Compact-binary: An efficient non-parameterized code for index compression

Ilias Nitsos[1], Georgios Evangelidis[1], and Dimitris Dervos[2]

[1] Department of Applied Informatics, University of Macedonia
156 Egnatia Str., 54006 Thessaloniki, Greece
{nitsos, gevan}@uom.gr
[2] Department of Information Technology, TEI
P.O. Box 14561, 54101 Thessaloniki, Greece
dad@it.teithe.gr

Abstract. Inverted file indexes are nowadays the most popular method for indexing text databases. Integer number compression codes are applied on the inverted document id lists to produce compact inverted file indexes. A class of index compression codes that are insensitive to the variations in the statistics of dynamic text collections are the non-parameterized codes. In the present study, we introduce *compact-binary (cb)*: a new non-parameterized coding scheme that combines the Golomb code and the binary representation of integers. The performance of the new code is compared to that of existing popular codes. Experimental results obtained from a number of TREC document collections reveal an overall 7,7% improvement over the most efficient of the existing non-parameterized codes. The outcome is backed by analysis and comprises a significant gain when one considers the large sizes of the target text database collections.

1 Introduction

The dramatic growth in the sizes of electronic text databases and the increasing use of the internet as a means for storing and retrieving textual data have given rise to the problem of manipulating huge sources of text. The main objective is to guarantee fast response to user posed queries.

Several indexing schemes have been developed for speeding up the search operations in text databases. The most popular one is the inverted file [3], [9], [17], [15], where much progress has been made due to great advances in the field of integer compression codes [6], [2], [5], [16], [8]. Several such codes exist, that allow the average pointer (integer) inside the inverted file to utilize less than 1 byte of storage space [15], thus, producing very compact indexes. Compressed indexes are also fast, because fewer disk I/O operations are required at retrieval time and the decompression cost is not very important [13].

In the present paper, we examine some of the compression codes and introduce a new one that is suitable for indexing dynamic document collections and achieves better performance when compared to the popular existing codes. Three variations of the new code are discussed. Each newly introduced variation improves the performance of the previous one.

In Section 2 we examine several codes that have been developed for compressing integers inside the inverted files and the models they are based on. The γ[2] and the Golomb code [5] are analyzed thoroughly. The proposed code and its three variations are described in Section 3. In Section 4, we consider the analysis of the third and best performing variation of the proposed code. The profile of the collections used for the experiments and the results obtained are described in Section 5. We conclude in Section 6.

2 Codes for Compressing Inverted Files

A very efficient index structure used in full-text retrieval systems is the inverted file index [3], [9], [17], [15]. Let us suppose, that we have a text database consisting of N documents, each represented by a positive integer $d \in [1, \ldots, N]$. For each distinct word t, an inverted list is created, that stores all the document numbers d containing t. All those inverted lists are stored in a single file, known as the inverted file [3].

The compression codes presented in this section are some of the most popular ones for compressing the integers stored in an inverted file and they utilize run length coding [5]. Instead of storing the absolute numbers d of the documents containing a word, they store their differences. For example, for word t, instead of storing the d list $\{2, 9, 10, 15, 16, 20\}$, the corresponding d-gap list $\{2, 7, 1, 5, 1, 4\}$ is constructed and stored. In general, this results in storing smaller and more frequent numbers. The initial list can be easily recomputed by adding the d-gaps.

The codes discussed in the subsections that follow, are based on models that consider the probability distribution of d-gap sizes and assign small bit codes to frequent d-gap sizes and larger bit codes to rare ones. In general, codes are divided into parameterized and non-parameterized, depending on whether statistical information, concerning the document collection being indexed, is used or not.

2.1 Non-parameterized codes

Unary code. In the unary coding scheme any positive integer x is represented by x-1 one bits followed by a zero. For example, number 5 in unary will be stored as 11110. This means that the bit length of integer x in unary is $len_u(x) = x$. A list of the unary codes for the first ten positive integers is shown in Table 1.

The unary code is equivalent to assigning a probability of 2^{-x} to gaps of length x [15].

Elias codes. A popular non-parameterized code for compressing integers, that results in significant savings, is the γ code described by Elias in [2]. This code suggests that any positive integer x should be stored according to the following encoding scheme:

- store number $1 + \lfloor log_2 x \rfloor$ in unary
- store the remainder $x - 2^{\lfloor log_2 x \rfloor}$ in binary using $\lfloor log_2 x \rfloor$ bits.

Table 1. Examples of codes for integers

				Golomb			
x	unary	γ code	δ code	b=2	b=3	b=6	b=7
1	0	0	0	00	00	000	0000
2	10	100	1000	01	010	001	0001
3	110	101	1001	100	011	0100	0010
4	1110	11000	10100	101	100	0101	0011
5	11110	11001	10101	1100	1010	0110	0100
6	111110	11010	10110	1101	1011	0111	0101
7	1111110	11011	10111	11100	1100	1000	0110
8	11111110	1110000	11000000	11101	11010	1001	10000
9	111111110	1110001	11000001	111100	11011	10100	10001
10	11111111110	1110010	11000010	111101	11100	10101	10010

It follows that the bit length of integer x in γ code is $len_{\gamma}(x) = 2\lfloor log_2 x\rfloor + 1$. The γ code is equivalent to assigning a probability of $1/2x^2$ to gap x [15].

Given the bitstream 111000111011 and knowing that it is encoded using the γ code, decoding proceeds as follows: Count all the one bits until the first zero is encountered; in this case 3 one bits. Ignore this first zero bit. Proceed by reading in a binary number consisting of as many of the following binary bits as the number of the ones originally read. In this example, the binary number is number 1, corresponding to the bitstream 001 (the 3-bit sequence after the first zero). The number retrieved is $2^3 + 1 = 9$. In general, if k one bits are read until the first zero bit is encountered, then k bits must be read in binary after the first zero bit, to produce the remainder r. The number retrieved will then be $x = 2^k + r$. Returning to the initial example, there are bits in the bitstream that have not been decoded. Working in the same way, number 7 is produced and there are no more bits left to continue. To visualize the decoding stages of the algorithm, the initial bitstream may be represented as 1110001,11011.

Another popular integer encoding scheme also introduced by Elias in [2] is the δ-code. In the case of the δ-code each positive integer number x is stored according to the following scheme:

– store number $1 + \lfloor log_2 x\rfloor$ using γ code.
– store the remainder $x - 2^{\lfloor log_2 x\rfloor}$ in binary using $\lfloor log_2 x\rfloor$ bits.

The bit length of integer x in δ is $l_{\delta}(x) = \lfloor log_2 x\rfloor + 2\lfloor log_2(1 + \lfloor log_2 x\rfloor)\rfloor + 1$ and a probability of $\frac{1}{2x(log_2 x)^2}$ is assigned to it [15].

A list of the γ and δ codes for the first ten positive integers is shown in Table 1.

Golomb code. According to this code, for a given parameter b, any positive integer x can be stored according to the following scheme:

– store number $q + 1$ in unary, where $q = \lfloor (x - 1)/b\rfloor$
– store the remainder $r = x - q \times b - 1$ in binary using either $\lfloor log_2 b\rfloor$ bits or $\lceil log_2 b\rceil$ bits for some values of b and $\lceil log_2 b\rceil$ bits for some other values of b.

It follows that the bit length of integer x in Golomb is at most $len_g(x) = \lfloor (x-1)/b \rfloor + 1 + \lceil log_2 b \rceil$.

A list of the Golomb codes for the first ten positive integers, for some values of the parameter b, is shown in Table 1. Different b values result in different groups of Golomb codes. For $b = 1$, the resulting Golomb code is identical to the unary code.

At the decompression process a number is calculated as $x = r + q \times b + 1$.

Let us see how the remainder r is stored. It is obvious that r receives values smaller than b: $r \in [0 \ldots b-1]$. In other words, there are b distinct values for r. Normally, one needs $\lceil log_2 b \rceil$ bits to encode b distinct values. But, we can do much better if we observe that we can encode b distinct values using $few = \lfloor log_2 b \rfloor$ bits for some of the values and $many = \lceil log_2 b \rceil$ for the rest, when $2^{few-1} + 2^{many-1} \geq b$. Let us suppose that we use few bits to store the first 2^{few-1} distinct b values. The first bit is set to 0, to indicate that few bits are used to encode the value. The rest $few - 1$ can be used to encode the actual number, and thus, 2^{few-1} distinct numbers can be encoded using few bits. We use $many$ bits to store the rest 2^{many-1} distinct b values. The first bit is set to 1, to indicate that $many$ bits are used to encode the value. The rest $many - 1$ bits are used to encode the actual number, and thus, 2^{many-1} distinct numbers can be encoded using $many$ bits. In total $max = 2^{few-1} + 2^{many-1}$ distinct values can be encoded, using few bits for the first 2^{few-1} of them and $many$ bits for the rest 2^{many-1} of them. If $b \leq max$, then the remainder can be stored using this strategy. If $b > max$ then we can use $many$ bits to encode the remainder. That is, 2^{many} distinct numbers can be encoded, since the first bit for the remainder is no longer used to indicate its size.

In the decoding phase, if either few or $many$ bits are used, then if the first bit of the binary part is zero (i.e., few bits are being used), the value for r equals to the value of binary number corresponding to the remaining $few - 1$ bits. If the first bit of the binary part is one, and the binary number corresponding to the remaining $many - 1$ bits is $rpart$, then $r = 2^{few-1} + rpart$. When just $many$ bits are used for the coding of the binary part, r equals to the value of the binary part fetched.

Given the bitstream 101001001001 and knowing that it is encoded using the Golomb code with $b = 6$, the decoding process is the following: Count all the one bits until the first zero is encountered; in this case $q = 1$ one bits. Because $max = 6 \leq b$, the binary part uses either $few = 2$ or $many = 3$ bits. The first bit of the binary part is 1. This means that $many$ bits have been used, thus, 2 more bits have to be fetched (in total $many = 3$). Those bits are 00, and so we have $rpart = 0$ and $r = 2^{few-1} + rpart = 2^1 + 0 = 2$. Finally, the number is calculated as $x = r + q \times b + 1$. In this case $x = 9$. More bits are left in the bitstream. The rest of the decoding process results in the list of integers {9,8,2}. To visualize the decoding algorithm, the initial bitstream can be represented as 10100,1001,001.

2.2 Parameterized codes

Golomb code for the global Bernoulli model. Let us suppose, that we have a text database consisting of N documents that contains n distinct words and f index pointers (i.e., f distinct "$document, word$" pairs).

The global Bernoulli model assumes that large text databases are homogeneous and the words are scattered uniformly across the N documents. This means that the probability that any randomly selected word appears in any randomly selected document is $p = f/(N \times n)$. This assumption leads to the conclusion that the probability of occurrence of a gap x for any word is $(1-p)^{x-1}p$. This is in fact the probability of having $x-1$ nonoccurrences of a particular word, each of probability $1-p$, followed by one occurrence of probability p.

The Golomb coding method can satisfy the probabilities generated by the global Bernoulli model when b is chosen to be $b = \lceil \frac{log_2(2-p)}{-log_2(1-p)} \rceil$ [4].

Golomb code for the local Bernoulli model. The difference between the global and the local Bernoulli model [1], [14], is that in the latter, each d-list is associated with a different b value. The value of the parameter for the d-gap list of word t is calculated as $b = \lceil \frac{log_2(2-p)}{-log_2(1-p)} \rceil$, with $p = f_t/N$, where f_t is the number of documents in the text database containing t. This means, that for each distinct word t in the text database, the value for f_t must also be stored together with the inverted list of the word, so that the b value is calculated at decoding time. The value for f_t can be stored at the head of each list, using the γ code [15]. When decompressing an inverted list, the f_t value is first decoded, then the b parameter is calculated, and d-decompression continues with the rest of the d-list. It has to be mentioned at this point that the Golomb code for the local Bernoulli model is much more efficient when compared to the global Bernoulli model. The latter produces indexes that are twice as large in size [15].

Other coding methods. Many other models and methods have been developed for coding gaps, like the skewed Bernoulli model [12], [8], the local hyperbolic model [11], the interpolative method [7], and the $u\gamma$-Golomb [10] but we will not get into details for those methods here.

In general, the local Bernoulli model and the Golomb code for the representation of d-gaps is the preferred choice in the case of static document collections because it combines acceptable compression with fast decoding. Unfortunately, in the case of dynamic document collections, parameterized codes are not suitable because they make use of statistical information concerning the collection being indexed. A non-parameterized compression code should be used instead.

3 Compact-binary

In this Section, we present three variations for a new non-parameterized method for storing integers. We call the new method compact-binary (cb) and the three variations will be referred to as cb_1, cb_2 and cb_3. The main idea behind cb is to store the exact binary representation of an integer and to select a compact method for storing the length of the representation, so that decoding is possible. Analysis and experimental measurements conducted on the fifth volume of the TREC collection, confirm that suitable codes for storing the above length, are the Golomb codes for $b = 2$ or 3. More specifically, as it will be explained in Section 4 the choice of $b = 1$ or $b > 3$ causes cb to produce less efficient codes.

Table 2. Compact-binary codes for integers

	cb_1		cb_2		cb_3	
x	b=2	b=3	b=2	b=3	b=2	b=3
1	0000	0000	00001	00001	00001	00001
2	0001	0001	0001	0001	001	001
3	001	001	001	001	0001	0001
4	0100	01000	0100	01000	0100	01000
5	0101	01001	0101	01001	0101	01001
6	0110	01010	0110	01010	0110	01010
7	0111	01011	0111	01011	0111	01011
8	100000	011000	100000	011000	100000	011000
9	100001	011001	100001	011001	100001	011001
10	100010	011010	100010	011010	100010	011010

3.1 First variation: cb_1

According to the cb_1 scheme, any positive integer $x \geq 3$ consists of two parts. The second part (p_2) is the exact binary representation of integer x, except for the most significant bit, that is known beforehand to be 1. The first part (p_1) is the length of p_2 encoded using a suitable Golomb code ($b = 2$ or 3). This is what we call the basic *cb* rule. We present the cb_1 codes ($b = 2$ and 3) for the first ten positive integers in Table 2.

The selection of 0000 and 0001 for the encoding of integers 1 and 2 requires special consideration. The basic *cb* rule for storing an integer is not applicable in the case of integer number 1. The binary representation of integer number 1 is 1 and if the most significant bit is to be omitted, then length 0 has to be stored as the length of the binary representation. Unfortunately no Golomb codes exist for number 0. On the other hand, using the basic *cb* rule, number 2 would be stored as 000. The problem of storing number 1 can be solved if 2 is to be stored as 0001, because then we can use the bit sequence 0000 for encoding number 1. During decoding, four zeros stand for integer number 1 and three zeros followed by a 1, stand for integer number 2. When, during decoding, the bit sequence does not start with 3 or 4 zeros, we use the basic *cb* rule.

Let us now explain the coding and decoding process using the *d*-gap list 16, 2, 9, 8, 1, 2, 5 as an example with $b = 3$. The corresponding exact binary representations of the above integers are, respectively: 10000, 10, 1001, 1000, 1, 10, 101. As expected, the first bit in all the representations is 1. Number 16 is to be stored as 100,0000. We make use of the comma only as a visual aid, in order to distinguish the first part from the second one. The first part is number 4 using the Golomb coding method for b = 3 (see Table 1 and it stands for the length of the second. The second part is the binary representation of integer number 16, after omitting the most significant bit. On the other hand number 2 is encoded as 0001, because of the exception that was discussed earlier. According to the basic *cb* rule, number 9 is stored as 011,001 and so on. The entire list, after the encoding process is over, will be stored as 100,0000(16)-0001(2)-011,001(9)-011,000(8)-0000(1)-0001(2)-010,01(5).

The process for decoding the above list is described next. The first number has been encoded using the basic *cb* rule, because the bit sequence does not start with 3 or 4 zeros. As a result, two parts must be calculated. The first part is a number stored using Golomb ($b = 3$). In this case, we follow the decoding rules for the Golomb code and it turns out that the encoded number takes up 3 bits and corresponds to number 4. The second part consists of the four bits that follow (0000). The actual number stored can be reconstructed by adding bit 1 as the most significant bit of the second part. In this case, the actual number stored is number 10000 (16). In the case of the second number, three zeros are encountered followed by a bit that is set to one. This means that the actual number stored is number 2, because of the exception discussed earlier. The same procedure is applied on the entire bit sequence and the original initial d-gap list is reconstructed in the end.

3.2 Second variation: cb_2

A drawback for cb_1 is that it uses 4 bits to store integer number 1. When common and function words such as “a”, “the”, “of” are indexed in a document collection (and in some languages, e.g., German and Hebrew, function words need to be included into the index), a frequently occurring d-gap in the d-gap lists is integer number 1, since these words tend to occur in almost every document. Other codes, such as the γ and the δ codes use a single bit to store number 1 (see Table 1).

In the case of cb_2, the encoding of 1 is treated differently. When a d-gap equal to 1 is encountered, the bit sequence 0000 is used to encode it. If the next, consecutive, d-gap is also 1, then a single extra 0 bit is used to indicate that the sequence of 1 d-gaps is not over yet, and so on. A final bit equal to 1 is appended to the bit sequence, in order to indicate that the d-gap sequence of 1s is over. The above modification suggests that a single d-gap equal to 1 is stored as 00001, two subsequent d-gaps equal to 1 are stored as 000001, three consecutive ones are stored as 0000001 and so on. It is expected that the performance of the code will be improved, since many d-gap lists tend to contain groups of consecutive 1s corresponding to common and function words present in the text database. The experimental results, presented in Section 5 reveal the improvement in cb_2 next to cb_1.

3.3 Third variation: cb_3

The cb_3 scheme is an attempt to further improve the cb_2 scheme, by swapping the codes for numbers 2 and 3. The first ten positive integers for the cb_3 scheme are presented in Table 2. In general, smaller d-gaps tend to appear more often in a document collection and they should receive smaller codes. The experimental results, presented in Section 5 reveal the improvement in cb_3 next to cb_2.

4 Analysis

In all compact-binary variations, the length of the code for any integer number $x > 3$ can be calculated by adding the lengths of p_1 and p_2. This means that:

$$len_{cb}(x) = len_{p_1} + len_{p_2} \tag{1}$$

According to the definition of *cb*, the second part of the code is the binary representation of number x without its most significant bit. This means that $len_{p_2} = \lfloor log_2(x) \rfloor$. The first part of the code is the length of the second part encoded in Golomb. This means that the length of the first part adds up to at most $len_g(\lfloor log_2(x) \rfloor) = \lfloor(\lfloor log_2(x)\rfloor - 1)/b\rfloor + 1 + \lceil log_2(b)\rceil$ bits (see Section 2). After the substitutions Equation (1) becomes:

$$len_{cb}(x) = \lfloor(\lfloor log_2(x)\rfloor - 1)/b\rfloor + 1 + \lceil log_2(b)\rceil + \lfloor log_2(x)\rfloor \tag{2}$$

Utilizing Equation (2) we calculated and compared the lengths between the different groups of *cb* codes produced for different values of b. When the b parameter took values greater than 3, relatively small integers received longer codes, whereas large integers received shorter codes. We note that most of the *d*-gaps stored in an inverted file are relatively small integer numbers, because they correspond to lists of frequently occurring words. On the other hand, large *d*-gaps appear in the lists of rare words and they do not appear often inside the inverted file. An efficient code should take into consideration these facts. We experimented with many text databases and applied many different groups of *cb* codes. For each group, the average number of bits required to store a pointer was calculated. The results we obtained were the ones we expected. As the value of the b parameter increased above 3 or became equal to 1, the efficiency of the code degraded. The most efficient groups of codes were the ones produced for b = 2 or 3 and they are the ones included in our presentation.

Below, we compare the bit lengths for integers coded using the δ code and cb_3 for b = 2 (referred to as cb_{3-2}) or 3 (referred to as cb_{3-3}). We do not compare our codes over the γ code, because it has been shown that the latter is outperformed by the δ code [8]. Moreover, we focus mainly on the the third variation of *cb* because it is designed to improve the performance of cb_1 and cb_2. The experimental results presented in Section 5 confirm that cb_3 performs better than cb_1 and cb_2.

For any integer $x > 3$ the bit length difference between the δ code and cb_{3-2} is exactly:

$$len_\delta(x) - len_{cb}(x) = 2\lfloor log_2(1 + \lfloor log_2 x\rfloor)\rfloor - \lfloor(\lfloor log_2(x)\rfloor - 1)/2\rfloor - 1 \tag{3}$$

This is an exact case equivalent for cb_{3-2} because $b = 2$ and $\lceil log_2 2\rceil = many = 1$ bits are used for r (see Section 2).

In the case of the δ code and cb_{3-3}, the bit length difference is at least:

$$len_\delta(x) - len_{cb}(x) = 2\lfloor log_2(1 + \lfloor log_2 x\rfloor)\rfloor - \lfloor(\lfloor log_2(x)\rfloor - 1)/3\rfloor - 2 \tag{4}$$

This is a worst case equivalent for cb_{3-3} because $b = 3$ and we use either $\lfloor log_2 3\rfloor = few = 1$ bits, or $\lceil log_2 3\rceil = many = 2$ bits to encode the number.

We can substitute $1 + \lfloor log_2 x\rfloor$, that is equal to the number of bits used for the binary representation of number x, with y. The above equations are respectively transformed into

$$len_{\delta}(x) - len_{cb}(x) = 2\lfloor log_2(y) \rfloor - \lfloor (y-2)/2 \rfloor - 1 \quad (5)$$

$$len_{\delta}(x) - len_{cb}(x) = 2\lfloor log_2(y) \rfloor - \lfloor (y-2)/3 \rfloor - 2 \quad (6)$$

Apparently, the latter equations are functions of y, i.e., the number of bits that are used for the binary representation of the integers. Table 3 lists bit length differences for Equations 5 and 6.

Table 3. Bit length differences

Number	Bit length difference		Number	Bit length difference	
	δ-cb_{3-2}	δ-cb_{3-3}		δ-cb_{3-2}	δ-cb_{3-3}
1	-4, 0 consequent	-4, 0 consequent	17-bit numbers	0	1
2	1	1	18-bit numbers	-1	1
3	0	0	19-bit numbers	-1	1
3-bit numbers	1	0	20-bit numbers	-2	0
4-bit numbers	2	2	21-bit numbers	-2	0
5-bit numbers	2	1	22-bit numbers	-3	0
6-bit numbers	1	1	23-bit numbers	-3	-1
7-bit numbers	1	1	24-bit numbers	-4	-1
8-bit numbers	2	2	25-bit numbers	-4	-1
9-bit numbers	2	2	26-bit numbers	-5	-2
10-bit numbers	1	2	27-bit numbers	-5	-2
11-bit numbers	1	1	28-bit numbers	-6	-2
12-bit numbers	0	1	29-bit numbers	-6	-3
13-bit numbers	0	1	30-bit numbers	-7	-3
14-bit numbers	-1	0	31-bit numbers	-7	-3
15-bit numbers	-1	0	32-bit numbers	-6	-2
16-bit numbers	0	2			

The results demonstrate (see Table 3) that the cb_{3-2} compression efficiency is equal to or better than that of the δ code for all integers in $[2 \ldots 2^{13}-1]$ or $[2 \ldots 8191]$. When the value of the b parameter is set to 3, then the particular variation of cb has equal or better compression efficiency than the δ code for all integers in $[2 \ldots 2^{22}-1]$ or $[2 \ldots 4194303]$. Especially in the case of integer number 1, cb_3 utilizes four bits of storage more than the δ code. To overcome this problem, the cb_3 algorithm utilizes only one extra bit when storing d-gaps of consecutive 1 values (see Section 3).

It is important for a compression method to assign small compression codes to small d-gaps, because the latter appear many times inside the inverted lists that correspond to frequent terms. Therefore, the decreased efficiency of cb_{3-2} and cb_{3-3} for large numbers has no negative implications in practice. As a final comment, concerning the slightly increased efficiency of cb_{3-3} over cb_{3-2} (see section 5): In general cb_{3-3} is found to generate smaller codes for very large numbers ($2^{10} = 1024$ and larger ones), but one must keep in mind, that this is the worst case equivalent for cb_{3-3}. For example the actual bit-difference between the δ code (110010011) and the cb_{3-3} code (1000011) for number 19 is two, whereas the worst case equivalent, estimates a difference of just one bit (see Table 3).

5 Experimental Results

The document collection used for the experiments presented in this paper, was the fifth volume of the TREC collection, that contains 475MB of articles from the Los Angeles Times (LAT) in the two year period from January 1, 1989 – December 31, 1990 and 470MB of documents from the Foreign Broadcast Information Service (FBIS) for the year 1994. The profile for the three collections LAT, FBIS and the concatenation of both, noted as LATFBIS is shown in Table 4.

Table 4. Testbed collections' profiles

Collection	Size	Distinct words (n)	Total documents (N)	Total pointers (f)
LAT	475MB	288823	130472	31193292
FBIS	470MB	247563	131167	34982316
LATFBIS	945MB	437864	261639	66175608

The following compression codes were considered:

– Elias γ code for all the gaps in all the lists of the inverted file. This code is referred to as "γ code".
– Elias δ code for all the gaps in all the lists of the inverted file. This code is referred to as "δ code".
– Golomb code for local Bernoulli model. An inverted list is created for every word in the collection. The list also stores the value for f_t using the γ code. The f_t overhead for each word is included in the final results. This code is referred to as "Golomb".
– The first, the second and the third variation of *cb* for all the gaps in all the lists of the inverted file, with the b parameter set to 2. These codes will be referred to as "cb_{1-2}", "cb_{2-2}" and "cb_{3-2}" respectively.
– The first, the second and the third variation of *cb* for all the gaps in all the lists of the inverted file, with the b parameter set to 3. These codes will be referred to as "cb_{1-3}", "cb_{2-3}" and "cb_{3-3}" respectively.

The results obtained are expressed in bits per pointer and they are listed in Table 5.

A first observation is that the compression code with the best performance is the "Golomb" code. This was expected, because the "Golomb" code is a local, parameterized code. This means that it makes use of statistical information concerning the document collection being indexed. This is the reason that this method is not suitable for dynamic environments. On the other hand, the rest of the codes appearing in Table 5, are non-parameterized and are easily applied on dynamic collections. Another,

Table 5. Experimental results expressed in bits per pointer

Compression method	Bits per pointer		
	LAT	FBIS	LATFBIS
"γ code"	7.92	6.97	7.49
"δ code"	7.36	6.58	7.02
"*Golomb*"	6.14	6.07	6.32
"cb_{1-2}"	7.19	6.62	6.94
"cb_{1-3}"	7.15	6.62	6.92
"cb_{2-2}"	6.83	6.16	6.54
"cb_{2-3}"	6.79	6.17	6.51
"cb_{3-2}"	6.81	6.14	6.51
"cb_{3-3}"	6.77	6.14	6.48

expected, result is that the "γ code" is outperformed by the "δ code". Similar results are presented in [8].

Commenting on the results obtained, all *cb* variations are seen to outperform the rest of the non-parameterized codes, with the exception of "cb_{1-2}" and "cb_{1-3}" for the FBIS collection. When applied on the latter collection, the *cb* code variations performed slightly worse than the "δ code". This means that the first *cb* variation can produce larger indexes compared to existing codes, probably because of its poor performance in the case of encoding *d*-gaps equal to 1. On the other hand, the rest of the *cb* variations, always produce significantly smaller indexes compared to existing non-parameterized codes and cb_1. The *cb*-algorithm enhancement for treating groups of consecutive 1s in the *d*-gap lists appears to have paid off.

Another observation is that *cb* produces better results when the value of b is equal to 3. This finding confirms the validity of the analysis made in Section 4. We have experimented with many values for the b parameter. The scheme was found to perform poorly when the b parameter obtained values above 4 and they are not included in our presentation. The most efficient codes were the ones produced for $b = 2$ or 3.

One final result is that the third variation performs slightly better than the second one. This was expected, because the third variation enhances the second by swapping the codes for numbers 2 and 3. The third variation assigns the smaller code to number 2 and the larger one to number 3. In general, the experimental results obtained, reveal that the best *cb* variation is "cb_{3-3}" (about 7.7% better than the *delta* code). It is a non-parameterized code with efficiency that comes close to the one of the "Golomb" code for the local Bernoulli model (about 2.5% worse).

Having experimented with many other text database collections, we have obtained results similar to the ones presented.

6 Conclusion

A new non-parameterized code (cb: compact-binary) is introduced and it is shown to improve the performance of inverted index file compression. The efficiency of the new code is considered by utilizing the 945 MB TREC-LATFBIS documents collection. The experimental results obtained reveal that the cb code comprises a 7,7% improvement over the δ code. Being of the non-parameterized type, the new code is applicable to dynamic document collections. Its performance is measured to fall behind that of the local Bernoulli model by only 2,5%: a most encouraging outcome, considering that the local Bernoulli model leads to a parameterized code that is applicable only to static document collections. Three variations of the proposed compression code are presented and the experimental results obtained are confirmed/backed by the corresponding analytical calculations.

References

1. Bookstein, A., Klein, S.T., Raita, T.: Model based concordance compression. In Storer and Cohn. (1992) 82–91
2. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory, **IT-21**. (1975) 194–203
3. Fox, E., Harman, D., Baeza-Yates, R., Lee, W.: Inverted Files. In: Frakes, W., Baeza-Yates, R. (eds): Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, NJ, Chapter 3. (1992) 28–43
4. Gallager, R.G., van Voorhis, D.C.: Optimal source codes for geometrically distributed alphabets. IEEE Transactions on Information Theory, **IT-21**. (1975) 228–230
5. Golomb, S.W.: Run-length Encodings. IEEE Transactions on Information Theory, **IT-21**. (1966) 399–401
6. Huffman, D.A: A method for the construction of minimum redundancy codes. Procedures IRE, **40**(9). (1952) 1098–1101
7. Moffat, A., Stuiver, L.: Exploiting clustering in inverted file compression. In Storer and Cohn. (1996) 82–91
8. Moffat, A., Zobel, J.: Paremeterised Compression for Sparse Bitmaps. 15th Ann Int'l SIGIR, Denmark (1992) 274–285
9. Moffat, A., Zobel, J.: Self-Indexing Inverted Files for Fast Text Retrieval. ACM Transactions on Database Systems, **14**. (1996) 349–379
10. Nitsos, I., Evangelidis, G., Dervos, D. : $u\gamma$-Golomb: A new Golomb Code Variation for the Local Bernoulli Model. To appear in Proceedings of the 7th East-European Conference on Advances in Databases and Informations Systems (ADBIS 2003), Dresden, Germany, September 3–6, 2003
11. Schuegraf, E.J.: Compression of large inverted files with hyperbolic term distribution. Information Processing and Managemant **12**. (1976) 377–384
12. Teuhola, J.: A compression method for clustered bit-vectors. Information Processing Letters, **7**(2). (1978) 308–311
13. Williams, H. E., Zobel, J.: Compressing Integers for Fast File Access. The Computer Journal, **42**. (1999) 193–201
14. Witten, I.H., Bell T.C., Nevill C. G.: Indexing and compressing full-text databases for CD-ROM. Journal of Information Science, **17**. (1992) 265–271
15. Witten, I.H., Moffat A., Bell T.C.: Managing Gigabytes. Compressing and Indexing Documents and Images. Academic Press (1999)
16. Witten, I.H., Neal, R., Cleary, J.G.: Arithmetic coding for data compression. Communications of the ACM, **30**(6). (1987) 520–541
17. Zobel, J., Moffat, A., Ramamohanarao K.: Inverted Files Versus Signature Files for Text Indexing. ACM Transactions on Database Systems, **23**. (1999) 369–410