IEICE TRANS. INF. & SYST., VOL.E100–D, NO.7 JULY 2017

PAPER

Well-Balanced Successive Simple-9 for Inverted Lists Compression

Kun JIANG[†,††a)], ***Member***, **Yuexiang YANG**[††], ***and*** **Qinghua ZHENG**[†], ***Nonmembers***

SUMMARY The growth in the amount of information available on the Internet and thousands of user queries per second brings huge challenges to the index update and query processing of search engines. Index compression is partially responsible for the current performance achievements of existing search engines. The selection of the index compression algorithms must weigh three factors, i.e., compression ratio, compression speed and decompression speed. In this paper, we study the well-known Simple-9 compression, in which exist many branch operations, table lookup and data transfer operations when processing each 32-bit machine word. To enhance the compression and decompression performance of Simple-9 algorithm, we propose a successive storage structure and processing metric to compress two successive Simple-9 encoded sequence of integers in a single data processing procedure, thus the name Successive Simple-9 (SSimple-9). In essence, the algorithm shortens the process of branch operations, table lookup and data transfer operations when compressing the integer sequence. More precisely, we initially present the data storage format and mask table of SSimple-9 algorithm. Then, for each mode in the mask table, we design and hard-code the main steps of the compression and decompression processes. Finally, analysis and comparison on the experimental results of the simulation and TREC datasets show the compression and decompression efficiency speedup of the proposed SSimple-9 algorithm.

key words: *Successive Simple-9, successive storage, inverted index, decompression performance*

1. Introduction

The ever-increasing numbers of web pages and user queries have brought huge challenges to search engines' storage, update and query performance [1], [2]. Index compression techniques can directly relieve the performance bottleneck; thus, evaluation metrics should include compression ratio, compression speed and decompression speed. The latest word-aligned compression techniques, e.g., Simple-9, FOR and PFOR, upgrade processing efficiency by reducing the number of branch operations [3]–[7]. Given a fixed-length integer list, if more integers can be processed in a single branch (called a data chunk), then the total number of branch operations can be reduced. However, one serious problem in the FOR series techniques (e.g., FOR and PFOR) is that more integers in a data chunk will lead to a larger exception integer, which will increase the bit width in the data chunk and result in a lower compression ratio. To solve this problem, compact compression algorithms such as VSE and AFOR find the optimal bit width by dynamic programming but greatly reduced the compression speed [8], [9]. Although the Simple-9 compression technique shows a much better compression ratio than do many other techniques, there exist a large number of data operations such as branch operations, table lookup and bit shifting that limit further performance improvements. By increasing the number of integers compressed in a single data chunk, our aim is to upgrade the compression and decompression performance of the Simple-9 technique without affecting its high compression ratio.

It is a long and difficult challenge to weigh the choice of appropriate index compression algorithms in both search engine companies and academia. Most of the compression algorithms can show good performance on one evaluation metric. One importance factor driving the choice of compression method to adopt in commercial search engines is the balance between different performance indicators [10], [11]. The Group Varint inverted index compression algorithm proposed and adopted by Google can greatly enhance the decompression performance of the Varint algorithm [1]. The algorithm can compress 4 integers as one data chunk and remains byte-aligned at the border of the compressed codeword. This feature can reduce the number of branch operations and upgrade compression and decompression performance. Research shows that the performance of Simple-9 can exceed that of Varint on compression ratio, compression speed and decompression speed. In this paper, we focuses on improving the word-aligned encoder Simple-9 and proposes the Successive Simple-9 (SSimple-9) compression algorithm in accordance with the successive storage metric. The goal is to reduce the number of branch operations by increasing the number of compressed integers in a data chunk. We need to redesign the storage structure of data area and status area and generate the mask tables spanning two adjacent machine words. By reducing the number of branch operations, shifting, and table lookup, we are aiming at further upgrading the compression and decompression speed of Simple-9.

The remainder of this paper is organized as follows. We provide related works on inverted index compressions in Sect. 2. In Sect. 3, we present the integer storage format of SSimple-9 compression. The compression and decompression process of the SSimple-9 compression algorithm is described in Sect. 4. In Sect. 5, the performance of the SSimple-9 compression algorithm is tested explicitly on the

Manuscript received November 20, 2016.
Manuscript revised March 2, 2017.
Manuscript publicized April 17, 2017.

[†]The authors are with the School of the Electronic and Information Engineering, Xi'an Jiaotong University, China.
[††]The authors are with the College of Computer, National University of Defense Technology, China.
a) E-mail: jk_365@126.com

DOI: 10.1587/transinf.2016EDP7466

simulation and TREC datasets. Finally, Sect. 6 concludes this paper and presents prospects for future work.

2. Related Works

The inverted index plays an important role in the efficient query processing of search engines with huge web datasets [12]. An inverted list can be viewed as an ordered list of integers, in which each entry of the list corresponds to a different term or word in the collection. The inverted lists can consist of many millions of postings, which could be approximately linear with the size of the collection [13]. To allow faster access and to limit the amount of memory needed, search engines use various compression techniques that can significantly reduce the size of the inverted lists. One common practice when storing an inverted list is to use *d*-gaps, in which it is possible to decrease the average value that must be compressed, resulting in a sequence of smaller numbers with a higher compression ratio. Instead of naively storing the raw integer in a 32-bit or 64-bit machine word, the main idea of index compression is to store each integer using as few bits as possible. The compressed representation of integer lists is called codeword sequence. Index encoders can be divided into bit-aligned encoders, byte-aligned encoders and word-aligned encoders.

Bit-aligned encoders assign a distinct codeword to each possible integer, and the compression process consists of representing each integer in the posting list with a predetermined codeword. According to the distribution the integers follow, these methods can be categorized into global and local methods. The distribution of the global method does not depend upon the input sequence, and all of the integers use the same compression model. Examples of this class of encoders are Unary Code, Gamma/Delta Code, and Golomb/Rice Code [13]–[16]. Local methods adjust the parameters of the model according to the change of the input, e.g., Interpolative Code, which leads to a higher compression ratio [17]. The bit-aligned compression algorithms have high compression ratios, but frequent bitwise operations will greatly reduce compression/decompression efficiency.

Byte-aligned encoder codes such as Varint and Group Varint represent an integer in bytes [1], [18]. The basic idea of Varint is to use the low 7 bits of a byte as the data area and the most significant bit as the status bit to indicate whether the byte is the last that stores the data of the integer. Varint compression can have a poor compression ratio because it requires one full byte to encode small integers. However, it requires only a single branching condition for each byte, and the decompression speed is much higher than the bit alignment compression algorithm. The Group Varint encoder compresses a group of integers as a chunk, and the status bits of all of the integers are stored together; the same is true for the data bits. Thus, it has a higher compression/decompression speed than does Varint, indicating that a chunk with a large number of integers can upgrade the compression and decompression procedure.

The most popular compression algorithms are those of the word-aligned encoders; these access and process each codeword that is aligned to a 32/64-bit word boundary. We divide word-aligned encoders into those that employ a fixed number of integer compressions (a list of codewords) and those that employ an unfixed number of integer compressions (single codeword).

The first group encode fixed $32m$ (m is a positive integer) integers in a data chunk with equal-length bit width, ensuring that the last codeword is word-aligned. Typical encoders are FOR, PFOR, VSE, and AFOR [6]–[9]. In FOR compression, the range of values in the chunk is coded first; then, all values in the chunk are written in reference to the range of values [6]. However, if an integer exists that is greater than other values in the chunk (called an exception), then the bit width will be assigned to adjust the exception integer. PFOR is an extension of FOR that is less vulnerable to exceptions, in which the normal integers are encoded with the same bit width but the exceptional integers are stored in a separate location. One serious problem with PFOR is that more integers in a data chunk will lead to a larger exception integer, which will result in a lower compression ratio. To solve this problem, compact compression algorithms such as OptPFD, VSE and AFOR attempt to find the optimal bit width but greatly reduced the compression speed [8], [9], [19]. OptPFD determines the proper bit width by weighing the total normal integers and exceptional integers, leading to the overall optimal compression ratio [19]. VSE and AFOR both use dynamic programming approaches to partition a fixed-length chunk into several sub-chunks, each with the same bit width [8], [9]. The search for optimal partitioning improves the compression ratio but leads to a serious performance bottleneck of the index update.

The second group of encoders [3]–[5], [20], [21], for example, Simple-X (X = 9, 16, 8*b*), encode as many integers as possible into one single codeword with different padding modes, as shown in Table 1. Although the Simple-X compression technique shows a much better compression ratio than do many other techniques, there exist a large number of data operations such as branch operations, table lookup and bit shifting that limit further performance improvements. By increasing the number of integers compressed in a single data chunk, our aim is to upgrade the compression and decompression performance of

Table 1 Pre-computed lookup table representing the 9 different padding modes for the use of the 28 data bits.

	Status area (4 bits)	Chunk length	Bit width (bits)	Wasted bits
0	#*a* (0000)	28	1	0
1	#*b* (0001)	14	2	0
2	#*c* (0010)	9	3	1
3	#d (0011)	7	4	0
4	#*e* (0100)	5	5	3
5	#*f* (0101)	4	7	0
6	#*g* (0110)	3	9	1
7	#*h* (0111)	2	14	0
8	#*i* (1000)	1	28	0

the Simple-9 technique without affecting its high compression ratio. Research shows that the performance of Simple-9 can exceed that of Varint on compression ratio, compression speed and decompression speed. Inspired by the Group Varint compression, we think it is feasible to combine successive Simple-9 compressed codewords together with new storage format, which can reduce the number of branch operations by increasing the number of compressed integers in a data chunk.

Recently, a series of group compression frameworks were proposed to speed up the processing by taking account of the SIMD instructions [22]–[24]. However, the bit width of all groups remains the same for vectorization; thus, there has been a certain degree of loss in compression ratio.

3. Successive Storage Format

To increase the number of integers that can be compressed together in a chunk by Simple-9 and reduce the number of branch operations in compression and decompression procedure, we propose the SSimple-9 compression. The main idea of the data storage format is designed as follows. We combine integers coded by two adjacent 32-bit Simple-9 coded machine words together; the most significant 8 bits of the first codeword are assigned as the status area. Then, the remaining 24 bits of the first codeword and the second 32-bit machine word are assigned as the data area. The 8-bit status area can represent a total of 16*16=256 cases, and only 9*9=81 cases are used for the padding modes of the SSimple-9 compression.

The 28-bit data area coded by Simple-9 cannot fit into the 24-bit data area in the first 32-bit machine word of SSimple-9. The remaining 4 bits coded by Simple-9 can be stored in the second 32-bit machine word of SSimple-9. Thus, the 24 bits of the first machine word and the most significant 4 bits of the second machine word can be used together to store the 9 padding modes of Simple-9, and the 28 bits of the second machine word can be used to store all of the padding modes of Simple-9. The storage format should be carefully redesigned because the padding modes cross the machine boundaries. Figure 1 shows one padding mode storage of SSimple-9 combined by mode #*e* and mode #*a* of Simple-9.

The least significant 5 bits of mode #*e* are divided into two parts, in which the first 4 bits are stored in the first 32-bit machine word, and the left 1 bit is stored in the second 32-bit machine word. In most padding modes of SSimple-9, some integers are stored across the adjacent 32-bit machine word, and decompression should be done by combining the bit sequence at the end of the first 32-bit machine word and the bit sequence at the front of the second 32-bit machine word. The compression ratio of SSimple-9 remains the same as Simple-9, but the storage format of the two machine words is completely changed. As previously described, the reason that word-aligned compression such as Simple-9 is faster than byte-aligned compressions is that the use of a machine-word process unit leads to fewer branch operations. In the decompression procedure, the integer sequence can be obtained by shifting the data area according to the exact mask table. To reduce the cost of branch operations further, the mask table can be hard-coded for loop-unrolled operations. For SSimple-9 compression, 81 cases of padding modes should be hard-coded for fast integer extraction, as shown in Table 2.

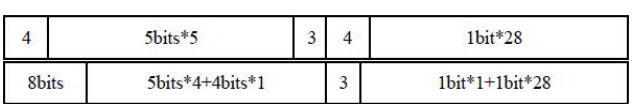

Fig. 1 An example shows the integer storage across the word boundary in SSimple-9.

4. Details of Compression and Decompression

4.1 Compression Description

We present the compression details of SSimple-9 in Algorithm 1 according to the data storage format. The input of the algorithm is an integer array d with length n, and the

Table 2 Pre-computed lookup table representing the 81 different padding modes for the use of the 56 data bits.

	Status area (8 bits)	The first part of data area (24 bits), b bits*k num	The second part of data area (32 bits), b bits*k num	Wasted bits
0	#*aa* (00000000)	1*24	1*4+1*28	0
…	…	…	…	…
8	#*ai* (00001000)	1*24	1*4+28*1	0
9	#*ba* (00010000)	2*12	2*2+1*28	0
…	…	…	…	…
11	#*bc* (00010010)	2*12	2*2+3*9	1
…	…	…	…	…
17	#*bi* (00011000)	2*12	2*2+28*1	
18	#*ca* (00100000)	3*8	3*1+1*28	1
…	…	…	…	…
26	#*ci* (00101000)	3*8	3*1+28*1	1
27	#*da* (00110000)	4*6	4*1+1*28	0
…	…	…	…	…
35	#*di* (00111000)	4*6	4*1+28*1	0
36	#*ea* (01000000)	5*4+4*1	1*1+1*28	3
…	…	…	…	…
40	#*ee* (01000100)	5*4+4*1	1*1+5*5	6
…	…	…	…	…
44	#*ei* (01001000)	5*4+4*1	1*1+28*1	3
45	#*fa* (01010000)	7*3+3*1	4*1+1*28	0
…	…	…	…	…
53	#*fi* (01011000)	7*3+3*1	4*1+28*1	0
54	#*ga* (01100000)	9*2+6*1	3*1+1*28	1
…	…	…	…	…
62	#*gi* (01101000)	9*2+6*1	3*1+28*1	1
63	#*ha* (01110000)	14*1+10*1	4*1+1*28	0
…	…	…	…	…
69	#*hg* (01110110)	14*1+10*1	4*1+9*3	1
…	…	…	…	…
71	#*hi* (01111000)	14*1+10*1	4*1+28*1	0
72	#*ia* (10000000)	24*1	4*1+1*28	0
…	…	…	…	…
80	#*ii* (10001000)	24*1	4*1+28*1	0

Algorithm 1 SSimple-9 Compression

```
Input: a sequence of numbers, d, of n integers.
 1: for k from 0 to n do
 2:     set j1 = 0, j2 = 0 and k' = k
 3:     for i from 0 to modenum[j1] do
 4:         if 2^bitlength[j1] <= d[k' + i] then
 5:             j1 + +
 6:             continue
 7:         end if
 8:     end for
 9:     k' + = modenum[j2]
10:     for i from 0 to modenum[j2] do
11:         if 2^bitlength[j2] <= d[k' + i] then
12:             j2 + +
13:             continue
14:         end if
15:     end for
16:     s = M[j1][j2] //status lookup table
17:     switch(s) do
18:         case_s:
19:             code_s(d,k,r)
20:             break
21:         ...
22:     end switch
23: end for
Output: a sequence of codewords r.
```

Algorithm 2 SSimple-9 Decompression

```
Input: a sequence of codewords r.
 1: for k from 0 to n do
 2:     two successive codewords, r1 and r2 in r
 3:     unsigned int s = r1 >>> 24
 4:     switch(s) do
 5:         case_s:
 6:             decode_s(r1,r2,d,k)
 7:             k+=out_s
 8:             break
 9:         ...
10:     end switch
11: end for
Output: integer list d with n numbers.
```

output of the algorithm is an integer array r with length less than n. The lookup table of Simple-9 is stored as two arrays (*modenum* and *bitlength*), provides the number of coded integers and the bit lengths of the different padding modes. The maximum value that can be represented in one padding mode is calculated using the bit length (line 4, line 11). The algorithm first chooses proper padding modes using Simple-9 for a given sequence of integers (line 3-15). With the successive padding mode j_1 and j_2, the status bits of SSimple-9 can be determined for the above Simple-9 coded integers (line-16). Then, the specific padding mode of SSimple-9 can be used to compress the above Simple-9 coded integers (line 17-22).

Given an integer sequence 1, 1, ..., 1, 31, 32 (27 1s and integer 31, 32), the compression steps with Simple-9 of these 29 integers are as follows. First, we use mode #*b* to present the first 14 1s and leave sequence 1, 1, ..., 1, 31, 32 (13 1s and integer 31, 32). Then, we use mode #*c* to present the following 9 1s and leave sequence 1, 1, 1, 1, 31, 32 (4 1s and integer 31, 32). Next, we use mode #*e* to present the following 5 integers 1, 1, 1, 1, 31. Finally, we use mode #*e* to present the last integer 32. The compression will cost 4 32-bit machine words to present the above sequence. When the hard-coding optimization is used, compression and decompression both require 4 table lookup and 4 branch instructions.

If the above integer sequence is compressed with two 64-bit machine words, the total data processing and branch instructions will be halved. The details of the SSimple-9 compression are as follows. First, we can obtain the two padding modes #*b* and #*c* by testing two adjacent 32-bit machine words and form the new padding mode #*bc*. Then, the status area can be stored as bit sequence 00010010 by a mask table lookup, and the data area can be stored as the 56-bit sequence 01010101010101010101010101, 00100100100100100100100100100. The actual data can be stored as two 32-bit integers. The first 32-bit integer will combine the status area 00010010 and 24-bit data area 010101010101010101010101. The second 32-bit integer will be the left 32 bits 01010010010010010010010010010010. The final two compressed integers are {307582293, 1380525202}. Next, we can also compress the remaining integers into {1141391903, 2080374784}. When the hard-coding optimization is used, the total integer sequence needs 2 table lookup and 2 branch instructions.

4.2 Decompression Description

The decompression procedure is depicted in Algorithm 2. The input and output of the algorithm are both integer arrays. For two successive 32-bits codewords, the status bits is first decoded by shifting the most significant 8 bits of the first codeword, which forms the following padding mode (line 3). Then, the corresponding mask can be obtained by hardcoded lookup in the mask table. Finally, the algorithm choose a proper hardcoded decoding function, in which the integers can be extracted by shifting a fixed bit-width defined by the given mask (line 4-10).

Given two codewords {307582293, 1380525202}, the decompression steps are as follows. First, change the first codeword 307582293 to the binary representation 00010010010101010101010101010101. Second, shift left to obtain the 8 status bits 00010010, and the padding mode #bc can be obtained by lookup in the mask table. Next, we change the second codeword 1380525202 to the binary representation 01010010010010010010010010010010 and append it to the 24-bit data area of the first codeword 010101010101010101010101. For the combination binary representation of the 56-bit data area 01010101010101010101010101, 01010010010010010010010010010010, we can extract the integer sequence by shifting left 2 bit-widths 14 times and a 3-bit width 9 times. Then, the integer sequence can be stored in an integer array as {1, ..., 1} (23 1s). Finally, the remaining two codewords {1141391903, 2080374784} can also be decoded to {1, 1,

Algorithm 3 code_69(d,k,r)

Input: integer array d and start index k.
1: $r'[0] = (r'[0] << 14) + d[k]$
2: $r'[0] = (r'[0] << 10) + (d[k+1] >>> 4)$
3: $r'[1] = (r'[1] << 4) + ((d[k+1] << 28) >>> 28)$
4: **for** i from 0 to 3 **do**
5: $\quad r'[1] = (r'[1] << 9) + d[k+2+i]$
6: **end for**
7: $r'[0] = s << 24 | r'[0]$
8: append $r'[0]$ and $r'[1]$ to r
Output: a sequence of codewords r.

Algorithm 4 decode_69(r_1,r_2,d,k)

Input: integer codewords r_1, r_2 and index k in d.
1: $d[k] = (r_1 << 8) >>> 18$
2: $d[k+1] = (r_1 << 22) >>> 18 | (r_2 >>> 27)$
3: $d[k+2] = (r_2 << 5) >>> 23$
4: $d[k+3] = (r_2 << 14) >>> 23$
5: $d[k+4] = (r_2 << 23) >>> 23$
Output: integer array d.

1, 1, 31, 32} with the same method. With the hard-coding metric, the decoding procedure of SSimple-9 needs 2 table lookup and 2 branch instructions overall, compared to 4 table lookup and 4 branch instructions of Simple-9.

As shown above, the hard-coded shifting and mask operations can be specifically designed to speed the compression and decompression. Algorithm 3 and Algorithm 4 present the loop-unrolled pseudocode of the compression and decompression of padding mode #*hg* ($14bit*2+9bit*3$). The input and output of the two algorithms are both 32-bit integers. There is a pair of hard-coded compression and decompression routines for each padding mode, and each pair is assigned a number as its status.

As depicted in Algorithm 3, the integers are checked from the smaller bit width to the larger bit width of different padding modes, and padding mode #*hg* ($14bit*2+9bit*3$) is finally selected. Thus, the first integer is temporarily stored at the last 14 bits of the first codeword (line 1). The first codeword is left shifted for 10bits, and the first 10 bits of the second integer are stored at the last 10 bits of the first codeword (line 2). The last 4 bits of the second integer are stored temporarily at the last 4 bits of the second codeword (line 3). The 3 9-bit-width integers are stored one by one at the last 27 bits of the second codeword with the shifting left of 9 bits 3 times (line 4-6). Finally, the status bits are stored at the first 8 bits of the first codeword (line 7). However, in the decoding procedure in Algorithm 4, all of the integers are extracted exactly as indicated by the definitions of different bit areas of padding mode #*hg*, particularly the cross-word boundary storage (line 2).

For an given sequence of integers, the padding modes of Simple-9 cannot be determined until checking the bit width of each integer while compressing. Thus, the number of overall branch operations for a given inverted lists (integer sequence) is hard to estimate before compressing, as shown in previous works. However, it is sure that the branch operations are halved for SSimple-9 compared to that in Simple-9. For a given sequence of n integers, there will be a branch operation for every data chunk in both compressing and decompressing. If the number of branch operations of the integer sequence for Simple-9 is m, we can only conclude that $n/28 \leq m \leq n$ before compressing, but the number of branch operations of the sequence for SSimple-9 will be surely halved to $m/2$. That is to say, the time complexity of branch operations is $O(n)$ for Simple-9 and surely $O(n/2)$ for SSimple-9, which always leads to high performance achievement for SSimple-9.

5. Experimental Results

5.1 Experimental Setup

The experimental evaluation is composed of synthetic data evaluation and real web TREC data evaluation. The synthetic dataset evaluations are performed on the platform constructed in [23], in which the clustering and uniform data models used are from [5]. The two models generate sets of distinct sorted integers that can be stored as d-gaps. In the uniform model, integers follow a uniform distribution between the given integer ranges. That is, randomly generate fixed number of integers within a ranges and convert them into sequences of d-gaps. In the clustering model, integers follow a clustering distribution. The clustering model is generated from a recursive processing of the uniform model. Given the integer range array $A[l \ldots r]$, we want to select f distinct numbers, where $f \leq r-l+1$. If $f < 10$, then f numbers in array A are selected by the uniform model. When $f \geq 10$, array A is divided into two sub-arrays $A[l \ldots m]$ and $A[m+1 \ldots r]$ (m is randomly selected). The two sub-arrays must both provide $f/2$ distinct numbers. There are three different sub-array data selection metrics. The first one is the selection in $A[l \ldots m]$, which obeys the uniform model, whereas the selection in $A[m+1 \ldots r]$ obeys the clustering model. The second one is the selection in $A[l \ldots m]$, which obeys the clustering model, whereas the selection in $A[m+1 \ldots r]$ obeys the uniform model. The third one is the selections in $A[l \ldots m]$ and $A[m+1 \ldots r]$; both obey the clustering model. The choice of the above cases is made randomly, with a probability of 0.25, 0.25 and 0.5, respectively. The gaps between these integers in the clusterings are relatively small, which are more compressible than are those of uniform data. Thus, the clustering data are more likely to simulate realistic data of web pages.

The real dataset evaluation is based on the Terrier information retrieval platform [25]. We used inverted lists obtained from two TREC Web test collections, WT2G and GOV2. The full name of TREC is Text REtrieval Conference, which is co-sponsored by the National Institute of Standards and Technology (NIST) and the U.S. Department of Defense (DOD). Its purpose was to support research within the information retrieval community by providing the infrastructure necessary for large-scale evaluation of text retrieval methodologies. The WT2G and GOV2 are two gen-

eral datasets provided by TREC for information retrieval evaluation. The TREC WT2G collection contains approximately 247 thousand documents, with an uncompressed size of 2 GB. The GOV2 collection contains approximately 25.2 million documents, with an un-compressed size of 426 GB. For the query processing evaluation, we use 10,000 queries randomly selected from the TREC2005 Efficiency Track Queries.

The synthetic data evaluation is used for debugging the compression algorithm, and the real web data evaluation is based on the search engine query processing. Thus, the experimental hardware environment differs; the former is on a personal laptop with a quad-core Intel(R) Core(R) i5-5200U processor running at 2.20 GHz with 8 GB of RAM and 12,288 KB of cache, and the latter is on a dedicated, otherwise idle server with an Intel(R) Xeon(R) E5-2640 processor running at 2.00 GHz with 128 GB of RAM and 20MB of L3 cache. All solutions were implemented in JAVA. In every experiment in which we report running time, the JVM was initially executed a certain number of times. The benchmark for warmup and the numbers are averaged over 3 independent runs when the JVM reached steady-state performance. Because the compression ratio remains unchanged for SSimple-9, the effective criteria we count are compression/decompression speeds.

5.2 Synthetic Data Evaluation

In this section, we use the two models of clustering data and uniform data to verify the validity of the proposed SSimple-9 algorithm. For the two models, we generate integer arrays that simulate docIDs in inverted lists. The integers are selected randomly in the range $[0, 2^{29})$ for both data models. In the first pass, we generated 2^{10} short arrays containing 2^{15} integers each. (We can also generate arrays with fewer integers, such as 2^9.) The average difference between successive integers within an array is thus $2^{29-15} = 2^{14}$. We expect the compressed data to use at least 14 bits per int (b/i) for the most sparse integers distributed in the given ranges. In the second pass, we generated only one long array of 2^{25} integers. (We can also generate array with more integers, such as 2^{28}.) In this case, the average distance between successive integers is $2^{29-25} = 2^4$. We expect the compressed data to use at least 4 b/i for the most sparse integers distributed in the given ranges. We choose compression ratio, compress speed and decompress speed as the criteria. To have a general comparison, we select the commonly used fast decompressive compressions Varint, Group Varint, Simple-9 and OptPFD as the baselines. The experimental results are shown in Tables 3–6. The compression ratio is averaged across the fixed number of integers within an increasing distribution ranges. The compress/decompress speed is measured in millions of integers per second (mi/s).

Tables 3–6 shows the great compression and decompression speed improvement of SSimple-9; the best case achieves a speed twice that of Simple-9 because the number of branch operations can be significantly reduced in the 64-bit SSimple-9 compression. We also find that the same algorithm is more compressive in clustering data than it is in uniform data because the gaps between the integers of clustering data are relatively small. Additionally, the same compression algorithm on long arrays is more compressive than that on short arrays because long arrays within a fixed range can have smaller integer gaps. Group Varint achieves the best compression speed because the compression process is relatively simple and can largely avoid branch operations that exist in the Varint compression. In addition, although OptPFD maintains the best decompression speed, its compression speed is extremely slow because the dynamic programming technique is used to calculate the optimal compression in the compression process of OptPFD. This characteristic also illustrates why the index compression algorithm used by most search engines not only considers the optimal decompression speed but also the index construction speed. The proposed SSimple-9 compression can significantly improve both compression and decompression speed without changing the compression ratios of Simple-9. In addition, the performance of Simple-9 is much better than that of Varint. Thus, we can conclude that the SSimple-9 algo-

Table 3 Results of SSimple-9 with uniform data: short arrays.

Compressions	Compression ratio (b/i)	Compress speed (mi/s)	Decompress speed (mi/s)
Varint	11.6	84.7	197.7
Group Varint	11.6	250.1	383.5
OptPFD	9.6	5.2	609.5
Simple-9	10.9	73.7	284.2
SSimple-9	10.9	143.8	494.5

Table 4 Results of SSimple-9 with uniform data: long arrays.

Compressions	Compression ratio (b/i)	Compress speed (mi/s)	Decompress speed (mi/s)
Varint	8.0	164.7	274.3
Group Varint	8.0	361.0	480.0
OptPFD	4.5	10.3	863.7
Simple-9	4.5	111.0	380.3
SSimple-9	4.5	197.3	797.0

Table 5 Results of SSimple-9 with clustering data: short arrays.

Compressions	Compression ratio (b/i)	Compress speed (mi/s)	Decompress speed (mi/s)
Varint	10.8	84.1	205.7
Group Varint	10.8	261.7	387.2
OptPFD	8.3	6.0	684.8
Simple-9	9.4	81.4	311.2
SSimple-9	9.4	122.9	546.2

Table 6 Results of SSimple-9 with clustering data: long arrays.

Compressions	Compression ratio (b/i)	Compress speed (mi/s)	Decompress speed (mi/s)
Varint	8.0	166.0	283.7
Group Varint	8.0	374.7	496.0
OptPFD	3.9	12.3	907.0
Simple-9	4.0	139.3	441.0
SSimple-9	4.0	158.7	711.3

rithm can greatly improve the performance of index building and query processing of search engines while maintaining a better compression ratio.

5.3 TREC Data Evaluation

In this subsection, we present a realistic web page data evaluation on TREC WT2G and GOV2. The criteria we use here are index size and query processing latency. From the synthetic dataset evaluation, we conclude that the SSimple-9 can be used to speed the processing of index building. However, the inverted indexes are first built with Elias-Gamma compressed and then recompressed to different compression algorithms on the Terrier platform. Thus, we will not present index building performance in this subsection. Our inverted lists include docIDs, term frequencies, field frequencies and term positions. We build docID-sorted inverted index structures with 1024 docIDs per chunk using the Elias-Gamma compression, removing the standard English stopwords, and applying Porter's English stemmer. Finally, the index is built by the Terrier platform with single-pass in-memory indexing and can be recompressed for different index compression algorithms.

The inverted lists are compressed using the original Elias-Gamma compression if the number of integers can only fit fewer than 28 bits of data area. The average index size of the inverted lists is provided in Table 7. The index size compressed by SSimple-9 is the same as that of Simple-9 but is much better than that of Group Varint. OptPFD achieves a better compression ratio than does Varint, Group Varint, Simple-9 or SSimple-9 because the dynamic programming metric is used to obtain optimal chunk splitting with a different compression bit width. Moreover, the best compression ratio can be achieved by Elias-Gamma with smaller datasets and by OptPFD with larger datasets because it is proper to compress smaller integers with Elias-Gamma and larger integers with OptPFD, which can handle exceptional values. However, our SSimple-9 and the original Simple-9 compressions can achieve an acceptable compression ratio.

Next, compared performance directly in a real searching context, i.e., by answering queries with the above WT2G and GOV2 indexes. We use topics 401-450 and topics 751-800 for querying the above WT2G and GOV2 indexes, respectively. We use disjunctive document-at-a-time as the index traversal technique and BM25 as the ranking function. The inverted lists related to the query terms are loaded into main memory at the beginning of each experiment. Every time we report the query latency, the JVM warm-up is necessary to maintain a steady performance state, and the results are averaged over 5 independent runs. The average query latency results are shown in Table 8. For each corpus, this table shows the query-processing speed measured in milliseconds per query.

As depicted in Table 8, the query processing performance of SSimple-9 is significantly improved compared with that of Simple-9, which is close to the performance of OptPFD. Additionally, the superior performance of Group Varint compared with that of Varint shows the effectiveness of the group compression metric. Because many factors can influence the speed of real web-query processing and decompression performance is only one factor, the results might not be as good as those shown in the synthetic evaluation. However, the performance of SSimple-9 compression is almost the same as that of OptPFD compression. In addition, because Terrier uses 1024 docIDs as a data compression chunk, word-aligned fixed number of integer compressions such as OptPFD need only use a different compression algorithm when the number of integers is less than 128. However, the Simple series of compression should use a different compression algorithm for every 1024 docIDs, which can affect overall performance. Thus, the query processing performance can be enhanced by using an unfixed chunk size for the Simple-9 series compression. In summary, the SSimple-9 compression does not achieve optimal performance on one single criterion. However, it shows well-balanced performance in compression speed, compression ratio and decompression speed.

6. Conclusions

To enhance the compression and decompression speed of Simple-9 compression, resulting in faster query processing performing search engines, we have proposed a novel SSimple-9 compression and designed the storage format and details of the compression/decompression process. In essence, the new compression improves the number of integers that can be compressed in a single data-processing procedure, thus shortening the process of branch operations, table lookup and data transfer operations when compressing the integer sequence. Experimental results on a synthetic dataset show that the compression and decompression procedures of the proposed SSimple-9 algorithm achieve a speed almost 2 times faster than that of Simple-9, and

Table 7 The index size of the inverted lists achieved by different compression techniques for the TREC WT2G and GOV2 datasets.

Compressions	WT2G (MB)	GOV2 (MB)
Elias-Gamma	104	10,099
Varint	258	19,359
Group Varint	258	19,359
OptPFD	146	9,679
Simple-9	148	10,367
SSimple-9	148	10,367

Table 8 Average query latency in milliseconds (ms) on TREC WT2G and GOV2 indexes.

Compressions	WT2G (ms)	GOV2 (ms)
Elias-Gamma	35.1	438.0
Varint	29.0	399.0
Group Varint	25.4	376.2
OptPFD	23.2	324.5
Simple-9	25.0	361.0
SSimple-9	23.7	331.4

query processing on the realistic TREC dataset shows significant performance gains. And it will show well-balanced performance in compression speed, compression ratio and decompression speed in commercial search engines. We also find that the two variants of Simple-9, i.e., Simple-16 and Simple-8*b*, can make full use of 4 status bits to present $2^4 = 16$ padding modes, and the successive versions might lead to further performance gains. Thus, future work can focus on enhancing variants of Simple-9, such as Simple-16 to improve compression ratio and Simple-8*b* to improve compression/decompression performance.

Acknowledgments

This work was supported by the China Postdoctoral Science Foundation (2016M602825).

References

[1] J. Dean, "Challenges in building large-scale information retrieval systems: Invited talk," Proc. Second ACM International Conference on Web Search and Data Mining, WSDM '09, New York, NY, USA, p.1, ACM, 2009.
[2] J.D. Brutlag, H. Hutchinson, and M. Stone, "User preference and search engine latency," Proc. ASA Joint Statistical Meetings, 2008.
[3] V.N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," Information Retrieval, vol.8, no.1, pp.151–166, 2005.
[4] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," Proc. 17th International Conference on World Wide Web, WWW '08, New York, NY, USA, pp.387–396, ACM, 2008.
[5] V.N. Anh and A. Moffat, "Index compression using 64-bit words," Software: Practice and Experience, vol.40, no.2, pp.131–147, 2010.
[6] J.P. Deveaux, A. Rau-Chaplin, and N. Zeh, "Adaptive tuple differential coding," International Conference on Database and Expert Systems Applications, DEXA 2007, pp.109–119, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
[7] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," 22nd International Conference on Data Engineering (ICDE '06), p.59, April 2006.
[8] F. Silvestri and R. Venturini, "VSEncoding: Efficient coding and fast decoding of integer lists via dynamic programming," Proc. 19th ACM International Conference on Information and Knowledge Management, CIKM '10, New York, NY, USA, pp.1219–1228, ACM, 2010.
[9] R. Delbru, S. Campinas, and G. Tummarello, "Searching web data: An entity retrieval and high-performance indexing model," Web Semantics: Science, Services and Agents on the World Wide Web, vol.10, pp.33–58, 2012.
[10] A. Trotman, X.F. Jia, and M. Crane, "Towards an efficient and effective search engine," SIGIR 2012 Workshop on Open Source Information Retrieval, 2012.
[11] G. Ottaviano, N. Tonellotto, and R. Venturini, "Optimal space-time tradeoffs for inverted indexes," Proc. Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, New York, NY, USA, pp.47–56, ACM, 2015.
[12] J. Zobel and A. Moffat, "Inverted files for text search engines," ACM Comput. Surv., vol.38, no.2, Article No. 6, July 2006.
[13] I.H. Witten, A. Moffat, and T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
[14] R. Rice and J. Plaunt, "Adaptive variable-length coding for efficient compression of spacecraft television data," IEEE Trans. Commun. Technol., vol.19, no.6, pp.889–897, 1971.
[15] P. Elias, "Universal codeword sets and representations of the integers," IEEE Trans. Inf. Theory, vol.21, no.2, pp.194–203, 1975.
[16] S. Büttcher, C. Clarke, and G.V. Cormack, Information Retrieval: Implementing and Evaluating Search Engines, MIT Press, 2016.
[17] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," Inf. Retr., vol.3, no.1, pp.25–47, July 2000.
[18] H.E. Williams and J. Zobel, "Compressing integers for fast file access," Computer Journal, vol.42, no.3, pp.193–201, 2002.
[19] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," Proc. 18th International Conference on World Wide Web, WWW '09, New York, NY, USA, pp.401–410, ACM, 2009.
[20] V.N. Anh and A. Moffat, "Index compression using fixed binary codewords," Proc. 15th Australasian Database Conference - Volume 27, ADC '04, Darlinghurst, Australia, pp.61–67, 2004.
[21] V.N. Anh and A. Moffat, "Improved word-aligned binary compression for text indexing," IEEE Trans. Knowl. Data Eng., vol.18, no.6, pp.857–861, June 2006.
[22] D. Lemire, L. Boytsov, and N. Kurz, "SIMD compression and the intersection of sorted integers," Software: Practice and Experience, vol.46, no.6, pp.723–749, 2016.
[23] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," Software: Practice and Experience, vol.45, no.1, pp.1–29, 2015.
[24] W.X. Zhao, X. Zhang, D. Lemire, D. Shan, J.-Y. Nie, H. Yan, and J.R. Wen, "A general SIMD-based approach to accelerating compression algorithms," ACM Trans. Inf. Syst., vol.33, no.3, pp.15:1–15:28, March 2015.
[25] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma, "Terrier: A high performance and scalable information retrieval platform," Proc. OSIR Workshop, pp.18–25, Citeseer, 2006.

Kun Jiang receieved his B.S., M.S., and Ph.D. degrees in Computer Science from College of Computer, National University of Defense Technology (NUDT), China, in 2008, 2011, and 2015, respectively. He is now a postdoctoral in the School of the Electronic and Information Engineering, Xi'an Jiaotong University, China. His research interests include information retrieval and data mining.

Yuexiang Yang is a Professor with College of Computer of National University of Defense Technology (NUDT), China. He received his Ph.D. degree in Computer Science from School of Computer of NUDT in 2006. His research interests include information retrieval, network security, architecture design of the Internet, and web service.

Qinghua Zheng is a Professor with School of Electronic & Information Engineering of Xi'an Jiaotong University, China. He is the winner of the National Funds for Distinguished Young Scientists and the Distinguished Professor for Yangtze River Scholar Project, is a candidate for "the New Century National Hundred Thousand-and-Ten Thousand Talents Project" and one of the first batch of leading scientists for the "Ten-Thousand Talents Project" for science and technology innovation. His major research fields include theory and technology of intelligent e-Learning environment, network public opinion and harmful information monitoring, and software reliability evaluation.