Index Compression using Fixed Binary Codewords

Vo Ngoc Anh **Alistair Moffat**

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

http://www.cs.mu.oz.au/

Abstract

Document retrieval and web search engines index large quantities of text. The static costs associated with storing the index can be traded against dynamic costs associated with using it during query evaluation. Typically, index representations that are effective and obtain good compression tend not to be efficient, in that they require more operations during query processing. In this paper we describe a scheme for compressing lists of integers as sequences of fixed binary codewords that has the twin benefits of being both effective and efficient. Experimental results are given on several large text collections to validate these claims.

Keywords: Index compression, integer compression, web searching.

1 Introduction

Document retrieval and web search engines index large quantities of text. Collection sizes range from hundreds of megabytes for organizational document collections, through to tens of terabytes for web search engines. Access to these collections is typically via term-based queries, where documents are selected from the collection and offered as answers based on the presence of query terms, and on an evaluation of the similarity between the query and the text of the document. For an introduction to these concepts, see Witten et al. [1999].

The index of a large collection is itself also large. It stores, for every word and number (and sometimes phrase) that appears in the collection, a list of the documents in which that concept appears. Hence, in raw terms, the index occupies a significant fraction of the space occupied by the retrieval system as a whole.

Fortunately, the index is amenable to compression, and the static costs associated with storing the index can be traded against dynamic costs associated with using it during query evaluation. For example, when stored uncompressed, access is fast, but the space consumption is high. Principled compression techniques, some of which are summarized below, are able to greatly reduce the space consumed, but at the potential cost of slowing access to the index. Typically, index representations that are *effective*, in that they obtain good compression, tend not to be *efficient*, in that they require more operations during query processing than do simpler methods.

In this paper we describe a scheme for compressing lists of integers as sequences of fixed binary codewords that has the twin benefits of being both effective and efficient. One way of looking at our results is to say that we build on a simple coding scheme, to improve the compression obtained without sacrificing speed. Alternatively, our work can be viewed as providing a faster implementation of principled compression methods. Either way, the result is the same – our method is both effective and efficient for representing the indexes associated with text retrieval systems. Experimental results are given on several large text collections to validate these claims.

2 Index compression

In an inverted file-based text retrieval system, for each term that appears anywhere in the collection of documents, an index list is constructed that contains the ordinal numbers of all of the documents that contain that term. For a term t, the index list has the structure

$$\langle f_t; d_1, d_2, d_3, \ldots, d_{f_t} \rangle ,$$

where f_t is the number of documents containing t, and d_i is an integer that identifies the document associated with the ith appearance of t.

Most index compression mechanisms transform the index list into a set of d-gaps by taking consecutive differences,

$$\langle f_t; d_1, d_2 - d_1, d_3 - d_2, \ldots, d_{f_t} - d_{f_t - 1} \rangle ,$$

and then using a code for integers that is either static, or is controlled by a small number of parameters.

For example, one representation that is in use in some implementations is the following *byte aligned code*. To code an integer x, the seven low-order bits of x are isolated. If the remaining high-order bits of x are all zero, the seven low-order bits are prefixed by a "1" bit and output as a single byte. On the other hand, if the high-order bits of x are non-zero, the seven low order bits are coded as a byte with a "0" prefix, and the entire process is repeated on the high-order bits. The result is that x is broken into seven-bit chunks, and as many bytes as are needed to code the non-zero chunks are written to the output file. Scholer et al. [2002] and Trotman [2003] consider the implementation of this scheme, and describe experiments to measure its practicality.

Using the byte-aligned method, integers up to 128 (presuming that the integers being handled are "one-origin" and satisfy $x \geq 1$) are represented in a single byte; integers up to $16{,}384$ in two bytes (or up to $2^{14} + 2^7 =$

16,512, if the codes are adjusted to ensure that the mapping is one to one); and so on. The benefit of the byte aligned code is that it is extremely fast to compute the transformation, as a relatively small number of masking and shifting operations are required per codeword.

Parameterized codes take advantage of the different average d-gaps in different inverted lists. The lists associated with common words contain many small d-gaps, while the lists associated with infrequent terms contain a smaller number of large d-gaps. One landmark code that adjusts to this differing density is that of Golomb [1966]. In a Golomb code, integer $x \geq 1$ is represented relative to parameter b by sending $\lfloor (x-1)/b \rfloor$ using a zero-origin unary code; and then $x - b \times \lfloor (x-1)/b \rfloor$ using a one-origin binary code of either $\lfloor \log_2 b \rfloor$ or $\lceil \log_2 b \rceil$ bits. Witten et al. [1999] give details of the Golomb code, and example codewords.

When the average d-gap is small, parameter b should also be small; and when the average d-gap is large, b should also be large. In particular, if term t appears in f_t randomly-selected documents, then the set of d-gaps for t can be assumed to be drawn from a geometric distribution with probability $p = f_t/N$, where N is the number of documents in the collection. Moreover, the choice $b = (\ln 2)/p$ results in a Golomb code that is a minimum-redundancy code (Huffman code) for the infinite probability distribution $\Pr(x) = (1-p)^{x-1}p$. That is, a Golomb code is an optimal code for terms that are randomly distributed across documents.

There are three ways in which Golomb codes can be improved on in practice. First, they are optimal only in the sense of discrete codewords, and when the probability p is high, use of arithmetic coding yields superior compression effectiveness. In an arithmetic code, symbols can be allocated bit-fractional codewords [Witten et al., 1999].

Second, words do not appear in documents at random, and in any collection there are terms that are more frequently used in some parts of the collection than in others – think about the words "Olympics" and "election", for example, both of which spring into prominence every four years. Hence, a compression mechanism that is sensitive to clustering might outperform Golomb coding on non-uniform inverted lists. One such mechanism is the interpolative code of Moffat and Stuiver [2000]. It uses binary representations to code the document numbers in each inverted list in a non-sequential order, so that as each number is coded, bounds on the range of that number are available that limit the number of bits required in the corresponding codeword. Moffat and Stuiver showed that for typical document collections, the interpolative code achieves better compression than a Golomb code.

The third drawback of Golomb codes is that they are less than desirable in an efficiency sense, since decoding them involves accesses to individual bits, which slows query processing. It is the quest for high decoding rates that has led to the use of byte-aligned codes in commercial systems. Brisaboa et al. [2003] have recently generalized the byte-aligned approach, and described a mechanism that allows a fractional number of bits to be used as the flag to indicate whether or not this is the last byte in a given codeword.

Here we take a different approach, and describe a mechanism that uses fixed binary codes for d-gaps, with selector codes interspersed periodically to indicate how many bits there are in a segment of subsequent binary codes. Details of this approach are provided in the next section, after a further static code is described.

3 Binary codes

Another classic integer representation is that of Elias [1975]. In the Elias C_γ code, integer $x \geq 1$ is coded in two parts – a zero-origin unary code for the value $\lfloor \log_2 x \rfloor$, and then a zero-origin binary code of that many bits to place x within the magnitude established by the unary part. In a sense, each binary part can be regarded as a set of "data" bits associated with x, and the unary part as a selector that indicates how many data bits there are. The coded stream is then an alternating sequence of selector and data components, where each data component is a flat binary code for a value between 0 and $2^k - 1$, for the integer k described by the selector part.

As a running example, consider the following list of d-gaps for some hypothetical term t for which $f_t = 12$:

$$\langle 12; 38, 17, 13, 34, 6, 4, 1, 3, 1, 2, 3, 1 \rangle .$$

An Elias code for this list would represent the data components of these twelve d-gaps in $5 + 4 + 3 + 5 + 2 + 2 + 0 + 1 + 0 + 1 + 1 + 0 = 24$ bits respectively. Because the selector part of each codeword is represented as a zero-origin unary value, a further $24 + 12 = 36$ bits are required for the twelve selectors. In total, the list requires 60 bits.

On the other hand, if a flat binary code was to be used across the whole list, each codeword would consume six bits, since the largest d-gap is 38, and $\lceil \log_2(38-1) \rceil = 6$ (minus one because the d-gaps are one-origin). A total of $12 \times 6 = 72$ bits would then be required, plus the additional one-off cost of indicating that each codeword was six bits long.

Other possibilities exist between these two extremes. For example, if the list is broken into two six-item halves, and each half treated independently, the first six d-gaps still require six bits each, but the second six require only two bits each. The overall saving is less than the suggested 24 bits, because two lots of codeword length information must now be transmitted; but even so, there is likely to be a net reduction compared to both the flat binary code and the Elias code.

The decomposition into sublists can be based on any useful split. For example, another possibility is

$$\langle 12; (6, 4 : 38, 17, 13, 34), (3, 1 : 6), \\ (2, 7 : 4, 1, 3, 1, 2, 3, 1) \rangle ,$$

where the notation $(w, s : d_1, d_2, \ldots, d_s)$ indicates that w-bit binary codes are to be used to code each of the next s values. Depending on how the selectors are represented, this decomposition might yield further savings.

With that example, we have now introduced the approach we pursue in this paper – a representation in which selectors span multiple sets of data bits, and the extent of that span is chosen so as to ensure an output bit stream that is as compact as is possible. The issues requiring resolution are thus:

- what data widths w and span values s are appropriate;
- how the data width and span values should be represented; and
- how to effectively decompose the initial list into segments, each of which can be handled by a permitted combination of data width and span.

data bits	span		
	s_1	s_2	s_3
-3	code 1		
-2	code 2	code 3	
-1	code 4	code 5	code 6
0	code 7	code 8	code 9
$+1$	code 10	code 11	code 12
$+2$	code 13	code 14	
$+3$	code 15		
max	code 16		

Table 1: Sixteen combinations of relative data size and span. The bits used for the data values are relative to the size of the last-used data bits value, except in the case of *max*, which is always the largest data bits value for this inverted list. The span is one of three fixed values s_1, s_2, or s_3.

In the remainder of this paper we address these three questions, with an emphasis throughout on retaining practicality. In particular, the solutions we articulate are not optimal, but are grounded in a desire to maintain fast decoding. So we eschew the unary codes associated with the selector parts of the Elias and Golomb representations, and instead use a fixed binary code for the selector parts as well as all of the data parts.

Selectors

In our initial investigation of possible binary codings for d-gaps we restricted each span so that the selector plus the associated data bits fitted within one complete machine word of 32 bits [Anh and Moffat, 2004]. The intention was to retain word-alignment, even if that meant wasting some bits within each word. A consequence of this choice was that the range of possible data widths was dictated by the divisors of 32. For example, it makes no sense to consider fitting three 9-bit codes into a word when three 10-bit codes fit equally well.

In this paper, the word-alignment constraint is relaxed, and we are free to choose any span, and also allow any data width. In a collection of N documents, the largest number of data bits that can be required is $\mathit{max} = \lceil \log_2(N-1) \rceil$, and the smallest is zero, for the integer 1. For a typical collection, *max* can be expected to be at least 20, and perhaps as large as 30.

This wide range of data sizes suggests that the selector needs to contain five bits just to convey the data width, let alone the accompanying span. However, it is also possible to code the data widths in fewer bits if they are taken to be relative to the previous one. Table 1 lists sixteen combinations of width and span, with the data width taken in each case to be relative to the width used in the previous segment. The lengths s_1, s_2, and s_3 governing the spans are discussed shortly.

For example, if $w = 5$ bit codes have most recently been emitted, then the selector value "code 6" indicates that the next s_3 gaps are all coded as $w = 5 - 1 = 4$ bit binary numbers.

Table 1 represents the simplest case, when $w - 3 \geq 0$ and $w + 3 < \mathit{max}$. In other situations, when w is closer to either 0 or *max*, the shape of the table is adjusted to avoid wasting selector possibilities.

As a more complete example, consider again the list

$$\langle 12; 38, 17, 13, 34, 6, 4, 1, 3, 1, 2, 3, 1 \rangle ,$$

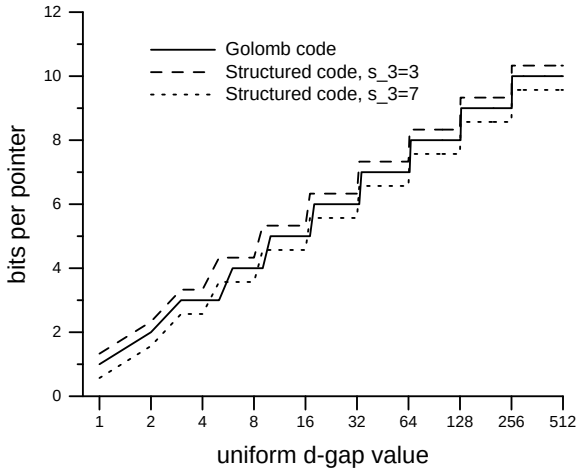

Figure 1: Cost per pointer of Golomb code and structured code, when all d-gaps are the same value, and when each selector requires four bits and covers a span of as many as either $s_3 = 3$ or $s_3 = 7$ gaps.

and suppose that $s_1 = 1$, $s_2 = 2$, and $s_3 = 4$. Using the differential codes listed in Table 1, and presuming that the initial value of the width w is taken to be *max* (in this case, 6) and that no table adjustments are made, the list can be coded as

$$\begin{array}{ll} \langle 12; & \\ \quad (\text{code } 9 : 38, 17, 13, 34), & |w = 6, s = 4| \\ \quad (\text{code } 1 : 6), & |w = 3, s = 1| \\ \quad (\text{code } 6 : 4, 1, 3, 1), & |w = 2, s = 4| \\ \quad (\text{code } 9 : 2, 3, 1) & |w = 2, s = 3| \\ \rangle , & \end{array}$$

requiring a total of 16 bits for the four selectors, plus $\sum w \times s = 6 \times 4 + 3 \times 1 + 2 \times 4 + 2 \times 3$ bits for the twelve data values. In total, $16 + 41 = 57$ bits are required, three bits fewer than the Elias code, and exactly the same number of bits as is required by a Golomb code.

Note that the sixteen combinations shown in Table 1 are chosen on a relatively ad-hoc basis. Nevertheless, the arrangement results in excellent compression being obtained, a claim that is defended below.

Representation

Given that there are sixteen possible selector values shown in Table 1, it is straightforward for each selector to be represented in four bits. By combining the width and the span into a single value, as many as s_3 gaps are indicated by a single four-bit selector.

Table 1 allows three different spans to be employed. The obvious assignment is to take $s_1 = 1$, $s_2 = 2$, and $s_3 = 3$. But other choices also make sense, including $s_1 = 1$, $s_2 = 2$, and $s_3 = 4$; and $s_1 = 1$, $s_2 = 3$, and $s_3 = 7$. Three-valued sequences starting at two might also be sensible – it might be cheaper to use a selector with a span of two that wastes a data bit or two, than to code each d-gap using the exact number of data bits.

Figure 1 shows the relative cost of using a Golomb code and the code described by Table 1 to code uniform lists of identical d-gaps, plotted as a function of the size of the d-gap, and presuming that either $s_3 = 3$ or $s_3 = 7$. This input is ideal for the structured code, as all d-gaps require the same data width, and hence each selector can span s_3 gaps. As can be seen from the graph, when

$s_3 = 3$, the code is little more expensive than a Golomb code, and when $s_3 = 7$ it is capable of outperforming the Golomb code, even for uniform data.

Another situation in which the new code outperforms the Golomb code is when the list of d-gaps is highly structured in another sense, with many large d-gaps in one half, and many small d-gaps in the other. The Golomb parameter b is calculated holistically, and each codeword then has a fixed minimum length dictated by b. On the other hand, in the new code, the half of the list with the small d-gaps will be coded with small data widths w.

Alternative versions of the code are categorized in terms of their set of permitted span values. For example, a 1-2-3 implementation makes use of $s_1 = 1$, $s_2 = 2$, and $s_3 = 3$. Experimental results showing the compression effectiveness of various span combinations appear in Section 5 below.

Parsing

Once the span values have been chosen, the next task is to parse the sequence of d-gaps into segments, each of which is handled by a single selector.

Optimal parsing can be achieved through the use of a shortest path labelling of a graph in which each node represents a combination of a d-gap and the data width used to code it, and each outgoing edge from that node represents one way in which a selector might be applied to cover some number of subsequent gaps. The cost of each edge in the graph is the number of bits consumed by the selector value, plus the product of the indicated span s and data width w values.

In a list of f_t gaps in which the maximum data width w is given by *max*, there are potentially as many as $f_t \times max$ nodes in the graph. In practice the number of interesting nodes in the graph is much smaller than this limit, as no node is needed if the value of a d-gap exceeds the nominal incoming data bit width. For example, a d-gap of 18 cannot be coupled with data widths of less than five, so four of the graph nodes counted in the expression $f_t \times max$ can be eliminated.

Each node in the graph might have as many as 16 outgoing edges, but again this is an upper bound, as no selector value can be used if any of the next s_i gaps are too large to be stored in the indicated number of bits w associated with that selector.

Consider again the same example sequence

$$\langle 12; 38, 17, 13, 34, 6, 4, 1, 3, 1, 2, 3, 1 \rangle .$$

The graph starts with a special source node $(0, max = 6)$ that represents the starting state, prior to any d-gaps being coded. From that node three edges are possible: one to a node $(1, 6)$ with cost 10 that indicates that a single d-gap is coded in six bits; a second to a node $(2, 6)$ with cost 16 that indicates that two d-gaps are each coded in six bits; and a third to the node $(4, 6)$ with cost 28 that indicates that a total of four d-gaps are each coded in six bits.

Similarly, from the node $(1, 6)$, five edges are possible: to node $(2, 5)$ (selector code 4); to node $(3, 5)$ (selector code 5); to node $(2, 6)$ (selector code 7); to node $(3, 6)$ (selector code 8); and to node $(5, 6)$ (selector code 9). None of the other eleven possible selectors are applicable, either because they make use of data widths that are too small for one or more of the upcoming d-gaps, or because they would imply the use of data widths greater than *max*. Figure 2 shows the permissible edges for the first few d-gaps in the example sequence.

Once the graph has been constructed, the minimum cost representation is found by starting at the source node of the graph, $(0, max)$, and solving a single-source shortest path problem to label each node in the graph with its minimum cost. The graph contains as many as *max* nodes with labels $(f_t, \cdot)$, and the least cost labelling across this set of nodes defines the best labelling of any path from the source node to a node that codes the final d-gap, d_{f_t}. Identification of the shortest path indicates how the sequence of d-gaps should be parsed and coded.

Finding the optimal parsing is a relatively complex process. Sub-optimal mechanisms might yield equally good representations in practice, with a (hopefully) small amount of effectiveness exchanged for speed of computation and ease of implementation. One standard technique for approximating computations of this kind is to use a *greedy* mechanism, choosing at each moment the permissible alternative that offers the best local economy. In a sense, the graph is traversed from source node to final nodes, without ever reducing the cost label assigned to any intermediate node, and never backtracking.

Taking the same example list again, and reducing it to a list of codeword lengths gives

$$\langle 12; 6, 5, 4, 6, 3, 2, 0, 1, 0, 1, 1, 0 \rangle .$$

Starting at a source node of $(0, max)$ opens up three possible first moves: use of $s_1 = 1$, and a single 6-bit code; use of $s_2 = 2$, and two 6-bit codes; or use of $s_3 = 4$, involving four 6-bit codes. To choose between these three moves, consideration is given as to whether the bit-saving arising from the use of a shared selector is greater than any bit wastage arising by making data components larger than minimally necessary. In the example list, moving from $s_1 = 1$ to $s_2 = 2$, wastes $6 - 5 = 1$ data bits, but saves four selector bits. Hence, $s_2 = 2$ is preferable to $s_1 = 1$. Similarly, in extending that same selector to $s_3 = 4$, two further data bits are wasted, which is less than the saving generated by extending the selector. Hence, the first selector can be confirmed as being $s_3 = 4$, and the next node on the chosen path through the graph is $(4, 6)$, with cost 28. The process then repeats from that node, comparing the number of wasted bits caused by each of s_1, s_2, and s_3, and making a choice that in that neighborhood minimizes the wastage.

All of the experiments reported below make use of this greedy approach, and the optimal approach has not been implemented. Given the excellent compression results attained by the approximate mechanism, it seems unlikely that any significant gain can accrue from switching to an optimal method, and it is not clear that the implementation and computation costs involved can be justified.

4 Extensions

Two further refinements are described in this section, and evaluated in the experiments described in Section 5.

Adding a multiplier

Each list has a different nature, and use of any fixed set of three values s_1, s_2, and s_3 is inevitably a compromise. The first extension considered is to add one more "knob" to the arrangement, in the form of a multiplier m. Suppose, for example, that a 1-2-3 arrangement of selectors is being used. The parameter m serves a role in each inverted list as a scale factor, so that when $m = 1$, the list is processed using selectors in the baseline 1-2-3 spectrum;

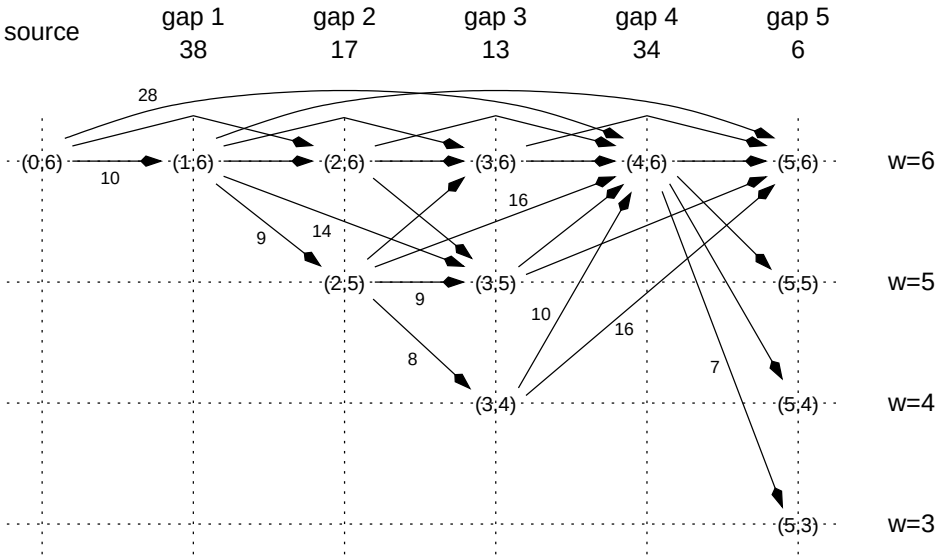

Figure 2: Part of the graph of possible parsings for the example sequence of d-gaps. Each of the as many as $f_t \times max$ graph nodes corresponds to a position in the sequence of d-gaps, and a data width w with which that d-gap was coded. The labels beside some of the edges represent their cost in bits.

when $m = 2$, the list is processed using a 2-4-6 combination; and so on. The best value of m is identified for each list using a brute force evaluation – starting at $m = 1$, the greatest value of $m \leq 8$ for which the cost of storing the list decreases, is chosen. The value of m is then stored in conjunction with that compressed list.

Extending the runs

In situations where the d-gaps in a list are of uniform magnitude, spans of significantly greater than s_3 might be warranted. The final modification to our method is to recognize this need, and provide an "escape" mode to accommodate long spans. After any span of s_3, a further 4-bit nibble is inserted into the data stream to allow for more d-gaps requiring the same number of data bits. Use of a nibble allows a number between 0 and 15 to be represented, meaning that in an extreme case, $15m$ gaps can be represented using an additional 4 selector bits. In this scheme the escape nibble is required after every occurrence of s_3, even if the next d-gap has a different number of data bits than the current set.

5 Experiments

In order to provide an experimental evaluation, the new mechanism has been incorporated in an information retrieval system, and efficiency and compression effectiveness measurements made on four large data collections. Those document sets are described in Table 2. The same four collections were also used for the experimentation carried out in connection with our word-aligned binary-based index representation [Anh and Moffat, 2004].

All of the collections are drawn from data 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 18 GB collection that was built by crawling the `.gov` domain in 2002 [Craswell and Hawking, 2002].

Attribute	Collection			
	WSJ	*TREC12*	*wt10g*	*.GOV*
Size (MB)	509	2072	10511	18538
Documents (10^3)	173	742	1692	1248
Terms (10^3)	296	1134	6789	5487
Pointers (10^6)	39	137	385	360
Blocks (10^3)	544	1910	9402	7833

Table 2: Test collections used in the experiments. All of the collections are derived from data collected as part of the *TREC* information retrieval project (see `trec.nist.gov`).

The index in our retrieval system is slightly more complex than was suggested above, and makes use of an *impact-sorted* organization [Anh et al., 2001], in which each inverted list consists of a sequence of *blocks* of document numbers, each of which shares a common score in terms of establishing document relevance. (Note that this paper does not discuss the scoring heuristics at all, and when we refer to effectiveness, we are discussing the extent to which the index is stored in a minimal amount of memory or disk space.)

That is, instead of maintaining one sorted index list per term, as many as $b = 10$ sorted index blocks per term are manipulated, where b is an attribute of the similarity heuristic governing the relevance calculation, and is not a parameter of the compression process. In this representation, the total cost of the index is the combined cost of storing all of the equal-impact blocks for all of the inverted lists. In each block, document numbers are stored as a sorted subset of the integers from 1 to N, the number of documents in that collection (Table 2).

Table 3 lists the total cost of storing compressed inverted indexes of the kind described in this paper, expressed in terms of bits per pointer stored. All 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

Method	Bits per pointer			
	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
Carryover-12	7.86	8.35	9.73	10.04
1-2-3	7.53	7.92	9.17	9.48
1-2-4	7.45	7.86	9.12	9.41
1-3-7	7.66	7.98	9.08	9.73
2-4-6	7.34	7.73	8.98	9.28
2-4-8	7.33	7.73	8.99	9.28

Table 3: Compression effectiveness, measured in bits per pointer averaged across the whole index, for an impact-sorted index with $b = 10$ surrogate weights. The first four rows are drawn from an equivalent table presented in Anh and Moffat [2004].

Method	Bits per pointer			
	WSJ	*TREC12*	*wt10g*	*.GOV*
1-2-3 $\times m$	7.21	7.62	8.88	9.14
1-2-4 $\times m$	7.22	7.58	8.81	9.15
1-2-3 $\times m$, escape	7.11	7.51	8.83	9.01
1-2-4 $\times m$, escape	7.11	7.48	8.82	9.02

Table 4: Compression effectiveness, measured in bits per pointer averaged across the whole index, for an impact-sorted index with $b = 10$ surrogate weights. The four mechanisms shown in this table are described in Section 4.

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.

The first half of Table 3 shows various base-line systems: the Golomb code; the interpolative code; the byte-aligned code described earlier; and the best of the word-aligned codes described by Anh and Moffat [2004]. The latter mechanism – dubbed "Carryover-12" – shares some attributes with the mechanism described in this paper. The principal difference between them is that in the earlier work we required that every selector describe a 32-bit word of binary codes, which meant that there were wasted bits at the end of most words. The mechanism described here rectifies that deficiency, by allowing codes to span word boundaries.

The bottom half of Table 3 then reports the compression effectiveness of several versions of the approach described in Section 3, parameterized in terms of the three values s_1, s_2, and s_3. Compared to the Carryover-12 mechanism, elimination of the wasted bits at the end of each 32-bit word is clearly advantageous, and compression effectiveness is uniformly improved. The two 2-4-x alternatives provide the most compact representation. In these, the four-bit selector is always spread over at least two codewords.

It was the observation that 2-4-6 is "double" 1-2-3, and that 2-4-8 is double 1-2-4, that led to the first extension described in Section 4. The first two rows of Table 4 show the additional gain in compression effectiveness brought about by this "$\times m$" extension, in which a locally best value for multiplier m is chosen for each block in the index. While the gain in effectiveness is small, it is consistent across all of the collections, and adding in this extra flexibility is of benefit. Adding the escape nibble then achieves a further slight gain in compression effectiveness, and the final row of Table 3 reflects compression rates within about 0.5 bits per pointer of those attained by a Golomb code, and equal to the Golomb code on the heterogenous collection *TREC12*.

Value of m	Fraction of population (%) by		
	blocks	selectors	pointers
1	77.88	13.83	3.54
2	4.37	5.24	2.57
3	8.06	27.41	23.26
4	3.68	14.10	15.06
5	2.85	36.17	49.37
6	1.34	2.62	4.18
7	0.87	0.38	0.60
8	0.95	0.25	1.42

Table 5: Fraction of index blocks, selector groups, and underlying pointers compressed using different values of m, summed across all four test collections, for method "1-2-4$\times m$, escape". The great majority of pointers are stored in blocks for which m is between 3 and 5.

Finally in this section, Table 5 shows the distribution of values of m, summed across the four collections. Column two of that table shows that the great majority of index blocks are coded using $m = 1$. However, these tend to be blocks with relatively few selectors and pointers in them, and the outcome is markedly different when the values of m are broken down by the number of selectors, or the number of pointers. Fewer than 4% of the pointers in these four indexes are best coded with $m = 1$, and nearly half of the pointers are coded in blocks for which $m = 5$. In this latter case the three spans in use are $s_1 = 5$, $s_2 = 10$, and $s_3 = 20$.

6 Decoding Speed

Our rationale in Anh and Moffat [2004] for working with word-aligned code was that access to whole words would be fast. The code presented in this paper allows codes to span word boundaries, and thus obtains better compression effectiveness. The obvious question now is whether or not access speed to the compressed index is compromised by the change.

Raw decoding speed is less important in an information retrieval environment than is query throughput rate. Figure 3 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 Figure 3 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 prepared, but without any of those answers being retrieved or presented to the user. In all cases, an impact sorted index with $b = 10$ is used, together with the fast query evaluation described by Anh et al. [2001], but without any use of pruning or early termination heuristics. The same hardware – a 933MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux – was used in these experiments.

The integer compression mechanisms described in this paper allow query throughput rates that are 50% higher than the bit-aligned Golomb code. The new mechanism is also faster than the byte-aligned code, and, to our sur-

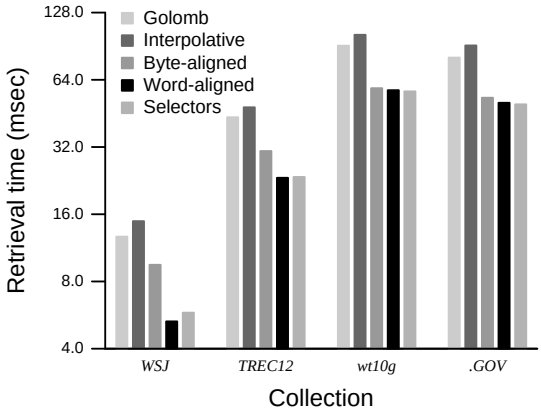

Figure 3: Impact of different coding schemes on query processing speed. The figure shows the average of the elapsed time (in milliseconds) between when a query enters the system and when a ranked list of the top $1{,}000$ answers is finalized, but not retrieved. The average is taken over $10{,}000$ artificial queries with mean length of three. The code denoted "Word-aligned" is the Carryover-12 mechanism described by Anh and Moffat [2004]. The code denoted "Selectors" is the "1-2-4 $\times$ m, escape" code described in this paper. The hardware used was a 933MHz Intel Pentium III with 1 GB RAM running Debian GNU/Linux.

Method	Pointers per selector			
	WSJ	*TREC12*	*wt10g*	*.GOV*
Carryover-12	4.07	3.83	3.29	3.18
Selectors	6.40	5.86	4.73	6.34

Table 6: Data components per selector, averaged across all of the blocks in the index for each collection. In the row labelled "Selectors", the "1-2-4 $\times$ m, escape" mechanism is used, and each selector spans approximately 50% more pointers than in the Carryover-12 technique.

prise, fractionally faster than the word-aligned Carryover-12 mechanism.

Table 6 shows the basis for the unexpected decoding speed. To construct the table, the average span of each selector was computed, for the Carryover-12 mechanism, and for the "1-2-4 $\times$ m, escape" mechanism. In the Carryover-12 arrangement, the hard limit placed by word boundaries keeps the average span small, and each (two-bit) selector corresponds to between three and four codewords. On the other hand, in the mechanism described in this paper, the spans are free to become larger if appropriate, and on average between five and six pointers are coded per selector. The new approach creates extra fiddling at word transitions, but on the whole, that cost is compensated for by having to deal with fewer selectors, which are a competing form of transition.

7 Summary

We have extended our previous results regarding binary-coded inverted files, and have described a new mechanism for use in information retrieval systems. The approach described here results in a reduction in index storage costs compared to our previous word-aligned version, with no cost in terms of query throughput.

Two variations on the basic method have also been described, as well as an approach that allows calculation of optimal parsing of the sequence of d-gaps. Additional experimentation is planned to determine the extent to which optimal parsing results in further compression gains.

Acknowledgement This work was supported by the Australian Research Council.

References

V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proc. 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42, New Orleans, LA, Sept. 2001. ACM Press, New York.

V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 2004. To appear.

P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Information Processing & Management*, 2003. To appear. Preprint available at `http://www.ted.cmis.csiro.au/TRECWeb/`.

N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proc. Symp. String Processing and Information Retrieval*, Manaus, Brazil, Oct. 2003. To appear.

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

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

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

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

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

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, and K. Järvelin, editors, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, Aug. 2002. ACM Press, New York.

I. Soboroff. Does wt10g look like the web? In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, and K. Järvelin, editors, *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 423–424, Tampere, Finland, Aug. 2002. ACM Press, New York.

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

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.