STREAM VBYTE: Faster Byte-Oriented Integer Compression

Daniel Lemire[a,*], Nathan Kurz[b], Christoph Rupp[c]

[a] *LICEF, Université du Québec, 5800 Saint-Denis, Montreal, QC, H2S 3L5 Canada*
[b] *Orinda, California USA*
[c] *Upscaledb, Munich, Germany*

Abstract

Arrays of integers are often compressed in search engines. Though there are many ways to compress integers, we are interested in the popular byte-oriented integer compression techniques (e.g., VByte or Google's VARINT-GB). Although not known for their speed, they are appealing due to their simplicity and engineering convenience. Amazon's VARINT-G8IU is one of the fastest byte-oriented compression technique published so far. It makes judicious use of the powerful single-instruction-multiple-data (SIMD) instructions available in commodity processors. To surpass VARINT-G8IU, we present STREAM VBYTE, a novel byte-oriented compression technique that separates the control stream from the encoded data. Like VARINT-G8IU, STREAM VBYTE is well suited for SIMD instructions. We show that STREAM VBYTE decoding can be up to twice as fast as VARINT-G8IU decoding over real data sets. In this sense, STREAM VBYTE establishes new speed records for byte-oriented integer compression, at times exceeding the speed of the `memcpy` function. On a 3.4 GHz Haswell processor, it decodes more than 4 billion differentially-coded integers per second from RAM to L1 cache.

Keywords: Data compression, Indexing, Vectorization, SIMD Instructions

1. Introduction

We frequently represent sets of document or row identifiers by arrays of integers. Compressing these arrays can keep the data closer to the processor and reduce bandwidth usage, and fit more data in memory or on a disk. Though data can always be compressed using generic algorithms such as Lempel-Ziv coding (LZ77), specialized compression algorithms for integers can be orders of magnitude faster. We consider codecs to compress 32-bit unsigned integers (in $[0, 2^{32})$). We are especially interested in compressing arrays where most integers are small. In addition to such arrays that arise naturally, sorted arrays of non-small integers can often be treated as arrays of small integers by considering the successive differences ("deltas"). So instead of compressing the integers $x_1, x_2, \ldots$ directly, we can compress the gaps between them

*Corresponding author. Tel.: 00+1+514 843-2015 ext. 2835; fax: 00+1+800 665-4333.

$x_1, x_2 - x_1, x_3 - x_2, \ldots$ There are many integer-compression algorithms applicable to arrays of small integers. One option are byte-oriented techniques [1]. In these formats, the main data corresponding to an integer is stored in consecutive whole bytes, and all bits within a given byte correspond to only one such integer. Though byte-oriented formats do not offer the best compression ratios or the best speeds, they are in widespread use within databases, search engines and data protocols in part because of their simplicity.

Historically, the most widely known byte-oriented compression algorithm is VByte. It writes a non-negative integer starting from the least significant bits, using seven bits in each byte, with the most significant bit set to zero when the following byte continues the current integer. Thus integers in $[0, 2^7)$ are coded using a single byte, integers in $[2^7, 2^{14})$ in two bytes and so on. For example, the integer 32 is written using a single byte (**0**0100000) and the integer 128 is written using two bytes (**1**0000000 and **0**0000001).

To decode VByte data, it suffices to iterate over the compressed bytes while checking the value of the most significant bit. Whenever the number of bytes required per integer is easily predictable—such as when most integers fit in $[0, 2^7)$—the absence of branch prediction errors allows high decoding speeds on modern-day superscalar processors capable of speculative execution.

Unfortunately, not all data is easily predictable. In some of these case, the decoding speed of VByte compressed data becomes a bottleneck. For this reason, Google developed VARINT-GB [2]. In VARINT-GB, numbers are compressed and decompressed in blocks of four. Instead of having to check one bit per compressed byte, only one control byte for every four integers needs to be processed. For unpredictable patterns, this block-based design can reduce the number of branch mispredictions by a factor of four.

Modern processors (e.g., ARM, POWER, Intel, AMD) have instructions that perform the same operation on multiple scalar values (e.g., the addition of two sets of four 32-bit integers). These instructions are said to be *single-instruction-multiple-data* (SIMD) while the algorithms designed to take advantage of SIMD instructions are said to be *vectorized*.

Stepanov et al. [1] considered the problem of vectorizing byte-oriented compression. They reported that there was little benefit to vectorizing VByte decoding. As an alternative to VByte and VARINT-GB, Stepanov et al. proposed a new patented byte-oriented format (VARINT-G8IU) designed with SIMD instructions in mind. It proved more than three times faster than VByte on realistic sets of document identifiers.

Plaisance et al. [3] revisted the VByte decoding problem: unlike Stepanov et al., their MASKED VBYTE decoder is twice as fast as a scalar VByte decoder. Though it should not be expected to be as fast as a decoder working on a format designed for SIMD instructions (e.g., VARINT-G8IU), it can help in systems where the data format is fixed.

There remained an open question: could VARINT-G8IU be surpassed? Could byte-oriented decoding be even faster? We answer this question by the positive. In fact, our proposal (STREAM VBYTE) can be twice as fast as VARINT-G8IU on realistic data.

2. SIMD Instructions

Since the introduction of the Pentium 4 in 2001, x64 processors have had vector instructions operating on 16-byte SIMD registers (called XMM registers). These registers can store four 32-bit integers.

A STREAM VBYTE decoder (§ 4) can be written with just two x64 SIMD assembly instructions:

- The `movdqu` instruction can load 16 bytes from memory into a SIMD register, or write such a register to memory. On recent Intel processors (e.g., Haswell) these operations have multicycle latency (5–6 cycles from L1 cache), but they also have high throughput. In ideal cases, two XMM loads and an XMM write can all be issued each CPU cycle.
- The shuffle (`pshufb`) instruction can selectively copy the byte values of one SIMD register v to another according to a mask m. If $v_0, v_1, \ldots, v_{15}$ are the values of the 16 individual bytes in v, and $m_0, m_1, \ldots, m_{15}$ are the bytes within m ($m_i \in \{-1, 0, 1, 2, \ldots, 15\}$), then `pshufb` outputs $(v_{m_0}, v_{m_1}, \ldots, v_{m_{15}})$ where $v_{-1} \equiv 0$. Once its operands are in registers, the `pshufb` instruction is fast: it has a latency of one cycle and a reciprocal throughput of one instruction per cycle.

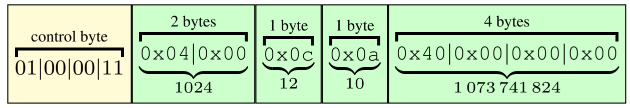
Both of these SIMD instructions are available in all processors supporting the SSSE3 instruction set, i.e., almost all x64 processors produced since 2010.

Differential Coding. When we have compressed "deltas", or successive differences $\delta_1, \delta_2, \ldots = x_1, x_2 - x_1, x_3 - x_2, \ldots$ instead of the original integers, we need to compute a prefix sum to recover the original integers $(\delta_1, \delta_1 + \delta_2, \delta_1 + \delta_2 + \delta_3, \ldots)$. The computation of the prefix sum can be accelerated by vector shifts and additions. We can compute the prefix sum of a vector of size 2^L using L shifts and L additions [4]. For example, consider the vector of delta values $(3, 4, 12, 1)$. We add to this vector a version of itself shifted by one integer ($(0, 3, 4, 12)$) to get $(3, 7, 16, 13)$. Finally, we add to this last vector a version of itself shifted by two integers $(0, 0, 3, 7)$ to get the prefix sum $(3, 7, 19, 20)$ of the original vector.

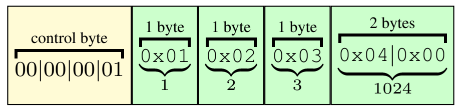
3. Byte-Oriented Integer Codecs

The VByte format uses the most significant bit of each byte as a control bit. Thus an integer is compressed to an array of $L+1$ bytes $b_0, b_1, \ldots, b_L$ such that $b_0, b_1, \ldots, b_{L-1}$ have their most significant bits set to 1 whereas the most significant bit of b_L is zero. That is, if we view bytes as unsigned integers in $[0, 2^8)$, we have that $b_0, b_1, \ldots, b_L \in [2^7, 2^8)$ and $b_L \in [0, 2^7)$. We can decode the compressed integer as $\sum_{i=0}^{L} (b_i \bmod 2^7) \times 2^{7i}$. Integers in $[2^{7(L-1)}, 2^{7L})$ are coded using L bytes for $L = 1, 2, \ldots$ The software to decode the VByte format is easy to write in standard C, but performance on inputs with mixed byte lengths L can suffer due to poor branch prediction.

For greater speed, Plaisance et al. [3] proposed the MASKED VBYTE decoder that uses SIMD instructions. It works directly on the standard VByte format. The MASKED VBYTE decoder gathers the most significant bits of an array of consecutive bytes using the `pmovmskb` x64/SSE2 instruction. Using look-up tables and a shuffle instruction



(a) Integer values 1024, 12, 10, 1 073 741 824 compressed using 9 bytes



(b) Integer values 1, 2, 3, 1024 using 6 bytes

Figure 1: Examples of blocks of four integers compressed with VARINT-GB.

(`pshufb`), MASKED VBYTE permutes the bytes to arrive at the decoded integers. We refer the interested reader to Plaisance et al. [3] for a detailed description.

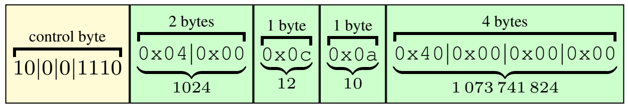
Whereas the VByte format interleaves actual integer data (using the least significant 7 bits of each byte) with control data (using the most significant bit of each byte), VARINT-GB stores the control data using a distinct control byte corresponding to a block of four integers [2]. See Table 1 and Fig. 1. To store an integer x, we use $\lceil \log_{2^8}(x+1) \rceil$ continuous bytes, since $x \in [0, 2^{\lceil \log_{2^8}(x+1) \rceil})$ always holds. Given four integers, x_1, x_2, x_3, x_4, the control byte stores the lengths $\lceil \log_{2^8}(x_i+1) \rceil$ for each of the four integers $i = 1, 2, 3, 4$. Indeed, because there are only four possible byte lengths ($\{1, 2, 3, 4\}$), each of the four individual byte lengths can be stored using just two bits. Thus a block in VARINT-GB used to store integers x_1, x_2, x_3, x_4 contains a control byte followed by $\sum_{i=1}^{4} \lceil \log_{2^8}(x_i+1) \rceil$ bytes. When the number of integers is not divisible by four, an incomplete block might be used. For this reason, it might be necessary to record how many integers are compressed (e.g., as a format header).

To decode VARINT-GB data, the control byte is first parsed, from which we extract the four lengths. The integers x_1, x_2, x_3, x_4 are then extracted (without any branching) and we advance to the next control byte. To achieve higher speed for highly compressible data, we find it useful to include an optimized code path for when all integers are small ($x_1, x_2, x_3, x_4 \in [0, 2^8)$).

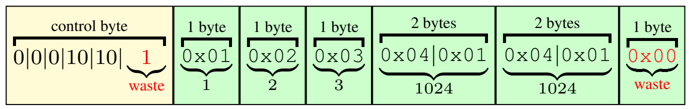
Like VARINT-GB, VARINT-G8IU also uses a control byte [1], but it does not describe a fixed number (four) of integers. Rather, VARINT-G8IU divides the compressed

Table 1: Overview of the byte-oriented integer compression techniques

name of decoder	data format	SIMD
VByte	7 data bits per byte, 1 bit as continuation flag	no
MASKED VBYTE [3]	identical to VByte	yes
VARINT-GB [2]	fixed number of integers (4) compressed to a variable number of bytes (4–16), prefixed by a control byte	no
VARINT-G8IU [1]	fixed number of compressed bytes (8) for a variable number of integers (2–8), prefixed by a control byte	yes
STREAM VBYTE (novel)	control bytes and data bytes in separate streams	yes



(a) Integer values 1024, 12, 10, 1 073 741 824 compressed using 9 bytes



(b) Integer values 1, 2, 3, 1024, 1024 compressed using 9 bytes

Figure 2: Examples of blocks of 9 compressed bytes representing different numbers of integers in the VARINT-G8IU format

output into blocks of eight data bytes for each control byte. Each block can contain between two and eight compressed integers. See Fig. 2. The control byte describes the next eight data bytes. Thus a compressed block in VARINT-G8IU is always made of exactly nine compressed bytes. Each bit in the control byte corresponds to one of the eight data bytes. A bit value of 0 indicates that the data byte completes a compressed integer, whereas a bit value of 1 indicates that the data byte is part of a compressed byte or is "wasted". Indeed, as shown in Fig. 2b, if the control byte ends with trailing ones, then they correspond to bytes that are wasted (they do not correspond to a compressed integer). A case with waste like the example of Fig. 2b might arise if the next integer to be coded is larger or equal to 2^8 so that it cannot fit in the remaining byte.

VARINT-G8IU can sometimes have worse compression than VByte due to the wasted bytes. However, VARINT-G8IU can compress a stream of integers in $[2^7, 2^8)$ using only nine bits per integer on average, against 16 bytes for VByte, so VARINT-G8IU can also, in principle, compress better. VARINT-GB has a slight size disadvantage compared to VARINT-G8IU when compressing streams of integers in $[0, 2^8)$ as it uses ten bits per integer against only nine bits per integer for VARINT-G8IU. However, this is offset by VARINT-GB's better compression ratio for larger integers.

4. The STREAM VBYTE format

Stepanov et al. [1] attempted to accelerate VARINT-GB by using SIMD instructions. The results were far inferior to VARINT-G8IU—despite the apparent similarity between the formats, with each having one control byte followed by some data.

To understand why it might be difficult to accelerate the decoding of data compressed in the VARINT-GB format compared to the VARINT-G8IU format, consider that we cannot decode faster than we can access the control bytes. In VARINT-G8IU, the control bytes are conveniently always located nine compressed bytes apart. Thus while a control byte is being processed, or even before, our superscalar processor can load and start processing upcoming control bytes, as their locations are predictable. Instructions depending on these control bytes can be reordered by the processor for best performance. However, in the VARINT-GB format, there is a strong data dependency: the location of the next control byte depends on the current control byte. This increases

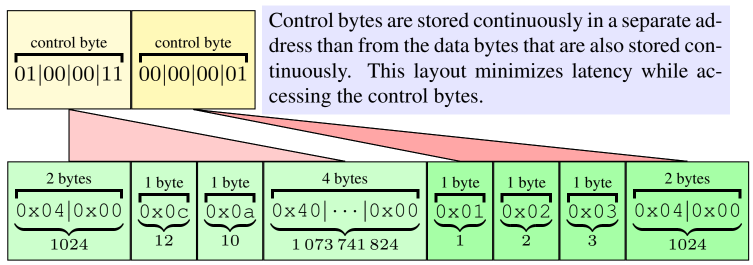


Figure 3: Compressed STREAM VBYTE bytes for $1024, 12, 10, 1\,073\,741\,824, 1, 2, 3, 1024$.

the risk that the processor remains underutilized, delayed by the latency between issuing the load for the next control byte and waiting for it to be ready.

But the VARINT-G8IU format has its own downside: it decodes a variable number of integers per control bytes (between two and eight inclusively). We expect that it is faster to store full SIMD registers (e.g., four integers) to memory with each iteration. Moreover, when using differential coding, it is more convenient and efficient to reconstruct the original integers from their deltas when they come in full SIMD registers.

Thankfully, we can combine the benefits of the VARINT-GB and the VARINT-G8IU formats: (1) having control bytes at predictable locations so that the processor can access series of control bytes simultaneously, without data dependency and (2) decoding integers in full SIMD registers. We use a format that is identical to that of VARINT-GB with the small, but important, difference that the control bytes are stored continuously in a separate stream from the data bytes. See Fig. 3. Since we record how many integers N are compressed, we can use the first $\lceil 2N/8 \rceil$ compressed bytes to store the control bytes followed by the data bytes.

Decoding a block requires no more than a handful of lines of code (see Fig. 4). At all times, we maintain a pointer into the stream of control bytes and a pointer into the stream of data bytes. They are initialized to the respective beginning of their streams.

- We start by retrieving the control byte. Given the control byte, we load from a 256-integer look-up table the number C of corresponding compressed bytes ($C \in [4, 16]$). See Fig. 4, line 3. For example, if the control byte is made of only zeros, then the sought-after length is four.
- Simultaneously, we load the next 16 data bytes in a 16-byte SIMD register. See Fig. 4, line 4. Depending on the control byte, we use only C of these bytes.
- From the control byte, we load a 16-byte shuffling mask for the `pshufb` instruction

```
1 // "databytes" is a byte pointer to compressed data
2 // "control" contains control byte
3 uint8_t C = lengthTable[control]; // C is between 4 and 16
4 __m128i Data = _mm_loadu_si128((__m128i *) databytes);
5 __m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
6 Data = _mm_shuffle_epi8(Data, Shuf);  // final decoded data
7 datasource += C;
```

Figure 4: Core of the STREAM VBYTE decoding procedure in C with Intel intrinsics

(line 5). There are 256 such masks, one for each possible value of the control byte.

- We apply the `pshufb` instruction on the data SIMD register using the shuffling mask (line 6). The result contains the uncompressed integers. If differential coding is used, the result contains four deltas which can be decoded (§ 2).
- Both pointers are advanced. The control-byte pointer is advanced by one byte whereas the data-byte pointer is advanced by C bytes (line 7).

Incomplete blocks (containing fewer than four integers) are decoded using a scalar function similar to that used for VARINT-GB. Likewise, when we detect that fewer than 16 data bytes remain, we use a scalar function. Further, we found it useful to use an optimized code path when we have four zero control bytes in sequence.

When the data is highly compressible, the STREAM VBYTE format stores long runs of control bytes set to zero. In some applications, it might be beneficial to compress such runs to improve compression (e.g., using run-length encoding).

Though details are outside our scope, we have implemented fast functions to append new integers to a compressed STREAM VBYTE array without having to recompress the data. We append extra data, and occasionally grow the control bytes stream.

5. Experiments

We implemented our compression software in C. We use the GNU GCC 4.8 compiler with the `-O3` flag. To ease reproducibility, our software is available online.[1]

We run the benchmark program on a Linux server with an Intel i7-4770 processor (3.4 GHz). This Haswell processor has 32 kB of L1 data cache and 256 kB of L2 cache per core with 8 MB of shared L3 cache. The machine has ample memory (32 GB of dual-channel DDR3 1600 RAM). Turbo Boost and Speed Step are disabled, so the processor runs consistently at its rated speed of 3.4 GHz. We measure wall-clock times to decompress data from RAM to L1 cache. All tests are single-threaded.

Search engines typically rely on posting lists: given a term, we create a list of all document identifiers corresponding to documents where the term appears. Instead of directly storing the identifiers, we use differential coding to reduce the average number of bits required to store each compressed integer. The document identifiers are sorted in increasing order ($x_1, x_2, \ldots$ where $x_i > x_{i-1}$ for $i = 2, 3, \ldots$), and we compress their successive differences (e.g., $\delta_1 = x_1 - 0, \delta_2 = x_2 - x_1, \ldots$). To recover the original identifiers during decompression, we need to compute a prefix sum ($x_i = \delta_i + x_{i-1}$). When using vectorized formats, we vectorize the computation of the prefix sum. All decoding times include differential coding to reconstruct the original integers.

We use a collection of posting lists extracted from the ClueWeb09 (Category B) data set. ClueWeb09 includes 50 million web pages. We have one posting list for each of the 1 million most frequent words—after excluding stop words and applying lemmatization. Documents are sorted lexicographically based on their URL prior to attributing document identifiers.[2]

---

[1] `https://goo.gl/6Op1t4` and `https://github.com/lemire/streamvbyte`

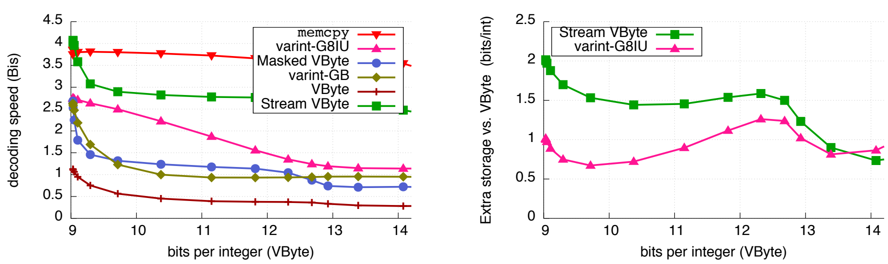
[2] The posting-list data is freely available (`http://goo.gl/DygoQM`).

We want to sort the posting lists by compressibility. For this purpose, the posting lists are grouped based on length: we store and process lists of lengths 2^K to $2^{K+1}-1$ together for all values of K needed by the corpus. While the correlation is not perfect, shorter lists tend to be less compressible than longer lists since their gaps tend to be larger.

We decode the compressed data sequentially to a buffer of 4096 32-bit integers (half the size of the $32\,\mathrm{kB}$ L1 cache). For each group and each decoder, we compute the average decoding speed in billions of 32-bit integers per second (Bis). So that disk access is not an issue, the data is loaded in memory prior to processing. The length of the arrays, their compressibility and the volume of data varies from group to group.

Our results are summarized in Fig. 5a. For STREAM VBYTE, the reported speed ranges from 4.0 Bis for highly compressible data to 1.1 Bis for less compressible data. The second fastest codec is VARINT-G8IU with speeds ranging from 2.7 Bis to 1.1 Bis. Next we have MASKED VBYTE and VARINT-GB with speeds ranging from 2.6 Bis to 0.5 Bis. Though VARINT-GB is more than $50\,\%$ faster than MASKED VBYTE when the data is poorly compressible, they are almost tied speed-wise when the data is more compressible. Finally, we find VByte with speed ranging from 1.1 Bis to 0.3 Bis. For all groups of posting lists, STREAM VBYTE is fastest of the algorithms considered. It is always at least $\approx 2.5\times$ faster than the conventional VByte decoder and sometimes nearly $5\times$ faster. Compared to VARINT-G8IU, STREAM VBYTE can be twice as fast. For reference, we also provide the copy speed of the uncompressed data by blocks of up to 4096 integers using the C function `memcpy`. For highly compressible data, STREAM VBYTE is faster than `memcpy` because fewer bytes are read. In our worst case, STREAM VBYTE decompresses at $70\,\%$ of the corresponding `memcpy` speed.

Though they are faster, both STREAM VBYTE and VARINT-G8IU use .5 to 2 more bits per integer than VByte in these tests: see Fig. 5b. STREAM VBYTE and VARINT-GB have almost exactly the same storage requirements due to their similar format. Also MASKED VBYTE and VByte have exactly the same compressed format.

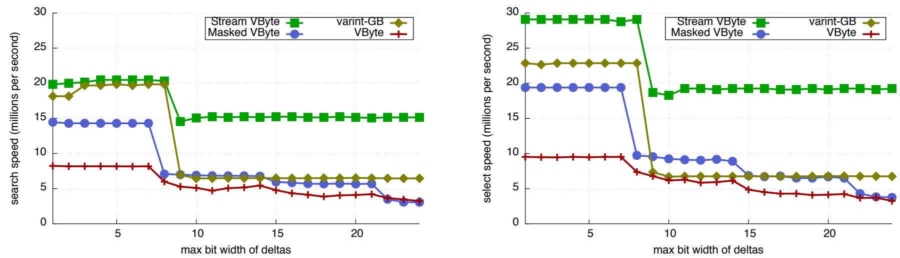


(a) Decoding speed in billions of integers per second (Bis) versus the compressibility of the data (in bits per integer for VByte)

(b) Extra storage space in bits per integer vs. the storage requirement of VByte

Figure 5: Results over sets of posting lists (ClueWeb)

Byte-oriented codecs make it convenient to program advanced operations directly on the compressed stream. We benchmark two such operations: (1) we seek the location of the first value greater or equal to a given target, and retrieve this value, and



(a) Seek speed

(b) Select speed

Figure 6: Seek and select results over blocks of 256 random integers

(2) we select the i^{th} integer. Given a bit width $b \leq 24$, we first generate an array of 256 random integers in $[0, 2^b)$: $\delta_1, \delta_2, \ldots$. The prefix sum is computed $(\delta_1, \delta_1 + \delta_2, \ldots)$ and used as input data. We omit VARINT-G8IU in this experiment. In Fig. 6a, we randomly seek a value in range. In Fig. 6b, we randomly select the value at one of the indexes. In these tests, STREAM VBYTE offers the best speed and is up to three times faster than VByte, with intermediate results for other codecs. The performance is noticeably better when all deltas compress down to a single byte due the simple and predictable code path: it happens when all deltas fit in 7 bits for VByte and MASKED VBYTE, and when they all fit in 8 bits for VARINT-GB and STREAM VBYTE.

Acknowledgments

This work is supported by the National Research Council of Canada, under grant 26143. We thank L. Boystov from CMU for sharing the posting list collection.

References

[1] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, P. S. Oberoi, SIMD-based decoding of posting lists, in: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11, ACM, New York, NY, USA, 2011, pp. 317–326.

[2] J. Dean, Challenges in building large-scale information retrieval systems: invited talk, in: Proceedings of the Second ACM International Conference on Web Search and Data Mining, WSDM '09, ACM, New York, NY, USA, 2009, pp. 1–1.

[3] J. Plaisance, N. Kurz, D. Lemire, Vectorized VByte Decoding, in: iSWAG '15, pp. 1–6. `http://arxiv.org/abs/1503.07387`.

[4] D. Lemire, L. Boytsov, N. Kurz, SIMD compression and the intersection of sorted integers, Softw. Pract. Exp. 46 (2016) 723–749.