Information Retrieval, 3, 25–47, 2000

Binary Interpolative Coding for Effective Index Compression

ALISTAIR MOFFAT alistair@cs.mu.oz.au
LANG STUIVER
Department of Computer Science and Software Engineering, The University of Melbourne, 3010, Australia

Abstract. Information retrieval systems contain large volumes of text, and currently have typical sizes into the gigabyte range. Inverted indexes are one important method for providing search facilities into these collections, but unless compressed require a great deal of space. In this paper we introduce a new method for compressing inverted indexes that yields excellent compression, fast decoding, and exploits clustering—the tendency for words to appear relatively frequently in some parts of the collection and infrequently in others. We also describe two other quite separate applications for the same compression method: representing the MTF list positions generated by the Burrows-Wheeler Block Sorting transformation; and transmitting the codebook for semi-static block-based minimum-redundancy coding.

Keywords: index compression, context-based model, document database

1. Introduction

The prevalent form of index structure in modern information retrieval systems is the *inverted index* (Witten et al. 1999). An inverted index consists of two main structures: a set of *inverted lists*, which record, for each term t that appears in the collection, the documents that contain that term; and a *vocabulary*, which maps each term to the address of its inverted list. In simplest form the inverted list for a term t has the structure

$$\langle f_t; d_{t,1}, d_{t,2}, d_{t,3}, \ldots, d_{t,f_t} \rangle,$$

where the term appears (once or more) in f_t documents, namely $d_{t,1}$, $d_{t,2}$, and so on. Using such lists, Boolean queries involving the operations AND, OR, and NOT are executed by forming the intersection, union, and complement respectively of the sets of document numbers they contain.

Another important form of query is the *ranked* query, in which a similarity score between the query and each document in the collection is calculated, and then some number of top-scoring documents are presented to the user as the answers. In assigning a score to a document most similarity heuristics take into account three factors: a decreasing function of the term frequency f_t; a decreasing function of some estimate of the length of the document; and an increasing function of the number of times each of the query terms appears in that document. Hence, if ranked queries are to be supported, it is also necessary to store with each document pointer the number of times the term appears within that document, giving the inverted list the form

$$\langle f_t;\, (d_{t,1},\ f_{d_{t,1},t}),\, (d_{t,2},\ f_{d_{t,2},t}),\, \ldots,\, (d_{t,f_t},\ f_{d_{t,f_t},t})\rangle,$$

where $f_{d,t}$ is the frequency of term t in document d.

While inverted indexes provide the functionality required in an information retrieval system, it is not without cost. Stored naively—perhaps as a sequence of four-byte document numbers and two-byte within-document frequency values—the inverted index can potentially occupy as much space as the text that is being indexed, since each word in the text also occupies about six bytes on average. Even refining this estimate, and allowing for the fact that each pointer represents (say) two word occurrences on average does little to assuage concerns about space, since even an indexing overhead of 50% is daunting; particularly if the text of the information retrieval system is stored compressed (Zobel and Moffat 1995).

Fortunately, inverted lists are themselves highly compressible, and the space they occupy can be substantially reduced. In this paper we first survey previous proposals for index compression mechanisms, and summarise their attributes. We then focus on the issue of locality—the tendency for terms to appear in clusters in the collection rather than at random. Section 3 describes previous attempts to exploit clustering, and discusses their drawbacks; then in Section 4 we describe a new *binary interpolative code* that exploits clustering well, and results in smaller compressed indexes than alternative approaches. Section 5 provides an analysis of the method, and shows that at worst it is only slightly inferior to Golomb coding, and can be arbitrarily better. Section 6 shows the application of the coding scheme to some of the large text databases associated with the TREC project. Two refinements that yield slightly improved compression are described in Section 7, and a number of potential drawbacks discussed. The coding method also has other applications, and two of these are presented in Section 8.

2. Index compression

There has been considerable interest in index compression over a period of more than twenty years (Bell et al. 1993, Bookstein and Klein 1991, Bookstein et al. 1992, Choueka et al. 1988, Choueka et al. 1986, Fraenkel and Klein 1985, Jakobsson 1978, Klein et al. 1989, McIlroy 1982, Schuegraf 1976, Witten et al. 1992). A common theme in these index compression methods is the reduction of each inverted list to a sequence of d-gaps, differences between consecutive document identifiers when the inverted list is sorted by document number. For example, consider a term t that appears in seven documents in a collection, namely documents 3, 8, 9, 11, 12, 13, and 17. The corresponding d-gaps are 3, 5, 1, 2, 1, 1, and 4, a sequence with much greater regularity than the original document numbers. In particular, frequent terms automatically give rise to large numbers of small d-gap values, so variable-length codes that favour small d-gaps with short codewords can be expected to yield compact representations.

The problem is to choose a code. Fixed codes, such as Elias's C_γ and C_δ codes (Elias 1975), perform reasonably well, typically representing each d-gap in large collections in around 6.5 bits (Bell et al. 1993). The C_γ and C_δ codes have the advantage of requiring

Table 1. Coding an inverted list using C_γ, C_δ, and a Golomb code.

d-gaps	3	5	1	2	1	1	4
Elias, C_γ	101	11001	0	100	0	0	11000
Elias, C_δ	1001	10101	0	1000	0	0	10100
Golomb, $b = 2$	100	1100	00	01	00	00	101

no parameters, making them easy to apply. If additional parameters are used to guide the construction of the code, better compression can be obtained. For example, the family of Golomb codes (Golomb 1966) parameterised by b are minimum-redundancy codes for geometric distributions $Pr(x) = (1 - p)^{x-1}p$ whenever $b = \lceil -\log(2 - p)/\log(1 - p)\rceil$ (Gallager and Van Voorhis 1975). If a random choice of f_t out of N possible document numbers is assumed then Golomb coding with $p = f_t/N$ should give good compression of inverted files. This is indeed the case, and Golomb coding on a per pointer basis typically averages about 5.5 bits per document pointer, or approximately 10–15% of the size of the source text, with the exact ratio depending upon document length and other such factors (Bell et al. 1993, Witten et al. 1999). Table 1 shows the representations that would be achieved for the example set of d-gaps for the C_γ, C_δ and Golomb codes, assuming that the collection contains $N = 20$ documents in total. For details of these codes see, for example, Witten et al. (1999, Chapter 3). For this simple list the Golomb code, at eighteen bits, is the most economical.

Other compression methods exist that consistently yield better compression than Golomb coding, a gain that is possible because term appearances in documents are in fact not independant random events. The non-randomness is a consequence of the fact that, in most databases, records are appended in chronological sequence, and topics move into and out of favour. Figure 1 shows in a pictorial manner the distribution of the 1,798 documents containing the word *hurricane* in a document collection consisting of 243,000 articles

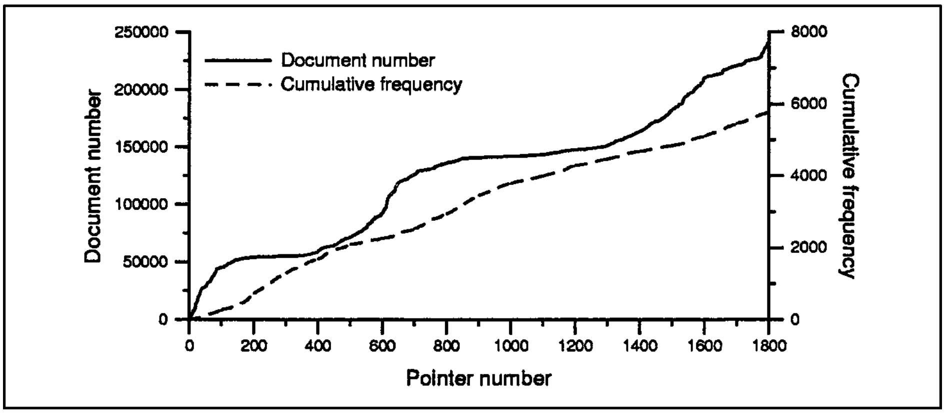


Figure 1. The 5,762 occurrences of "hurricane" in 1,798 documents of the 242,918 documents in the AP collection.

totalling 732 megabytes drawn from the Associated Press news service. The solid line shows the document number associated with the xth pointer in the inverted list, while the dashed line shows the xth cumulative sum $\sum_{i=1}^{x} f_{d_{t,i},t}$ of the within-document frequencies $f_{d,t}$ in the index list.

In figure 1 plateau regions on the solid line indicate clusters, since many pointers (on the horizontal axis) correspond to a relatively small range of document numbers (on the vertical axis). Conversely, regions of high gradient indicate the zones between clusters. The Bernoulli model encapsulated in the Golomb code assumes a straight line relationship; this is clearly not the case for this term.[1] Similar term clusters occur when collections are built by following explicit topic threads, as is the case for most web indexes.

Moffat and Zobel (1992) report experiments on a number of methods that skew the probability distribution in various ways so as to try and better capture the "true" distribution of d-gaps. Of the methods explored, the "batched LLRUN" mechanism—a modified form of the LLRUN technique of Fraenkel and Klein (1985)—worked consistently well. In the original LLRUN each d-gap x is coded in two components: first, a minimum-redundancy (Huffman) code for the value $\lfloor \log_2 x \rfloor$, with the codewords calculated according to the observed frequencies of all d-gaps in all inverted lists; and then a $\lfloor \log_2 x \rfloor$-bit binary number for the value $x - 2^{\lfloor \log_2 x \rfloor}$. That is, the LLRUN code is identical to the Elias C_γ code, except that a Huffman code is used to transmit $\lfloor \log_2 x \rfloor$ rather than a unary code. The "batching" process modifies the first part of each codeword, and rather than using a single Huffman table for all of the prefix parts, a set of Huffman codes is used, one code for each distinct value of $\lfloor \log_2 f_t \rfloor$, where f_t is the number of documents that contain the term. Improved compression results, since the Huffman codes for high frequency terms will favour small d-gap values, while those for low frequency terms will assign relatively long first-part codewords to the buckets that contain small d-gaps. The batched LLRUN technique is used as a baseline in the experiments reported below.

In this paper we re-examine the problem of index compression, and describe a surprisingly simple integer coding method—*binary interpolative coding*—that gives excellent performance when compressing the indexes of a range of large test databases; allows fast encoding and decoding; and is, even in the worst case, only slightly inferior to the proven robustness of Golomb coding.

3. Exploiting clustering

Bookstein et al. (1994, 1997) propose the use of a multi-state Markov model as a way of exploiting the opportunity presented by the tendency of terms to cluster. They suggest that the inverted list be thought of as a bitstring, with a 1-bit in the kth place indicating that a term appears in the kth document. That is, they suggest that the example inverted list $\langle 7; 3, 8, 9, 11, 12, 13, 17 \rangle$ (with $N = 20$) should be viewed as a bitstring `00100001101110001000`. This bitstring is then processed through a Markov model with three (or more) states, one state to indicate "definitely in a cluster", one to indicate "definitely not in a cluster", and the remainder indicating various degrees of uncertainty about whether or not the bitstring is currently in a cluster. As each bit of the bitstring is processed, it is coded relative to the probabilities for the current conditioning class, and then the appropriate transition performed. Bookstein et al. give results showing a small but discernible improvement in compression performance compared to previous methods.

There are, however, a number of drawbacks to their approach. The first is a pragmatic concern: in using a bitstring representation, they require the use of an arithmetic coder, which has the potential to substantially slow index decoding and adversely affect query response times in the information retrieval system supported by the index. Second, they require that a number of parameters be stored either implicitly or explicitly: transition probabilities out of each state must be maintained, and transmission of these as part of each inverted list might be a substantial overhead. To ameliorate this cost, Bookstein et al. suggest that the parameters be quantised into a number (four, say) of broad classes, and that a small number of bits (two, say) be added to each inverted list to select one of these classes. This quantisation detracts from the compression achieved, and means that considerable effort is needed at index creation time to choose the best set of parameters. The third and perhaps most telling problem is that clusters may not be apparent over just a small number of documents, and the use of a bitstring approach rather than d-gaps ignores this possibility. For example, consider again the distribution illustrated in figure 1. The two main clusters are quite pronounced (at pointers 200–400, and 900–1300), yet, of the several hundred document pointers within those two clusters, only about 50 correspond to d-gaps of one, and another 40 to d-gaps of two.

An alternative is to use conditioning classes based upon d-gaps rather than a binary bitstring. One or perhaps several small previous d-gaps is indicative of being in a cluster, so when this is observed the probability distribution used in the coding of the next d-gap can be biased in favour of small values. For example, Golomb codes might still be used, but with the parameter b used for each d-gap based upon a weighted average of the k previous d-gaps for this inverted list for some small value of k. One obvious risk with this method is that the penalty for accidentally coding a long d-gap with $b = 1$ is very high; some kind of exponential code (Teuhola 1978) should almost certainly be used rather than a strict Golomb code.

Another possible approach is to develop minimum-redundancy codes for quantised previous d-gaps: if the last d-gap is g, then $\lceil \log_2 g \rceil$ might be used to index an array of $\lceil \log_2 N \rceil$ minimum-redundancy codes.

We have explored both of these methods, and found that they do indeed offer improved index compression performance. However we have also discovered a quite different mechanism for index compression that exploits clustering implicitly, rather than explicitly, yet obtains even better compression rates.

4. Binary interpolative compression

Consider again the inverted list

$$\langle 7; 3, 8, 9, 11, 12, 13, 17 \rangle$$

with $N = 20$. Suppose that the value of the second pointer is somehow known before the first must be coded. In the example, if it is known that the second pointer is to document 8 then the first pointer must be restricted to some document number in the range 1 to 7

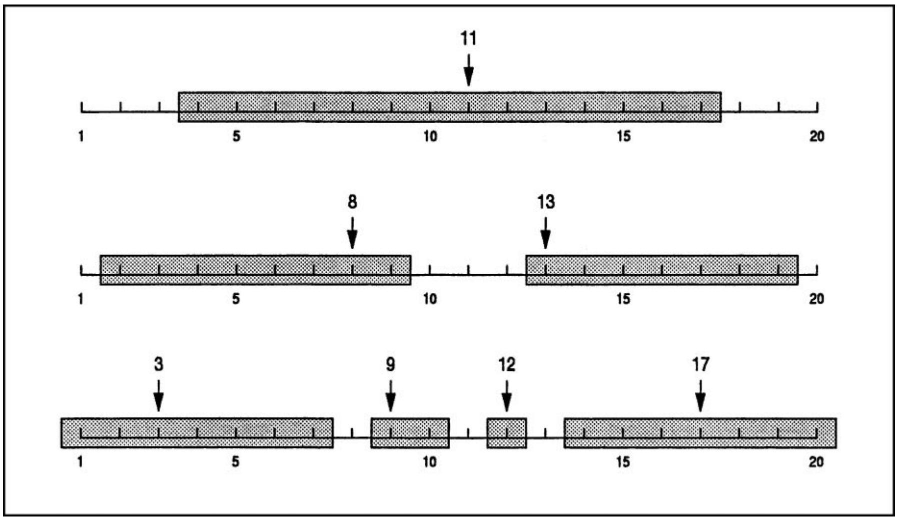


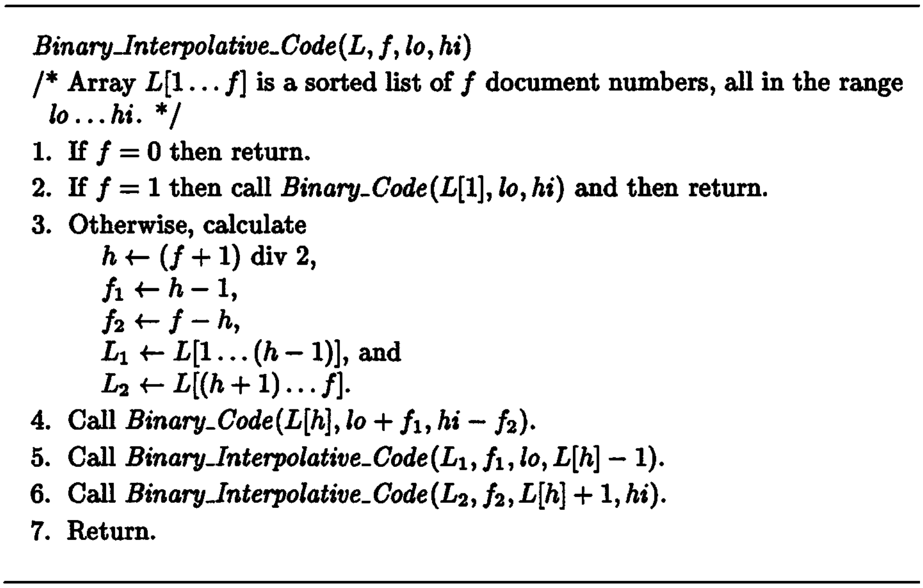
Figure 2. Representing the list ⟨7; 3, 8, 9, 11, 12, 13, 17⟩ using binary interpolative coding.

inclusive. A simple assignment of codewords then suffices to represent this first document pointer in three bits.

Suppose now that the fourth as well as the second document numbers are known. The fourth document pointer is to document 11, so the third pointer is constrained to the range 9 to 10. Again, a simple code—in this case one bit long—can be used to represent the third pointer. The brevity of this codeword is a direct consequence of the fact that there is a cluster, and that both the upper and lower bounding pointers are also in the cluster. As an even more extreme example, if both the fourth and the sixth pointers are known (to documents 11 and 13 respectively), then the fifth document pointer can be represented using a codeword zero bits long—it must be to document 12. Finally, if $N = 20$ is known, the last document number, in the range $14 \,..\, 20$, can also be coded in 3 bits. The restrictions that allow the coding of these three pointers are shown in the third section of figure 2.

This representation is based upon the supposition that the second, fourth, and sixth pointers are known. To represent them, a list $\langle 3; 8, 11, 13\rangle$ must have been previously coded. The same technique can be used for this list too. If the second pointer (to document 11) is known, then the first pointer (to document 8) takes at most four bits. Indeed, since there must be a pointer to the left and a pointer to the right of this document, the range can be further narrowed to $2 \,..\, 9$ inclusive, and a three bit code can be used. By a similar argument the third pointer must lie between $13 = 12 + 1$ and $19 = 20 - 1$ inclusive, and $3 = \lceil \log_2 7 \rceil$ bits suffice. The middle section of figure 2 shows the coding of these two values.

The only remaining problem is to code the fourth pointer. A five-bit code in the range $1 \,..\, 20$ inclusive is certainly adequate, and if the knowledge that there are three document pointers to the left and three to the right of this middle pointer is exploited, the range can be narrowed to $4 \,..\, 17$ inclusive, and four bits suffice, as is shown in the top section of figure 2. If we now consider figure 2 in its entirety, in the first step, one value (the fourth



```
Binary_Interpolative_Code$(L, f, lo, hi)$
/* Array $L[1 \ldots f]$ is a sorted list of $f$ document numbers, all in the range
  $lo \ldots hi$. */
1. If $f = 0$ then return.
2. If $f = 1$ then call Binary_Code$(L[1], lo, hi)$ and then return.
3. Otherwise, calculate
      $h \leftarrow (f + 1)$ div 2,
      $f_1 \leftarrow h - 1$,
      $f_2 \leftarrow f - h$,
      $L_1 \leftarrow L[1 \ldots (h - 1)]$, and
      $L_2 \leftarrow L[(h + 1) \ldots f]$.
4. Call Binary_Code$(L[h], lo + f_1, hi - f_2)$.
5. Call Binary_Interpolative_Code$(L_1, f_1, lo, L[h] - 1)$.
6. Call Binary_Interpolative_Code$(L_2, f_2, L[h] + 1, hi)$.
7. Return.
```

Figure 3. Binary Interpolative Coding.

one in the list, to document number 11) must be coded, and it is constrained to lie within the shaded region. The second step then transmits two values, and each of them is more tightly bounded by virtue of the fact that the middle value of the list is now known to be document 11. Finally, a list of four values is transmitted, each in its own even more tightly specified range.

The detailed process of calculating ranges and codes is described by the pseudo code in figure 3. Function *Binary_Code*(x, lo, hi) is assumed to encode a number $lo \leq x \leq hi$ in some appropriate manner. The simplest mechanism for doing this (as assumed in the description above) requires $\lceil \log_2(hi - lo + 1) \rceil$ bits. For example, from figure 2, it can be seen that the sequence of (x, lo, hi) triples processed is $(11, 4, 17)$, $(8, 2, 9)$, $(3, 1, 7)$, $(9, 9, 10)$, $(13, 13, 19)$, $(12, 12, 12)$, and $(17, 14, 20)$. With this simple implementation of *Binary_Code*, the corresponding codewords are 4, 3, 3, 1, 3, 0 and 3 bits long, for a total of 17 bits, one bit less than the Golomb code shown in Table 1. Note that in figure 3 explicit recursion has been used for ease of illustration. Should memory space be at a premium, an explicit stack can also be used. In either case, processing of a sequence of f values requires $O(\log f)$ memory.

But this simple implementation is wasteful. More compact representations result from the use of a centered minimal binary code of the form used by Howard and Vitter in their FELICS image compression method (Howard and Vitter 1993). (The HINT Hierarchical INTerpolation image compression method (Roos et al. 1988) also shares some aspects in common with the interpolative method described here.) The centered minimal code works in the following way. Suppose that a number in the range $1 \, . \, . \, r$ is to be coded. A simple binary code assigns codewords $\lceil \log_2 r \rceil$ bits long to all values 1 through r, and wastes $2^{\lceil \log_2 r \rceil} - r$ codewords. That is, $2^{\lceil \log_2 r \rceil} - r$ of the codewords can be shortened by one bit without loss

Table 2. Centered mininal binary coding.

	Value x to be coded								
Range r	1	2	3	4	5	6	7	8	9
1	—								
2	0	1							
3	00	1	01						
4	00	01	10	11					
5	000	01	10	11	001				
6	000	001	10	11	010	011			
7	000	001	010	11	011	100	101		
8	000	001	010	011	100	101	110	111	
9	0000	001	010	011	100	101	110	111	0001

of unique decodability. When $r = 14$, for example, two of the codewords can be three bits long. In Golomb coding the shorter codewords are allocated first, so that the codewords assigned to $x = 1, 2, 3, 4, \ldots, 14$ would be $000, 001, 0100, 0101, \ldots, 1111$ respectively. For reasons that will become clear in Section 5, in the case of binary interpolative coding it is important to allocate the shorter codewords to the *centre* of the range of integers being coded. For the same case of $r = 14$, the two three-bit codewords should thus be allocated to $x = 7$ and $x = 8$, with six four-bit codewords allocated to $1 \leq x \leq 6$ and six four-bit codewords to $9 \leq x \leq 14$. This coding situation is exactly what was illustrated in the first part of figure 2. In this case $r = hi - lo + 1 = 17 - 4 + 1 = 14$, and one bit less is used than might be otherwise be supposed when $x = 11 - lo + 1 = 8$ is coded. Note that the exact allocation of codewords is immaterial, provided that their lengths are correct and no codeword is a prefix of another; note also that there will always be an even number of longer codewords, so the codeword allocation can always be symmetric. Table 2 shows, for some small values of r, one way that the required assignment of codewords to values x might be achieved.

From Table 2 and figure 2 it is clear that a similar one bit gain arises when the final value of the example shown in figure 2 is coded. In this case $lo = 14$, $hi = 20$, and there are $r = 7$ numbers in range; the centered binary code assigns the single short codeword to $x = 17 - lo + 1 = 4$, the value being coded, and so a two bit code suffices. Hence, using binary interpolative coding the entire example list is represented in 15 bits, three less than the Golomb code.

5. Analysis

Now consider the number of bits required by the binary interpolative representation. Suppose that a sorted list of f values in the range $1 \mathinner{\ldotp\ldotp} N$ is to be coded. The number of bits produced will be accounted for in the reverse order in which they are generated, in the same manner in which the earlier example worked backwards considering first a list of $\lceil f/2 \rceil$ values, supposing the other $\lfloor f/2 \rfloor$ values to have been already encoded. It is also assumed initially that a number in the range $1\,..\,r$ requires a code of exactly $\log_2 r$ bits, as if the binary numbers in the range were assigned equal probabilities of $1/r$ and then arithmetically coded (Moffat et al. 1998). The difference in coding effectiveness between this ideal bound and the centered minimal binary code is examined below.

If the even-numbered values in a list of f items are known, the cost of binary coding the remaining $\lceil f/2 \rceil$ odd-numbered values is bounded above by

$$\left\lceil \frac{f}{2} \right\rceil \cdot \log_2 \frac{N - \lfloor f/2 \rfloor}{\lceil f/2 \rceil}\,, \tag{1}$$

since the concavity of the log function implies that the sum of the logarithms of the $\lceil f/2 \rceil$ individual ranges is maximised when all are equal. Let $C(N, f)$ be the maximum cost in bits of coding the entire list of f values. Equation (1) implies that

$$C(N, f) = \begin{cases} 0 & \text{when } f = 0, \\ \dfrac{f+1}{2} \cdot \log_2 \dfrac{N - (f-1)/2}{(f+1)/2} + C(N, (f-1)/2) & \text{when } f \text{ is odd,} \\ \dfrac{f}{2} \cdot \log_2 \dfrac{N - f/2}{f/2} + C(N, f/2) & \text{when } f \text{ is even.} \end{cases}$$

Using this recurrence, we now show that if $f < N/2$ and

$$C(N, f) \le f \cdot \left(2 + \log_2 \frac{N}{f}\right) \tag{2}$$

then

$$\begin{aligned} C(N, 2f) &\le (2f) \cdot \left(2 + \log_2 \frac{N}{2f}\right) \\ C(N, 2f+1) &\le (2f+1) \cdot \left(2 + \log_2 \frac{N}{2f+1}\right), \end{aligned}$$

and hence, by induction—taking as a basis the observation that $C(N, 1) = \log_2 N \le 2 + \log_2 N$—that Eq. (2) is true for all values $1 \le f \le N$.

There are two cases to consider: $2f$, and $2f + 1$. These two cases are treated in the two derivations that follow:

$$\begin{aligned} C(N, 2f) &= \frac{2f}{2} \cdot \log_2 \frac{N-f}{f} + C(N, f) \\ &\le f \cdot \log_2 \frac{N-f}{f} + f \cdot \left(2 + \log_2 \frac{N}{f}\right) \\ &\le 2f + f \log_2 N - f \log_2 f + f \log_2 N - f \log_2 f \\ &= (2f) \cdot \left(2 + \log_2 \frac{N}{2f}\right)\,, \end{aligned}$$

and

$$\begin{aligned}
C(N, 2f+1) &= \frac{(2f+1)+1}{2} \cdot \log_2 \frac{N-f}{f+1} + C(N,\ f) \\
&\le (f+1) \cdot \log_2 \frac{N-f}{f+1} + f \cdot \left(2 + \log_2 \frac{N}{f}\right) \\
&\le (2f+1) \cdot (2 + \log_2 N - \log_2(2f+1)) \\
&\qquad + f \cdot (2\log_2(2f+1) - \log_2 f - \log_2(f+1) - 2) \\
&\qquad + \log_2(2f+1) - \log_2(f+1) - 2 \\
&= (2f+1) \cdot (2 + \log_2 N - \log_2(2f+1)) \\
&\qquad + f \cdot \log_2 \frac{(2f+1)^2}{4f(f+1)} + \log_2 \frac{2f+1}{2(f+1)} - 1 \\
&\le (2f+1) \cdot \left(2 + \log_2 \frac{N}{2f+1}\right),
\end{aligned}$$

with the final inequality holding because

$$\log_2 \frac{4f^2+4f+1}{4f^2+4f} = \log_2 \left(1 + \frac{1}{4f^2+4f}\right) \le \frac{1}{f}.$$

That is, in an amortised sense the cost per value coded is at most $2 + \log_2(N/f)$ bits. This compares very well with the best that a Golomb code attains. The entropy of the geometric distribution $Pr(x) = (1-p)^{x-1}p$ is given by $-((\log_2(1-p))/p + \log_2 p) \approx \log_2 e + \log_2(1/p) \approx 1.44 + \log_2(N/f)$ bits per pointer, and is a lower bound on the compression effectiveness for a Golomb code.

The weakness of the previous analysis is, of course, that it assumes the use of perfect binary codes. To remedy this weakness the inefficiency introduced by the use of non-perfect binary codes must also be calculated. In a completely unconstrained environment the centered binary codeword for a value in $1\,..\,r$ might be as much as one bit longer than $\log_2 r$. At face value, this would imply that the best that can be said about the binary interpolative code is that it requires at most $3 + \log_2(N/f)$ bits per pointer in an amortised sense.

This is, however, unnecessarily pessimistic. Suppose that r is the current range, and the current middle value $L[h]$ is to be coded out of r possibilities. When a codeword longer than $\log_2 r$ bits is emitted, it must be because the sublists L_1 and L_2 (see figure 3) are non-symmetric. One must have a range sufficiently larger than $r/2$, and one sufficiently smaller than $r/2$, that the central region of the code—to which codewords *shorter* than $\log_2 r$ bits are allocated—is avoided. Indeed, the greater the inefficiency in the coding of the central value, the more unbalanced the relative sizes of the two partitions must be, and in the limit, to actually attain an inefficiency of one bit, the central value $L[h]$ must be equal to either $lo + f_1$ or $hi - f_2$ (and the range $r = (hi - f_2) - (lo + f_1) + 1$ must be equal to one more than a large power of two).

That is, there are two factors acting in opposition. Unbalanced partitions are necessary to introduce inefficiency in the coding of the middle value $L[h]$, but unbalanced partitions result in savings in the subsequent recursive representations, since the codes in a short partition inherit a shared saving as a consequence of being more crowded than was assumed in the derivation of Eq. (2). The tension between these two effects means that the maximum inefficiency is considerably less than one bit per pointer. We now quantify this argument.

Suppose that f is one less than a power of two. The value $L[h]$ can be thought of as being at the root of a subtree of f values that eventually has $(f+1)/2$ leaf calls and $(f-1)/2$ internal recursive calls. There are $(f-1)/2$ calls in total on each side of the principal call that establishes the central value $L[h]$. Let $0 \le s < 1$ represent the fraction of $r = hi - lo + 1$ (the total coding range) that is allocated to short codewords when $L[h]$ is coded,

$$s = \frac{2^{\lceil \log_2 r \rceil} - r}{r}\,.$$

If a long codeword is required, the extra cost is given by

$$\log_2(1+s), \tag{3}$$

since $(1+s)r$ is the next greatest power of two, and the analysis above only allowed $\log_2 r$ bits for the binary code. Offset against this are the savings that result as a consequence of the imbalance in the two partitions—one side is now of width at most $r(1-s)/2$ and the other of width at least $r(1+s)/2$, where in the original analysis both partitions were assumed to be of width $r/2$. In each side of the subsequent recursions there are $(f-1)/2$ values eventually binary coded, and so the saving caused by the imbalance of the partition is given by

$$\begin{aligned}&\frac{f-1}{2}\left(2\,\log_2\frac{r}{2} - \log_2\frac{r(1-s)}{2} - \log_2\frac{r(1+s)}{2}\right)\\ &= -\frac{f-1}{2}(\log_2(1-s) + \log_2(1+s)).\end{aligned} \tag{4}$$

Subtracting Eq. (4) from Eq. (3) gives the extra cost incurred when a long codeword of minimal disruption is used:

$$\begin{aligned}&\log_2(1+s) + \frac{f-1}{2}(\log_2(1-s) + \log_2(1+s))\\ &= \log_2\left[(1+s)^{(f+1)/2}(1-s)^{(f-1)/2}\right].\end{aligned} \tag{5}$$

When $f = 1$ and a subtree contains a single leaf code, Eq. (5) is, unsurprisingly, minimised as s becomes asymptotically close to 1, and in the limit takes on the value one. When just one value is being coded, the inefficiency can be as much as one bit. But when $f = 3$ the maximum value of Eq. (5) occurs at $s = 1/3$ and is just 0.2451. Table 3 lists the maximum inefficiencies possible for other values of f, calculated using numerical techniques. As the

Table 3. Bounds on inefficiency arising through the use of a centered minimal binary code compared with arithmetic coding.

f	minimising s	$\log_2[(1+s)^{(f+1)/2}(1-s)^{(f-1)/2}]$
1	1.0000	1.0000
3	0.3333	0.2451
7	0.1429	0.1034
15	0.0667	0.0481
31	0.0323	0.0233
63	0.0159	0.0145
127	0.0079	0.0057

subtrees get larger, the possible loss gets very small. This is why the use of the centered minimal binary code is integral to the description of binary interpolative coding.

Over a whole list of integers, half of them are coded in a tree of size $f = 1$, one quarter are coded as the root of a tree of size $f = 3$, one eighth in a tree of size $f = 7$, and so on. Hence, the worst-case inefficiency introduced by the use of integral-length codewords (rather than an arithmetic code as was assumed in Eq. (2)) is bounded above by

$$\frac{1.000}{2} + \frac{0.2451}{4} + \frac{0.1034}{8} + \frac{0.0481}{16} + \frac{0.0233}{32} + \frac{0.0145}{64} + \cdots \approx 0.5783$$

bits per symbol. That is, the cost of an interpolative code for f integers in the range $1 \,..\, N$ is never more than

$$f \cdot \left(2.5783 + \log \frac{N}{f}\right) \tag{6}$$

bits. Note that this is a worst-case bound, and holds for all combinations of f and N, and, once N and f are fixed, for any set of f distinct integers in $1 \,..\, N$.

In practice three other effects act to further reduce the average cost. First, there is a strong averaging tendancy for the central numbers in each list and sublist to be near the middle of their range, and thereby benefit from a codeword shorter than $\log_2 r$ bits. Second, the effect of range-narrowing—making use of the knowledge that $L[h]$ lies in $lo + f_1 \,..\, hi - f_2$ rather than in $lo \,..\, hi$—has been completely ignored in the analysis. In the extreme case, when $f = N$, range-narrowing means that no bits whatsoever are required to transmit the f values, since all of the middle values $L[h]$ are constrained to a range of $r = 1$.

The third effect that acts to make the analysis pessimistic is that Eq. (5) supposes that an adversary may choose s freely at each step. In fact, s is predetermined by the sequence of values, and it does not appear to be possible for the worst-case value of s to occur for every sublist that is coded. When $f = 1$ the worst-case s can clearly be achieved, and the list

$$\langle 1, 2^k + 1, 1, 2^k + 1, 1, 2^k + 1, 1, 2^k + 1, 1, \ldots \rangle$$

requires $2.5 + \log_2(N/f)$ bits to represent, partly because the $f = 1$ sublists are worst-case in terms of s, and partly because the $f = 3$, $f = 7$, etc., sublists are also (somewhat fortuitously) structured such that $\log_2 r$ is very close to the number of bits emitted by the centered binary code. But if the worst-case values of s at both $f = 1$ and $f = 3$ are to be achieved the list must have the structure

$$\langle 1, 2^k + 1, 1, 2^{k+1} + 1, 1, 2^k + 1, 1, 2^{k+1} + 1, 1, \ldots, \rangle,$$

which forces the consumption of fewer than $\log_2 r$ bits in the $f = 7$, $f = 15$, etc., sublists, and as a consequence appears experimentally to require on average only $2.427 + \log_2(N/f)$ bits per pointer. That is, Eq. (5) is an upper bound that has not been demonstrated to be tight.

For these three reasons, the average behaviour of the interpolative code is much better than the bound of Eq. (6). Figure 4 summarises experiments that demonstrate this claim. To produce figure 4, sequences of randomly generated values were compressed using a Golomb code, Elias's C_δ code, and the binary interpolative code. In the first set of experiments, illustrated on the left in figure 4, $f = 1{,}000{,}000$ gaps were drawn from a geometric distribution and compressed using the three methods, for each of a large number of average gaps N/f in the range $1 < N/f \leq 100$. The curves plotted show the amount in bits per symbol by which each of the coding methods exceeds the calculated self-entropy of the generated distribution. Not surprisingly, the Golomb code has a very small excess codelength for all values of N/f, since it is a minimum-redundancy code for this distribution. At the other extreme, the Elias C_δ code has an excess codelength that grows as N/f increases. Between

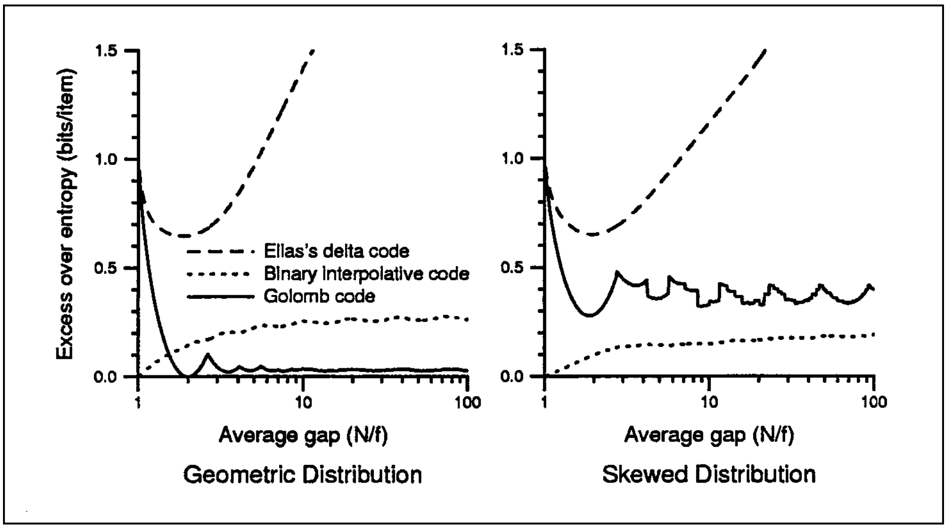


Figure 4. Compression results for geometric and skewed geometric distributions of $f = 1{,}000{,}000$ items: average inefficiency relative to self-entropy (bits per item), as a function of the observed average gap of the distribution.

these two is the interpolative code, with an excess codelength that appears experimentally to asymptotically approach an inefficiency value of approximately 0.27 bits per symbol, meaning that on average each pointer costs $(1.44+0.27)+\log_2(N/f) = 1.81+\log_2(N/f)$ bits, considerably less than upper bound guaranteed by Eq. (6).

The second graph in figure 4 shows the same three methods, but applied to a distorted sequence of values. For each value of N/f the sequence of $f = 1{,}000{,}000$ geometrically distributed gaps was broken into chunks of 100 contiguous values. The chunks were then considered in groups of three. In the first chunk in each group each gap was multiplied by a factor of 0.2, while in the other two chunks each value was multiplied by a factor of 1.4. This process created artificial clusters of gaps much less than the average, and placed about 1/3 of the values to be coded into these clusters, but retained the same overall average gap. As can be seen from the second graph in figure 4, the inefficiency of the Golomb code increases, because the parameter used to control the code is set globally based upon the overall average gap, which was unchanged by the transformation. Now the interpolative code—which automatically adjusts to exploit the presence of the clusters—provides better compression. Note also the superiority of the interpolative code when N/f is small, even for the geometrically distributed gaps. The Golomb code always allocates at least one bit per item, whereas on dense lists (and dense clusters within sparse lists) the interpolative code can compress using, on average, less than one bit per item.

We conclude this section with a negative result. Suppose that the symbols $1, 2, 3, \ldots, n$ have probabilities $p_1 \geq p_2 \geq p_3 \geq \cdots \geq p_n$—that is, that the probability distribution is non-increasing. On such a distribution the Elias C_γ code is *universal* (Elias 1975), in that the average number of bits per symbol required by the code for any non-increasing probability distribution is bounded above by $2H + 1$, where $H = \sum_{i=1}^{n}(-p_i \cdot \log_2 p_i)$ is the self-entropy of the probability distribution.

The same is not true of the interpolative code. Consider the non-increasing probability distribution

$$p_i = \begin{cases} 1-q & \text{when } i = 1, \\ q/(n-1) & \text{when } 2 \leq i \leq n \end{cases}$$

on n symbols. The average value of this distribution is given by

$$\sum_{i=1}^{n}(i \cdot p_i) = (1-q) + q \cdot \frac{n(n+1)/2 - 1}{n-1}$$

and is such that for large n and small q the ratio

$$\frac{\log_2 \sum_{i=1}^{n}(i \cdot p_i)}{H} = \frac{(1-q) + q \cdot \frac{n(n+1)/2-1}{n-1}}{-(1-q)\log_2(1-q) + q \cdot \log_2(n-1)}$$

grows without limit. Nor is it the case that Eq. (5) is sufficiently imprecise that the range narrowing effect acts to eliminate this growth. When q is small, each non-unit symbol in the recursive representation has approximately $\log(1/q)$ non-minimal intervals prior to it in the code sequence (higher in the tree) that can be unambiguously associated with it. Hence, for a fixed and small value of q, doubling n causes the average number of bits required by the interpolative coder to increase by $q\ \log_2(1/q)$, while the entropy only grows by q. That is, the cost of the interpolative coder can be arbitrarily large compared to the self-entropy, and the interpolative code is not universal in the sense that the Elias C_γ code is.

As is shown in the next section, this negative result does not detract from the usefulness of the code.

6. Experimental compression results

Table 4 shows various parameters of five large document collections that we have used for our experiments. Each of the first four collections are homogeneous, drawn from a single source: the Associated Press news service; The San Jose Mercury newspaper; the Wall Street Journal; and magazines published by the Ziff company. The final TREC collection includes the first four plus additional text drawn from a variety of other sources, and so contains a mixture of document styles and subjects. The TREC collection is used internationally as a testbed for research in information retrieval techniques (Harman 1995). The values listed are for a complete representation of the collection, including all words, and all strings of digits. No stopping or other index reduction techniques are applied in any of the results listed in this section.

Table 5 shows the compression achieved by several different index compression methods. Elias's C_δ code achieves remarkably good compression, considering that it is a static code.

Table 4. Parameters of test collections.

	Collection				
Parameter	AP	SJM	WSJ	ZIFF	TREC
Size (MB)	732	287	510	764	3,193
Documents (N)	242,918	90,257	173,252	293,121	1,078,166
Words ($\sum_t F_t$)	120,931,705	42,904,466	83,082,345	125,023,270	506,798,322
Distinct words (n)	274,752	12,8951	156,796	264,273	720,417
Pointers ($\sum_t f_t$)	58,635,644	20,802,740	38,081,424	50,702,686	203,784,900

Table 5. Index compression, document pointers only, bits per pointer averaged over all of the inverted lists in the collection.

	Collection				
Method	AP	SJM	WSJ	ZIFF	TREC
Elias, C_δ	6.46	6.51	6.53	5.52	6.24
Golomb	5.20	5.23	5.20	5.47	5.87
LLRUN	5.26	5.29	5.28	4.84	5.38
Markov-3	5.04	5.00	5.04	4.80	5.42
Interpolative	5.18	5.14	5.20	4.70	5.10

Golomb coding, and the batched LLRUN method of Moffat and Zobel (1992) achieve slightly better compression. The row captioned "Markov-3" gives estimated compression results for the three-state Markov machine model proposed by Bookstein et al. (1994) These results are estimates rather than precise because of the complexity of implementation required for the Markov method. They were generated assuming that each inverted list (bitstring) was coded in a semi-static manner and that the optimal parameters could be determined by a first pass through that entry, but that the parameters need not be stored in the compressed entry. We did, however, assume that two bits per entry should be counted to select parameters to be used for that entry, and, in the manner of the experiments undertaken by Bookstein et al., only applied the Markov model to inverted lists of 60 or more pointers. Lists shorter than this were represented using a Bernoulli model and arithmetic coding—in effect, a one state machine parameterised by f_t alone. The compression rates listed in Table 5 are thus a lower bound on the compression attained by the Markov method. All of the other results are for actual compression of the complete inverted file, and have been verified by decompressing the file and comparing with the original.

As can be seen from the table, the interpolative method gives good compression, particularly on the heterogeneous TREC collection, in which clustering is the most pronounced. Note that the Markov-3 method obtains a large fraction of its benefit from the use of arithmetic coding: for example, an arithmetic coded Bernoulli model for all inverted lists yields 5.12 on AP, 0.08 bits per pointer better than the 5.20 bits per pointer attained by the model-equivalent Golomb code. Finally, note that the interpolative method is fast in encoding and decoding, being only slightly slower than Golomb coding.

If ranked queries are to be supported then within-document frequencies must be added to the inverted file. These can be compressed in exactly the same manner as the document pointers themselves: if there are f_t entries in some inverted list and a total of F_t occurrences of that term in the collection then the sequence of f_t cumulative sums of the $f_{d,t}$ values also forms a strictly increasing integer sequence, and all of the previous compression methods are applicable. Table 6 shows the cost, in terms of bits per pointer, for storing the within-document frequencies for the five test collections. The interpolative method again provides excellent performance. Note that the best fixed code for within-document frequencies is the Elias C_γ code rather than the C_δ code; this is because most of the $f_{d,t}$ values are small.

If word-level operations such as adjacency or proximity queries (for example, to support queries of the form "find the documents where *index* and *compression* appear within ten

Table 6. Index compression, within-document frequencies only, bits per pointer averaged over all inverted lists.

	Collection				
Method	AP	SJM	WSJ	ZIFF	TREC
Elias C_γ	1.97	1.97	2.01	2.11	2.08
Golomb	1.79	1.81	1.87	2.10	2.12
LLRUN	1.96	1.97	1.99	2.10	2.06
Markov-3	1.49	1.49	1.61	1.79	1.80
Interpolative	1.59	1.60	1.69	1.74	1.73

Table 7. Index compression, word pointers, bits per pointer averaged over all inverted lists.

	Collection			
Method	AP	SJM	WSJ	ZIFF
Elias, C_δ	13.98	13.82	13.91	13.24
Golomb	11.77	11.79	11.64	11.54
LLRUN	11.29	11.35	11.21	10.72
Markov-3	11.77	11.82	11.63	11.54
Interpolative	11.44	11.51	11.34	10.81

words of each other”) are to be supported a word-level index is needed. In simplest form a word-level index treats every word in the collection as an individual document. Table 7 shows the compression obtained for a word-level indexes for the five test collections.

Note the increased size, since there are now in total $\sum_t F_t$ pointers stored rather than $\sum_t f_t$ pointers, and each pointer is in the range $1 \, .. \sum_t F_t$ rather than $1 \, .. \, N$. These two effects combine to make word-level indexes roughly four times bigger than document-level indexes. Indeed, a word-level index is a lossy representation of the words of the database without any of the whitespace or punctuation, so it is not surprising that the best compression attainable for a word-level index is to about 25% of the original text size. Memory limits prevented the creation of a word-level index for the TREC collection.

7. Refinements

In this section we describe two refinements to the interpolative method that each result in slight but consistent improvement in compression.

The first refinement is easiest to illustrate with an example. Assume that some inverted list contains 10 pointers. The description of interpolative coding given in figure 3 codes the 5th pointer, and then recurses on one list of four pointers and one list of five. These two lists in turn give rise to recursive calls with f equal to 1, 2, 2, and 2. Now suppose instead that the first pointer processed in the original list was the 8th. Then the left recursion is on seven pointers, and leads to a simple recursion pattern as per the example given earlier in Section 4. The right recursion is on two pointers, and is the only two-pointer recursion in the entire process. Because the middle values of each range are favoured by the shorter codewords, two-pointer recursions—coding a pointer that on average can be expected to lie one third of the way through the range—are, on average, slightly more expensive for what they achieve than are three- or one-pointer recursions. The same argument applies to four-pointer recursions. Minimising the number of non-$2^k - 1$ recursions is desirable, and is achieved by always coding the pointer that occupies the largest power of two location in the list, and accepting an early large non-balanced recursion—in which sheer weight of numbers will encourage a central value for the coded number—to avoid many small non-balanced recursions later in the process. The first row of Table 8 shows the compression achieved by this modification. There is a slight but uniform improvement.

Table 8. Performance of refinements, document pointers only, bits per pointer averaged over all of the inverted lists in the collection.

	Collection				
Method	AP	SJM	WSJ	ZIFF	TREC
Interpolative, original	5.18	5.14	5.20	4.70	5.10
Interpolative, balanced recursion	5.14	5.10	5.15	4.66	5.06
Interpolative, reordered codelengths	5.13	5.09	5.15	4.61	5.03

The second refinement arises from the observation that at the base of the sequence of recursive calls, when $f = 1$, it is no longer appropriate to assume that middle values are more likely and allocate them the shorter codewords. Instead it should be the outer regions of the range that are allocated the shorter codewords, since if there is any clustering the pointer can be expected to lie close to one or the other of the two bounding pointers, and if there is no clustering then the codeword allocation is immaterial. Hence, when $f = 1$, we reverse the allocation of codewords, and favour the extremities with the one-bit-shorter codes. The results of applying this refinement (in conjunction with the first refinement) are shown in the second row of Table 8. Again, there is a slight but definite improvement in compression effectiveness.

Finally, before moving on to other applications of the interpolative method, a number of drawbacks need to be considered. In most information retrieval systems the space occupied by the index is an important consideration, but by no means the only one. Ease of construction and speed of access when answering queries are also important factors, and it is in these two aspects that simple methods of index coding—such as Elias coding—have benefits. To construct an index using the interpolative code each inverted list must be known in entirety, and in a dynamic retrieval system this may not be possible. The binary-search nature of index access may also prove problematic. For Boolean-style queries, partial decoding of the index using skip-ahead pointers of the form described by Moffat and Zobel (1996) is possible, at the cost of a slight increase in index size. In each case the skip is attached to the left subtree of a value, and indicates the bit address at which the right subtree commences, with the codes stored in the order generated by a pre-order traversal of the tree (Moffat et al. 1995) rather then the level-order representation implied above.

For ranked queries, however, the situation is more complex. The frequency-sorted indexes of Persin et al. (1996) group the pointers of each inverted list into discrete sections, all with the same within-document frequency value, and while the interpolative method can be applied to each section, it is not clear that clustering of terms (and hence the compression advantage) will be preserved. Other recent mechanisms for providing fast ranked querying (Anh and Moffat 1998) may prove to be even less amenable to the use of the interpolative code.

8. Other applications

While index compression has been the main application presented here, interpolative coding is not limited to the representation of inverted files. In this section two other applications are

Table 9. Part of the permuted text formed by applying the Block Sort transformation to a small initial section of the King James Bible.

Symbol	Context	MTF value
`i`	`n the audience of the people of the`	1
`i`	`n the audience of the sons of Heth,`	1
`I`	`n the beginning God created the hea`	49
`i`	`n the bottle, and she cast the chil`	2
`i`	`n the breadth of it; for I will giv`	1
`o`	`n the camels, and followed the man:`	4
`o`	`n the carcases, Abram drove them aw`	1
`i`	`n the cave of Machpelah, in the fie`	2
`i`	`n the cave of the field of Machpela`	1

described: to Block-Sorting, a general purpose compression technique developed recently by Burrows and Wheeler (1994); and to the representation of preludes for block-based semi-static minimum-redundancy coding (Turpin and Moffat 2000).

Block-Sorting, or the Burrows-Wheeler transform (Burrows and Wheeler 1994; Fenwick 1996), is a highly effective compression scheme which permutes the input text with a reversible transformation. A full description of the technique is beyond the scope of this article, and the reader is referred to, for example, Witten et al. (1999, Chapter 2). Table 9 shows a small part of the permuted text for the *Bible*. The characters in the first column have been sorted according to their following context, the strings in the second column. Once sorted, the characters in the first column are transmitted to the decoder. One way in which this can be done is shown in the third column, which lists move-to-front (MTF) values for the characters of the first column. For example, when the upper-case `I` is encountered, a total of 48 distinct other symbols have been encountered since `I` was last coded, and so `I` is coded as the integer 49. The next character in the permuted text, the third lower-case `i` in the first column of Table 9, is then coded as the integer 2, since one distinct other character (the `I`) has been processed since the last time `i` was coded.

Lexicographically similar following contexts tend to be preceded by one of only a small number of distinct characters, meaning that most of the MTF values to be transmitted are small. For example, there are only 3 distinct characters in the 80 characters following the fragment shown in Table 9. Hence, just as with index compression, representations of the integers in which small values are naturally assigned short codes are of interest. Moreover, different following contexts will result in different degrees of clustering. While a statistical compressor based upon global observed frequencies can be used to represent the MTF values, such a mechanism does not allow for any local adaptation within the list of MTF values. Nor is it clear how such adaptation could be embodied, since all of the contextual information has been stripped out as the MTF values are generated.

The interpolative code can be directly applied to the list of MTF values exactly as if it were a set of d-gaps. The resulting compression process is both faster than if arithmetic

Table 10. Block-sorting compression: effectiveness measured in bits per character, and throughput measured in MB per CPU minute on a DEC Alpha, with programs written in C and compiled with gcc, and a blocksize of 900kB. The data file is the 510MB WSJ collection.

Method	Compression (bits/char)	Throughput (MB/min) Encoding	Throughput (MB/min) Decoding
Arithmetic coding	2.47	6.8	15.3
Interpolative coding	2.17	7.9	25.3
bzip–2.0	2.07	9.9	29.9

coding is used, and also, as a result of the localised sensitivity of the interpolative code, more effective. In particular, long runs of ones in the MTF list—which occur when the next character is the same as the previous character—are handled very economically by the interpolative coder.

Table 10 summarises our experiments with a Block-Sorting compression system. The first row shows compression effectiveness and throughput efficiency results for a straightforward implementation of the Burrows-Wheeler transformation when coupled with an arithmetic coder (Moffat et al. 1998) based upon global probabilities for the various MTF values. The second row describes the behaviour of the same program, but with the arithmetic coder replaced by the interpolative coder. Better compression is obtained because the interpolative coder is sensitive to the extent and degree of the clustering present in the list of MTF values. The third row shows the performance of the *bzip–2.0* software developed by Julian Seward;[2] it is also based upon the Burrows-Wheeler transformation, but is a carefully engineered implementation and obtains superior compression at a faster throughput rate. Among its many features, *bzip–2.0* isolates runs of symbols in the source text and treats them differently; isolates runs of ones in the MTF list and treats them differently; and makes use of a small set of minimum-redundancy codes, selecting the best code for each section of the file. This latter feature represents explicit handling of the variable density that the interpolative code recognises implicitly.

A final application in which we have used interpolative codes is block-based semi-static minimum-redundancy coding. In a semi-static coder a statistics prelude must be transmitted from encoder to decoder, indicating what codeword is to be assigned to each symbol in the source alphabet. If the message is being processed as a sequence of blocks, the prelude for each block must also include a subalphabet selection component which identifies, out of a universe of possible symbols, those which appear in that block. Identification of the subalphabet then allows more economical transmission of the codeword lengths, since only symbols that appear in the block need to be allocated codewords.

Table 11 summarises the cost of three different subalphabet selection mechanisms when applied in a block-based manner to the minimum-redundancy coding of two different messages derived from the WSJ collection. The first message is the words of the collection, represented as ordinal integers assigned in appearance order. For example, for the source text “singing do wah diddy diddy, dum diddy do”, the message would be the integer sequence “1, 2, 3, 4, 4, 5, 4, 2”. The second test message was the non-words of WSJ, where the non-words are the character strings that separate the words. For the same example text,

Table 11. Average cost (bits per symbol, averaged over the whole message) for different methods of specifying the subalphabet for block-based minimum-redundancy coding. The two test messages are respectively the words of the WSJ collection and the non-words of the WSJ collection, both represented as integers assigned in order of first appearance. Each message contains approximately 86 million symbols, drawn from an alphabet of size $N = 287{,}578$ in the case of the words, and $N = 8{,}193$ in the case of the non-words.

Message	Block size	Elias, C_γ	Golomb	Interpolative
WSJ words	10,000	1.659	2.163	1.392
	100,000	0.500	0.660	0.429
	1,000,000	0.115	0.143	0.098
WSJ nonwords	10,000	0.050	0.072	0.043
	100,000	0.013	0.016	0.012
	1,000,000	0.003	0.003	0.003

the sequence of non-word numbers would be “1, 1, 1, 1, 2, 1, 1”. In combination with the lexicons that translate integer symbol numbers into character strings, the sequence of non-words and words allow lossless reproduction of the source text. Moffat et al. (1998) discuss such a compression regime in more detail. Because the symbol numbers are assigned sequentially, the subalphabet actually used in each block tends to be clustered at low symbol numbers—symbols that appear early in the text have a higher probability of appearing in any given block than do symbols that make their first appearance later.

The interpolative code is sensitive to this variable density, and provides the best compression of the three methods tested. When the blocks are small the subalphabet is sparse, and the per-symbol cost of specifying the subalphabet is high. When the blocks are large, most of the symbols appear in most of the blocks, and the cost is much less. The ability of the interpolative code to deal effectively with both sparse and dense sets of values is the key to its success in this application.

9. Conclusions

We have described a novel method for representing strictly ascending integer sequences. The new method is particularly suited to situations in which clustering is anticipated, and experiments with the inverted indexes of five large text databases show the method to yield superior compression performance for both document pointers and within-document frequencies. Two other applications have also been described.

The new method is substantially easier to implement than any of the previous methods that achieve comparable performance, and the absence of any need for explicitly specified parameters means that it can be used without detailed knowledge of probabilities or frequency distributions.

Acknowledgments

This material was presented in preliminary form at the 1996 IEEE Data Compression Conference, Snowbird, UT, March 1996. Andrew Turpin (University of Melbourne) undertook

the experimental work summarised in Table 11, and that table is derived from data presented by Turpin and Moffat (2000).

Notes

1. The two plateaux regions correspond to Hurricane Gilbert in September 1988 and Hurricane Hugo in September 1989, reinforced—as a result of general discussion of natural disasters—by the San Francisco earthquake of 17 October 1989.
2. `http://www.muraroa.demon.co.uk/`.

References

Anh VN and Moffat A (1998) Compressed inverted files with reduced decoding overheads. In: Croft WB, Moffat A, van Rijsbergen CJ, Wilkinson R and Zobel J, Eds., Proc. 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Melbourne, Australia. ACM Press, New York, pp. 290–297.

Bell TC, Moffat A, Nevill-Manning CG, Witten IH and Zobel J (1993) Data compression in full-text retrieval systems. Journal of the American Society for Information Science, 44(9):508–531.

Bookstein A and Klein ST (1991) Compression of correlated bit-vectors. Information Systems, 16(4):387–400.

Bookstein A, Klein ST and Raita T (1994) Markov models for clusters in concordance compression. In: Storer JA and Cohn M, Eds., Proc. 1994 IEEE Data Compression Conference. IEEE Computer Society Press, Los Alamitos, California, pp. 116–125.

Bookstein A, Klein ST and Raita T (1997) Modeling word occurrences for the compression of concordances. ACM Transactions on Information Systems, 15(3):254–290.

Bookstein A, Klein ST and Ziff DA (1992) A systematic approach to compressing a full-text retrieval system. Information Processing & Management, 28(6):795–806.

Burrows M and Wheeler DJ (1994) A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California.

Choueka Y, Fraenkel AS and Klein ST (1988) Compression of concordances in full-text retrieval systems. In: Proc. 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Grenoble, France. ACM Press, New York, pp. 597–612.

Choueka Y, Fraenkel AS, Klein ST and Segal E (1986) Improved hierarchical bit-vector compression in document retrieval systems. In: Proc. 9'th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy. ACM, New York, pp. 88–97.

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

Fenwick P (1996) The Burrows-Wheeler transform for block sorting text compression: Principles and improvements. The Computer Journal, 39(9):731–740.

Fraenkel AS and Klein ST (1985) Novel compression of sparse bit-strings—Preliminary report. In: Apostolico A and Galil Z, Eds., Combinatorial Algorithms on Words, Volume 12. Springer-Verlag, Berlin, pp. 169–183. Nato ASI Series F.

Gallager RG and Van Voorhis DC (1975) Optimal source codes for geometrically distributed integer alphabets. IEEE Transactions on Information Theory, IT-21(2):228–230.

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

Harman DK (1995) Overview of the second text retrieval conference (TREC-2). Information Processing & Management, 31(3):271–289.

Howard PG and Vitter JS (1993) Fast and efficient lossless image compression. In: Storer JA and Cohn M, Eds., Proc. 1993 IEEE Data Compression Conference. IEEE Computer Society Press, Los Alamitos, California, pp. 351–360.

Jakobsson M (1978) Huffman coding in bit-vector compression. Information Processing Letters, 7(6):304–307.

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

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

Moffat A, Neal RM and Witten IH (1998) Arithmetic coding revisited. ACM Transactions on Information Systems, 16(3):256–294. Source software available from `http://www.csse.unimelb.edu.au/~alistair/arith_coder/`.

Moffat A and Zobel J (1992) Parameterised compression for sparse bitmaps. In: Belkin NJ, Ingwersen P and Pejtersen AM, Eds., Proc. 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Copenhagen. ACM Press, New York, pp. 274–285.

Moffat A and Zobel J (1996) Self-indexing inverted files for fast text retrieval. ACM Transactions on Information Systems, 14(4):349–379.

Moffat A, Zobel J and Klein ST (1995) Improved inverted file processing for large text databases. In: Sacks-Davis R and Zobel J, Eds., Proc. 6th Australasian Database Conference, Singapore. World Scientific, pp. 162–171.

Persin M, Zobel J and Sacks-Davis R (1996) Filtered document retrieval with frequency-sorted indexes. Journal of the American Society for Information Science, 47(10):749–764.

Roos P, Viergever MA, van Dijke MC and Peters JH (1988) Reversible intraframe compression of medical images. IEEE Transactions on Medical Imaging, 7(4):328–336.

Schuegraf EJ (1976) Compression of large inverted files with hyperbolic term distribution. Information Processing & Management, 12:377–384.

Teuhola J (1978) A compression method for clustered bit-vectors. Information Processing Letters, 7(6):308–311.

Turpin A and Moffat A (2000) Housekeeping for prefix coding. IEEE Transactions on Communications, 48(4).

Witten IH, Bell TC and Nevill CG (1992) Indexing and compressing full-text databases for CD-ROM. Journal of Information Science, 17:265–271.

Witten IH, Moffat A and Bell TC (1999) Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed. Morgan Kaufmann, San Francisco.

Zobel J and Moffat A (1995) Adding compression to a full-text retrieval system. Software—Practice and Experience, 25(8):891–903.