IEICE TRANS. INF. & SYST., VOL.E98–D, NO.11 NOVEMBER 2015

LETTER

Highly Compressed Lists of Integers with Dense Padding Modes

Kun JIANG[†a)], ***Student Member***, **Xingshen SONG**[†], ***and*** **Yuexiang YANG**[†], ***Nonmembers***

SUMMARY Index compression is partially responsible for the current performance achievements of Internet search engines. Among many latest compression techniques, Simple9 can pack as many integers as possible into a single 32-bit machine word using 9 different padding modes. However, the number of wasted bits in Simple9 remains large. In previous works, researchers have focused on reducing the unused trailing bits of the padding modes and have proposed various additional modes that make full use of the cases of the status bits. Instead, we focus on the wasted bits in the integer list, padding extra zeros for a complete dense mode when the number of integers is not enough to fit a complete mode. More precisely, we first propose a novel index compression method called SimpleD with dense padding modes to achieve a more compact storage compared with that of Simple9. We then design an innovative metric for extracting the inserted extra zero integers during the decoding phase. Experiments on the TREC WT2G and GOV2 datasets show that our encoder outperforms Simple9 while still retaining a very fast decompression speed.

key words: *inverted indexes, index compression, dense padding mode, compression ratio*

1. Introduction

An inverted index can be seen as an ordered list of integers, where each entry of the list corresponds to a different term or word in the dataset. For each term, the index contains an inverted list consisting of a number of postings describing all of the places where the term occurs. Postings in each list are typically sorted by docID, or sometimes by impact factor, etc. The set of terms is called the lexicon, which is relatively small in most cases. However, the inverted lists may consist of millions of postings, which could be roughly linear with the size of the dataset. In this letter, we refer to the index compression as compression on docID-sorted inverted lists [1]. 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. Instead of naively storing the raw integer in a 32-bit machine word, the main idea of index compression is to store each integer using as few bits as possible. One common practice while storing an inverted list is to use d-gaps where possible to decrease the average value that needs to be compressed, resulting in a sequence of smaller numbers with a higher compression ratio. There exists much research on index compression in the literature; see Lemire and Boytsov's [2] very recent survey.

Index encoders can be divided into integer encoders and integer list encoders. Integer encoders assign a distinct codeword to each integer. Elias-gamma, Elias-delta and variable-bytes encodings are all integer encoders [2]. These encoders are said to be bit-aligned or byte-aligned encodings because their codewords may cross the boundary of a machine word. During decoding, this requires different bitwise shift operations that slow down the decoding speed. Integer list encoders are specifically designed to compress lists of integers and may encode any of them by considering their neighbors in the list, thereby achieving higher compression or providing faster decompression. A workaround to this has been attempted by aligning each codeword to a 32-bit word boundary. Typical encoders are Binary Interpolative Coding [3], Simple9 [4] and PForDelta [5]. In this letter, we focus on improving the word-aligned encoder Simple9, which provides efficient decompression performance.

Simple9 coding [4] is a typical word-aligned encoder, where each 32-bit word stores a set of binary codes and each integer corresponds to an equal-length bit slot. While bit operations are required to unpack each word, there are no single bit accesses, and straight-line decompression remains fast. However, Simple9 has wasted bits in the equal-length bit slot due to its sparse padding modes. In this letter, we present an extension of Simple9, called SimpleD, which allows for better compression effectiveness than Simple9 and comparable decoding performance. SimpleD has the additional advantage of making full use of the equal-length bit slot. When the number of integers is not enough to fit a complete padding mode, we insert extra trailing zero integers for a complete dense data padding mode.

The rest of the letter is organized as follows: Sect. 2 gives a brief overview of the Simple9 encoder. Section 3 presents an in-depth description of our dense padding mode and the decompression metric. Section 4 reports experimental results, and Sect. 5 provides concluding remarks.

2. Simple9 Encoder

The basic idea of Simple9 is to pack as many integers as possible into a single 32-bit machine word. This is done by using the first 4 bits of a word as a status to describe the 9 possible padding modes of the remaining 28 data bits: 28 1-bit integers, 14 2-bit integers, 9 3-bit integers (one bit unused), 7 4-bit integers, 5 5-bit integers (three bits unused), 4 7-bit integers, 3 9-bit integers (one bit unused), 2 14-bit in-

Manuscript received April 28, 2015.
Manuscript revised July 5, 2015.
Manuscript publicized August 19, 2015.

[†]The authors are with College of Computer, National University of Defense Technology, China.
a) E-mail: jiangkun@nudt.edu.cn

DOI: 10.1587/transinf.2015EDL8102

tegers, or 1 28-bit integer (Table 1), with each integer (slot) having the same bit length. For example, we can store {509, 510, 511} as three 9-bit values in a word, with the highest 4 bits of the word reflecting the mode used. Decompression is accomplished by reading the value of the status bits and using a pre-computed lookup table over the status bits to select the appropriate padding mode to extract all of the integers in the remaining 28 data bits. This procedure can be optimized by hardcoding each of the 9 cases using fixed bit shift and mask operations on the data bits and using a switch operation on the status bits to select the mode.

Simple9 wastes data bits in two ways, by having only 9 cases instead of the 16 that can be expressed with 4 status bits, and by having unused bits in several of the padding modes. We note here that there exist two variants of Simple9 called Relate10 and Carryover12 that, in some cases, achieve slightly better compression than Simple9 [4]. Relative10 shrinks the selector to just 2 bits, generating new padding modes with 30 data bits for less trailing unused bits, and the padding modes can be interpreted relative to the mode value of the previous word. Carryover12 makes advantage of the trailing unused bits of the current word to hold the mode value for the next word. Another variant of Simple9 that reduces the number of unused bits is Slide [6], in which the codeword straddle 32 bits word boundaries to avoid trailing unused bits, but it exhibits higher decoding complexity. Zhang et al. [7] have proposed Simple16, a more compact encoding schema for fitting 16 different padding modes of integers and leaving no unused trailing bits within a 32-bit word. These variants can also be implemented efficiently using a switch statement and hardcoding for each mode. The results show that they match the speed of Simple9 while achieving slightly better compression.

All of these variants focus on minimizing the unused bits at the end of the modes or making full use of the cases of the status bits. There is not much consideration of the wasted bits in each slot of the padding mode itself, especially when the index of the exception is larger than the number of the padding mode. Actually, the bit length of the mode used in the list is not decided by the maximum integer but by the number of integers that can be multiplied by the bit length to fit a 32-bit word. This could lead to wasted bits in every slot. In this letter, we propose a novel dense padding metric that focuses on mode selection, which can reduce the wasted bits when coding the integer list. We believe our technique is orthogonal to the above variants, and the idea can be added to them for further improvement.

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

Status (4 bits)	Number of integers coded	Length of each integer (bits)	Wasted bits
#a	28	1	0
#b	14	2	0
#c	9	3	1
#d	7	4	0
#e	5	5	3
#f	4	7	0
#g	3	9	1
#h	2	14	0
#i	1	28	0

3. Dense Padding Modes

3.1 Padding Extra Integers

Given a sequence of integers and a padding mode, the Simple9 encoding step performs one pass over the list to check if all of the integers can be represented by the given bit length. At some point, when an exception number is superior to the maximum value that the mode can represent, we should choose the next mode with a larger bit length and a smaller number of integer slots. If the exception occurs at an index larger than the number of slots in the next mode, we know that the next mode can certainly be selected for coding, and there exist wasted bits in all of the slots in the next mode.

With a sequence of 28 integers with 27 1s and 1 potential exception value of 32, we try to compress the list using the padding modes from #a to #i. We find that it cannot be represented using mode #a. We use mode #b to represent only 14 1s and leave {1, 1, . . . , 1, 32}. Then, we use mode #c to code 9 1s and leave {1, 1, 1, 1, 32}. Next, we use mode #e to code these 5 numbers. This could require a total of 4 words to code the numbers. In fact, if there are 28 1s and one exception number 32, we need just 2 words to code the entire sequence. Why are more numbers more compact than less numbers when using Simple9 compression?

To solve this problem, we revisit the padding mode of the original Simple9 and find that there is huge gap in the number of integers coded between two adjacent modes, especially for modes with more numbers coding smaller integers. The padding mode that the exception value occurs for can be used to represent all of the integers if the numbers are more than the limit of the next padding mode. For instance, when mode #a cannot be used to code the entire 28 number sequence, the numbers that can be coded are more than the next mode #b. If we use mode #b, we use 2 bits to represent only 14 1s, leaving 1 wasted bit per slot.

If we insert 0s at the end of the integer list when the exception occurs at a position larger than the number of integers coded by the next mode, we can obtain a complete run of integers that fit the current mode with more numbers and leave the exception value for the next 32-bit word. With the above sequence of integers, the compression step first performs one pass over the list with mode #a to test if all of the integers can be represented by 1 bit. All of the integers satisfy the condition except for the last one. We know that the number of integers that satisfy mode #a (27) is larger than the number of integers coded by the next mode #b (14). We insert a 0 at the right of the 27 1s, making a complete run of integers for padding mode #a. For the remaining integer, 32, we use another 32-bit word to code it. Overall, the length of the codeword is 64 bits. Had a Simple9 code been used, a total of 128 output bits would have been generated.

Algorithm 1 is the enhanced SimpleD encoder, which

Algorithm 1 Compress (int[] d, int[] r)

Input: a sequence of numbers, d, of n integers.

```
1:  while k < n do
2:      set j = 0 and c = 0;
3:      for i from 0 to modenum[j] do
4:          if 2^bitlength[j] <= d[k + i] then
5:              if i > modenum[j + 1] then
6:                  codednum = i;
7:                  break;
8:              else
9:                  j++;
10:                 continue;
11:             end if
12:         end if
13:         c = (c << bitlength[j]) + d[k + i];
14:     end for
15:     if codednum! = modenum[j] then
16:         c <<= (modenum[j] − codednum) ∗ bitlength[j];
17:     end if
18:     set k = k + modenum[j], c = c|(j << 28) and write c to r;
19: end while
```

Output: a sequence of codewords, r.

retains the use of the current padding mode when the suited integers are more than the number of coded integers of the next mode (lines 4-7). We need to insert extra trailing zero bits for a complete current padding mode (lines 15-16). This operation is conducted by shifting the codeword to the left by a couple of bits to make the highest integer reach the left side of the 28 data bits. When the suited integers are less than the number of coded integers of the next mode, we can choose the next padding mode (lines 8-10). The lookup table 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).

3.2 Decoding of Dense Padding Modes

During the decoding phase, the most important thing is to recognize the inserted zero integers, as the zero integers have been inserted only at the end of a 32-bit word, and there do not exist any zeros in the integer sequence (the docIDs in an inverted list are different from each other). The couple of trailing zero integers we extracted with the slot bit length all belong to the ones we inserted.

Firstly, we need to count the consecutive trailing zero bits on the rightmost side of the word. For a 32-bit word v, we have attempted many methods of counting the number of trailing zero bits, and the fastest one uses the De Bruijn sequence [8], [9], as in Algorithm 2. The expression $(v\& - v)$ extracts the one least significant bit and its trailing zero bit sequence from v. The constant 0x7DCD629 is a De Bruijn sequence, which produces a unique pattern of bits into the high 5 bits (right shift for 27 bits) for each possible bit position that it is multiplied against. There are many such values constructed by taking an Eulerian cycle of an $(n-1)$−dimensional De Bruijn graph [8].

Algorithm 2 ZeroCount (v)

Input: unsigned int v.
static const int $MultiplyDeBruijnBitPosition[32] = \{0, 1, 23, 2, 29, 24, 14, 3, 30, 27, 25, 18, 20, 15, 10, 4, 31, 22, 28, 13, 26, 17, 19, 9, 21, 12, 16, 8, 11, 7, 6, 5\}$;

Output: $MultiplyDeBruijnBitPosition[(((v\&-v) * 0x7DCD629)) >> 27]$.

Secondly, as the status bits decide which padding mode has been used, we can extract the inserted extra zero integers by calculating the number of zero bits divided by the bit length of each mode. Given a 32-bit codeword of 0011, 0101, 0100, 1011, 1101, 0110, 1010, 0000, from the leading 4 status bits, we can find that padding mode #d is used for the data bits with 4 bit lengths. We then use a hardcoding mask to extract the result integers efficiently. The resulting sequence is {5, 11, 11, 13, 6, 10, 0}. The number of zero bits of the trailing codeword is 5. The inserted extra zero integer is $\lfloor 5/4 \rfloor = 1$. Finally, we can delete the trailing zero integers. That is 1 zero for the above example, and we yield six integers {5, 11, 11, 13, 6, 10} as the final result.

Algorithm 3 Decompress (int[] r, int[] d)

Input: one codeword, w, from r.

```
1: unsigned int status = w >>> 28;
2: compute the bit length l from the status bits;
3: switch (status)
4:       shift and read integers of equal length l store them to r;
5: unsigned int z = ZeroCount(w);
6: delete the right ⌊z/l⌋ integers from r;
```

Output: add the sequence of integers, r, to the integer list, d.

Algorithm 3 describes the decompression process for a sequence of codewords. The hardcoding part of shifting for each integer is reduced (line 4). The decoded integers are truncated after the normal decoding phase (line 6). The possible drawback of the SimpleD encoder is its slightly increased complexity for the zero counting.

4. Experimental Results

Index encoders are usually evaluated in terms of compression ratio and decompression speed. In contrast, compression speed is somewhat less critical. As our dense padding technique is orthogonal to other variants of Simple9, we only use Simple9 as the baseline and leave integration and comparison of our dense padding modes with other variants as future work. All our implementation code is written in Java on the Terrier 4.0 IR platform [10] and is available at http://github.com/deeper2/SimpleD. Our experiments were performed on a dedicated, otherwise idle, Intel(r) Xeon(r) E5-2640 processor running at 2.00 GHz with 128 GB of RAM and 20 MB of L3 cache.

We use inverted lists obtained from TREC WT2G and TREC GOV2 datasets. The TREC WT2G dataset contains approximately 247 thousand documents with an uncompressed size of 2 GB, and the GOV2 dataset contains approximately 25.2 million documents with an uncompressed

Table 2 Average index size in MB of the inverted lists for the TREC WT2G and GOV2 datasets.

	WT2G	GOV2_#1	GOV2_#2	GOV2
Simple9	148	354	2,967	10,367
SimpleD	146	350	2,933	10,262

Table 3 Average query latency in milliseconds on the TREC WT2G and GOV2 indexes.

	WT2G	GOV2_#1	GOV2_#2	GOV2
Simple9	35.0	48.3	210.7	626.4
SimpleD	34.7	48.0	208.4	621.8

size of 426 GB. We sequentially select documents from the GOV2 dataset, and generate one 10GB dataset and one 100GB dataset , named GOV2_#1 and GOV2_#2 respectively. We build docID-sorted inverted indexes with 1024 docIDs per block, using the two encoders, respectively, removing the standard English stopwords, and applying Porter's English stemmer. Our inverted lists include docIDs, term frequencies, field frequencies and term positions. Table 2 shows the average index size of the inverted lists for the four TREC datasets. As seen from the table, the inverted lists size of SimpleD is reduced compared with that of Simple9 by more than 1% on the four different datasets.

Instead of comparing the raw decoding speeds of the two encoders, i.e., the number of integers decoded or encoded per millisecond or the average bits per integer, as is usually done, we decided to compare the performance directly in a real searching context, i.e., answering queries with the above WT2G and GOV2 indexes. We use topics 401-450 and topics 751-800 for querying the above 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.

Table 3 shows the average query latency results for four indexes. As seen, the query processing speeds of the two encoders are almost the same for all indexes. There is no significant performance gap between the two encoders. Furthermore, we categorize topics 751-800 by the length of the query terms, and report the performance only for the TREC GOV2 dataset due to space limitation. Table 4 shows the average query latency for different number of query terms. We can find that the average query latency of the two encoders remains almost the same with different query length. This means that the slightly increased complexity of the zero counting SimpleD can be compensated for by more compact indexes. Overall, our SimpleD encoder achieves more compact storage than the Simple9 baseline with comparable query latency, as seen in the above experimental results.

5. Conclusions and Further Work

We have proposed a new dense mode padding technique to enhance the compression ratio of Simple9. We have padded extra trailing zero integers for a complete dense data padding mode when the number of integers is not enough to fit a complete padding mode, and we have designed an innovative metric for extracting the inserted extra zero integers during the decoding phase. The experimental results on the TREC WT2G and GOV2 datasets show that our proposed encoder can achieve a better compression ratio while still retaining a comparable query processing latency under real search conditions.

We believe an in-depth understanding of the list distribution can be helpful in achieving a better compression ratio if we provide detailed padding modes for common integer ranges. Additionally, the document reordering technique can be used to generate more runs of 1s, which is beneficial for our dense padding modes. Further work can be done to further enhance the padding modes on common integer list ranges and to study other variants of Simple9 for less wasted bits in the coding slots.

Table 4 Average query latency in milliseconds for different number of query terms on the TREC GOV2 indexes.

	1	2	3	4	>= 5
Simple9	59.5	367.0	647.7	895.8	1,267.4
SimpleD	57.5	379.0	652.5	905.9	1,260.8

References

[1] J. Zobel and A. Moffat, "Inverted files for text search engines," ACM computing surveys (CSUR), vol.38, no.2, p.6, 2006.
[2] 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.
[3] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," Information Retrieval, vol.3, no.1, pp.25–47, 2000.
[4] V.N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," Information Retrieval, vol.8, no.1, pp.151–166, 2005.
[5] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, pp.59–59, IEEE, 2006.
[6] 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, 2006.
[7] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," Proceedings of the 17th international conference on World Wide Web, pp.387–396, ACM, 2008.
[8] J. Thas, "A combinatorial problem," Geometriae dedicata, vol.1, no.2, pp.236–240, 1973.
[9] S.E. Anderson, "Bit twiddling hacks," URL: http://graphics. stanford. edu/˜ seander/bithacks. html, 2005.
[10] 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.