Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Information Processing and Management 42 (2006) 407–428

INFORMATION PROCESSING & MANAGEMENT

www.elsevier.com/locate/infoproman

Unique-order interpolative coding for fast querying and space-efficient indexing in information retrieval systems

Cher-Sheng Cheng, Jean Jyh-Jiun Shann, Chung-Ping Chung [*]

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, 30050 Taiwan, R.O.C.

Received 17 May 2004; accepted 9 February 2005
Available online 30 March 2005

Abstract

This paper presents a size reduction method for the inverted file, the most suitable indexing structure for an information retrieval system (IRS). We notice that in an inverted file the document identifiers for a given word are usually clustered. While this clustering property can be used in reducing the size of the inverted file, good compression as well as fast decompression must both be available. In this paper, we present a method that can facilitate coding and decoding processes for interpolative coding using recursion elimination and loop unwinding. We call this method the unique-order interpolative coding. It can calculate the lower and upper bounds of every document identifier for a binary code without using a recursive process, hence the decompression time can be greatly reduced. Moreover, it also can exploit document identifier clustering to compress the inverted file efficiently. Compared with the other well-known compression methods, our method provides fast decoding speed and excellent compression. This method can also be used to support a self-indexing strategy. Therefore our research work in this paper provides a feasible way to build a fast and space-economical IRS.
© 2005 Elsevier Ltd. All rights reserved.

Keywords: Inverted index compression; Inverted file; Prefix-free coding; Interpolative coding; Fast decoding

1. Introduction

Information retrieval systems (IRSes) are widely used in many applications such as search engines, electronics libraries, e-commerce, electronics news, genomic sequence analysis, etc. (Kobayashi & Takeda, 2000; Williams & Zobel, 2002). To provide fast data retrieval, IRSes require an indexing structure so that

[*] Corresponding author.
E-mail addresses: jerry@csie.nctu.edu.tw (C.-S. Cheng), jjshann@csie.nctu.edu.tw (J.J.-J. Shann), cpchung@csie.nctu.edu.tw (C.-P. Chung).

0306-4573/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2005.02.002

the desired data can be located quickly. Compared with the signature file, the Pat tree, and the bitmap, the inverted file is the most suitable indexing structure for an IRS due to its quick response time, high compression efficiency, scalability, and support for various search techniques (e.g. Boolean, ranked, phrase, and proximity queries) (Faloutsos, 1985; Frakes & Baeza-Yates, 1992; Witten, Moffat, & Bell, 1999; Zobel, Moffat, & Ramamohanarao, 1998). Compression of inverted files is essential to large-scale IRSes. This is because the total time of transferring a compressed inverted list and subsequently decompressing it is potentially much less than that of transferring an uncompressed inverted list. All the well-known inverted file compression methods trade off between compression ratio and decompression time. Can we develop a method that has both the advantages of compression ratio and fast decompression?

1.1. Inverted file compression and its difficulties

An inverted file contains, for each distinct term t in the collection, an inverted list of the form

$$IL_t = <f_t; id_1, id_2, \ldots, id_{ft}>,$$

where frequency f_t is the total number of documents in which t appears, and id_i is the identifier of the document that contains t. To process a query, the IRS retrieves the inverted lists of the terms appearing in the query, and then performs some set operations, such as intersection ($\cap$) and union ($\cup$), on the inverted lists to obtain the answer list (Frakes & Baeza-Yates, 1992; Witten et al., 1999).

A popular compression technique (Witten et al., 1999) is to sort the document identifiers of each inverted list in increasing order, and then replace each document identifier (except the first one) with the distance between itself and its predecessor to form a list of d-gaps. For example, the inverted list $<5; 13, 18, 22, 35, 42>$ can be transformed into d-gap representation as $<5; 13, 5, 4, 13, 7>$. Although every document identifier is distinct, their d-gaps show some probability distributions. Many coding methods, such as unary coding (Elias, 1975), γ coding (Elias, 1975), Golomb coding (Golomb, 1966; Witten et al., 1999), skewed Golomb coding (Teuhola, 1978), batched LLRUN coding (Fraenkel & Klein, 1985; Moffat & Zobel, 1992), variable byte coding (Scholer, Williams, Yiannis, & Zobel, 2002), and word-aligned "Carryover-12" mechanism (Anh & Moffat, 2005), have been proposed for compressing inverted lists by estimates for these d-gaps probability distributions. The more accurately the estimate, the greater the compression can be achieved. In this paper, Golomb coding means the "Local Bernoulli model" described in Witten et al. (1999).

The document identifiers for any given word are not uniformly distributed, since the documents in the collection are inserted in chronological order and the word's popularity changes over time (Moffat & Stuiver, 2000). These document identifiers tend to be clustered, and inverted file compression may benefit if this clustering can be taken into account. Based on the d-gap technique, some coding methods, such as skewed Golomb coding and batched LLRUN coding, can capture clustering of documents via accurate estimates to achieve satisfactory compression performance. However, the estimates in these methods are relatively sophisticated, which require more decompression time, so they are not yet applied in real IRSes.

Recently, Moffat and Stuiver (2000) have proposed interpolative coding. It is independent of the estimates for the d-gaps probability distributions. By using clustering with a recursive process of calculating ranges and codes in an interpolative order, superior compression performance can be achieved. However, interpolative coding is computationally expensive due to a stack required in its implementation, which prohibits it from being widely used in real-world IRSes.

Therefore, to solve problems such as the slow response time and the large disk space required in large scale IRSes, a new method that provides high speed decoding and exploits clustering well to achieve good compression should be developed.

1.2. Research goal

In terms of query throughput rates, Trotman (2003) shows that for small inverted lists Golomb coding is recommended, whereas for large inverted lists variable byte coding is recommended. Furthermore, Anh and Moffat (2005) show that word-aligned "Carryover-12" mechanism allows a query throughput rate that is higher than Golomb coding and variable byte coding, regardless of the lengths of the inverted lists. Although these compression methods provide high query throughput rates, their compression efficiencies need to be improved.

In this paper, we develop a new method based on interpolative compression combined with a *d*-gap compression scheme. We call it the unique-order interpolative coding. The results of this research showed that the unique-order interpolative coding can take advantage of document identifier clustering in inverted lists to achieve good compression performance. Nevertheless, the decoding speed of this new method is even faster than that of Golomb coding and word-aligned "Carryover-12" mechanism.

This paper is organized as follows. In Section 2, we present the interpolative coding that is the most space efficient method known to compress inverted files. In Section 3, we present the unique-order interpolative coding. Then we show the quantitative analysis and the simulation results in Sections 4 and 5. In Section 6, we present some possible applications of the unique-order interpolative coding. Finally, Section 7 presents our conclusion.

2. Interpolative coding

2.1. Algorithm description

Moffat and Stuiver (2000) have proposed a compression technique called interpolative coding. It makes full use of the clustering in a recursive process of calculating ranges and codes, and demonstrates superior compression performance. In this method, the storing order as well as lower bound *lo* and upper bound *hi* of every document identifier x are calculated, and then function *Binary_Code*(x, lo, hi) is called to encode x in some appropriate manner. The simplest mechanism uses only binary code to encode x in $\lceil \log_2(hi - lo + 1) \rceil$ bits. The algorithm is described in Fig. 1.

This interpolative coding is best illustrated with an example. Consider the inverted list $<7; 1, 2, 5, 6, 8, 10, 13>$ in a collection of $N = 20$ documents. According to the algorithm in Fig. 1, the middle item in the list, the identifier 6, is encoded. This identifier must take on a value ranged from 1 to 20. Additionally, since there are three other identifiers on each side of this middle item, its possible value range is further limited to from 4 to 17. We represent this fact with $(x, lo, hi) = (6, 4, 17)$, indicating that the document identifier x is within the range $lo \ldots hi$. Once the coding of document identifier 6 is accomplished, the three document identifiers on the left-hand side may take on values 1–5 and those three on the right-hand side 7–20. According to the same rule, the three document identifiers on the left can be processed first, followed by those three on the right. Therefore, the complete sequence of (x, lo, hi) triples generated by algorithm *Interpolative_Code* are $(6, 4, 17)$, $(2, 2, 4)$, $(1, 1, 1)$, $(5, 3, 5)$, $(10, 8, 19)$, $(8, 7, 9)$, and $(13, 11, 20)$. Using the simplest implementation of *Binary_Code*, the corresponding codewords are 4, 2, 0, 2, 4, 2, and 4 bits long.

Using a centered minimal binary code, the compression efficiency of interpolative coding can be further improved (Moffat & Stuiver, 2000). The centered minimal binary code works in the following way. Support that a number in the range $1 \ldots r$ is to be coded. A simple binary code assigns codewords $\lceil \log_2 r \rceil$ bits long to all values 1 through r, and wastes $2^{\lceil \log_2 r \rceil} - r$ codewords. That is, $2^{\lceil \log_2 r \rceil} - r$ of the codewords can be shortened by one bit without loss of unique decodability. These minimal codewords are then centered on the encoding range. Numbers at the extremes of the range requiring one bit more for storage than those in the center.

```
Algorithm Interpolative_Code(IL, f, lo, hi);
Input: IL ( $IL[1 \ldots f]$ is a sorted list of f document identifiers, all in the range lo...hi)
Output: bitstring to represent $IL[1 \ldots f]$
begin
    if $f = 0$ then return;
    if $f = 1$ then output bitstring by invoking Binary_Code(IL[1], lo, hi) and then return;
    $h := (f+1)$ div 2;
    $f_1 := h-1$;
    $f_2 := f-h$;
    $L_1 := IL[1 \ldots (h-1)]$;
    $L_2 := IL[(h+1) \ldots f]$;
    Output bitstring by invoking Binary_Code(IL[h], $lo+f_1$, $hi-f_2$);
    Call Interpolative_Code($L_1, f_1$, lo, IL[h]-1);
    Call Interpolative_Code($L_2, f_2$, IL[h]+1, hi);
end
```

Fig. 1. Interpolative coding.

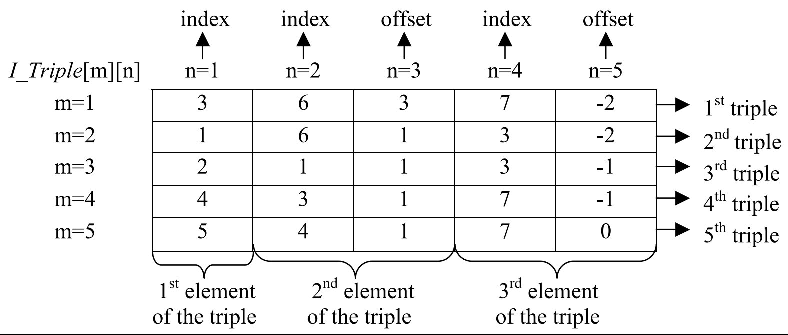
2.2. Observation and improvement

The major overhead of interpolative coding is that a recursive process is used to calculate the order and range of every document identifier. Although a recursive process can be converted to a non-recursive one (Tenenbaum, Langsam, & Augenstein, 1990), the converted code requires a stack, which makes the coding and decoding very slow. This is why interpolative coding is not widely used in IRSes.

We observed that the calculation of the order and range for every document identifier can be accelerated by storing partial results in memory. Consider a general inverted list $IL_t = <f_t; id_1, id_2, \ldots, id_{ft}>$, where f_t is the number of documents containing term t, $id_k < id_{k+1}$, and all document identifiers are within the range 1–N. Using the interpolative coding method in Fig. 1, for every f_t, we can obtain the full sequence of triples for the list. Some examples are shown in Table 1. These triple sequences are useful for interpolative coding to calculate the order and range for each document identifier. For example, consider the inverted list $IL_t = <f_t = 5; id_1 = 1, id_2 = 2, id_3 = 5, id_4 = 7, id_5 = 8>$ for a collection of $N = 10$ documents. The values of this list can be calculated using $f_t = 5$ triples in Table 1. The full sequence of triples are $(id_3, 3, N-2) = (5,3,8)$, $(id_1, 1, id_3 - 2) = (1,1,3)$, $(id_2, id_1 + 1, id_3 - 1) = (2,2,4)$, $(id_4, id_3 + 1, N-1) = (7, 6,9)$, and $(id_5, i\ d_4 + 1, N) = (8,8,10)$. Storing such a table containing a full set of triple sequences in memory is helpful for the coding and decoding processes of interpolative coding. Compared with the method in Fig. 1, this improved method eliminates need for a stack in the document identifier order and range calculation, saving a large amount of time.

Table 1
Some examples of the full sequence of triples for the general inverted list

f_t	The full sequence of triples for the general inverted list
1	$(id_1, 1, N)$
2	$(id_1, 1, N-1)$, $(id_2, id_1 + 1, N)$
3	$(id_2, 2, N-1)$, $(id_1, 1, id_2 - 1)$, $(id_3, id_2 + 1, N)$
4	$(id_2, 2, N-2)$, $(id_1, 1, id_2 - 1)$, $(id_3, id_2 + 1, N-1)$, $(id_4, id_3 + 1, N)$
5	$(id_3, 3, N-2)$, $(id_1, 1, id_3 - 2)$, $(id_2, id_1 + 1, id_3 - 1)$, $(id_4, id_3 + 1, N-1)$, $(id_5, id_4 + 1, N)$



I_Triple[m][n]	n=1 (index)	n=2 (index)	n=3 (offset)	n=4 (index)	n=5 (offset)	
m=1	3	6	3	7	-2	1st triple
m=2	1	6	1	3	-2	2nd triple
m=3	2	1	1	3	-1	3rd triple
m=4	4	3	1	7	-1	4th triple
m=5	5	4	1	7	0	5th triple
	1st element of the triple	2nd element of the triple		3rd element of the triple		

Fig. 2. Given a general inverted list IL_t: $<f_t = 5; id_1, id_2, id_3, id_4, id_5>$, and set $id_{f_t+1} = id_6 = 0$ and $id_{f_t+2} = id_7 = N$. The corresponding triples: $(id_3, 3, N-2)$, $(id_1, 1, id_3-2)$, (id_2, id_1+1, id_3-1), $(id_4, id_3+1, N-1)$, (id_5, id_4+1, N) can be represented with the *I_Triple*[f_t][5].

The triples for each f_t can easily be represented as a two-dimensional array *I_Triple* consisting of f_t rows and 5 columns. This representation for $f_t = 5$ is shown in Fig. 2. The first row of the array represents the first triple, and the second row represents the second triple, and so forth. The first column is used to denote the index of the document identifier in the inverted list for the first element of the triple. For example, *I_Triple*[3][1] is 2, meaning the first value of No. 3 triple is id_2. The second and third columns denote the index of the document identifier in the inverted list and the offset for the second element of the triple. For example, *I_Triple*[3][2] and *I_Triple*[3][3] are two 1s, meaning the second value of No. 3 triple is $id_1 + 1$. Finally, the fourth and fifth columns denote the index of the document identifier in the inverted list and the offset for the third element of the triple. For example, *I_Triple*[3][4] and *I_Triple*[3][5] are 3 and -1, meaning the third value of No. 3 triple is $id_3 - 1$. To make this representation more practical and convenient, two extra values are used for each inverted list: $id_{ft+1} = 0$ and $id_{ft+2} = N$. Therefore, the first triple (id_3, 3, $N-2$) in Fig. 2 can be represented as 3, 6, 3, 7, and -2. The algorithm in Fig. 3 can be used to generate the corresponding triples for each f_t and store them in *I_Triple*[f_t] [5]. For a sub-inverted list $IL[index\ldots(index+k-1)]$ among $id_{lo\_\text{index}} + lo$ and $id_{hi\_\text{index}} + hi$, *Compute_I_Triple*$(index, k, lo\_index, lo, hi\_index, hi)$ can be called to generate the corresponding triples and store them in a two-dimensional array *I_Triple*.

2.3. Remark

Although the procedure *Compute_I_Triple* in Fig. 3 also uses recursive process, it can be processed off-line and one can store the *I_Triple*s of different f_ts in memory. This can reduce the on-line decoding time dramatically. With the *I_Triple*, one can easily find minimal binary code in encoding an inverted list, as shown in the following:

```
for $m := 1$ to $f_t$ do
    output bitstring by invoking Binary_Code(IL[I_Triple[m][1]],
                                             IL[I_Triple[m][2]] + I_Triple[m][3],
                                             IL[I_Triple[m][4]] + I_Triple[m][5]);
```

However, this improved method still requires large memory space. For example, each triple contains five integers. If an integer takes 4-byte storage space, the memory requirement for a triple is 20 bytes. Therefore, in an inverted list with f_t document identifiers, $20 \times f_t$ bytes are required. The maximum f_t in present IRSes

```
Algorithm Generate_I_Triple(IL, f, N);
Input: IL ( IL[1...f] is a sorted list of f document identifiers, all in the range 1...N, and to simplify
        the algorithm we set IL[(f + 1)] to 0, and IL[(f + 2)] to N)
Output: I_Triple[f][5] to represent the triples
begin
  n:=1;  /* n is a global variable*/
  Compute_I_Triple(1, f, f+1, 1, f+2, 0);  /* generate I_Triple[f][5] */
  return I_Triple;
end

procedure Compute_I_Triple(index, k, lo_index, lo, hi_index, hi)
begin
  if k=0 then return;
  if k=1 then
    I_Triple[n][1]:=index;
    I_Triple[n][2]:=lo_index;
    I_Triple[n][3]:=lo;
    I_Triple[n][4]:=hi_index;
    I_Triple[n][5]:=hi;
    n++;
    return;
  h:=(k-1)/2;
  f_1:=h;
  f_2:=k-h-1;
  I_Triple[n][1]:=h+index;
  I_Triple[n][2]:=lo_index;
  I_Triple[n][3]:=lo+f_1;
  I_Triple[n][4]:=hi_index;
  I_Triple[n][5]:=hi-f_2;
  n++;
  Compute_I_Triple (index, f_1, lo_index, lo, index+h, -1);
  Compute_I_Triple (index+h+1, f_2, index+h, 1, hi_index, hi);
end
```

Fig. 3. The algorithm for generating *I_Triple*.

can reach up to thousands or millions, which means the memory space required for *I_Triple* storage is ten thousands or even ten millions of bytes. This makes it impossible using memory to accelerate coding and decoding with interpolative code. Furthermore, using *I_Triple* to encode and decode requires extra memory access time, which makes the decoding speed slow.

3. Unique-order interpolative coding

The recursive process makes the decoding of interpolative coding slow. Although using memory to store partial results of the recursive process can accelerate the coding and decoding of interpolative coding, a large amount of memory is required to store the *I_Triple* for each f_t. We develop a new method called unique-order interpolative coding in which only one *I_Triple* is required for the entire coding and decoding process of all inverted lists no matter how many different values of f_t are present. Then we introduce loop unwinding to replace *I_Triple* with constant values. The number of memory accesses to *I_Triple* can therefore be reduced, which accelerates the whole process.

3.1. The coding method

This section presents the details of our proposed coding method. Two key decisions are to be made in the coding method.

3.1.1. Decomposition of an inverted list into blocks to take advantage of interpolative coding

In an inverted list $IL_t = <f_t; id_1, id_2, \ldots, id_{ft}>$, f_t is the number of documents containing term t, $id_k < id_{k+1}$, and all document identifiers are within the range 1–N. A group size g is first determined. Then IL_t is divided into $m = \lceil \frac{f_t}{g} \rceil$ blocks, each having g document identifiers except possibly the last block. We define the first document identifier in each block to be a boundary pointer, the document identifiers between boundary pointers to be inner pointers, and those in the last block except the boundary pointer to be residual pointers. IL_t can then be compressed as follows. The boundary pointers and its subsequent residual pointers together can be regarded as a sub-inverted list, and a suitable *d*-gap compression scheme with high decoding speed can be used for compression. The inner pointers in each block are compressed via interpolative coding. With this new method (see Fig. 4), each inner block contains a constant number $(g - 1)$ of inner pointers, enabling the use of only one *I_Triple* in coding and decoding. Compared with interpolative coding, this new method allows document identifiers to be stored in a fixed order, hence the name unique-order interpolative coding. When $f_t \leqslant g$ or $m = 1$ or $g = 1$, no inner pointers are present, and we apply only a *d*-gap compression scheme.

3.1.2. Choice of a suitable coding method for boundary and residual pointers

Compared with the *d*-gaps of a traditional *d*-gap compression scheme, the *d*-gaps of unique-order interpolative coding extracted from every group of document identifiers are potentially much larger and may cause a decrease in compression efficiency. Therefore, a suitable coding method is required to encode the boundary pointers to improve compression efficiency. To simplify implementation, the boundary and residual pointers are encoded with the same method.

In this paper, we recommend Golomb coding and *r* coding for encoding the *d*-gaps of unique-order interpolative coding. Golomb coding is very suitable for encoding the *d*-gaps of unique-order interpolative coding, since the *d*-gaps extracted from every group of document identifiers are roughly of the same length. Using γ coding is also a relatively economical choice when the document identifiers in an inverted list are also close together, and the *d*-gaps are small. Other coding methods are not disregarded. We are still

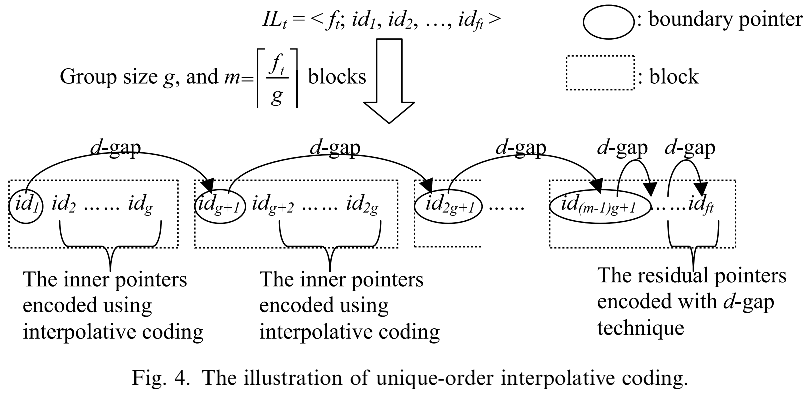

$IL_t = <f_t; id_1, id_2, \ldots, id_{ft}>$
: boundary pointer
Group size g, and $m=\left\lceil \frac{f_t}{g} \right\rceil$ blocks
: block
d-gap
d-gap
d-gap
d-gap
d-gap
id_1 id_2 …… id_g
id_{g+1} id_{g+2} …… id_{2g}
id_{2g+1} ……
$id_{(m-1)g+1}$ ……id_{ft}
The inner pointers encoded using interpolative coding
The inner pointers encoded using interpolative coding
The residual pointers encoded with d-gap technique

Fig. 4. The illustration of unique-order interpolative coding.

looking for a faster and more compact coding method to encode the *d*-gaps of unique-order interpolative coding.

To improve the compression efficiency of the *d*-gaps of unique-order interpolative coding, the value *g* is subtracted from the *d*-gap of all boundary pointers (except the first one) without loss of unique decodability. This approach works the best when the original *d*-gaps are small.

3.2. Illustration

This unique-order interpolative coding is best illustrated with an example. Given an inverted list $<11; 5, 8, 12, 13, 15, 18, 23, 28, 29, 32, 33>$, let the group size *g* be 4, the document identifiers 5, 15, and 29 are therefore the boundary pointers, the document identifiers 32 and 33 are the residual pointers, and the others are the inner pointers. Let $[id_i, id_{i+1}, \ldots, id_j]$ represent $id_i, id_{i+1}, \ldots, id_j$ encoded in interpolative code. Since the two successive boundary pointers must be known before interpolative coding of the inner pointers, the boundary pointer of each block is stored before coding of the inner pointers. Therefore, the inverted list is to be stored as

$$<11; 5, 15, [8, 12, 13], 29, [18, 23, 28], 32, 33>,$$

where $[8, 12, 13]$ and $[18, 23, 28]$ are in interpolative codes, and 5, 15, 29, 32, 33 in *d*-gaps. The resulted list is

$$<11; 5, 10(= 15 - 5), [8, 12, 13], 14(= 29 - 15), [18, 23, 28], 3(= 32 - 29), 1(= 33 - 32)>.$$

Next, since there are three document numbers between each pair of boundary pointers, the list can be simplified as

$$<11; 5, 7(= 10 - 3), [8, 12, 13], 11(= 14 - 3), [18, 23, 28], 3, 1>.$$

In decoding, the first two *d*-gaps, 5 and 7, are retrieved to obtain the first two boundary pointers, which are 5 and 15 (=5 + 7 + 3). Interpolative coding is then used to obtain $[8, 12, 13]$. Then, the *d*-gap, 11, is retrieved to obtain the next boundary point, 29 (=15 + 11 + 3), and interpolative coding is used to obtain $[18, 23, 28]$. Finally, the residual pointers 32 (=3 + 29) and 33 (=1 + 32) are obtained by the remaining *d*-gaps.

Now, consider a general inverted list $IL_t = <f_t; id_1, id_2, \ldots, id_{ft}>$ encoded using unique-order interpolative coding with group size $g = 4$, the IL_t can be represented as

$$\begin{array}{lllll} <f_t; id_1, & id_5, & [id_2, & id_3, & id_4], \\ & id_9, & [id_6, & id_7, & id_8], \\ & id_{13}, & [id_{10}, & id_{11}, & id_{12}], \quad \ldots>, \end{array}$$

where $id_1, id_5, id_9, id_{13}$ are encoded using a *d*-gap coding method and $[id_2, id_3, id_4], [id_6, id_7, id_8], [id_{10}, id_{11}, id_{12}]$ are encoded using interpolative coding. The example list can be further represented (using triple representation in Section 2) as

$$\begin{array}{ll} < f_t; id_1, & id_5 - id_1 - 3, (id_3, id_1 + 2, id_5 - 2), (id_2, id_1 + 1, id_3 - 1), (id_4, id_3 + 1, id_5 - 1), \\ & id_9 - id_5 - 3, (id_7, id_5 + 2, id_9 - 2), (id_6, id_5 + 1, id_7 - 1), (id_8, id_7 + 1, id_9 - 1), \\ & id_{13} - id_9 - 3, (id_{11}, id_9 + 2, id_{13} - 2), (id_{10}, id_9 + 1, id_{11} - 1), (id_{12}, id_{11} + 1, id_{13} - 1), \quad \ldots>. \end{array}$$

We observed that the *I_Triple* for $[id_i, id_{i+1}, id_{i+2}]$ can be converted to the *I_Triple* for $[id_{i+4}, id_{i+5}, id_{i+6}]$ by adding 4 (which is the value of *g*) to the indices of document identifiers in the *I_Triple* for $[id_i, id_{i+1}, id_{i+2}]$. Therefore, only one *I_Triple* is required in coding and decoding, which accelerates the whole process. If we

```
Algorithm Unique_Order_Interpolative_Code(IL, f, N, g);
Input: IL (IL[1...f] is a sorted list of document numbers, all in the range 1...N), and
        group size g(an integer);
Output: Bitstring (the compressed inverted list IL)
begin
  if $f \leq g$ then  // compressed by Golomb coding
      $b := \lceil 0.69 \times N / f \rceil$;
      prev_document_identifier:=0;
      for i:=1 to f
          append Golomb_Code(IL[i]-prev_document_identifier, b) to Bitstring;
          prev_document_identifier:= IL[i];
  else  // compressed by unique-order interpolative coding
      $m = \lceil f / g \rceil$;
      $b := \lceil 0.69 \times N / (f - (m-1) \times (g-1)) \rceil$;

      // encode the first boundary pointer
      append Golomb_Code(IL[1], b) to Bitstring;

      // generate I_Triple
      n:=0;
      I_Triple:=Compute_I_Triple(2, g-1, 1, 1, g+1, -1);

      for i:=0 to (m-2) do
          index:=$i \times g$;

          // encode boundary pointer
          append Golomb_Code(IL[index+g+1]-IL[index+1]-g+1, b) to Bitstring;

          // encode inner pointers
          for j:=1 to g-1 do
            append Binary_Code(IL[index+I_Triple[j][1]],
                               IL[index+I_Triple[j][2]]+I_Triple[j][3],
                               IL[index+I_Triple[j][4]]+I_Triple[j][5]) to Bitstring;

      // encode residual pointers
      for i:=(m-1) $\times$ g+2 to f
         append Golomb_Code(IL[i]-IL[i-1], b) to Bitstring;

  return BitString;
end
```

Fig. 5. Unique-order interpolative coding (using Golomb coding to encode boundary and residual pointers).

use Golomb coding to encode boundary pointers and residual pointers, this new coding method can be shown as the program in Fig. 5.

3.3. Implementation optimization

This section presents how to use loop unwinding to accelerate the encoding and decoding of unique-order interpolative coding. Note that once the group size g is determined, the program in Fig. 5 can be further accelerated. For example, for $g = 4$, the following program segment in Fig. 5

```
for $i := 0$ to $(m-1)$ do
    $index := i \times g$;

    //encode boundary pointer
    append Golomb_Code(IL[index + g + 1] − IL[index + 1] − g + 1, b) to Bitstring;

    // encode inner pointers, 8 memory accesses are required for encoding each inner
    // pointer: 5 for I_Triple accesses and 3 for IL accesses
        for $j := 1$ to $g - 1$ do
            append Binary_Code(IL[index + I_Triple[j][1]],
                               IL[index + I_Triple[j][2]] + I_Triple[j][3],
                               IL[index + I_Triple[j][4]] + I_Triple[j][5]) to Bitstring;
```

can be converted to

```
for $i := 0$ to $(m-1)$ do
    $index := i \times 4$;

    // encode boundary pointer
    append Golomb_Code(IL[index + 5] − IL[index + 1] − 3, b) to Bitstring;

    // loop unwinding, only 3 memory accesses of IL are required for encoding each
    // inner pointer
    append Binary_Code(IL[index + 3], IL[index + 1] + 2, IL[index + 5] − 2) to Bitstring;
    append Binary_Code(IL[index + 2], IL[index + 1] + 1, IL[index + 3] − 1) to Bitstring;
    append Binary_Code(IL[index + 4], IL[index + 3] + 2, IL[index + 5] − 1) to Bitstring;
```

In other words, once the group size g has been determined, the *I_Triple* accesses in loop can be eliminated in unique-order interpolative coding. So the $8 - 3 = 5$ times memory accesses for each document identifier can be avoided, which in turn accelerates the encoding process. By using the same approach, the decoding of unique-order interpolative coding can also be accelerated.

4. Analysis

Give an inverted list $IL = <f; id_1, id_2, \ldots, id_f>$, where $id_k < id_{k+1}$, and all document identifiers are within the range 1–N. As stated in Section 3, the first step in unique-order interpolative coding is to determine the group size g. Once g is determined, the IL will be divided into $m = \lceil \frac{f}{g} \rceil$ blocks, with the first $(m - 1)$ blocks containing g document identifiers and the last block containing $f - (m - 1)g$ document identifiers. The boundary pointers and the residual pointers will be coded by efficient prefix-free coding methods such as Golomb coding and γ coding, in d-gap manner, and the inner document identifiers will be coded by the interpolative coding.

Let the function $F(N,f)$ represent bits needed for compressing the f document identifiers ranging from 1 to N. Theoretically, the following approximate formulas can then be achieved (Elias, 1975; Gallager & Van Voorhis, 1975; Golomb, 1966; Mcllroy, 1982; Moffat & Stuiver, 2000).

$$\text{Golomb coding:}\quad G(N,f) \leqslant f \times \left(2 + \log_2 \frac{N}{f}\right) \tag{1}$$

$$\gamma \text{ coding}:\ \gamma(N,f) \leqslant f \times \left(1 + 2 \times \log_2 \frac{N}{f}\right) \tag{2}$$

$$\text{Interpolative coding}:\quad I(N,f) \leqslant f \times \left(2.5783 + \log_2 \frac{N}{f}\right) \tag{3}$$

If Golomb coding is used to encode the boundary pointers and residual pointers, then the maximum number of bits required to store these $f - (m - 1)(g - 1)$ boundary and residual pointers is

$$(f - (m-1)(g-1)) \times \left(2 + \log_2 \frac{N}{f - (m-1)(g-1)}\right) \tag{4}$$

If we use γ coding to encode these pointers, then the maximum number of bits required is

$$(f - (m-1)(g-1)) \times \left(1 + 2 \times \log_2 \frac{N}{f - (m-1)(g-1)}\right) \tag{5}$$

Based on Eq. (3), the number of bits required to code the inner pointers ($(m - 1)$ groups, $(g - 1)$ document identifiers in each group) is

$$\sum_{i=1}^{m-1} \left[(g-1) \times \left(2.5783 + \log_2 \frac{N_i}{g-1}\right)\right], \quad \text{where } N_i = id_{g\times i+1} - id_{g\times(i-1)+1} - 1 \tag{6}$$

Since

$$\sum_{i=1}^{m-1} N_i \leqslant N \tag{7}$$

and the sum of the logarithms of the $(m - 1)$ individual ranges is maximized when all $\frac{N_i}{g-1}$ are equal, one obtains

$$\sum_{i=1}^{m-1} \left[(g-1) \times \left(2.5783 + \log_2 \frac{N_i}{g-1}\right)\right] \leqslant (m-1)(g-1) \times \left(2.5783 + \log_2 \frac{N}{(m-1)(g-1)}\right) \tag{8}$$

Therefore, if Golomb coding is used to encode the boundary and residual pointers, then the maximum number of bits required by the unique-order interpolative coding is at most

$$(f - (m-1)(g-1)) \times \left(2 + \log_2 \frac{N}{f - (m-1)(g-1)}\right) + (m-1)(g-1) \times \left(2.5783 + \log_2 \frac{N}{(m-1)(g-1)}\right) \tag{9}$$

Or if we use γ coding, it is

$$(f - (m-1)(g-1)) \times \left(1 + 2 \times \log_2 \frac{N}{f - (m-1)(g-1)}\right) + (m-1)(g-1) \times \left(2.5783 + \log_2 \frac{N}{(m-1)(g-1)}\right) \tag{10}$$

Eqs. (9) and (10) can be simplified under the condition that no residual pointers exist. For example, when $f = (m - 1)g + 1$, Eq. (9) can be rewritten as:

Table 2
Some examples of the maximum number of bits required for unique-order interpolative coding if Golomb coding is used to encode boundary pointers under the condition that no residual pointers exist

g	Maximum number of bits required
2	$f \times \left[3.29 + \log_2 \frac{N}{f}\right]$
4	$f \times \left[3.25 + \log_2 \frac{N}{f}\right]$
8	$f \times \left[3.05 + \log_2 \frac{N}{f}\right]$
16	$f \times \left[2.88 + \log_2 \frac{N}{f}\right]$
32	$f \times \left[2.76 + \log_2 \frac{N}{f}\right]$

$$f \times \left[\frac{2 + \log_2 g + 2.5783 \times (g-1) + (g-1) \times \log_2\left(\frac{g}{g-1}\right)}{g} + \log_2 \frac{N}{f}\right] \tag{11}$$

and some examples of the maximum number of bits required for unique-order interpolative coding are derived in Table 2.

The results in Table 2 showed that when Golomb coding is used to encode boundary pointers, the maximum number of bits required in unique-order interpolative coding has inverse relationship with group size g: the maximum number of bits decreases with increase in group size g and increases with decrease in g. On the other hand, if the number of document identifiers is less than $(g + 1)$, unique-order interpolative coding cannot be used. We design an experiment in Section 5 to find a suitable group size g.

The results in Eqs. (9), (10), and Table 2 can be improved if Eq. (3) can be improved. For example, the maximum number of bits required for interpolative coding to encode an inverted list with three document identifiers ranging from 1 to N is

$$\lceil \log_2(N-2) \rceil + \lceil \log_2 a \rceil + \lceil \log_2 b \rceil \tag{12}$$

since the middle item requires $\lceil \log_2(N-2) \rceil$ bits, and the left and right items require $\lceil \log_2 a \rceil + \lceil \log_2 b \rceil$ bits where a, b are two positive integers and $a + b = (N - 1)$. Since

$$\lceil \log_2(N-2) \rceil < 1 + \log_2 N \tag{13}$$

and

$$\lceil \log_2 a \rceil + \lceil \log_2 b \rceil < (1 + \log_2 a) + (1 + \log_2 b) < 2 + \log_2 \frac{N}{2} + \log_2 \frac{N}{2} \tag{14}$$

hence

$$\lceil \log_2(N-2) \rceil + \lceil \log_2 a \rceil + \lceil \log_2 b \rceil < 3 \times \left(1.92 + \log_2 \frac{N}{3}\right) \tag{15}$$

We replace Eq. (3) with Eq. (15) when group size $g = 4$, and the maximum number of bits required for the unique-order interpolative coding under the condition that no residual pointers exist is therefore

$$f \times \left[2.76 + \log_2 \frac{N}{f}\right] \tag{16}$$

Compared with the figure in Table 2, a much tighter upper bound is obtained.

To further understand the characteristics of unique-order interpolative coding, we conducted following experiments. We used encoding methods such as Golomb coding, skewed Golomb coding, batched LLRUN coding, interpolative coding, variable byte coding, Carryover-12 mechanism, unique-order interpolative coding 1 (group size $g = 4$; boundary pointers and residual pointers by Golomb coding), unique-order interpolative coding 2 (group size $g = 4$; boundary pointers and residual pointers by γ coding) in compression. In the first experiment (Table 3(a)), $f = 1,000,000$ gaps were drawn from a geometric distribution and compressed using the eight methods. The Golomb coding performs the best, since it is a minimum-redundancy code for geometric gap distribution (Gallager & Van Voorhis, 1975). Compared with other methods, unique-order interpolative coding is not suitable for a geometric distribution when $2 < \frac{N}{f} < 256$. But when $\frac{N}{f}$ increases, the performance of unique-order interpolative coding 1 improves proportionally. When $\frac{N}{f} \leqslant 2$, the results of unique-order interpolative coding 2 are satisfying. For most cases in the first experiment, both variable byte coding and Carryover-12 mechanism are inefficient in compression.

In the second experiment, for each value of $\frac{N}{f}$ the sequence of $f = 1,000,000$ geometrically distributed gaps was broken into chunks of 200 contiguous values. The chunks were then placed in groups of five. In the first three chunks of each group, all gaps were multiplied by a factor of 0.1; whereas in the other two chunks all gaps were multiplied by a factor of 2.35. This process created artificial clusters of gaps much similar than the average, and about 60% of the values were coded into these clusters, while the overall average gap remained the same. This better resembles the distribution of real document collections. The results are shown in Table 3(b). Compared with skewed Golomb coding, batched LLRUN coding, and interpolative coding, the compression efficiency of Golomb coding is not as good as others, meaning it is unable to exploit clustering well. The compression results of unique-order interpolative coding for a skewed geometric distribution are better than that for a geometric distribution. This means that unique-order interpolative coding does take a good advantage of the clustering property. For $\frac{N}{f} \leqslant 32$, we prefer to use the unique-order

Table 3
Compression results for geometric and skew geometric distributions of $f = 1,000,000$ gaps: average bits per gap

Coding methods	Average gap N/f											
	1	2	4	8	16	32	64	128	256	512	1024	2048
(a) Geometric distribution												
Golomb coding	1.00	2.33	3.30	4.39	5.43	6.45	7.46	8.47	9.47	10.47	11.47	12.47
Skewed Golomb coding	1.00	2.53	3.51	4.60	5.64	6.66	7.67	8.68	9.68	10.68	11.68	12.68
Batched LLRUN coding	1.00	2.27	3.46	4.50	5.53	6.52	7.52	8.52	9.52	10.52	11.52	12.53
Interpolative coding	0.00	2.15	3.45	4.59	5.66	6.69	7.70	8.71	9.71	10.71	11.71	12.72
Variable byte coding	8.00	8.00	8.00	8.00	8.00	8.14	9.08	10.93	12.87	14.24	15.07	15.52
Carryover-12 mechanism	1.07	2.88	4.11	5.17	6.18	7.38	8.75	9.90	10.58	12.30	14.41	15.56
Unique-order interpolative coding 1	3.00	4.19	5.13	5.97	6.76	7.53	8.29	9.06	9.89	10.77	11.68	12.77
Unique-order interpolative coding 2	0.25	2.33	3.91	5.31	6.64	7.92	9.19	10.45	11.70	12.96	14.21	15.46
Self-entropy	0.00	2.00	3.24	4.35	5.40	6.42	7.43	8.44	9.44	10.44	11.43	12.43
(b) Skewed geometric distribution												
Golomb coding	1.40	2.60	3.30	4.29	5.33	6.37	7.39	8.40	9.40	10.40	11.40	12.41
Skewed Golomb coding	1.80	2.31	2.92	3.76	4.80	5.79	6.80	7.82	8.82	9.83	10.83	11.83
Batched LLRUN coding	1.40	2.31	2.86	3.60	4.61	5.66	6.70	7.71	8.71	9.71	10.70	11.71
Interpolative coding	0.84	1.53	2.07	2.90	3.97	5.07	6.15	7.19	8.21	9.23	10.23	11.24
Variable byte coding	8.00	8.00	8.00	8.00	8.10	8.58	9.38	10.11	10.63	11.28	12.43	13.80
Carryover-12 mechanism	1.07	2.36	2.90	3.72	4.84	6.02	6.98	7.9	9.35	10.90	12.08	12.57
Unique-order interpolative coding 1	3.60	3.96	4.30	4.80	5.51	6.30	7.11	7.94	8.76	9.60	10.51	11.62
Unique-order interpolative coding 2	1.25	1.90	2.47	3.33	4.53	5.88	7.21	8.53	9.81	11.07	12.33	13.60
Self-entropy	0.97	1.77	2.30	3.05	4.06	5.10	6.15	7.18	8.19	9.19	10.19	11.20

interpolative coding 2; while for $\frac{N}{f} > 32$, we suggest unique-order interpolative coding 1. Similar to that for a geometric distribution, the unique-order interpolative coding 1 performs better as $\frac{N}{f}$ becomes larger. Again, both variable byte coding and Carryover-12 mechanism are inefficient in compression for most cases in the second experiment. From Table 3(b), interpolative coding can even outperform self-entropy. This is due to the fact that interpolative coding does not use the gap value in encoding directly, but instead uses a minimal binary code to encode every gap after it is converted to a triple.

5. Experiments

An experimental information retrieval system was implemented to evaluate the various coding methods. Experiments were conducted on some real-life document collections, and query processing time and storage requirements for each coding method were measured.

5.1. Document collections and queries

Five document collections were used in the experiments. Their statistics are listed in Table 4. In this table, N denotes the number of documents; n is the number of distinct terms; F is the total number of terms in the collection; and f indicates the number of document identifiers that appear in an inverted file.

Collection *Bible* is the King James version of the Bible, in which each verse is considered as a document. The second collection, *DBbib*, is a set of citations to papers appearing in the database literature. The third and fourth collections, *FBIS* (Foreign Broadcast Information Service) and *LAT* (LA Times), are disk 5 of the TREC-6 collection that are used internationally as a test bed for research in information retrieval techniques (Voorhees & Harman, 1997). The final collection *TREC* includes the *FBIS* and *LAT* collections.

Since effectiveness of coding methods relies heavily on clustering of documents, inverted files for these collections were built with a Greedy-NN algorithm (Shieh, Chen, Shann, & Chung, 2003). These inverted files were then used to test the advantages and shortcomings of various coding methods.

We followed the method (Moffat & Zobel, 1996) to evaluate performance with random queries. For each document collection, 1000 documents were randomly selected to generate a query set. A query was generated by selecting words from a word list of a specific document, combined by some randomly generated Boolean operators ANDs and ORs. To form the document word list, words in the document were case folded, and stop words such as "the" and "this" were eliminated. For example, a query word list may be "inverted file document collection built", a query may be "(inverted <AND> file <AND> document <AND> collection) <OR> built". For each query, there existed at least one document in the document collection that satisfied the query. The generated queries followed a Zipf-like distribution $P \sim 1/\rho^{0.55}$, where P is the probability of accessing each query, and ρ is the popularity rank for the test query stream. This is

Table 4
Statistics of document collections

	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Documents N	31,101	32,472	130,471	131,896	262,367
# of terms F	884,746	2,320,610	72,922,893	72,087,460	145,010,353
Distinct terms n	8965	58,536	214,310	168,251	317,393
# of document identifier count f	701,304	1,694,491	28,628,698	32,483,656	61,112,354
Average gap size $N \times n/f$	398	1122	977	683	1363
Total size (Mbytes)	4.69	21.30	470	475	945

widely believed to closely resemble the distribution of real queries (Breslau, Cao, Fan, Phillips, & Shenker, 1999).

5.2. Compression performance of unique-order interpolative coding

In this section, Golomb coding was used to code both boundary pointers and residual pointers. This is due to the fact that the average gap sizes in Table 4 are relatively big, Golomb coding was recommend according to Table 3(b). The compression result is shown in Table 5, and the metric used is the average number of *bits per document identifier BPI*, defined as follows:

$$BPI = \frac{\text{The size of the compressed inverted file}}{\text{number of document identfiers } f}.$$

For each term t, the cost of using r coding to encode the frequency f_t is calculated and included in the presented results.

Note that for group size $g = 4$ and $g = 8$, unique-order interpolative coding achieved good compression. For a simple implementation, we suggest using $g = 4$. In the following experiments, Golomb coding was used to code both boundary pointers and residual pointers for unique-order interpolative coding, and group size g was set to 4 unless otherwise stated.

5.3. Compression performance of different coding methods

We now compare the effectiveness of the eight coding methods: γ coding, Golomb coding, batched LLRUN coding, skewed Golomb coding, interpolative coding, variable byte coding, Carryover-12 mechanism, and unique-order interpolative coding. For each term t, the cost of using r coding to encode the frequency f_t is calculated and included in the presented results. Moreover, any necessary overheads, such as the complete set of models and model selectors for the batched LLRUN coding, are also calculated and included. However, the cost of storing the parameter b for each inverted list in Golomb coding (Witten et al., 1999) is not calculated nor included. This is because the parameter b for each inverted list in Golomb coding can be calculated via stored frequency f_t using Witten's approximation. The results are shown in Table 6. Notice that:

1. Both variable byte coding and Carryover-12 mechanism are inefficient in compression of inverted files.

Table 5
Compression performance of unique-order interpolative coding versus different group size g

Group size g	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
1	6.11	6.20	5.27	5.31	5.49
2	5.64	5.47	4.84	4.91	4.99
3	5.61	5.31	4.80	4.89	4.94
4	5.46	5.11	4.66	4.74	4.78
5	5.52	5.13	4.71	4.80	4.82
6	5.52	5.10	4.71	4.79	4.81
7	5.47	5.04	4.65	4.74	4.75
8	5.42	4.98	4.59	4.68	4.69
9	5.47	5.01	4.64	4.72	4.73
10	5.51	5.03	4.67	4.75	4.76

Table 6
Compression performance of different coding methods

Coding methods	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
γ coding	6.58	5.96	5.38	5.63	5.63
Golomb coding	6.11	6.20	5.27	5.31	5.49
Batched LLRUN coding	5.52	4.88	4.63	4.78	4.84
Skewed Golomb coding	5.92	5.75	5.04	5.07	5.10
Interpolative coding	5.37	4.89	4.58	4.65	4.62
Variable byte coding	9.10	9.54	8.88	8.89	8.84
Carryover-12 mechanism	7.14	7.99	6.23	6.13	5.95
Unique-order interpolative coding	5.46	5.11	4.66	4.74	4.78

2. For the other coding methods, the compression efficiencies of both γ coding and Golomb coding are relatively low because of the simple models they use.
3. The compression efficiencies of batched LLRUN, skewed Golomb, interpolative, and unique order interpolative coding methods are relatively good. This shows that clustering is a good compression aid.
4. The compression efficiency of unique-order interpolative coding is only inferior to that of interpolative coding, meaning that it does take a good advantage of the clustering property.

5.4. Search performance of different coding methods

The query processing time includes (1) disk access time, (2) decompression time, and (3) document identifiers comparison time. Experiments showed that disk access time and decompression time occupy more than 90% of query processing time. And document identifier comparison time is not a function of the coding method used. Therefore the search performance metric is defined as

$$\text{Search Time (ST)} = \text{Disk Access Time (AT)} + \text{Decompression Time (DT)}.$$

And the speedups of all coding methods relative to Golomb coding, for all test collections, were calculated.

All experiments described in this section were run on an Intel P4 2.4GHz PC with 256MB DDR memory running Linux operating system 2.4.12. The hard disk was 40 GB, and the data transfer rate was 25 MB/s. Intervening processes and disk activities were minimized during experimentation. All decoding mechanisms were written in *C*, complied with *gcc*, and optimized as follows:

1. Replaced sub-routines with macros.
2. Careful choice for compiler optimization flags.
3. Implementation used 32-bit integers, as that is the internal register size of the Intel P4 CPU.
4. Implemented the integer logarithm function $\lceil \log_2(i) \rceil$ with a lookup table.
   Let z be a 256-entry array, and z[k] be $\lceil \log_2(k+1) \rceil$ where $0 \leqslant k \leqslant 255$. The function $\lceil \log_2(i) \rceil$ can be implemented in *C* as follows (v is the returned value of $\lceil \log_2(i) \rceil$):

```
do {
    register int_i = (i)−1;
   (v) =_B_i≫16 ? (_B_i≫24 ? 24 + z[_B_i≫24] : 16 + z[_B_i≫16]):
                  (_B_i≫ 8 ? 8 + z[_B_i≫8] : z[_B_i]);
} while (0);
```

5. Implemented the integer logarithm function $\lfloor \log_2(i) \rfloor$ also with a lookup table.
   The array z is the same as that used in the function $\lceil \log_2(i) \rceil$. The function $\lfloor \log_2(i) \rfloor$ can be implemented in *C* as follows (v is the returned value of $\lfloor \log_2(i) \rfloor$):
```
do {
      register int_i = (i);
      (v) =_B_i≫16 ? (_B_i≫24 ? 23 + z[_B_i≫24] : 15 + z[_B_i≫16]):
                              (_B_i≫ 8 ? 7 + z[_B_i≫8] : z[_B_i] - 1);
} while (0);
```
6. A 256-entry lookup table is used to locate the exact bit location of the first "1" bit in a byte.
   For example, in the byte 00101000 the first "1" bit is in location 3. This can accelerate the decoding process of unary codes because no bit-by-bit decoding is required.
7. Access to binary codes with masking and shifting operations, and no bit-by-bit decoding is required.

With these optimizations, decoding of a document identifier only required tens of ns, and no bit-by-bit decoding is required.

Other optimizations included: The Huffman code of batched LLRUN coding was implemented with canonical prefix codes (Turpin, 1998). The canonical prefix codes can be decoded via fast table look-up. And for the interpolative coding method, recursive process was transformed to non-recursive process, at the cost of an explicit stack (Tenenbaum et al., 1990).

The search performance measurements are shown in Table 7. Key findings are:

1. Although variable byte coding and Carryover-12 mechanism gave fast decoding, *r* coding and unique-order interpolative coding achieved higher query throughput rates. This is because the disk access time (AT) of variable byte coding and Carryover-12 mechanism is much higher than that of *r* coding and unique-order interpolative coding.
2. For collection *DBbib*, the decoding times (DT) of *r* coding and unique-order interpolative coding are less than that of Carryover-12. This is because a large portion of the *d*-gaps of frequently used query terms for *DBbib* is of value 1. Both *r* coding and unique-order interpolative coding can encode these *d*-gaps very economically. This also makes the decoding times of *r* coding and unique-order interpolative coding for these *d*-gaps very low.
3. Batched LLRUN coding, skewed Golomb coding, and interpolative coding gave better compression rates than Golomb coding. However, their complex decoding mechanisms prohibited them from being used in real-world IRSes.
4. Experimental results showed that *r* coding, Carryover-12 mechanism, and unique-order interpolative coding were recommended for real-world IRSes. Their query throughput rates were all much higher than that of Golomb coding.
5. To obtain better compression rates, Golomb coding and unique-order interpolative coding use a minimal binary code in their codewords. To decode a minimal binary code, "toggle point" calculations are required and slow down query evaluation. Rice coding is a variant of Golomb coding where the value *b* is restricted to be a power of 2. The advantage of this restriction is that there is no "toggle point" calculation required. The disadvantage of this restriction is the slightly worse compression than that of Golomb coding. If we use Rice coding to encode the boundary and residual pointers in unique-order interpolative coding and use a simple binary code to encode the (x, lo, hi) triples for the inner pointers, there is no "toggle point" calculation required for unique-order interpolative coding. Table 8 showed that Rice coding allowed query throughput rates of approximately 8% higher than Golomb coding, and unique-order interpolative coding without "toggle point" calculation allowed query throughput

Table 7
Search performance of different coding methods (AT is the disk access time, DT is the decoding time, ST = AT+DT is the search time, and SP is the performance relative to the Golomb coding)

Coding method	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
γ coding					
AT(us)	125	202	1125	1168	2149
DT(us)	70	188	952	980	1696
ST(us)	195	390	2077	2148	3845
SP	1.14	1.50	1.20	1.23	1.20
Golomb coding					
AT(us)	131	306	1282	1321	2422
DT(us)	92	280	1200	1314	2179
ST(us)	223	586	2482	2635	4601
SP	1.00	1.00	1.00	1.00	1.00
Batched LLRUN coding					
AT(us)	116	381	1101	1134	2086
DT(us)	130	192	1688	1771	3013
ST(us)	246	573	2789	2905	5099
SP	0.91	1.02	0.89	0.91	0.90
Skewed Golomb coding					
AT(us)	117	331	1120	1150	2097
DT(us)	122	201	1492	1577	2696
ST(us)	239	532	2612	2727	4793
SP	0.93	1.10	0.95	0.97	0.96
Interpolative coding					
AT(us)	111	137	1024	995	1916
DT(us)	243	688	3094	3266	5598
ST(us)	354	825	4118	4261	7514
SP	0.63	0.71	0.60	0.62	0.61
Variable byte coding					
AT(us)	214	918	3134	3489	5506
DT(us)	22	90	336	388	633
ST(us)	236	1008	3470	3877	6139
SP	0.95	0.58	0.72	0.68	0.75
Carryover-12 mechanism					
AT(us)	145	311	1498	1491	2566
DT(us)	52	190	765	825	1368
ST(us)	197	501	2263	2316	3934
SP	1.13	1.17	1.10	1.14	1.17
Unique-order interpolative coding					
AT(us)	113	182	1066	1076	2011
DT(us)	82	169	1041	1041	1909
ST(us)	195	351	2107	2117	3920
SP	1.14	1.67	1.18	1.24	1.17

rates of approximately 30% higher than Golomb coding. Experimental results further showed that the decoding time of unique-order interpolative coding without "toggle point" calculation is even less than that of Carryover-12 mechanism.

Table 8
Search performance of Rice coding and unique-order interpolative coding[a] (AT is the disk access time, DT is the decoding time, ST = AT + DT is the search time, and SP is the performance relative to the Golomb coding)

Coding method	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Rice coding					
AT(us)	133	286	1305	1345	2462
DT(us)	74	267	1004	1069	1808
ST(us)	207	553	2309	2414	4270
SP	1.08	1.06	1.07	1.09	1.08
Unique-order interpolative coding[a]					
AT(us)	119	193	1128	1137	2127
DT(us)	55	141	747	772	1363
ST(us)	174	334	1875	1909	3490
SP	1.28	1.75	1.32	1.38	1.32

[a] The boundary and residual pointers are encoded in Rice codes, the (x, lo, hi) triples for the inner pointers are encoded in simple binary codes, and group size g is 4.

6. Experimental results showed that a good coding method must be characterized by both high compression ratio and high decompression rate. The unique-order interpolative coding is such a good method.

6. Other applications

Unique-order interpolative coding, like interpolative coding, can be directly applied to encode strictly ascending integer sequences. One such example is encoding of within-document frequencies of inverted lists. If ranked queries are to be supported, it is also necessary to store with each document identifier the frequency of the term appearing within that document, giving the inverted list the form:

$$<f_t; (id_1, f_{t,1}), (id_2, f_{t,2}), \ldots, (id_{ft}, f_{t,ft})>,$$

where f_t is the number of documents containing term t, $id_k < id_{k+1}$, and $f_{t,i}$ is the frequency of term t in document i, $1 \leqslant i \leqslant f_t$. The within-document frequencies can be compressed in exactly the same manner of compressing document pointers: if there are f_t entries in an inverted list and a total of F_t occurrences of that term in the collection, the sequence of cumulative sums of the $f_{t,i}$ values also forms a strictly increasing integer sequence, and all of the existing compression methods are applicable. Because the within-document frequencies are typically small, according to Table 3(b), unique-order interpolative coding should use γ coding to encode within-document frequencies. Table 9 shows the cost, in bits per pointer, of storing the within-document frequencies for the five test collections. Test results showed that unique-order interpolative coding achieved very good compression, second to only the interpolative coding. Considering also the performance results in Section 5.4 and implementation cost, we conclude that the unique-order interpolative coding is very suitable for encoding within-document frequencies of inverted lists.

Unique-order interpolative coding can also support a self-indexing strategy with a little additional storage space. Typically, a query evaluation involves only a few document identifiers in an inverted list (Moffat & Zobel, 1996; Vo & Moffat, 1998). However, most compressed inverted lists do not support random accesses. An inverted list must be completely decompressed in order to be randomly accessed to any document identifier, and the full decompression is expensive. Recently, Moffat presented the skipped inverted lists (using self-indexing strategy) to support random access and reduce the query response time (Moffat &

Table 9
Within-document frequency index compression of all inverted lists, in average bits per pointer

Coding methods	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Unary coding	1.26	1.37	2.55	2.22	2.37
γ coding	1.38	1.44	2.14	2.00	2.07
Golomb coding	1.30	1.50	2.29	2.09	2.20
Batched LLRUN coding	1.38	1.44	2.14	2.00	2.05
Skewed Golomb coding	1.45	1.60	2.39	2.26	2.35
Interpolative coding	0.86	0.92	1.78	1.77	1.75
Variable byte coding	8.11	8.19	8.04	8.02	8.03
Carryover-12 mechanism	2.04	2.75	3.22	2.99	3.07
Unique-order interpolative coding[a]	0.96	1.02	1.92	1.76	1.84

[a] The boundary and residual pointers are encoded in r codes and group size g is 4.

Zobel, 1996). The technique is to divide the inverted list into blocks, and store the first document identifier of each block (called critical document identifier) separately from other document identifiers. Document identifier search proceeds in two steps. The first step is searching in the critical document identifier list and the second step is searching in one targeted block. Experimental results show that the self-indexing strategy can improve response time for both Boolean AND queries and ranked queries. The only shortcoming of the self-indexing strategy is that every critical document identifier requires some extra bits to specify the location of the next critical one.

Compared with the self-indexing strategy by Moffat, the unique-order interpolative coding is more efficient in generating skipped inverted lists. We illustrate this with an example. In Fig. 6, when the group size $g = 4$, each boundary pointer can be regarded as a critical document identifier in the self-indexing strategy, while the number of bits needed to store the inner pointers can be calculated using the value of the two boundary pointers and Eq. (17).

$$\text{Max. required bits of inner pointers when } g = 4 = \begin{cases} 0 & \text{if } N = 3 \\ 2 & \text{if } N = 4 \\ 3(k+1)+1 & \text{if } 4 < N < 3 \times 2^k + 3 \\ 3(k+1)+2 & \text{if } 3 \times 2^k + 3 \leqslant N \end{cases} \tag{17}$$

where N is the gap of two successive boundary pointers and $k = \lceil \log_2(N-2) \rceil - 2$. Eq. (17) is the upper bound of Eq. (12) (in Section 4) expressed in closed form, which can be obtained by simulation and validated by experiment. Since the number of bits for each inner pointer is known, those inner pointers that are useless in set operations during query processing can be skipped easily. Such a method is called skipped unique-order interpolative coding. The results in Table 10 showed that this method does not require extra

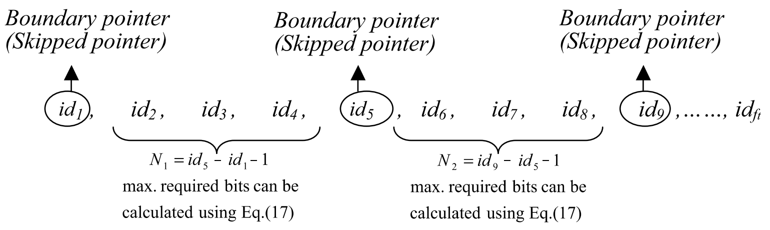

Boundary pointer (Skipped pointer)
Boundary pointer (Skipped pointer)
Boundary pointer (Skipped pointer)
id1, id2, id3, id4, id5, id6, id7, id8, id9, ……., idft
N1 = id5 - id1 - 1
max. required bits can be calculated using Eq.(17)
N2 = id9 - id5 - 1
max. required bits can be calculated using Eq.(17)

Fig. 6. Skipped unique-order interpolative coding with group size $g = 4$.

Table 10
Compression performance of Golomb coding and skipped unique-order interpolative coding

Coding methods	Collection				
	Bible	*DBbib*	*FBIS*	*LAT*	*TREC*
Golomb coding	6.11	6.20	5.27	5.31	5.49
Skipped unique-order interpolative coding	5.87	5.50	5.06	5.14	5.16

bits to specify the location of the critical document identifiers, and the storage consumed is even less than that of a Golomb code. This method is also simple, so we suggest that it be widely used in IRSes.

7. Conclusion

This paper proposes a novel coding method, the unique-order interpolative coding, for compressing inverted files in IRSes. This method is much easier to implement than interpolative coding. Furthermore, it is custom designed to suit the clustering property of document identifiers, a property that has been observed in real-world document collections. Experiments with the inverted files of five test databases show that this method yields superior performance in both fast querying and space-efficient indexing. Also shown is that it can support the self-indexing strategy efficiently. This work shows a feasible way in building a responsive and storage-economical IRS.

Acknowledgement

This work was support by National Science Council, ROC: NSC92-2213-E009-065. A shorter version (Cheng, Shann, & Chung, 2004) of this paper was presented at the ITCC'2004, April 5–7, 2004, Las Vegas, Nevada.

References

Anh, V. N., & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval, 8*(1), 151–166.

Breslau, L., Cao, P., Fan, L., Phillips, G., & Shenker, S. (1999). Web caching and zipf-like distributions: evidence and implications. In *Proceedings of eighteenth annual joint conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '99), New York, March* (pp. 126–134). Los Alamitos, CA: IEEE Computer Society Press.

Cheng, C. S., Shann, J. J. J., & Chung, C. P. (2004). A unique-order interpolative code for fast querying and space-efficient indexing in information retrieval systems. In Pradip K. Srimani, et al. (Eds.). *Proceedings of ITCC 2004 international conference on information technology: coding and communications, Las Vegas, Nevada, April* (Vol. 2, pp. 229–235). Los Alamitos, CA: IEEE Computer Society Press.

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

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys, 17*(1), 49–74.

Frakes, W. B., & Baeza-Yates, R. (1992). *Information Retrieval: Data Structures and Algorithms*. Upper Saddle River, NJ: Prentice Hall.

Fraenkel, A. S., & Klein, S. T. (1985). Novel compression of sparse bit-string preliminary report. In A. Apostolico & Z. Galil (Eds.), *Combinatorial Algorithms on Words. NATO ASI Serials F* (Vol. 12, pp. 169–183). Berlin: Springer-Verlag.

Gallager, R. G., & Van Voorhis, D. C. (1975). Optimal source codes for geometrically distributed alphabets. *IEEE Transactions on Information Theory, IT-21*(2), 228–230.

Golomb, S. W. (1966). Run Length Encoding. *IEEE Transactions on Information Theory, IT-12*(3), 399–401.

Kobayashi, M., & Takeda, K. (2000). Information retrieval on the web. *ACM Computing Surveys, 32*(2), 144–173.

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

Moffat, A., & Zobel, J. (1992). Parameterised compression for sparse bitmaps. In N. Belkin, P. Ingwersen, & A. M. Pejtersen (Eds.), *Proceedings of 15th annual international ACM-SIGIR conference on research and development in information retrieval, Copenhagen, June* (pp. 274–285). New York: ACM Press.

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

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

Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, & K. Järvelin (Eds.), *Proceedings of the 25th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 222–229). Tampere, Finland, New York: ACM Press.

Shieh, W. Y., Chen, T. F., Shann, J. J., & Chung, C. P. (2003). Inverted file compression through document identifier reassignment. *Information Processing and Management, 39*(1), 117–131.

Tenenbaum, A. M., Langsam, Y., & Augenstein, M. J. (1990). *Data structures using C*. Englewood CLiffs, NJ: Prentice-Hall.

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

Trotman, A. (2003). Compressing inverted files. *Information Retrieval, 6*(1), 5–19.

Turpin, A. (1998). Efficient prefix coding. Phd. thesis. Melbourne: University of Melbourne.

Vo, A. N., & Moffat, A. (1998). Compressed inverted files with reduced decoding overheads. In R. Wilkinson, B. Croft, & C. van Rijsbergen (Eds.), *Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval* (pp. 290–297). Melbourne, New York: ACM Press.

Voorhees, E., & Harman, D. (1997). Overview of the sixth text retrieval conference (TREC-6). In E. M. Voorhees & D. K. Harman (Eds.), *Proceedings of the sixth Text REtrieval Conference (TREC-6)* (pp. 1–24). Gaithersburg, MD: NIST.

Williams, H. E., & Zobel, J. (2002). Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering, 14*(1), 63–78.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing on documents and images* (2nd ed.). San Francisco, CA: Morgan Kaufmann Publishers.

Zobel, J., Moffat, A., & Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems, 23*(4), 453–490.

Cher-Sheng Cheng received the B.S. and M.E. degrees in computer engineering from the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, ROC in 1994 and 1996, respectively. Currently he is pursuing the Ph.D. degree in computer science and information engineering at the National Chiao Tung University, Hsinchu, Taiwan, Republic of China. His research interests include computer architecture, parallel and distributed systems, and information retrieval.

Jean Jyh-Jiun Shann received the B.S. degree in Electronic Engineering from Feng-Chia University, Taichung, Taiwan, Republic of China in 1981. She attended the University of Texas at Austin from 1982 to 1985, where she received the M.S.E. degree in Electrical and Computer Engineering in 1984. She was a lecturer in the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC, while working towards the Ph.D. degree. She received the degree in 1994 and is currently an Associate Professor in the department. Her current research interests include computer architecture, parallel processing, and information retrieval.

Chung-Ping Chung received the B.E. degree from the National Cheng-Kung University, Tainan, Taiwan, Republic of China in 1976, and the M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering. He was a lecturer in electrical engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Hsinchu, Taiwan, Republic of China, where he is a professor. From 1991 to 1992, he was a visiting associate professor of computer science at the Michigan State University. From 1998, he joined the Computer and Communications Laboratories, Industrial Technology Research Institute, ROC as the Director of the Advanced Technology Center, and then the Consultant to the General Director. He is expected to return to his teaching position after this three-year assignment. His research interests include computer architecture, parallel processing, and parallelizing compiler.