Information Retrieval, 8, 151–166, 2005

Inverted Index Compression Using Word-Aligned Binary Codes

VO NGOC ANH
ALISTAIR MOFFAT alistair@cs.mu.oz.au
Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia

Received June 18, 2003; Revised December 24, 2003; Accepted January 6, 2004

Abstract. We examine index representation techniques for document-based inverted files, and present a mechanism for compressing them using word-aligned binary codes. The new approach allows extremely fast decoding of inverted lists during query processing, while providing compression rates better than other high-throughput representations. Results are given for several large text collections in support of these claims, both for compression effectiveness and query efficiency.

Keywords: index compression, integer coding, index representation

1. Introduction

The usefulness of an information retrieval system depends upon a range of factors, including the amount of processor and disk resources required to store documents and generate answers to queries; the quality of the returned answers; and the interface presented by the system to its user. The first of these three factors is essentially economic, and directly contributes to the cost of providing search services. The second and third factors determine the extent to which users "like" the service, and thus the amount that they are willing to pay for it. There may also be tradeoffs possible between the three factors. For example, a crude search method may be very fast, but yield low-quality answers. Improvements in any of these three areas, or in the tradeoffs between them, are of direct relevance to any provider of information retrieval services.

In this work we focus on the first of the three factors. Resource costs arise in a number of ways – the cost of storing the raw data once is has been harvested from the web or some other source; the computational cost of building the data into an information retrieval system; the storage cost associated with index and vocabulary structures once they have been constructed; and the computational cost of using the structures to determine the proposed answers to queries.

There has been considerable investigation over many years into efficient mechanisms for indexing text, for storing the resultant indexes, and for using them to resolve user-initiated queries (Salton 1989, Frakes and Baeza-Yates 1992, Witten et al. 1999, Baeza-Yates and Ribeiro-Neto 1999). Common to all efficient implementations is a mechanism for representing the index in a compact manner, usually by reducing each index list to a sorted sequence of document identifiers; then taking the difference between consecutive values to form a list of *d*-gaps; and ultimately treating the representation of the *d*-gaps as an integer coding problem. Index compression also provides faster query processing than might be otherwise possible, as smaller amounts of data need to be transferred from disk (Zobel and Moffat 1995, Williams and Zobel 1999). In particular, the relativity between disk and CPU speeds on current hardware is such that with most compression schemes, data can be decoded faster than it can be delivered from the disk, resulting in a net decrease in access time if it is stored compressed. Trotman (2003) provides a detailed exploration of this behavior.

In this paper we describe a new compression technique for inverted file indexes. The new mechanism provides compression rates that are inferior to the best that have been previously achieved, but has the advantage of requiring exceptionally low computational effort at decoding time. That is, by allowing some inefficiency in the compressed representation, the new code provides fast access into the compressed inverted lists, and overall provides faster query processing than previous techniques. We give compression results for several large text collections in support of these claims. We also give retrieval throughput results that show the overall benefit of the new approach compared to previous index coding mechanisms.

2. Index representations

In an inverted index, the integer document identifiers associated with each distinct index term are collected together into an inverted list. In a typical large collection there are hundreds of thousands of such lists, ranging in length from one document identifier through to millions of document identifiers. A vocabulary notes which terms appear in the collection, and the locations on disk of their inverted lists.

Query resolution consists of searching the vocabulary for the query terms; fetching and decoding some (or all) of some (or all) of the corresponding inverted lists; and computing a similarity score for each document mentioned in the parts of the inverted lists that get processed. The documents with the highest similarity scores are then presented as the "answers" to the query. Witten et al. (1999) provide a description of the structures and mechanisms involved when queries are processed using an inverted file index. Alternative approaches are described by Persin et al. (1996) and Anh et al. (2001).

Each of the inverted lists consists of a set of integer document numbers. The standard way of representing these is to sort them into document-order, and then take differences between consecutive values. For example, the document-sorted list

$$\langle 4, 10, 11, 12, 15, 20, 21, 28, 29, 42, 62, 63, 75, 95\rangle$$

is reduced to the set of *d*-gaps

$$\langle 4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12, 20\rangle.$$

For terms *t* that occur in many of the documents in the collection, the list of *d*-gaps is long, but on average the values must be small, since the sum of the *d*-gaps cannot exceed *N*, the number of documents in the collection. On the other hand, terms with short inverted lists can contain large *d*-gaps, but cannot contain very many of them. That is, while large *d*-gaps can occur, they cannot be common. If the inverted list for a term t contains f_t entries, then the average *d*-gap in that list cannot exceed N/f_t.

A wide range of representations have been described for efficiently representing *d*-gaps. Witten et al. (1999) explain these and provide pointers into the original literature. Here we summarize the properties of a few key techniques, to set a context for the mechanism described in the next section.

If the documents containing a given term t can be assumed to be a random subset of the documents in the collection, then the set of *d*-gaps associated with term t will conform to a geometric distribution, and a Golomb code (see Witten et al. (1999) for a description) provides a minimum-redundancy representation. In essence, the Golomb code is optimal among the universe of prefix codes if the documents containing the term are randomly scattered. Golomb codes are also relatively straightforward to implement, and have been found to provide good compression effectiveness on typical document collections, over the full spectrum of scale.

If the documents containing a term t are not a random subset, but are instead clustered in some way as t moves into and out of use in the collection (for example, consider the word "Chernobyl" in a collection organized chronologically or geographically), better compression can be obtained. The binary interpolative code of Moffat and Stuiver (2000) is sensitive to localized clustering, and in extreme cases can reduce a whole run of unit *d*-gaps to just a few bits of output, still making use of simple binary coding mechanisms. Blandford and Blelloch (2002) took these ideas to the next level, and considered the possibility of reordering the documents in the collection so as to magnify clustering effects and thus minimize the cost of storing the index, but we do not apply their technique here, and leave the collection in its original ordering.

A wide range of ad-hoc mechanisms have also been described. The static codes of Elias (again, see Witten et al. (1999) for details) fall into this category—they give plausibly good compression, and have the advantage of not requiring any tuning or parameter setting. In a sense, they trade away compression effectiveness in any particular situation in favor of universality. However, Elias codes decode at the same rate as Golomb codes, and are no easier to implement.

Other tradeoffs are possible. In particular, it is interesting to consider compromises that swap compression effectiveness for decoding speed, on the grounds that a modest amount of additional disk storage to hold the index may well be warranted if query processing using the index can be accelerated. The best examples of these tradeoffs come through the use of nibble- and byte-aligned codes, which avoid the bit-by-bit processing costs associated with the Golomb, Elias, and interpolative techniques.

The commonest byte-aligned coding regime uses one bit in each byte as a flag, and the other seven bits for data. During encoding, if the number to be coded fits into a seven-bit integer, then those bits are output in a byte with a leading "0". Otherwise, the seven low-order bits are written in a byte with a leading "1"; the other high-order bits of the number are shifted right by seven bits; one is subtracted from them; and the process repeated. During decoding, if the flag bit in any byte is "1", the next byte must be fetched, incremented, and prepended, and then its flag bit checked. When a byte with a "0" flag is reached, the number is complete.

The net effect is that the numbers 1 to $128 = 2^7$ are stored in a single byte; the numbers 129 to $16{,}512 = 2^{14} + 2^7$ in two bytes; and so on—a kind of byte-level Golomb code. This code has been used in a number of commercial and research retrieval systems over the years, because of its low-tech blend of variable-length coding and byte-wise operation, and is considered in detail in the experiments of Scholer et al. (2002) and Trotman (2003).

Many other ad-hoc codes can also be constructed. For example, in a nibble-aligned code, the first three bits of the first four-bit nibble of each code might be used to record a number (between zero and seven) of additional nibbles to be fetched and concatenated to make a codeword. In this code, d-gaps of 1 and 2 would be coded into a single nibble; d-gaps of 3 to $34 = 2^5 + 2^1$ into two nibbles, or one byte; d-gaps of 35 to $546 = 2^9 + 2^5 + 2^1$ into three nibbles, and so on. On most computer architectures accessing byte- and half-byte-aligned nibbles is significantly faster than building codewords composed of individual bits, meaning that substantial query-time performance gains can be achieved. Scholer et al. (2002) and Trotman (2003) provide experimental evidence of the efficacy of byte-aligned codes in a query processing system. Byte-aligned codes have also attracted attention for word-based text compression (de Moura et al. 2000), where the alignment is beneficial to compressed searching.

The problem with ad-hoc codes is that they are insensitive to any attributes of the distribution of d-gaps being handled, and represent a Henry Ford-like "any color you like, so long as it is black" approach to compression. In a typical inverted index there is considerable variation in the d-gap distributions that must be handled: between one inverted list and the next; between one block of an inverted list and the next; and even between different parts of the same block of the same inverted list.

3. Fast decompression

This section introduces an alternative paradigm for representing a list of d-gaps, and three ways in which that paradigm might get implemented. The three implementations—dubbed "Simple-9", "Relative-10", and "Carryover-12"—offer different compromises between decoding complexity and compression rate. All combine reasonably good compression with fast decoding.

Simple-9

The basic idea of the new approach is that it represents a hybrid between bit-aligned and byte-aligned codes. It allows codewords that can be as short as one bit, but enforces a regular pattern within each compressed word, together with a periodic alignment that greatly reduces the cost in the decoder of extracting individual codewords from the incoming bitstream. It is also automatically sensitive to the local distribution of d-gaps, in the way that the interpolative code is sensitive.

Normally in designing a code the objective is to assign the minimum possible number of output bits to each input symbol. Here we adopt a slightly different tack, and assign to each fixed output word the maximum possible number of input symbols. In the new code, each 32-bit word stores a number of binary integers. (A similar code can readily be applied to 64-bit double words should such hardware become commonplace.) Different words store different numbers of integers, but within each word each integer is represented using exactly the same number of bits. In the first variant a total of 28 such *data bits* are used in each 32-bit word. (The use of the remaining 4 bits in each word is explained shortly.) For example, if the next 28 d-gaps in an inverted list are all either 1 or 2, then 28 one-bit binary codes can be used to code them. On the other end of the spectrum, if the next d-gap is greater than $16{,}384 = 2^{14}$ then it must be coded into a single word as a 28-bit number. Between these two extremes, seven 4-bit codes, or five 5-bits codes, or four 7-bit codes, and so on, could be used, depending on the values of the upcoming d-gaps. Numbers greater than 2^{28} cannot be represented in this system, meaning that the number of documents in the collection is limited to approximately 256 million. We do not regard this as a serious limitation, since very large collections are typically partitioned into manageable subcollections.

Table 1 shows the nine possible ways in which the 28 data bits can be partitioned into equal length binary integers, together with the number of bits (of the 28) that are wasted in each arrangement because the codewords are of fixed length. The remaining four bits in each 32-bit word are reserved for a *selector* field that indicates which row of the table is being used in this word—a value in the range **a** through to **i**. That is, each 32-bit output word consists of a 4-bit selector code, describing how the rest of the word is partitioned, and then a sequence of flat binary codewords describing a variable number of d-gaps, totalling not more than 28 bits. As many binary codes as can fit are packed into each 32-bit output word, meaning that at most 3 bits per word are wasted and not allocated to any part of the code.

To see this process in action, consider the list of d-gaps that was used earlier as an example:

$$\langle 4, 6, 1, 1, 3, 5, 1, 7, 1, 13, 20, 1, 12, 20\rangle.$$

Table 1. Nine different ways of using 28 bits for flat binary codes (method Simple-9).

Selector	Number of codes	Length of each code (bits)	Number of unused bits
a	28	1	0
b	14	2	0
c	9	3	1
d	7	4	0
e	5	5	3
f	4	7	0
g	3	9	1
h	2	14	0
i	1	28	0

If it is applicable, row **a** yields the most compact representation. But because there is a value greater than $2 = 2^1$ in the first 28 d-gaps to be coded, row **a** cannot be employed. Similarly, there is a value greater than $4 = 2^2$ in the first 14 entries, making use of row **b** impossible. But none of the first 9 entries are greater than $8 = 2^3$, so row **c** can be used to drive the first word of output. Allowing for the fact that a d-gap of one maps to a code of 000, the first output word is thus:

$$\mathbf{c}, 011, 101, 000, 000, 010, 100, 000, 110, 000, \cdot,$$

in which **c** is coded into four bits (as 0010 or some such), and the commas are purely for visual separation, and do not appear in the output. The final "·" represents the 32nd bit of the word, which is unused in the row **c** partitioning.

The remainder of the list of d-gaps (starting at 13, with five further values to be coded) is processed the same way, and yields a second word of compressed output:

$$\mathbf{e}, 01100, 10011, 00000, 01011, 10011, \ldots,$$

where now three bits are unused in the 32-bit word. That is the end of the list of d-gaps, so the compressed inverted list is complete.

In total, two 32-bit words suffice to code this list of fourteen d-gaps. Had a Golomb code been used, a total of 58 output bits would have been generated, which would similarly consume 64 bits if byte-padding or word-padding takes place on each compressed inverted list.

To explore the effectiveness of this approach, inverted lists of varying density were generated randomly for a collection containing a nominal N = 10,000,000 documents. Figure 1 shows the outcome of that experiment. Each vertical column of data points in

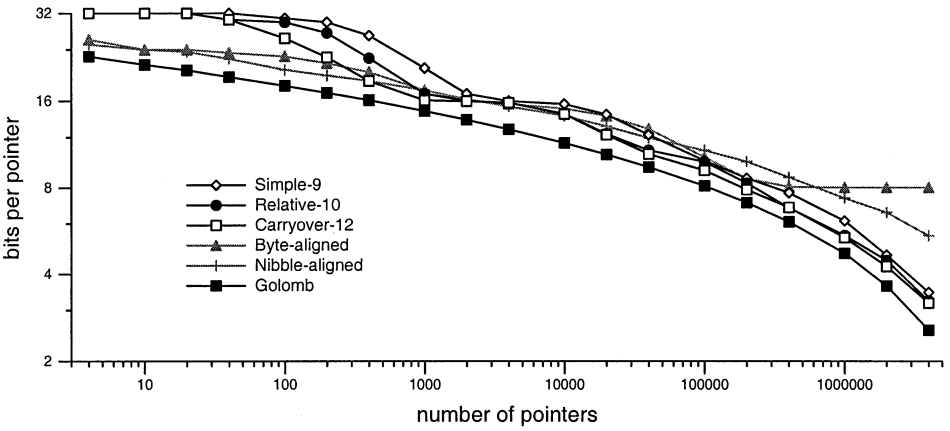

Figure 1. Bits per pointer for various integer coding methods, as a function of the number of pointers randomly generated for hypothetical inverted lists for a collection containing N = 10,000,000 documents. Compression is measured in average bits per pointer.

the graph corresponds to a value of f_t between 4 and 4,000,000. To calculate each data point, a random set of f_t distinct values was chosen, each between 1 and 10,000,000; sorted into order; and then d-gaps extracted. The exact number of bits required by each of several different coding methods to represent the list of d-gaps was then divided by f_t to get an average cost per d-gap, in bits per pointer. No allowance was made for byte rounding, or, in the case of the Golomb code, for the cost of storing the controlling parameter b.

The best code on the resulting geometric distributions is the Golomb code—it is ideal for the d-gaps that arise from randomly selected subsets, and sets a baseline that none of the other mechanisms are able to equal. The interpolative code (not shown on the graph) parallels the Golomb code, and is only fractionally worse than it. The bit-aligned and nibble-aligned codes described in the previous section are also reasonably effective when the size of each d-gap is large, but become limited by their alignment constraints for small d-gaps, and cannot represent pointers in fewer than 8 bit and 4 bits respectively, no matter how dense the list.

The solid line marked "Simple-9" in figure 1 shows the behavior of the code described in Table 1. (The suffix on the name arises because there are nine alternative partitionings of the word. Other variants are discussed shortly.) The new representation is considerably worse than the other methods for short inverted lists. The bad performance is a consequence of the quantum jump from 14 bit codes for d-gaps of less than 16,385 up to 28 bit codes for larger d-gaps, and the absence of any compromise in between. In effect, any number bigger than $2^{14} = 16{,}384$ is coded using a full 32-bit word. Furthermore, occasional values less than this limit are also coded into 32-bit words—two consecutive sub-16,384 d-gaps are required before a word containing two 14-bit codes can be emitted, and in a sparse inverted list the chances of this are slim.

On the other hand, the Simple-9 method works very well on long inverted lists. In these lists the majority of the d-gaps are uniformly small, and the ability to generate words containing 1, 2, 3, 4, 5, or 7 bit codes results in a pleasingly compact representation—better than either the byte-aligned or the nibble-aligned codes against which it is being compared.

Relative-10

The use of four bits for the selector code means that seven of the sixteen possible selector combinations are unused, a loss of almost one bit per word. One way of recovering this loss would be to eliminate a row of the table (perhaps row **e**, since use of it wastes the most bits), and choose amongst eight possibilities for each word, and use only three bits for the selection code. Doing so would allow 29 data bits in each word. Unfortunately, 29 is prime, and so the extra bit can only be used when rows **a** and **i** are in operation.

Another option would be to introduce a variable lengh selector component, so that (for example) two 15-bit codewords could be signalled by a 2-bit selector, and nine 3-bit codewords indicated by a selector that could be as long as 5 bits without affecting the overall utilization within the word. The drawback of this approach is that it reintroduces the bit operations we are striving to eliminate.

A more interesting approach is to shrink the selector to just 2 bits, leaving 30 data bits. The benefit of this is obvious—30 has many factors, and provides a wider range of partitionings

Table 2. Different ways of using 30 bits for binary codes. Only four of the ten rows in the table are available at any given time, with the subset determined by the value of the selector in the previous word. The various combinations are denoted row **a** to row **j**, and the legal transfers between rows are shown in Table 3 (method Relative-10).

Selector	Number of codes	Length of each code (bits)	Number of unused bits
a	30	1	0
b	15	2	0
c	10	3	0
d	7	4	2
e	6	5	0
f	5	6	0
g	4	7	2
h	3	10	0
i	2	15	0
j	1	30	0

than does the previous 28-bit option. Table 2 lists the ten partitionings available for 30 data bits, and shows that there are only two partitionings that result in unused bits.

The problem now is the selector. With only two bits available, we are restricted to just four choices for each word. To make the most use of those four combinations, they are interpreted relative to the selector value of the previous word, using four options: one row less; the same row; one row more; or the last row. That is, if the previous word made use of row r, then the two-bit selector in this word identifies one of row $r - 1$, r, $r + 1$, and row **j**. At the extremities of the table (when $r =$ **a** and $r =$ **j**, for example) the choices of next row are altered so as to always provide four viable alternatives. Table 3 shows the transfer matrix that dictates which rows of Table 2 are permitted to follow each other.

Encoder and decoder must agree on an initial value of "current selector" in order to process the first word. In all of the experiments described here, row **j** was assumed as an initial configuration.

A final improvement is to note that Table 3 is pessimistic, in that it allows for a *d*-gap as large as 2^{28} at all times. To provide the maximum number of options in each word, when it is known that the maximum *d*-gap in the inverted list fits into a binary code of 14 bits or less, the transfer table is adjusted to move the column of "3"s to the left, pushing over other codes as appropriate.

If the set of *d*-gaps is homogeneous, the bulk of the rows will get coded using an appropriate row of Table 2. The occasional large *d*-gap may result in a sudden shift to row **j** (or the alternative maximum row in use if it is less than **j**) and the consequent use of some long codewords, but the current state can then migrate back up the table to the natural position for this list.

Figure 1 shows the compression effectiveness of this "Relative-10" mechanism, evaluated against the methods that were described earlier. Because it permits 15-bit codewords, there is

Table 3. The transfer table, showing which rows of Table 2 are available in the context of the current row. At any given time four rows are available. The last row of the table must always be available, unless a preliminary scan shows that the largest d-gap in the list can be represented in fewer than 28 bits.

	Possible next selector values									
Current selector	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**
a	0	1	2							3
b	0	1	2							3
c		0	1	2						3
d			0	1	2					3
e				0	1	2				3
f					0	1	2			3
g						0	1	2		3
h							0	1	2	3
i							0	1	2	3
j							0	1	2	3

a marked improvement compared to Simple-9 in the section of the graph in which average d-gaps are in the range 10,000 to 100,000. Relative-10 also provides a more effective representation on very long inverted lists, since when the average d-gap is small, two extra bits of each word are being used for data.

Carryover-12

In Table 2, when 7-bit and when 4-bit codewords are being used, there are two spare bits in each word. In the third variant of the word-aligned approach, "Carryover-12", those spare bits are used to store the selector value for the next word of codes, thereby allowing that subsequent word to employ 32 rather than 30 data bits.

There are also other situations in which "spare" bits can be created in a word. For example, when there are 30-data bits, and the next two numbers are both less than 2^{14} = 16,384, then the allocation of the 30 bits as two 15-bit numbers is wasteful. Adding a "two codewords each of 14 bits" row to the table allows the pair of data values to be stored, while at the same time permitting inclusion of the next selector.

Similarly, when 32 data bits are being used, allocation of five 6-bit codes leaves two bits unused, and allocation of two 15-bit codewords (rather than two 16-bit codewords, which is retained as an additional option) similarly permits storage of the next selector.

That is, in this Carryover-12 variant, each word contains either 32 data bits, if the selector for this word can be fitted into the previous word; or contains a selector plus 30 data bits, if it cannot. In either case, if there are spare bits at the end of the current word, they are used to hold the selector for the next word in the sequence. Table 4 summarizes the situation, and shows the twelve options that are now possible, for each of 30 data bits and 32 data bits.

Table 4. Different ways of using 30 or 32 bits for binary codes. If the previous word had space for a 2-bit selector in bits that would otherwise be unused, all 32 bits in the current word are available for data. If the necessary selector is not available in the previous word, the first 2 bits in this word are used for the selector, and the remaining 30 bits are available for data. In either case, if two bits in this word remain available after the codewords are allocated, they are used to store the selector for the next word. The various combinations are denoted row **a** to row **l** (method Carryover-12).

	Previously selector available			No previous selector		
Selector	Length of each code (bits)	Number of codes	Bits for next selector?	Length of each code (bits)	Number of codes	Bits for next selector?
a	1	32		1	30	
b	2	16		2	15	
c	3	10	Yes	3	10	
d	4	8		4	7	Yes
e	5	6	Yes	5	6	
f	6	5	Yes	6	5	
g	7	4	Yes	7	4	Yes
h	8	4		9	3	Yes
i	10	3	Yes	10	3	
j	15	2	Yes	14	2	Yes
k	16	2		15	2	
l	28	1	Yes	28	1	Yes

Figure 1 also plots the compression performance of the Carryover-12 scheme on randomly generated inverted lists of different densities. Further gains in effectiveness are obtained on relatively sparse lists compared to the other two word-aligned variants, as now 16-bit codewords are possible. The Carryover-12 code also obtains better compression on dense inverted lists.

The possible drawback of the Carryover-12 mechanism is its increased complexity. Figure 2 gives pseudo-code that describes the decoding process for a single value, and it is clear that several more operations per word are required compared to either of the other two variants. To establish the extent of this additional cost, we undertook detailed experiments with four large document collections.

Other approaches

There are, of course, other possibilities. One attractive option is to add further rows to Table 4, and allow more than one selector to be inserted into the vacant bits of a word. For example, a row might be included in the left half of the table that codes a single 20-bit number, then allows six relative selectors to be inserted. The corresponding rule in the right half of the table would provide for five more selectors. In this variant the decoder maintains a queue of pending selectors it has retrieved from the trailing end of previous words, and only

```
if bits_remaining < bits_per_code then
    if bits_remaining < 2 then
        read a new word
        set selector ← the first two bits of the word
        set bits_remaining ← 30
        set table ← right
    else
        set selector ← the last two bits of the word
        read a new word
        set bits_remaining ← 32
        set table ← left
    set rownum ← transfer_table[rownum, selector]
    set bits_per_code ← table[rownum]
extract the integer stored in the next bits_per_code bits of the word
set bits_remaining ← bits_remaining − bits_per_code
```

Figure 2. Decoding one integer d-gap using the Carryover-12 mechanism. When *table* is "*left*", the second column of Table 4 is accessed as array *table*; and when *table* is "*right*", the fifth column of Table 4 is used. Array *transfer_table* in this pseudo-code is an inverse of the mapping table shown in Table 3, and is indexed by a current row number and an integer between 0 and 3, to yield a next row number.

extracts a new selector from the leading bits of any word if the queue has become empty. The number of rows in the controlling table increases, and the transfer table requires more care, since it may no longer be appropriate to move between adjacent rows. Preliminary experiments with this approach indicated that compression effectiveness was very slightly improved compared to Carryover-12, but at the cost of a more complex decoder. We have not included this mechanism in the results reported in the next two sections.

4. Compression performance

To establish the speed of the new mechanisms, and allow them to be compared against existing techniques in a practical setting, we embedded them into a test-bed information retrieval system. This section describes experiments to measure the effectiveness of the Simple-9, Relative-10, and Carryover-12 codes that were described in the previous section, and compares them in this task to a range of benchmarks. The next section extends the experimentation, and gives throughput rates for an actual retrieval system.

The discussion in Section 2 suggested that each inverted list is stored as a single list of d-gaps, to make a *document sorted index*. One issue not yet addressed is the need to also store the within-document frequency values $f_{d,t}$ that are a component of most similarity scoring mechanisms (Witten et al. 1999). Storing the frequency of each term in a document along with the pointer would require interleaving the words of two distinct streams, each with its own set of selectors and codewords—one for the document numbers, and one for the within-document frequencies.

Recent information retrieval architectures have used variant index organizations that are more amenable to word-aligned codes. In particular, our test system makes use of an *impact-sorted index* (Anh et al. 2001). In an impact-sorted index (and similarly in the *frequency-sorted indexes* of Persin et al. (1996)) the $f_{d,t}$ values (or their logical equivalent) are factored out, and each inverted list consists of a sequence of *blocks*. Within a block all pointers have the same *impact* value, which is stored only once per block, if at all; and so the set of pointers in a block can be considered to be simple sorted list of document numbers of the kind assumed in Sections 2 and 3.

The blocks in each inverted list are sorted in decreasing order of impact, and are not necessarily all examined while any particular query is being resolved. That is, in this organization query evaluation pruning is effected by only processing a subset of the blocks in each of the index lists. Collection of the "important" pointers into the first few blocks of each inverted list, and factoring out of the impact value itself, makes the impact-sorted structure an ideal match for the word-aligned codes, as there is no need to support any random access capabilities within each inverted list.

Each block of the inverted list for each term can be presumed to consist of three header fields, followed by a list of *d*-gaps:

$$\left\langle n_b, s, r, \left[d_1, d_2, d_3, \ldots, d_{n_b}\right]\right\rangle$$

where n_b is the number of *d*-gaps in this block, s is the integer-valued impact associated with the block, r is the maximum row index in the corresponding table of combinations that is required for this block, and d_i is the ith *d*-gap. In typical applications, s is a small integer between 1 and 10.

Table 5 describes the four test collections used in our investigation. All are drawn from the document sets distributed as part of the long-running *TREC* information retrieval project. The *WSJ* collection is the concatenation of the *Wall Street Journal* subcollections on disks one and two of the *TREC* corpus (Harman 1995). The collection labelled *TREC12* is the concatenation of all nine subcollections on the first two *TREC* disks, including the *WSJ* data. The collection *wt10g* is 10 GB of web data collected as part of the Very Large Collection TREC track in 2000 (Bailey et al. 2003, Soboroff 2002); and the collection *.GOV* is the

Table 5. Test collections used, showing their size in megabytes; the number of documents they contain; the number of distinct terms after case-folding and stemming; the number of document pointers in the inverted index; and the number of blocks in the impact-sorted inverted index. All of the collections are derived from data collected as part of the *TREC* information retrieval project (see `trec.nist.gov`).

	Collection			
Attribute	*WSJ*	*TREC12*	*wt10g*	*.GOV*
Size (MB)	508.5	2071.7	10511.1	18537.6
Documents (10^3)	173.4	741.9	383.4	1247.8
Terms (10^3)	295.6	1133.9	2320.0	5486.9
Pointers (10^6)	38.6	137.1	92.8	360.1
Blocks (10^3)	543.6	1909.6	3304.2	7832.9

Table 6. Compression effectiveness, measured in bits per pointer averaged across the whole index, for an impact-sorted index with 10 different impact values. Costs include block headers and word-padding costs on each inverted list, but not components usually associated with the vocabulary structure, such as f_t. The code "Simple-9" uses a 4-bit selector in each word, and 28 data bits, as shown in Table 1. The row "Relative-10" uses a 2-bit selector that is relative to the previous word, and 30 data bits, as shown in Table 2. The code "Carryover-12" stores relative selectors in the spare bits of the previous word, if possible, and uses either 30 or 32 data bits, as shown in Table 4.

	Collection			
Method	*WSJ*	*TREC12*	*wt10g*	*.GOV*
Golomb	6.62	7.54	8.59	8.43
Interpolative	6.64	6.88	7.86	8.49
Byte-aligned	10.07	10.23	10.79	11.38
Nibble-aligned	9.50	9.58	10.37	11.13
Simple-9	8.66	9.05	10.35	11.13
Relative-10	8.19	8.60	10.02	10.35
Carryover-12	7.86	8.35	9.73	10.04

18 GB collection that was built by crawling the .gov domain in 2002 (Craswell and Hawking 2002).

In our experimental environment, a retrieval system is built for each collection as a monolithic whole. The total cost of the index is thus the combined cost of storing all of the equal-impact blocks for all of the inverted lists, where the document numbers stored in each block are a sorted subset of the integers from 1 to N, the number of documents in that collection (Table 5), and where each inverted list contains as many as 10 blocks.

Table 6 lists the total cost of storing compressed inverted indexes of this kind for different compression methods, expressed in terms of bits per pointer stored. Values listed are inclusive of all of the data stored in each inverted list, plus any byte and word alignment costs at the end of blocks and whole inverted lists. They do not include the cost of the vocabulary file, which is both constant for all of the compression methods, and relatively small. In the case of the Golomb code, the parameter b is calculated as $b = 0.69 \times (N/n_b)$, where N is the number of documents in the collection, and n_b is the number of document numbers in this block (Witten et al. 1999). That is, no additional overhead is incurred by the Golomb code.

For example, on collection *.GOV* the Golomb code stores the index in 8.43 bits per pointer on average, and as it (Table 5) contains 360.1×10^6 pointers, the total index costs approximately $8.43 \times 360.1 \times 10^6/8/1024/1024 = 361.9$ MB, or about 1.9% of the collection being indexed.

As can be seen from the table, the word-aligned codes described in this paper are inferior in an effectiveness sense to the bit-aligned Golomb and interpolative codes. The latter coding method works particularly well when the collection is non-homogeneous (*TREC12* and *wt10g*). On the other hand, the word-aligned codes are never worse than either the byte-aligned code or the nibble-aligned codes, and represent a considerable saving in storage space compared to them.

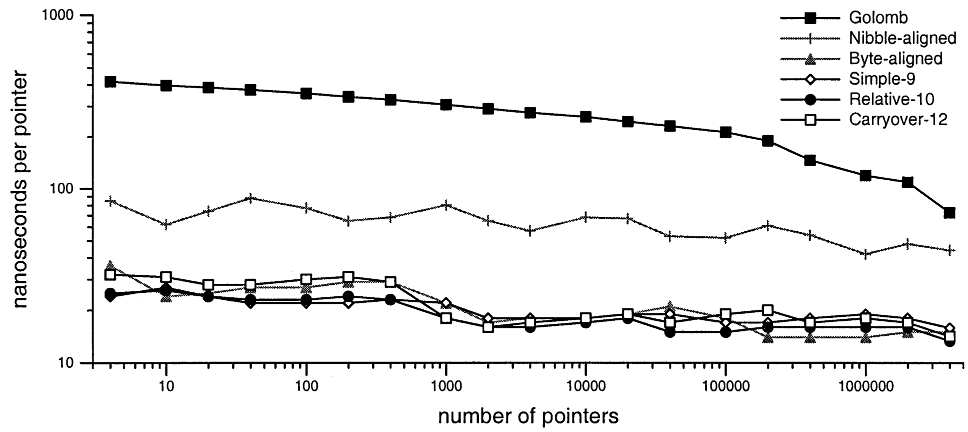

Figure 3. Decoding speed measured in CPU nanoseconds per pointer, as a function of the number of pointers randomly generated for hypothetical inverted lists for a collection containing $N = 10{,}000{,}000$ documents. These experiments were conducted on a 933 MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux.

5. Decoding speed

The more interesting question is whether or not the word-aligned methods suffer any penalty in decoding speed compared to the byte-aligned code. Figure 3 plots decoding cost, in CPU nano-seconds per pointer decoded, as measured in an experiment identical to that shown in figure 1. That is, synthetic index lists were generated containing a random subset of an assumed $N = 10{,}000{,}000$ documents; compressed using the various representations being considered; and then decoded under controlled conditions. The time taken on a 933 MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux to decode the entire index list (working from an array of binary words in memory, and generating an output array of integer d-gaps, also in memory) was divided by the number of pointers in the list to get a per-pointer decompression rate.

As can be seen, the Golomb code is expensive compared to the other mechanisms, and is not competitive. (The interpolative code, which is not plotted, was a further 10% slower than the Golomb code.) Nibble-aligned coding is moderately fast, but outperformed by the byte-aligned and word-aligned codes. The slight extra complexity of the Carryover-12 mechanism (figure 2) means that it is fractionally slower than the Simple-9 and Relative-10 variants, but all three operate at high speed, and are direct competitors with the byte-aligned code.

In the final analysis, raw decoding speed is less important in an information retrieval environment than is query throughput rate. Table 7 shows average query resolution times in milliseconds per query averaged over 10,000 random queries containing on average three terms each. For example, three of the queries used on the *wt10g* collection were "digital board", "fever forehand", and "exceptional captured operate circuits contamination".

The times shown in Table 7 are the average elapsed time between the moment the query enters the system, and the time at which a ranked list of 1,000 answer documents has been

Table 7. Impact of different coding schemes on query processing speed. Each value is the average of the elapsed time (in milliseconds) between when a query enters the system and when a ranked list of the top 1,000 answers is finalized, but not retrieved. The average is taken over 10,000 artificial queries with mean length of three. The hardware used was a 933 MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux.

	Collection			
Method	*WSJ*	*TREC*	*wt10g*	*.GOV*
Golomb	12.7	43.5	91.0	80.5
Interpolative	14.9	48.2	101.8	91.2
Byte-aligned	9.5	30.7	58.7	53.2
Nibble-aligned	8.6	31.9	76.1	68.0
Simple-9	7.7	28.6	56.6	47.2
Relative-10	5.8	23.3	55.0	47.9
Carryover-12	5.3	23.3	57.5	50.5

prepared, but without any of those answers being retrieved or presented to the user. In all cases, an impact sorted index with 10 different impact levels was used, together with the fast query evaluation process described by Anh et al. (2001), but without any use of pruning or early termination heuristics. The same hardware—a 933 MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux—was used in these experiments.

Our word-aligned integer compression mechanisms allow query throughput rates that approach twice those of the bit-aligned Golomb code, and are better than the throughput rates achieved by the byte-aligned code.

6. Conclusion

The word-aligned schemes proposed in this paper provide a new trade-off point between compression efficiency and compression effectiveness when coding the inverted indexes used in large information retrieval systems. They handsomely outperform Golomb codes for decoding speed and query processing throughput, while at the same time permit markedly improved compression effectiveness compared to the well-known byte-aligned compression mechanism.

Common to all three variants is the use of each output word to store a number of flat binary codes in a uniform manner. This scheme can thus only be used when the probability distribution is smooth—the sequence of d-gaps $\langle 1, \ell, 1, \ell, 1, \ell, \ldots\rangle$, where ℓ is a large number, has a very high degree of structure for a more principled compression mechanism to exploit, but is expensive for the word-aligned coder. However such pathological distributions are most unlikely in the application used as an example in our work, and for the four document collections used in our experiments the word-aligned code has clear benefits.

Other approaches are also possible. In particular, we have been experimenting recently with a variant of the Carryover-12 mechanism in which binary codes can span word boundaries. Preliminary results indicate that compression effectiveness can be improved, without degrading query response times (Anh and Moffat 2004).

Software. An implementation of the Carryover-12 mechanism is available from `http://www.cs.mu.oz.au/˜alistair/carry/`.

Acknowledgment

This work was supported by the Australian Research Council.

References

Anh VN, de Kretser O and Moffat A (2001) Vector-space ranking with effective early termination. In: Croft WB, Harper DJ, Kraft DH and Zobel J, Eds., Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New Orleans, LA, Sept. ACM Press, New York, pp. 35–42.

Anh VN and Moffat A (2004) Index compression using fixed binary codewords. In: Schewe K-D and Williams H, Eds., Proc. 15th Australasian Database Conference, Jan. Dunedin, New Zealand, pp. 61–67.

Baeza-Yates R and Ribeiro-Neto B (1999) Modern Information Retrieval. ACM Press, New York.

Bailey P, Craswell N and Hawking D (2003) Engineering a multi-purpose test collection for web retrieval experiments. Information Processing & Management, 39(6):853–871.

Blandford D and Blelloch G (2002) Index compression through document reordering. In: Storer JA and Cohn M, Eds., Proc. 2002 IEEE Data Compression Conference, April, IEEE Computer Society Press, Los Alamitos, CA. pp. 342–351.

Craswell N and Hawking D (2002) Overview of the TREC-2002 web track. In: Voorhees EM and Harman DK, Eds., The Eleventh Text REtrieval Conference (TREC 2002) Notebook, Nov. Gaithersburg, MD. National Institute of Standards and Technology. NIST Special Publication SP 500-251, pp. 248–257, available at `http://trec.nist.gov/pubs/trec11/t11_proceedings.html`.

de Moura ES, Navarro G, Ziviani N and Baeza-Yates R (2000) Fast and flexible word searching on compressed text. ACM Transactions on Information Systems, 18(2):113–139.

Frakes WB and Baeza-Yates R (1992) Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, NJ.

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

Moffat A and Stuiver L (2000) Binary interpolative coding for effective index compression. Information Retrieval, 3(1):25–47.

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.

Salton G (1989) Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley, Reading, MA.

Scholer F, Williams HE, Yiannis J and Zobel J (2002) Compression of inverted indexes for fast query evaluation. In: Beaulieu M, Baeza-Yates R, Myaeng SH and Jarvelin K, Eds., Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August, Tampere, Finland, ACM Press, New York, pp. 222–229.

Soboroff I (2002) Does wt10g look like the web? In: Beaulieu M, Baeza-Yates R, Myaeng SH and Järvelin K, Eds., Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August, Tampere, Finland, ACM Press, New York, pp. 423–424.

Trotman A (2003) Compressing inverted files. Information Retrieval, 6:5–19.

Williams HE and Zobel J (1999) Compressing integers for fast file access. The Computer Journal, 42(3):193–201.

Witten IH, Moffat A and Bell TC (1999) Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edition. 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.