Economical Inversion of Large Text Files

Alistair Moffat The University of Melbourne

ABSTRACT: To provide keyword-based access to a large text file it is usually necessary to *invert* the file and create an *inverted index* that stores, for each word in the file, the paragraph or sentence numbers in which that word occurs. Inverting a large file using traditional techniques may take as much temporary disk space as is occupied by the file itself, and consume a great deal of cpu time. Here we describe an alternative technique for inverting large text files that requires only a nominal amount of temporary disk storage, instead building the inverted index in compressed form in main memory. A program implementing this approach has created a paragraph level index of a 132 Mbyte collection of legal documents using 13 Mbyte of main memory; 500 Kbyte of temporary disk storage; and approximately 45 cpu-minutes on a Sun SPARCstation 2.

1. Introduction

Full-text databases are an important way of storing and accessing information. Newspaper archives, office automation systems, and on-line help facilities are but a few of the many applications. One common method of providing the index needed for efficient keyword-based query processing on such a database is to create an *inverted file* — a file that contains, for every term, a list of all documents that contain that term. Given an inverted file it is then straightforward to identify the documents that contain any boolean combination of the queried terms.

Here we address the task of creating the inverted file. We call this the *inversion* of the input text. When the input text is small, inversion is simple. All that is necessary is a single pass over the input data building a *lexicon* (also referred to as a *vocabulary*) of the distinct terms as they appear, and recording, at this first, and each subsequent occurrence, the document number. At the end of the input this in-memory structure is traversed and written to disk.

The limitation of this approach is main memory capacity. Even if we are prepared to (bravely) assume that the database will contain fewer than 65,536 documents, linked list storage of word occurrences will require 6 bytes per word, about the same amount as originally consumed by the word in the input text. For this approach to be viable, main memory must be as large as the text that is being inverted. On current workstations this poses no problems for databases of up to 1–5 Megabyte or so. However we are also interested in significantly larger databases—10 Mbyte, 100 Mbyte, and, perhaps, 1,000 Mbyte.

When main memory is insufficient secondary storage must be used. Moreover, a different approach is required, since secondary storage cannot be used in the same random access manner as primary memory. For example, attempting to invert a file of 1,000,000 words using linked lists on disk (which is what will happen if virtual memory is being mapped, via page faults, into a physical main memory that is too small) would require perhaps 2,000,000 head seeks, and, at 20 msec per seek, might take 12 hours or more of constant disk activity. It is much more economical to write a file of 'word-number, document-number' pairs in document number order, and then sort by word-number.

The drawback of this sort-based approach is the use of large amounts of temporary disk space. The same example file of 1,000,000 words would generate an intermediate file of about 8 Mbyte, coding the numbers as 32-bit integers; and then, during its several passes over the data, the sort would require another 8 Mbyte of temporary storage space. In total, 16 Mbyte of secondary storage capacity must be allocated to invert a file that was probably originally only 5–6 Mbyte, and, if stored compressed, actually requires only 2–3 Mbyte. If inversion is an operation that is periodically carried out as the database is extended, this peak load disk requirement must effectively become the 'normal' amount of secondary storage allocated to the database. Indeed, both Lesk [4] and Somogyi [8] have specifically warned users of their inverted file retrieval systems that large amounts of temporary disk space will be necessary. Inverting by sorting is also somewhat slow. Multiple passes over the data are required, with every item involved in every pass.

Here we describe an alternative technique for in-memory inversion of large text files. Our method stores document numbers in primary memory, but compressed in a bit-vector rather than as a linked list of integers. We have applied our technique to a database of 132.1 Mbyte containing 23,100,786 words and 261,829 documents, where each document corresponded loosely to a paragraph of text. The inversion required a total of 45 cpu minutes on a Sun SPARCstation 2; 13 Mbyte of main memory; and about 500 Kbyte of temporary disk storage. More generally, within any given amount of main memory we can invert (at paragraph level) databases approximately 10 times larger than can be handled by linked list techniques.

In the next section we describe a simple prefix code for representing positive integers, and give bounds on the total number of output bits generated under certain conditions. Section 3 then describes the practical application of the code to the problem of inverting a text file. Sections 4 and 5 detail the results of experiments using the inversion technique; and Section 6 describes a number of possible directions in which the method might be extended.

2. *Coding Positive Integers*

At the heart of the new technique is a simple code for representing positive integers. Suppose that $x \geq 1$ is a value to be stored, and that b is some positive power of 2 (the choice of b will be discussed below). To store x we first code $((x - 1) \text{ div } b)$ 1-bits, then a 0-bit, and then $((x - 1) \bmod b)$ in binary using $\log_2 b$ bits. Some sample codes for small values of x and various values of b are shown in Table 1. To make the table a little easier to understand the boundary between the unary and binary section within each codeword has been indicated with a comma. This comma does not, of course, appear in the actual output code.

x	$b = 1$	$b = 2$	$b = 4$	$b = 8$
1	0,	0,0	0,00	0,000
2	10,	0,1	0,01	0,001
3	110,	10,0	0,10	0,010
4	1110,	10,1	10,11	0,011
5	11110,	110,0	10,00	0,100
6	111110,	110,1	10,01	0,101
7	1111110,	1110,0	10,10	0,110
8	11111110,	1110,1	10,11	0,111
9	111111110,	11110,0	110,00	10,000

Table 1: Examples of block codes

The encoding process is simple, and C source code is shown in Figure 1. It is assumed the routine 'PUTBIT' disposes (somehow) of one bit of the code being generated; and that 'logbase2(b)' returns $\log_2 b$. The decoder is similarly straightforward.

This prefix code is a special case of a more general code first described by Golomb [3], and then further investigated by Gallager Van Voorhis [2]; McIlroy [6]; and Moffat & Zobel [7]. In the applications considered by those authors b was not restricted to be a power of two. Here the special case when b is a power of two is of particular interest.

```
encode(x, b) {
        x = x-1;
        while (x>=b) {
                PUTBIT(1);
                x = x-b;
        }
        PUTBIT(0);
        for (i=logbase2(b)-1; i>=0; i=i-1)
                PUTBIT((x>>i) & 0x1);
}
```

Figure 1: Encoding Process

Suppose now that a sequence of values $X = (x_i)$, $1 \leq i \leq p$ is to be coded, and that $\Sigma_{i=1}^{p} x_i \leq N$. The behaviour we are interested in is captured by the following lemmas.

Lemma 1 *The total number of bits B_w required to code X is bounded by*

$$B_w(b) \leq p \cdot (1 + \log_2 b) + \frac{N - p}{b} \qquad (1)$$

Proof Each of the values coded will contain $\log_2 b$ bits for the binary component, and one more 0-bit to mark the transition between prefix and suffix parts. Each prefix 1-bit corresponds to exactly b, and since each value $x_i \geq 1$ and the total value coded by prefix bits is at most $(N - p)$, the number of prefix bits is at most $(N - p)/b$.

Lemma 2 *For fixed values $1 \leq p \leq N/2$, (1) is minimised (for b a power of two) at b_w, where*

$$b_w = 2^{\left\lfloor \log_2 \frac{N-p}{p} \right\rfloor}, \qquad (2)$$

and it is assumed that any ties are to broken by taking the largest minimising value.

Proof The proof is by contradiction. Note that (2) implies

$$\frac{b_w}{2} \leq \frac{N - p}{2p} < b_w \leq \frac{N - p}{p} < 2b_w.$$

Suppose first that $b \leq (N - p)/2p$ is the minimising value. Consider $b' = 2b$. The value of (1) will increase by p in the first term if b is replaced by b', but will decrease by $(N - p)/2b$ in the second.

This latter is not less than p, because $b \leq (N - p)/2p$. Thus $B_w(b') \leq B_w(b)$, and b is not the largest minimising value for (1).

Similarly, suppose that $b > (N - p)/p$ is the minimising value. Taking $b' = b/2$ instead makes the first term of 1 decrease by p, and the second increase by $(N - p)/b$. In this case the increase is strictly less than p, and again b cannot be the minimising value. The only remaining possibility is that b_w is, as required, the largest minimising value.

When $p > N/2$ the minimising value is $b_w = 1$.

We should also justify the claim that it is sufficient to restrict b to be a power of 2. In general (for unrestricted choice of b) the binary component of the code will require either $[\log_2 b]$ bits, for $0 \leq (x - 1) \bmod b < 2^{[\log_2 b]} - b$, and $[\log_2 b]$ bits for $2^{[\log_2 b]} - b \leq (x - 1) \bmod b < b$. For example, if $b = 6$ the codes for remainders of 0 to 5 will be 00, 01, 100, 101, 110, and 111, corresponding respectively to $x_i = 6k + 1, 6k + 2, \ldots, 6k + 6$ for integral k.

To show that choosing b a power of two minimises the total cost of the coding we suppose that we face an *adversary* who informs us of the values of N and p, asks us for our choice of b, and then decides on particular values x_i intended to force the code to consume the maximal number of bits. Suppose further that we choose a value of b that is not a power of 2. To maximise the number of bits while minimising the 'cost' in terms of x_i, the adversary will choose $x_i = 2^{[\log_2 b]} - b + 1 \pmod b$ as often as possible, since, compared with choosing $x_i = 1 \pmod b$, one extra bit is consumed for a cost of less than b; indeed, at a cost less even than $2^{[\log_2 b]}$. We can thus restrict the adversary's power to force extra bits of code by choosing $b' = 2^{[\log_2 b]}$, in which case the adversary must choose $x_i = 1 \pmod b$ as often as possible.

This argument is based upon the worst that can happen, and does not imply that every list will be most economically coded by choosing b a power of two. However, as we shall see in the next section, it is this worst case bound that allows the new inversion strategy. For applications where an average case bound might be more accurate, such as storing the inverted file on disk, other choices of b might be appropriate [7].

3. *Application to Inversion*

The inversion process consists of two passes over the text to be inverted.

The first pass is a standard 'word frequency' program that scans the database and records N, the number of documents, and, for each word w, the number of documents p_w that the word appears in. At the end of the input the value of N, together with all of the pairs 'w, p_w' are written to an intermediate lexicon file on secondary storage. At the same time as it is accumulating values p_w the first pass might also be accumulating other statistics about the input text. For example, the first pass we have been using also calculates Huffman codes for a word-based compression model [7].

The second pass reads this lexicon and builds a search structure. In this case, a sorted array is sufficient, since no insertions will be required and a linked structure would require extra space for pointers. For each word w the second pass calculates b_w according to Lemma 2; and B_w according to Lemma 1. A bit-vector of B_w bits (in reality, $\lceil B_w/8 \rceil$ bytes) is dynamically allocated, and initialised to zero.

The text is then processed again. The entry for each word is located by binary search in the lexicon and a code for the *gap* between this occurrence and the most recent previous occurrence of the word is appended to the corresponding bit-vector. By coding the gaps rather than absolute document numbers we can be sure that the conditions of Lemmas 1 and 2 will be met: namely, that the sum of the p values coded not exceed N, and so we can be sure that B_w bits will be sufficient. At the end of input for the second pass the array is scanned in lexicographic order, either decompressing each bit-vector back to a sequence of absolute pointers, or, more economically, writing the compressed bit-vector to be decompressed when required. In this latter case either p or b_w would also need to be written to the output file.

4. *Experimental Results*

We have tested this technique on three collections of documents. Table 2 shows the sizes of these collections. Database *Manuals* was

	Collection Name *Manuals*	*GNUbib*	*Comact*
Text Size (Mbyte)	5.15	14.12	132.11
Distinct Words	27,554	70,866	68,074
Word Occurrences	958,744	2,575,411	23,100,786
Documents	2,496	64,344	261,829
Average Words per Document	384	40	88

Table 2: Sizes of Document Collections

a collection of Unix manual pages[1], including embedded formatting commands. Database *GNUbib*[2] was included to give a comparison with previous work—Somogyi [8] has described the difficulty of inverting *GNUbib* using a sort-based approach. The database stores 64,000 citations to journal articles, technical reports, conference papers, and books, all stored in 'refer' format [5, 9]. Each citation was taken to be a 'document' for inversion purposes. It was the desire to invert the third database that was the initiator of this investigation; it is a collection of 261,829 pages of legal text storing the complete Commonwealth Acts of Australia, from federation (in 1901) to 1990. We wished to build an inverted index for *Comact,* but did not have available the more than 200 Mbyte of temporary disk storage that would have been required by a disk-based inversion.

Table 3 shows the various time and space costs of inverting these databases when run on a Sun SPARCstation 2. The 'first pass processor times' reported in the second section of the table include the cost of counting not only the frequency of all of the words, but also the frequency of all of the intervening non-words, *and* the cost of building a Huffman code for each of these two sets of frequencies suitable for use in a word-based compression module [7]. Profiling indicated that removal of these calculations to produce a 'suitable for inverting only' first pass would reduce the running time by about 20%. The second pass times include the cost of 'decompressing' the compressed bit-vectors to generate an output stream of 32-bit integers. If the inverted file is to be used as part of a compressed full-text retrieval system [7]

1 /usr/man/man[1–8]/* on a Sun SPARCstation

2 GNUbib is available from the Free Software Foundation.

	Collection Name *Manuals*	*GNUbib*	*Comact*
Main Memory (Mbyte):			
— first pass data structures	0.94	2.37	2.34
— second pass data structures	0.65	1.70	1.62
— second pass bit-vectors	0.25	1.95	10.93
Memory Required	0.89	3.65	12.55
Processor Time (cpu-seconds):			
— first pass	67	230	1220
— second pass	53	174	1420
Time Required	120	404	2640
Secondary Storage (Mbyte):	0.20	0.53	0.50

Table 3: Inverting Document Collections

then clearly it is much more economical on storage to simply write the compressed bit-vectors and *not* decompress them; the decompression of a particular vector should be performed by the retrieval program if and when that bit-vector is required.

We indexed all words, where a 'word' was defined to be any non-empty string of up to 15 alphanumeric characters, provided that the maximum number of numeric characters was not greater than four. The latter constraint was added after initial experimentation showed that the page numbers of *Comact* ran in an unbroken run from 1 to 261,829 and produced an unpleasantly large number of 'words'. Strings of alphanumerics longer than 15 characters, or containing more than 4 numeric characters were broken, with the offending character marking the start of a new 'word'. We did not remove any *stop words,* and indexed even single character and single digit words. The removal of stop words would not have the same effect on memory requirements as for linked list approaches, since frequently occurring words are allocated short codes anyway; nevertheless, a non-trivial amount of storage might still be saved. For example, the bit-vector for 'the' on *Comact* was coded with $b = 1$, but still required 32 Kbyte.

The figures for pass one memory space and the lexicon space on secondary storage have been adjusted to remove all 'non-inversion' components, and are accurate for an 'inversion only' first pass. The actual combined first pass, building Huffman trees for the words and non-words, required about 30% more primary memory than is listed.

For *Comact* the second pass of the inversion required a main memory allocation of just 10% of the initial file size, and ran without fuss on the 48 Mbyte test workstation. The bound of Lemma 1 was very tight; of the 10.93 Mbyte allocated to bit-vectors for *Comact,* more than 99% was used. With most organisations typically operating at least one machine with 128 Mbyte or more we would expect to be able to carry out paragraph-level in-memory inversion for files of up to about 1 Gbyte.

5. Word and Byte Level Inversion

There will also be times when it is necessary to invert a file to generate, for each distinct word, a list of occurrences in terms of ordinal word numbers—*word level inversion*—or even in terms of byte offsets within the file—*byte level inversion.*

Our technique can also be used in these situations, with p becoming the number of occurrences of the word (rather than the number of documents that contain the word) and N becoming, respectively, the total number of words in the file, and the total size in bytes of the file.

The space requirements during the second pass are, however, much larger. There are two reasons for this. Firstly, there are more pointers to be stored. In a document level index multiple occurrences within a document of any particular word are represented with a single pointer. On *Comact* this effect meant that the 23,100,786 words corresponded to only 14,219,077 stored pointers. For a word or byte level index all words must, of necessity, correspond to a pointer, and so even if the pointers can be coded as compactly, the index must grow by at least this ratio.

The second reason the inverted index grows is that each stored pointer is more descriptive, identifying a value in a larger range, and so requiring on average more bits. For example, *Comact* has 261,829 documents, 23,100,786 words, and 138,524,314 bytes; and if pointers were simply coded as minimal binary codes, would require 18, 25, and 28 bits respectively. The code described in Section 2 automatically makes this adjustment as p and N change; and so we were ex-

	Collection Name		
	Manuals	*GNUbib*	*Comact*
Word Level Inversion:			
— second pass bit-vectors (Mbyte)	1.39	3.73	31.76
— second pass time (cpu-sec)	75	196	1740
Byte Level Inversion:			
— second pass bit-vectors (Mbyte)	1.67	4.53	38.88
— second pass time (cpu-sec)	81	206	1840

Table 4: Word and Byte Level Inversion

pecting a substantial increase in memory requirements for the compressed bit-vectors when we attempted word and byte level inversion.

Table 4 shows the increased resources required by this level of inversion. As expected, primary memory requirements increased significantly. Nevertheless, we were still able to invert *Comact* within the 48 Mbyte main memory of our test machine.

6. *Extending the Technique*

Other codes can also be used to compress the lists of pointers. Elias [1] describes a number of prefix codes for coding positive integers that have the necessary property that small integers are represented with short codes. For example, his γ code represents integer x by $\lfloor \log_2 x \rfloor$ 0-bits, followed by the binary representation of x (which must start with a 1-bit) in $1 + \lfloor \log_2 x \rfloor$ bits. If the appearances of a word are scattered randomly in the text and each gap is about N/p, γ coding the list of pointer differences will require approximately $2p \log_2(N/p)$, almost twice the bound given by Lemmas 1 and 2. On the other hand, if the appearances of a word are tightly clustered then the γ code will give the more succinct representation. In this case the first pass of the two pass method should count the exact number of bits required by the γ code rather than let the second pass estimate an upper bound based on the words occurrence count. Elias also described a δ code, in which γ (rather than unary) is used to code the value $\lfloor \log_2 x \rfloor$ indicating how many suffix bits should be decoded.

Table 5 shows the amount of main memory that would be required by γ and δ at the various inversion levels we have considered.

Inversion Level	*Manuals* γ	*Manuals* δ	*GNUbib* γ	*GNUbib* δ	*Comact* γ	*Comact* δ
Documents	0.26	0.25	1.73	1.52	8.63	8.19
Words	1.66	1.43	4.70	3.97	36.72	32.13
Bytes	2.22	1.81	6.23	5.14	50.64	42.12

Table 5: Coding using δ and γ (Mbyte)

On both *GNUbib* and *Comact* the δ code is better than the block code of Section 2 when inverting at document level, but the advantage is lost when inverting at word or byte level. The advantage of the block code is that it gives good compression on a variety of texts and indexing levels, and the space required can be bounded above given knowledge of p— often a value that is already being calculated for other uses by a multi-purpose first pass.

There is, however, one situation in which the use of a non-parameterised prefix code becomes necessary. Probably the biggest disadvantage of our inversion method is the need for two passes over the input data. If we are prepared to trade increased space for decreased time, we can consider a one pass approach, in which a fixed code is used to represent the gaps between pointers. In this case memory allocation must become more dynamic. The first appearance of a word installs it into the lexicon, and allocates a first quantum of space for the bit-vector, perhaps one 32-bit word. At subsequent appearances the space remaining in the bit-vector must be checked, and, if it is inadequate, a larger space allocated and the current contents of the bit-vector copied into the expanded area. The old bit-vector would then be returned to a pool of free space.

Provided that the sequence of bit-vector sizes is geometric (rather than arithmetic), the time spent copying bit-vectors will not dominate. For example, if the sequence of bit-vector sizes is 1 word, 2 words, 4 words, 8 words, and so on, the maximum number of times any particular bit is copied is less than 2, averaged over the bits contained in each bit-vector. The drawback of this dynamic approach is that too much space will always be allocated. If we assume that each bit-vector grows to fill on average half of its last extension, 33% more space will be required than by an equivalently coded two pass approach. Another possible sequence of bit-vector sizes is the Fibonacci numbers: 1, 2, 3, 5, 8, 13, which is, roughly speaking, a geometric sequence with a ratio of 1.62 between successive terms. Using the Fibonacci sequence reduces the overhead space to about 20%, but increases the bound on the number of times each item gets moved to 2.6, averaged over each bit-vector. Thus, for a realistic bound on the space needed by a one pass approach the values in Table 5 should be increased by at least 20%. This still results in an underestimate, since it assumes that every bit-vector that is relinquished because of expansion can be later reused; this will not be the case. Smaller multipliers on the geometric sequence continue to give tighter bounds on the space overhead, but continue to increase the running time. The savings from moving from two passes to a one pass approach will be quickly eroded. More importantly, peak memory usage will be increased, both through the overheads on the encoding, and the need for a dynamic data structure (rather than a static 'pointer-less' structure) to store the lexicon (in 'first pass data structures' Table 3).

7. Summary

We believe that our work will be useful to practitioners for two reasons. Firstly, we have described a paradigm for inverting large text files using main memory as a random access device storing compressed bit-vectors. Previous inversion techniques using secondary storage were unable to build inverted files in a random access fashion, and so were forced to use multi-pass sorting strategies and large amounts of disk work space. Main memory is still, of course, a more expensive resource than disk, and we do not claim to have replaced disk-based inversion techniques. Nevertheless, we have moved the boundary between the two strategies, and sort-based inversion will now only be needed for *very* large databases.

The second contribution is the particular encoding that we have used. The most important property of this code is that it is sensitive to word frequency, and so results in compact representations for frequently appearing words. Moreover, the sensitivity is captured by a single parameter. A Huffman code or similar might also have been used on each bit-vector, but because a Huffman code has many parameters, would have markedly increased both the disk space needed to store the lexicon and the memory space required during the two passes of the inversion process. Other encodings with a small number of parameters, such as Elias' δ code, or a single Huffman code for the entire inverted file, suffer from not being sensitive to the frequencies of individual words. The final advantage of the code is that it is possible to calculate a tight bound on the number of bits required to code a bit-vector, again given just the single parameter p; this means that the inversion can be performed in the second pass rather than needing a third (with the second counting the coded length of each bit-vector). For these reasons the code itself, independently of the application we have described, will also be of interest.

The disadvantage of our approach is the need for two passes. However in the application for which the inversion method was designed [7] we were *already* making two passes over the data, and so the extra cost was small. Moreover, when processing text on the large scale described here, taking extra time is only a minor irritation, but taking extra primary or secondary memory can make the difference between success and failure.

Acknowledgements

The author gratefully acknowledges the assistance of Justin Zobel, who participated in a number of helpful discussions during the project. This work was supported by the Australian Research Council.

References

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

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

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

M. Lesk. Grab—inverted indexes with low storage overhead. *Computing Systems,* 1(3):207–220, Summer 1988.

M. Lesk. Some applications of inverted indexes on the Unix system. In *Unix Programmers Manual, Volume 2A.* Bell Laboratories, 1988.

M. D. McIlroy. Development of a spelling list. *IEEE Transactions on Communications,* COM-30(1):91–99, January 1982.

A. M. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. Technical Report 91/21, Department of Computer Science, The University of Melbourne, Parkville 3052, Australia, November 1991.

Z. Somogyi. The Melbourne University bibliography system. Technical Report 90/3, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia, March 1990.

B. Tuthill. Refer—a bibliography system. In *Unix User's Manual Supplementary Documents.* 4.2 Berkeley Software Distribution, 1984.

[submitted Nov. 18, 1991; revised Jan. 24, 1992; accepted Feb. 12, 1992]