverted lists allows both disk space and query processing time to be reduced.

All of these times are exclusive of the cost of searching the lexicon for the term, and of seeking in the inverted file to the location of the desired inverted list. At typical CPU

and disk speeds these operations might add a further 0.020 seconds for each term in the query. Here again compression serves to improve performance: the compressed inverted file is about one sixth the size of an uncompressed equivalent, and so average seek times are less.

Scaling these calculations to the average situation encountered in the Boolean TREC queries, a query of 10 terms might be expected to require 0.6 seconds to identify a list of answer documents using a skipped inverted file. On the other hand, an unskipped compressed index might take as long as 2 seconds; and an uncompressed index would take a similar length of time.

For comparison, it is also interesting to estimate the performance of another indexing method advocated for conjunctive Boolean queries—the bitsliced signature file [10, 28]. In this case at least 10 bitslices of 212 Kb each (one slice per query term, each of one bit per document in the collection) must be fetched and conjoined. Transfer time alone accounts for more than a second. The comparison becomes even more decisive on queries involving fewer terms. The inverted file index becomes faster, because fewer terms and lists must be fetched. But with a signature file index some minimal number of bitslices must always be processed to reduce false match rates to an acceptable level, usually in the range 6–12. Moreover, a signature file index is typically several times larger than a compressed inverted file, even after the insertion of skips.

Multi-level signature file organisations reduce processing time by forming "super" signatures for blocks of records, so that record signatures for a block are investigated only if all query terms appear somewhere in the block [20, 28]. While they reduce the amount of data transferred from disk during query evaluation, these methods do not reduce the size of the index. Nor do they address the other drawbacks of signature files: the need to check for false matches; the difficulties presented by long records that set a high proportion of the bits in their signatures; and the lack of support for ranked queries.

4.3 Additional levels of skipping

Given that the insertion of skips reduces query processing time by up to 60–80%, an obvious extension is to allow indexing into the list of skips—that is, to apply the same solution recursively.

Consider a second level of skipping, where p_2 synchronisation points are provided into the list of p_1 records. Making the same assumptions as above,

$$T_d = t_d \left(2p_2 + \frac{kp_1}{p_2} + \frac{kp}{2p_1} \right),$$

which is minimised when

$$\begin{aligned} p_2 &= (1/2)k^{2/3}p^{1/3} \\ p_1 &= (1/2)k^{1/3}p^{2/3} \end{aligned}$$

yielding

$$T_d = 3t_d\, k^{2/3} p^{1/3},$$

which, when $k = 60$ and $p = 60,000$, is 0.004 seconds, a further saving of 0.005 seconds. In this case the index contains $p + 2p_1 + 2p_2$ pointers, which is a 25% overhead on the T_r time, and makes the total time $T = 0.037$ seconds, a slight additional gain.

In the limit, with h levels of skipping and $p_1, p_2, \ldots, p_h$ synchronisation points,

$$T_d = t_d \left(2p_h + \sum_{i=1}^{h-1} \frac{kp_i}{p_{i+1}} + \frac{kp}{2p_1} \right)$$

which is minimised by

$$p_i = (1/2) k^{i/(h+1)} p^{(h-i+1)/(h+1)}$$

with minimal value

$$T_d = t_d (h+1) k^{h/(h+1)} p^{1/(h+1)}.$$

This latter expression, when considered as a function of h, is itself minimised when

$$h = \left(\log_e \frac{p}{k} \right) - 1.$$

Using this "minimal CPU" set of values, the total processing time is

$$T = t_d (h+1) k^{h/(h+1)} p^{1/(h+1)} + t_r \left(p + 2 \sum_{i=1}^{h} p_i \right).$$

For the same example values of p and k, the CPU cost is minimised when $h = 6$ and there are six levels of skips; the nominal CPU time decreases to $T_d = 0.003$ seconds, but the total number of skips at all levels is over 17,000, an overhead of nearly 60% on the inverted list. The increase in reading time absorbs all of the CPU gain, and with $h = 6$ the total cost climbs back up to 0.051 seconds.

Table 3 shows calculated costs of various levels of skipping using this model. The first row shows the cost if neither compression nor skipping are employed.

h	$p + 2\sum_{i=1}^{h} p_i$	T_d	T_r	T
—	60,000	≈ 0	0.180	0.180
0	60,000	0.150	0.030	0.180
1	61,898	0.009	0.031	0.040
2	66,600	0.004	0.033	0.037
3	72,906	0.003	0.036	0.039
4	80,046	0.003	0.040	0.043
5	87,662	0.003	0.044	0.047
6	95,560	0.003	0.048	0.051

Table 3: Predicted processing time in seconds, $p = 60{,}000$ and $k = 60$

Other overheads associated with multi-level skipped indexes mean that there is little likelihood of improvement beyond that obtained with one level of skipping.

4.4 Implementation

Since the exact value of $k = |C|$ that will be used while processing any given inverted list depends the terms in the query and is highly variable, it is appropriate to make a simple approximation. With this in mind, several different inverted files were built with skips inserted into each list assuming that k had some fixed value for all of the terms in the query. Let L be the value of k for which the index is constructed. The assumed values of L, and the resultant size of each index, are shown in Table 4. In all cases the d-gaps were coded using a Golomb code; the $f_{d,t}$ values with a γ code; and both components of the skips with a Golomb code, as described in Section 2. To ensure that inverted files did not become too large, we imposed a minimum blocksize, requiring that that every skip span at least four document numbers.

Parameter	Size	
	Mb	%
No skipping	184.36	100
$L = 1$	186.14	101
$L = 10$	188.95	102
$L = 100$	194.74	106
$L = 1{,}000$	205.38	111
$L = 10{,}000$	220.33	120
$L = 100{,}000$	230.21	125

Table 4: Size of skipped inverted files

Small values of k are likely to occur for terms processed late in a Boolean query, and so the $L = 1$ inverted file should yield the best performance on queries with many terms. On the other hand, short queries will have high values of k, and so the $L = 10{,}000$ index should perform well for queries of just a few terms. There is, of course, no gain from skipping if the query consists of a single term.

The most expensive regime—the use of variable skips of as little as four pointers with $L = 100{,}000$—increases the inverted file size by about 25%. However, the original inverted file without skipping is compressed to less than 10% of the actual text being indexed, and in this context the space cost of skipping is still small.

4.5 Performance on Boolean queries

The time taken to process the sample Boolean queries, for unskipped indexes and skipped indexes constructed with L equal to 1, 100, and 10,000, is shown in Figure 6. Each point is the average of five runs on an otherwise idle Sun Sparc 10 Model 512 using local disks, and is the CPU time taken from when the query is issued until the list of answer document numbers is finalised. It does not include the time taken to retrieve and display answers. The different curves show the time spent decoding inverted lists, for each of the skipped inverted

files and for an unskipped index. For a 5-term query, for example, the $L = 1{,}000$ skipped index requires the least processor time, at about 20% of the cost of the unskipped inverted file. As expected, when more terms are added to the queries the smaller values of L become more economical overall, and for 20-term queries the $L = 1$ index requires less than 10% of the CPU effort of the original index.

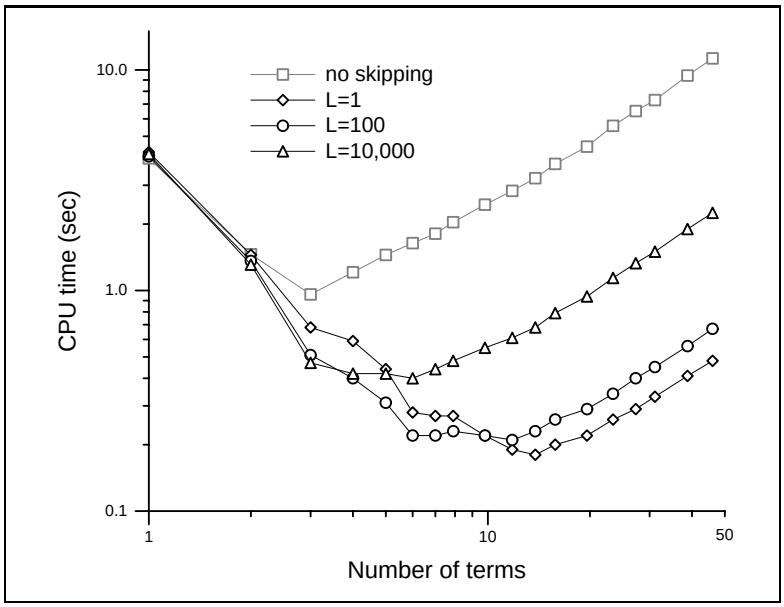

Figure 6: CPU time required to process Boolean queries

The elapsed time taken for queries shows the same trend, but with a common overhead. Table 5 lists, for the same experiments, elapsed time from moment of query issue until a list of answer documents has been calculated. Note the somewhat smaller savings, caused by the fact that, irrespective of the skipping regime being used, all of every inverted list must still be read into memory. Even so, for a wide range of query lengths skipping allows answers to be located 4 to 6 times faster than if there is no skipping. Any value $100 \le L \le 10,000$ is appropriate for typical queries.

Number of terms	Unskipped	Skipped		
		$L = 10{,}000$	$L = 100$	$L = 1$
2	1.48	1.33	1.37	1.46
4	1.22	0.42	0.42	0.61
8	2.03	0.50	0.25	0.28
16	3.77	0.82	0.29	0.23
32	7.33	1.54	0.46	0.31

Table 5: Elapsed time required to process Boolean queries (seconds)

As can be seen from Figure 6, single-term queries are expensive because of computations that must be done on a "per answer" basis, and which dominate the cost of inverted list

processing when the number of answers is large. The most expensive of these operations in our paged database is resolving a list of paragraph numbers into a list of unique document numbers. For example, in the non-paged version of TREC just 0.2 seconds is required to determine the list of 47,052 answer documents that contain the term "expanded"; but 4.7 seconds is required for the same query in the corresponding paged TREC. The difference is the cost of resolving 60,610 page numbers into the same final list of 47,052 document numbers. Queries with only a small number of answers are, of course, spared this cost, and this is why the curves in Figure 6 initially decrease.

Another way to include skipping in an index is to vary the L parameter for each inverted list, rather than use the same value for the entire inverted file. Because the terms in any query are processed in increasing f_t order, there might be some advantage to supposing that the number of candidates is large when f_t is small, and small when f_t is large. There would, however, still be guesswork in attempting to predict a best value of L for each inverted list. Furthermore, Figure 6 and Table 5 show that, within a broad range of query sizes, the time taken is relatively insensitive to the exact value of L used.

4.6 General Boolean queries

Conjunctive Boolean queries are by no means the only type of query supported by retrieval systems. More generally, queries can be formed as a conjunction of disjunctions such as

("data" OR "text" OR "index") AND
("compression" OR "compaction") AND
("strategy" OR "algorithm" OR "process" OR "method").

Skipping can also be used to speed these queries. The inverted lists for all of the terms in each disjunction must be simultaneously skipped through, and the candidate allowed to remain if it appears in any of them. The initial set of candidates should be formed by fully resolving one of the disjunctions; again, the one likely to result in the smallest result should be chosen. One rule for doing this is to suppose that there is no overlap between the terms in any disjunction, and so choose the disjunction containing the smallest total number of term appearances. As subsequent conjuncts are considered, each term in the conjunct is processed with the same value of k, and so savings similar to those demonstrated above can be expected.

5 Ranked Queries

Ranked queries are more like disjunctive Boolean queries than conjunctive queries, in that any document containing any of the terms is considered as a candidate. Nevertheless, it is possible to exploit skipping to reduce the time taken by ranked queries. The crucial observation is that it is possible to change from disjunctive to conjunctive mode for frequent terms without impacting retrieval effectiveness. This change, the time savings that result,

and the effect the change has on memory requirements of ranked query evaluation, are considered in this section.

5.1 Reduced-memory ranking

Consider again the ranking process outlined in Figure 2. The dominant demand on memory space is the set of accumulators A in which the cosine contributions are built up. They could be stored in an array that has one element for each document in the database, and this is the usual method described in the information retrieval literature [3, 17, 18]. Alternatively, if the number of non-zero accumulators is small compared to the number of documents stored, a dynamic data structure such as a balanced search tree or a hash table can be employed to store the set of accumulators [5]. In this case the document identifier for each non-zero accumulator must also be stored, so that the set can be searched, together with pointers or other structural information. In total, as many as sixteen to twenty bytes of memory might be consumed for each non-zero accumulator, compared with four if an array is used. Nevertheless, provided that at most 20–25% of the documents have $A_d > 0$ there is a net saving in space compared to storage in an array. For the TREC queries about 75% of the documents have non-zero accumulators (even after the removal of stop words), and so this change is not sensible. This means that memory space during query evaluation can be a significant problem; for example, using an array the paged TREC consumes over 7 Mb of random-access memory for accumulators. What is needed is some heuristic for limiting the number of non-zero accumulators so that a dynamic accumulator structure can be employed.

One simple strategy for restricting the number of accumulators is to order query terms by decreasing weight, and only process terms until some designated stopping condition is met. Figure 7 shows a modified ranking process in which no more terms are processed after the number of non-zero accumulators exceeds an *a priori* bound K. We designate this strategy as *quit*. Other possible stopping conditions that could be used at step 3c of this algorithm would be to place a limit on the number of terms considered or on the total number of pointers decoded; or to place an upper bound on the term frequency f_t, and only process terms that appear in fewer than x% of the documents, for some predetermined value x.

Quitting has the advantage of providing a short-circuit to the processing of inverted lists and hence faster ranking, but at the possible expense of poor retrieval performance, depending upon how discriminating the low weighted terms are.

An alternative to the *quit* strategy is to continue processing inverted lists after the bound on the number of accumulators is reached, but allow no new documents into the accumulator set. This *continue* algorithm is illustrated in Figure 8. Both *quit* and *continue* generate the same set of approximately K candidate documents, but in a different permutation, so when the top r documents are extracted from this set and returned, different retrieval effectiveness can be expected.

The *continue* algorithm has two distinct phases. In the first phase, accumulators are added freely, as in the *quit* algorithm. This phase is similar to evaluation of an OR query,

1. Order the words in the query from highest weight to lowest.
2. Set $A \leftarrow \emptyset$; A is the current set of accumulators.
3. For each term t in the query,
   (a) Retrieve I_t, the inverted list for t.
   (b) For each $\langle d, f_{d,t} \rangle$ pair in I_t,
      i. If $A_d \in A$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
      ii. Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
   (c) If $|A| > K$, go to step 4.
4. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
5. Identify the r highest values of C_d.

Figure 7: *Quit* algorithm for computing *cosine* using approximately K accumulators

and the processing of each inverted list is a disjunctive merge of the list with the structure of accumulators. In the second phase, existing accumulator values are updated but no new accumulators are added. This phase is more akin to evaluation of an AND query, as some (perhaps most) of the identifiers in each inverted list are discarded. In this phase the set of accumulators is static, and it is cheaper at step 4b to traverse the set in document number order, comparing it against each inverted list, than it is to search the previous dynamic structure looking for each inverted file pointer.

Figure 9 shows result of experiments with the *quit* and *continue* algorithms. It plots retrieval effectiveness as a function of k, the number of accumulators actually used. In each case k is slightly greater than the target value K, as an integral number of inverted lists is processed for each query. The values shown against every third point in this graph are the average number of terms processed to yield that volume of accumulators; for example, only 8.2 terms are needed to generate an average of 27,000 accumulators. The difference between *quit* and *continue* is marked, and, perhaps surprisingly, even the mid to low weight terms appear to contribute to the effectiveness of the cosine rule—ignoring them leads to significantly poorer retrieval.

Also surprising is that the *continue* strategy, with restricted numbers of accumulators, is capable of better retrieval performance than the original method of Figure 2 in which all documents are permitted accumulators. In fact, retrieval effectiveness peaks when the number of accumulators is only 1% of the number of documents, at which point an average of just eight terms per query have been processed and allowed to create accumulators. It appears that the mid to low weight terms, while contributing to retrieval effectiveness, should not be permitted to select documents that contain none of the more highly weighted terms.

Suppose that a small percentage of the documents in a collection are permitted accumulators during ranked query evaluation. A few infrequent terms will be processed in disjunctive mode, but then all remaining terms can be processed in conjunctive mode. Hence, skipping can again be employed to improve response time. In the case of the TREC queries,

1. Order the words in the query from highest weight to lowest.
2. Set $A \leftarrow \emptyset$.
3. For each term t in the query,
   (a) Retrieve I_t.
   (b) For each $\langle d, f_{d,t} \rangle$ pair in I_t,
      i. If $A_d \in A$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
      ii. Otherwise, set $A \leftarrow A + \{A_d\}$, calculate $A_d \leftarrow w_{q,t} \cdot w_{d,t}$.
   (c) If $|A| > K$, go to step 4.
4. For each remaining term t in the query,
   (a) Retrieve I_t.
   (b) For each d such that $A_d \in A$,
   if $\langle d, f_{d,t} \rangle \in I_t$, calculate $A_d \leftarrow A_d + w_{q,t} \cdot w_{d,t}$.
5. For each document d such that $A_d \in A$, calculate $C_d \leftarrow A_d / W_d$.
6. Identify the r highest values of C_d.

Figure 8: *Continue* algorithm for computing *cosine* using approximately K accumulators

$p = f_t = 75{,}000$ was observed, and $K = 10{,}000$ might be chosen—in our experiments this resulted in an average retrieval figure as good as the full cosine method. Using these values, skipping is estimated to reduced the CPU time from 0.188 seconds per term to 0.137 seconds, making the same assumptions as before. Hence, on a query of 42 terms, total CPU time is predicted to decrease from 7.9 seconds to about 5.8 seconds. The saving is less dramatic than for Boolean queries, but nevertheless of value.

5.2 Experimental results

The experiments illustrated in Figure 9 show that values K greater than about 0.2% of N give good retrieval effectiveness, where N is the number of documents in the collection and K is the accumulator target of Figure 8. Hence, we might assume for typical TREC queries that an index constructed with $L = 10{,}000$ is appropriate for general use. The value of K used to control the query can be set differently for each request processed, but the inverted file must be built based upon some advance supposition about L. For Boolean queries the value of k must be guessed in advance, but has no effect upon the correctness of the set of answers. In contrast to this, for ranked queries it is possible to make an accurate estimate of k—it is just a little greater than K—but the effect on retrieval accuracy cannot be exactly predicted.

Table 4 in Section 4 shows the sizes of the inverted files that were generated for different values of L. These indexes were also used with the *continue* algorithm of Figure 8, to answer the 50 queries constructed from the TREC topics. Values of K both close to and widely differing from the target value L for which each index was constructed were tested. The results of these experiments are plotted in Figure 10. The times shown are CPU-seconds,

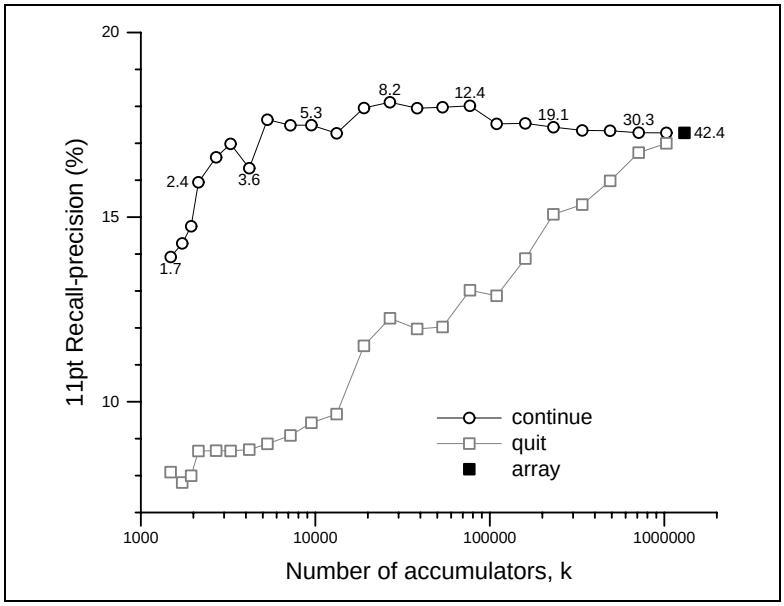

Figure 9: *Quit* vs. *continue*: retrieval effectiveness

measured from the time the query is issued until a list of the top $r = 200$ documents has been determined, and includes all components of the ranking process. As before, it does not include the time required to locate, fetch, and present those documents.

As can be seen from Figure 10, the predictions as to running time were accurate. In the best situation, when the number of accumulators is small and the inverted file has been constructed for a small number of accumulators, the time has been reduced from about 9 seconds to a little over 2 seconds, a substantial saving. For more conservative values of k the running time is halved. The analysis has also correctly predicted that time varies with the blocksize, and that each skipping regime performs best when $k \approx L$. For example, the $L = 1{,}000$ inverted file gives the best performance when the number of accumulators is small.

We also ran experiments on the same set of queries, but without excluding the stopwords. Some representative performance figures comparing stopped and unstopped queries are shown in Table 6. In both restricted accumulator cases the inverted file used was self-indexing and constructed with $L = 10{,}000$, and a hash table was used for the set A. The "unlimited" results followed the description of Figure 2, and used an array to maintain the set of accumulators, with all terms selecting candidates and participating in the final ranking. As can be seen, retrieval performance is largely unaffected by the stopping of high frequency terms and limitations on the number of accumulators, but processing time improves with both heuristics. The column headed "Pointers decoded" records the total number of compressed numbers processed, where each $\langle d, f_{d,t} \rangle$ pair is counted as 1, and each skip as 2. These bear a close relationship to the CPU times listed in the final column. The column "Terms processed" shows the average number of terms per query in the unlimited case, and

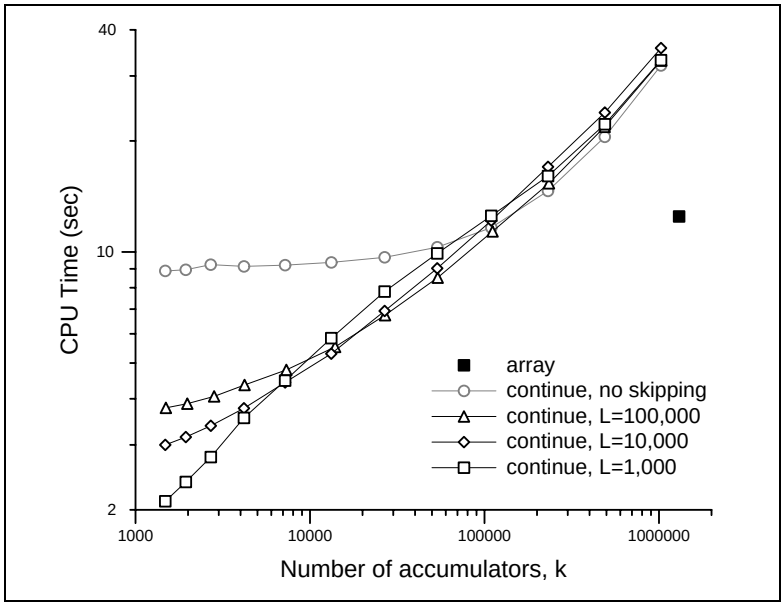

Figure 10: CPU time required to process ranked queries

the average number of terms per query processed in conjunctive mode and disjunctive mode for the *continue* method.

Query type	Terms processed	Eleven point effectiveness (%)	Actual accumulators k	Pointers decoded	CPU time (sec)
Stopped					
K = unlimited	42.4	17.3	1,304,115	3,131,050	12.5
K = 10,000	6.1+36.3	17.3	13,238	1,617,275	5.6
Unstopped					
K = unlimited	64.6	17.4	1,733,517	19,955,961	45.9
K = 10,000	6.2+58.4	17.4	13,558	5,142,752	16.1

Table 6: Stopped vs. unstopped queries, index constructed with $L = 10{,}000$

As expected, the amount of processing time saved through the use of skipping is much more when the query is not already stopped. This is because the stop words include those that are the most frequent, and it is on frequent terms that the greatest savings can be achieved. The unstopped TREC queries averaged 64.6 terms, and over 300,000 document numbers per term per query. With $k = 10{,}000$ the analysis of Section 2 predicts that an unskipped compressed index requires 0.750 seconds; and a skipped index 0.274 seconds. For a 65-term query these estimates correspond to about 49 seconds and 18 seconds respectively. The results of Table 6 validate these predictions. The slightly better than expected performance improvement arises because of two factors. First, the model takes no account of the amount of calculation required, and the restricted accumulators implementation performs

only a fraction of the floating point operations of the full cosine method. Second, the inverted lists are of widely differing length, and the savings are disproportionately greater on lists containing many pointers.

For ranked queries there is no advantage in using more than one level of skipping. Table 7 summarises the expected performance for ranked queries (in the same format as previously used in Table 3) for the two situations already considered: when $k = 10{,}000$ and $p = 75{,}000$ (stopped queries), and when $k = 10{,}000$ and $p = 300{,}000$ (unstopped queries).

h	$p + 2\sum_{i=1}^{h} p_i$	T_d	T_r	T
0	75,000	0.188	0.038	0.226
1	102,386	0.137	0.057	0.188
2	132,890	0.147	0.066	0.213
3	164,254	0.165	0.082	0.247

(a) $k = 10{,}000$ and $p = 75{,}000$

0	300,000	0.750	0.150	0.900
1	354,772	0.274	0.177	0.451
2	427,620	0.233	0.214	0.447
3	506,362	0.234	0.253	0.487

(b) $k = 10{,}000$ and $p = 300{,}000$

Table 7: Predicted processing time in seconds for terms in a ranked query

If the same index is to be used for both Boolean and ranked queries then a compromise parameter must be chosen at the time the index is constructed, since the value of k typical of Boolean queries is much less than the usual range for ranked queries. For Boolean queries of 5–10 terms and ranked queries of 40–50 terms, Figures 6 and 10 show that a value such as $L = 1{,}000$ is a reasonable compromise. Alternatively, if speed on both types of query is at a premium, two different indexes might be constructed. This is not as extravagant as it may at first appear—recall that the indexes are compressed, and each occupies just 10% of the space of the text being indexed. Two compressed skipped inverted files together still consume less than half of the space required by a single uncompressed inverted index or signature file.

6 Conclusions

We have shown that the memory and CPU time required for querying document collections held on disk can be substantially reduced. Our techniques are of particular importance when large, static, collections are being distributed on relatively slow read-only media such as CD-ROM. In these situations, when database access is to 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 and processing time.

The normal cost of the use of compression, which results in a massive saving of space, is increased processing time. However, by introducing skipping into the inverted lists, in effect making them self-indexing, substantial time savings also accrue. For Boolean queries skipped and compressed inverted lists allow both space and time savings of the order of 80% when compared with uncompressed indexes.

We have also shown that ranking can be effected in substantially less memory than previous techniques. The saving in memory space derives from the observation that the number of non-zero accumulators can be safely held at a small percentage of the number of documents. Based on this observation we have described a simple rule that allows the memory required by the document accumulators—the partial similarities—to be bounded. This "restricted accumulators" method then opens the way for self-indexing inverted files to be employed to speed ranked queries. Time savings of about 50% can be achieved without measurable degradation in retrieval effectiveness.

In combination, the methods we have described allow three important resources—memory space, disk space, and processing time—to be simultaneously reduced.

Acknowledgements

We would like to thank Neil Sharman, who undertook the bulk of the implementation effort. This work was supported by the Australian Research Council, the Collaborative Information Technology Research Institute, and the Centre for Intelligent Decision Systems.

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] A. Bookstein, S.T. Klein, and T. Raita. Model based concordance compression. In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 82–91, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.

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

[4] Y. Choueka, A.S. Fraenkel, and S.T. Klein. Compression of concordances in full-text retrieval systems. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 597–612, Grenoble, France, June 1988. ACM Press, New York.

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

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

[7] R. Elmasre and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummins, USA, 1989.

[8] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, 1985.

[9] C. Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 63–82, Montreal, Canada, June 1985. ACM Press, New York.

[10] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[11] E.A. Fox, D.K. Harman, R. Baeza-Yates, and W.C. Lee. Inverted files. In Frakes and Baeza-Yates [13], chapter 3, pages 28–43.

[12] A.S. Fraenkel and S.T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183, Berlin, 1985. Springer-Verlag.

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

[14] R.G. Gallager and D.C. Van Voorhis. Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, IT–21(2):228–230, March 1975.

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

[16] D.K. Harman, editor. *Proc. TREC Text Retrieval Conference*, Gaithersburg, Maryland, November 1992. National Institute of Standards Special Publication 500-207.

[17] D.K. Harman. Ranking algorithms. In Frakes and Baeza-Yates [13], chapter 14, pages 363–392.

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

[19] F.K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[20] A.J. Kent, R. Sacks-Davis, and K. Ramamohanarao. A signature file scheme based on multiple organisations for indexing very large text databases. *Journal of the American Society for Information Science*, 41(7):508–534, 1990.

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

[22] G. Linoff and C. Stanfill. Compression of indexes with full positional information in very large text databases. In R. Korfhage, E. Rasmussen, and P. Willett, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 88–97, Pittsburg, June 1993. ACM Press, New York.

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

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

[25] K.J. McDonell. An inverted index implementation. *Computer Journal*, 20(1):116–123, 1977.

[26] 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.

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

[28] R. Sacks-Davis, A.J. Kent, and K. Ramamohanarao. Multi-key access methods based on superimposed coding techniques. *ACM Transactions on Database Systems*, 12(4):655–696, 1987.

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

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

[31] 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.

Symbols used

Symbol	Meaning
a_i	bit address of ith skip
A	set of accumulators
A_d	accumulator for document d
C	set of candidates
C_d	*cosine* for document d
d	document identifier
f_t	number of documents containing term t
$f_{x,t}$	frequency of t in document or query x
h	number of skip levels
I_t	inverted list for term t
k	number of candidates or accumulators
K	accumulator target

Symbol	Meaning
L	skipping parameter for index
N	number of documents
p	number of pointer pairs
p_i	number of pairs at ith skip level
q	query
r	number of answers
t	term
t_d	time to decode $\langle d, f_{d,t}\rangle$
t_r	time to transfer $\langle d, f_{d,t}\rangle$
T	total time
T_d	total decode time
T_r	total transfer time
$w_{d,t}$	weight of t in document d
W_d	weight of document d