Compact-binary: an Efficient Non-parameterized Code for Index Compression

Ilias Nitoso1, Georgios Evangelidis1, and Dimitris Dervos2

1 Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece
Email: {nitoso, geven}@um.edu.gr
2 Department of Information Technology, TEI of Thessaloniki
54101 Thessaloniki, Greece
Email: 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, 15, 17], where much progress has been made due to great advances in the field of integer compression codes [2, 5, 6, 8, 16]. 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, 15, 17]. 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 \( \text{len}_u(x) = x \). A list of the unary codes for the first ten positive integers is shown in Table 2.1.

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 \( \gamma \) 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.
It follows that the bit length of integer \( x \) in \( \gamma \) code is \( \text{len}_\gamma(x) = 2 \cdot \lceil \log_2 x \rceil + 1 \). The \( \gamma \) 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 \( \gamma \) 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 \( \delta \)-code. In the case of the \( \delta \)-code each positive integer number \( x \) is stored according to the following scheme:

- store number \( 1 + \lceil \log_2 x \rceil \) using \( \gamma \) code.
- store the remainder \( x - 2^{\lceil \log_2 x \rceil} \) in binary using \( \lceil \log_2 x \rceil \) bits.

The bit length of integer \( x \) in \( \delta \) is \( \text{len}_\delta(x) = \lceil \log_2 x \rceil + 2 \lceil \log_2 (1 + \lceil \log_2 x \rceil) \rceil + 1 \) and a probability of \( \frac{1}{2x^2} \) is assigned to it [15].

A list of the \( \gamma \) and \( \delta \) codes for the first ten positive integers is shown in Table 2.1.

**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 \( \lceil \log_2 b \rceil \) 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 $\text{len}_b(x) = \left\lceil (x - 1)/b \right\rceil + 1 + \lfloor \log_b x \rfloor$.

A list of the Golomb codes for the first ten positive integers, for some values of the parameter $b$, is shown in Table 2.1. 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 $\lceil \log_b x \rceil$ bits to encode $b$ distinct values. But, we can do much better if we observe that we can encode $b$ distinct values using $\text{few} = \left\lfloor \log_2 b \right\rfloor$ bits for some of the values and $\text{many} = \lceil \log_2 b \rceil$ for the rest, when $2^{\text{few}-1} + 2^{\text{many}-1} \geq b$. Let us suppose that we use $\text{few}$ bits to store the first $2^{\text{few}-1}$ distinct $b$ values. The first bit is set to 0, to indicate that $\text{few}$ bits are used to encode the value. The rest $\text{few} - 1$ can be used to encode the actual number, and thus, $2^{\text{few}-1}$ distinct numbers can be encoded using $\text{few}$ bits. We use $\text{many}$ bits to store the rest $2^{\text{many}-1}$ distinct $b$ values. The first bit is set to 1, to indicate that $\text{many}$ bits are used to encode the value. The rest $\text{many} - 1$ bits are used to encode the actual number, and thus, $2^{\text{many}-1}$ distinct numbers can be encoded using $\text{many}$ bits. In total $\text{max} = 2^{\text{few}-1} + 2^{\text{many}-1}$ distinct values can be encoded, using $\text{few}$ bits for the first $2^{\text{few}-1}$ of them and $\text{many}$ bits for the rest $2^{\text{many}-1}$ of them. If $b \leq \text{max}$, then the remainder can be stored using this strategy. If $b > \text{max}$ then we can use $\text{many}$ bits to encode the remainder. That is, $2^{\text{many}}$ 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 $\text{few}$ or $\text{many}$ bits are used, then if the first bit of the binary part is zero (i.e., $\text{few}$ bits are being used), the value for $r$ equals to the value of binary number corresponding to the remaining $\text{few} - 1$ bits. If the first bit of the binary part is one, and the binary number corresponding to the remaining $\text{many} - 1$ bits is $r_{part}$, then $r = 2^{\text{few}-1} + r_{part}$. When just $\text{many}$ 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 $\text{max} = 6 < b$, the binary part uses either $\text{few} = 2$ or $\text{many} = 3$ bits. The first bit of the binary part is 1. This means that $\text{many}$ bits have been used, thus, 2 more bits have to be fetched (in total $\text{many} = 3$). Those bits are 00, and so we have $r_{part} = 0$ and $r = 2^{\text{few}-1} + r_{part} = 2^1 + 0 = 2$. 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 \( b = \left\lceil \frac{\log(2-p)}{\log(1-\frac{p}{N})} \right\rceil \) [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 \( b = \left\lceil \frac{\log(2-p)}{\log(1-\frac{p}{N})} \right\rceil \), 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 \( \gamma \) 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 [8, 12], the local hyperbolic model [11], the interpolative method [7], and the \( u, \gamma \)-Golomb [16] 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 \( cb1 \), \( cb2 \) and \( cb3 \). 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.
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 \( (b = 2 \text{ or } 3) \). This is what we call the basic \( cb \) rule. We present the \( cb_1 \) codes \( (b = 2 \text{ and } 3) \) 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 1000000. 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 2.1 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).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( cb_2 )</th>
<th>( cb_2 )</th>
<th>( cb_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=2</td>
<td>b=3</td>
<td>b=2</td>
<td>b=3</td>
</tr>
<tr>
<td>10000</td>
<td>0000</td>
<td>00001</td>
<td>00001</td>
</tr>
<tr>
<td>20001</td>
<td>0001</td>
<td>0001</td>
<td>001</td>
</tr>
<tr>
<td>3001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>4010</td>
<td>0100</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>50101</td>
<td>01001</td>
<td>01001</td>
<td>01001</td>
</tr>
<tr>
<td>601010</td>
<td>01010</td>
<td>01010</td>
<td>01010</td>
</tr>
<tr>
<td>70111</td>
<td>01011</td>
<td>0111</td>
<td>01011</td>
</tr>
<tr>
<td>801000011000</td>
<td>100001011000</td>
<td>100001011000</td>
<td></td>
</tr>
<tr>
<td>901000111001</td>
<td>100001101001</td>
<td>100001101001</td>
<td></td>
</tr>
<tr>
<td>100101011010</td>
<td>100101101010</td>
<td>100101101010</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Compact-binary codes for integers
The process for decoding the above list is described next. The first number has been encoded using the basic $ch$ 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 $\gamma$ and the $\delta$ codes use a single bit to store number 1 (see Table 2.1).

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:
\[ \text{len}_{cb}(x) = \text{len}_{p_1} + \text{len}_{p_2} \] (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 \( \text{len}_{p_2} = \lfloor \log_2(x) \rfloor \). 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 \( \text{len}_{p_2} (\lfloor \log_2(x) \rfloor + 1) + \lfloor \log_2(b) \rfloor \) bits (see Section 2). After the substitutions Equation (1) becomes:

\[ \text{len}_{cb}(x) = \lfloor (\lfloor \log_2(x) \rfloor - 1)/b \rfloor + 1 + \lfloor \log_2(b) \rfloor + \lfloor \log_2(x) \rfloor \] (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 \( k \)-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 \( k \)-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 \( \delta \) code and \( cb_b \) for \( b = 2 \) (referred to as \( cb_{b=2} \)) or 3 (referred to as \( cb_{b=3} \)). We do not compare our codes over the \( \gamma \) code, because it has been shown that the latter is outperformed by the \( \delta \) 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 \( \delta \) code and \( cb_{b=2} \) is exactly:

\[ \text{len}_{b}(x) - \text{len}_{cb}(x) = 2\lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor - \lfloor (\lfloor \log_2(x) \rfloor - 1)/2 \rfloor - 1 \] (3)

This is an exact case equivalent for \( cb_{b=2} \) because \( b = 2 \) and \( \lfloor \log_2(1) \rfloor = \text{many} = 1 \) bits are used for \( r \) (see Section 2).

In the case of the \( \delta \) code and \( cb_{b=3} \), the bit length difference is at least:

\[ \text{len}_{b}(x) - \text{len}_{cb}(x) = 2\lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor - \lfloor (\lfloor \log_2(x) \rfloor - 1)/3 \rfloor - 2 \] (4)

This is a worst case equivalent for \( cb_{b=3} \) because \( b = 3 \) and we use either \( \lfloor \log_2(1) \rfloor = \text{few} = 1 \) bits, or \( \lfloor \log_2(3) \rfloor = \text{many} = 2 \) 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
\[ \text{len}_a(x) - \text{len}_c(x) = 2 \lfloor \log_2(y) \rfloor - \lfloor (y - 2)/2 \rfloor - 1 \] (5)

\[ \text{len}_a(x) - \text{len}_c(x) = 2 \lfloor \log_2(y) \rfloor - \lfloor (y - 2)/3 \rfloor - 2 \] (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>
<thead>
<tr>
<th>Number</th>
<th>( \delta - \text{cb}_3, \delta - \text{cb}_3 )</th>
<th>17-bit numbers</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-4, 0 consequent</td>
<td>17-bit numbers</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>18-bit numbers</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>19-bit numbers</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>5-bit numbers</td>
<td>2</td>
<td>21-bit numbers</td>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>6-bit numbers</td>
<td>2</td>
<td>22-bit numbers</td>
<td>-3</td>
<td>0</td>
</tr>
<tr>
<td>7-bit numbers</td>
<td>2</td>
<td>23-bit numbers</td>
<td>-3</td>
<td>-1</td>
</tr>
<tr>
<td>8-bit numbers</td>
<td>2</td>
<td>24-bit numbers</td>
<td>-4</td>
<td>-1</td>
</tr>
<tr>
<td>9-bit numbers</td>
<td>2</td>
<td>25-bit numbers</td>
<td>-4</td>
<td>-1</td>
</tr>
<tr>
<td>10-bit numbers</td>
<td>2</td>
<td>26-bit numbers</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>11-bit numbers</td>
<td>1</td>
<td>27-bit numbers</td>
<td>-5</td>
<td>-2</td>
</tr>
<tr>
<td>12-bit numbers</td>
<td>1</td>
<td>28-bit numbers</td>
<td>-6</td>
<td>-2</td>
</tr>
<tr>
<td>13-bit numbers</td>
<td>1</td>
<td>29-bit numbers</td>
<td>-6</td>
<td>-3</td>
</tr>
<tr>
<td>14-bit numbers</td>
<td>0</td>
<td>30-bit numbers</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>15-bit numbers</td>
<td>0</td>
<td>31-bit numbers</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>16-bit numbers</td>
<td>0</td>
<td>32-bit numbers</td>
<td>-6</td>
<td>-2</td>
</tr>
</tbody>
</table>

Table 3. Bit length differences

The results demonstrate (see Table 3) that the \( \text{cb}_3 \) compression efficiency is equal to or better than that of the \( \delta \) 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 \( \text{cb} \) has equal or better compression efficiency than the \( \delta \) code for all integers in \([2 \ldots 2^{22} - 1]\) or \([2 \ldots 4194303]\). Especially in the case of integer number 1, \( \text{cb}_3 \) utilizes four bits of storage more than the \( \delta \) code. To overcome this problem, the \( \text{cb}_3 \) 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 \( \text{cb}_3 \) and \( \text{cb}_3 \) for large numbers has no negative implications in practice. As a final comment, concerning the slightly increased efficiency of \( \text{cb}_3 \) over \( \text{cb}_3 \) (see section 5): In general \( \text{cb}_3 \) 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 \( \text{cb}_3 \). For example the
actual bit-difference between the δ code (110010011) and the \( cb_{2-3} \) 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>
<thead>
<tr>
<th>Collection</th>
<th>Size</th>
<th>Distinct words (n)</th>
<th>Total documents (N)</th>
<th>Total pointers (f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAT</td>
<td>475MB</td>
<td>288823</td>
<td>130472</td>
<td>31193292</td>
</tr>
<tr>
<td>FBIS</td>
<td>470MB</td>
<td>247563</td>
<td>131167</td>
<td>34982336</td>
</tr>
<tr>
<td>LATFBIS</td>
<td>945MB</td>
<td>437864</td>
<td>261639</td>
<td>66175608</td>
</tr>
</tbody>
</table>

Table 4. Testbed collections’ profiles

The following compression codes were considered:

- Elias \( \gamma \) code for all the gaps in all the lists of the inverted file. This code is referred to as \( \gamma \) code”.
- Elias \( \delta \) code for all the gaps in all the lists of the inverted file. This code is referred to as \( \delta \) 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 \( f_i \) using the \( \gamma \) code. The \( f_i \) 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 \( “cb_{2-2}” \), \( “cb_{2-2}” \) and \( “cb_{2-2}” \) 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 \( “cb_{3-3}” \), \( “cb_{3-3}” \) 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

<table>
<thead>
<tr>
<th>Compression method</th>
<th>LAT FBIS LATFBIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;γ code&quot;</td>
<td>7.92 6.97 7.49</td>
</tr>
<tr>
<td>&quot;δ code&quot;</td>
<td>7.36 6.68 7.02</td>
</tr>
<tr>
<td>&quot;Golomb&quot;</td>
<td>6.14 6.07 6.32</td>
</tr>
<tr>
<td>&quot;db1-2&quot;</td>
<td>7.79 6.62 6.94</td>
</tr>
<tr>
<td>&quot;db1-3&quot;</td>
<td>7.15 6.62 6.92</td>
</tr>
<tr>
<td>&quot;db2-2&quot;</td>
<td>6.83 6.16 6.54</td>
</tr>
<tr>
<td>&quot;db2-3&quot;</td>
<td>6.79 6.17 6.51</td>
</tr>
<tr>
<td>&quot;db3-2&quot;</td>
<td>6.81 6.14 6.51</td>
</tr>
<tr>
<td>&quot;db3-3&quot;</td>
<td>6.77 6.14 6.48</td>
</tr>
</tbody>
</table>

Commenting on the results obtained, all cb variations are seen to outperform the rest of the non-parameterized codes, with the exception of "cb1-2" and "cb1-3" 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 cb1. 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 "cb2-3" (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/verified by the corresponding analytical calculations.

References