MSc thesis

Computer Science

Select-based random access to variable-byte encodings

Jussi Timonen

March 5, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. D.U. Mind, Dr. O. Why

Examiner(s)

Prof. D.U. Mind, Dr. O. Why

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Jussi Timonen

Työn nimi — Arbetets titel — Title

Select-based random access to variable-byte encodings

Ohjaajat — Handledare — Supervisors

Prof. D.U. Mind, Dr. O. Why

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	March 5, 2020	20 pages, 21 appendice pages

Tiivistelmä — Referat — Abstract

Use this otherlanguage environment to write your abstract in another language if needed.

Topics are classified according to the ACM Computing Classification System (CCS), see https://www.acm.org/about-acm/class: check command `\classification{}`. A small set of paths (1–3) should be used, starting from any top nodes referred to bu the root term CCS leading to the leaf nodes. The elements in the path are separated by right arrow, and emphasis of each element individually can be indicated by the use of bold face for high importance or italics for intermediate level. The combination of individual boldface terms may give the reader additional insight.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

ulkoasu, tiivistelmä, lähdeluettelo

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms specialisation line

Contents

1 Introduction

Enormous datasets are a common case in today's applications. Compressing the datasets is beneficial, because they naturally decrease memory requirements but also are faster when compressed data is read from disk (Zobel and Moffat, 1995).

One of the leading methods of data compression is variable-length coding (Salomon, 2007), where frequent sequences of data are represented with shorter codewords. Because the sequences of data have different lengths when compressed, it is not trivial to determine the exact location of a certain element in the compressed data. If this is required, the usual data compression algorithms are inefficient. Fortunately this is not a requirement compression algorithms usually need to fulfill. However, random access of compressed data is useful in compressed data structures. It saves storage space, bandwidth and increases the likelihood of data already being in cache (Scholer et al., 2002). In most compression methods, the only way to do this is to decompress data from the beginning.

A variable-byte encoding based integer compression method with constant time random access was first introduced by (Brisaboa et al., 2009). They used clever block reorganizing and *rank* data structure to achieve random access. Their solution is currently the only published solution for the problem and it has been widely adopted.

An alternative solution with *select* data structure is proposed and explained in detail later in this thesis. Comparison to *rank* implementation by Brisaboa et al is shown with different implementations of *rank* and *select* and several different data sets. Proposed method does not use data reorganizing, but rather capitalizes on the assumption that data is encoded the same way it is in the source. Because variable-byte encoding does not assign new codewords to data when encoding, data can be read straight from the memory and reassembling the integer from blocks is not needed.

Rank method is superior when dataset contains mostly small integers. With larger integers, multiple *rank* calls are needed and *select* method performs better. Additionally, the proposed *select* method offers quick access to next or previous element.

2 Variable-bytes

Variable-byte (VB) encoding (Williams and Zobel, 1999) is a method for compressing unsigned integers via omitting leading zero bits that would be present in a longer fixed width word. In normal data sets, VB encoding loses in compression performance to generic algorithms like Huffman encoding or Lempel-Ziv encoding, but is generally faster to decode (Brisaboa et al., 2009) and sometimes preferred due to its simplicity.

A good data set for VB encoding is a list of mostly small integers with a need to support larger ones. A search engine may use an inverted index of words in documents. For each word, a list of document IDs where the word appears is stored. It may also store locations of the word in document for advanced search purposes. Usually these lists are preprocessed and stored as an inverted index or gaps, storing the difference to previous number instead of the actual number (Manning et al., 2008). Common words have a lot of entries in these lists, but because of gap storing the integers are small. In contrast, rare words have only a few entries but the integers stored are larger. These lists are excellent data sets for VB encoding.

Variable-byte encoding originates from and is used in MIDI music file format (MIDI Manufacturers Association and others, 1996) and several applications have a similar implementation of VB. Apache Lucene has vInt datatype. Wireless Application Protocol has a variable length unsigned integer uintvar, Google Protocol Buffers has a Base 128 Varint (Google, 2019), Microsoft .Net framework offers "7BitEncodedInt" in BinaryReader and BinaryWriter classes and IBM DB2 has a variable byte (Bhattacharjee et al., 2009).

Later, VB encoding was to be found efficient in compressing integer lists. It was first experimented as a tool for compressing inverted index lists of word locations in documents (Scholer et al., 2002). That yielded excellent records, and since then many different approaches have been introduced.

More recent studies have taken a look into the machine code level for VB and applied SIMD (Single instruction, multiple data) instructions to VB (Lemire et al., 2018; Plaisance et al., 2015). The bit operations in VB are simple and therefore modifying the code to use SIMD instructions is straightforward and the speed improvements are significant.

Elias Delta and Gamma codes (Elias, 1975) are popular encoding methods for foremen-

Original number	first block	second block	third block	fourth block
4	10100			
17	00001	10001		
620	01100	00110	10010	
60201	11001	00010	01011	11110

Table 2.1: VByte encoded integers, block size 4. Continuation bit underlined.

tioned data sets. Their encoding process assigns short bit array codes to small integers, which is why they outperform VB encoding on datasets with a lot of small integers (Williams and Zobel, 1999).

2.1 VB encoding

VB encoding splits each integer into blocks of b bits and adds a continuation bit to the block to form chunks of length $1 + b$. The continuation bit is set to 1 only on the block containing the least significant bits of the integer. This information is used in decoding to signal if the current integer continues in the next chunk.

For example, block length is set to 4 and 42 is a 16 bit unsigned integer that is to be encoded. The standard 8-bit representation of 42 is `00000000 00101010`. This is then split to blocks of 4 bits, resulting in `0000 0000 0010 1010`. The block with least significant bits is given a continuation bit 1. The block with the next least significant bits is given a continuation bit 0. The rest of the blocks are omitted, because they all are empty. The result of this is `00010 11010`, which is the number 42 compressed with VB encoding and block length of 4. A pseudo code with block length 4 is shown in Figure 2.1. Prepend adds an element to the beginning of the list and extend adds all the elements of the second list to the end of the first list.

Smaller block length can yield a better compression rate at the cost of more bit manipulation, while bigger block lenghts need less bit manipulation and offer less effective compression. On the other hand, bigger block length means a smaller percentage of added continuation bits. Generally block length of 7 has been used because it tends to perform well on average and handling chunks as bytes is convenient (Manning et al., 2008).

```
function VBEncodeNumber(n)
    bytes ← list
    while true do
        prepend(bytes,n mod 128)
        if n < 128 then
            break
        n ← n div 128
    bytes[len(bytes)-1] += 128
    return bytes

function VBEncode(numbers)
    bytestream ← list
    for each n ∈ numbers do
        bytes ← VBEncodeNumber(n)
        extend(bytestream,bytes)
    return bytestream
```

Figure 2.1: VByte encoding

2.2 VB decoding

VB decoding reverses the encoding steps: encoded chunks are read until a chunk with 1 as continuation bit is found. Continuation bits are removed from all the chunks and the blocks are concatenated to form the original integer. A pseudo code implementation of VB decoding with block length of 7 is shown in Figure 2.2. Append adds an element to the end of the list. If block length is changed, additional bit operation steps when reading the data may be needed.

As how the encoding example ended, the encoded message was `00010 11010` with block length of 4 and the goal is to decode a 16 bit unsigned integer. The block from the first chunk is extracted and added to n, making $n =$`10` (bit representation of 2). The continuation bit was 0 in this chunk, which means the encoded integer continues to the next block. A bitwise shift to the left equal to block length is applied to n, changing $n =$`100000` (bit representation of 32). Then the chunk reading process is repeated. The block of the next chunk is added to n, making $n =$`101010` (bit representation of 42). The continuation bit of this chunk is 1, which means the decoding for this number is complete.

```
function VBDecode(bytestream)
    numbers ← list
    n ← 0
    for each b ∈ bytestream do
        if b < 128 then
            n ← 128 × n + b
        else
            n ← 128 × n + b - 128
            append(numbers,n)
            n ← 0
    return numbers
```

Figure 2.2: VByte decoding

3 Directly addressable codes

The ability to handle large amounts of data fast is one of the key challenges in the field of search engines. Compressed data structures are applied to fit the data into cache, memory or even hard drive. Direct access to any element in a compressed list or array is one of the usual requirements in compressed data structures. It is not natively possible to decode the i-th element in variable-byte compression algorithms, because the position of the element in the compressed list depends on the length of the preceding compressed data. Direct access is achievable with supporting data structures. A naive solution is to store the location of each element to an array, but it adds a very large overhead which removes the benefit of compression.

3.1 Rank and Select

Rank and select are two succinct data structure operations which operate on a bit array B. $Rank_1$(B, i) gives the sum of 1 bits between B[0] and B[i] and $select_1$(B, i) gives the index of i-th 1 bit in B. Both operations can be implemented to work in constant time (Gog et al., 2014). To use rank or select in indexing, the set 1 bits in B should reflect to the element locations in the encoded data.

For most compression algorithms, this requires B to be created in addition to the existing data and the length of B usually has to be close to the length of the data, because encoded element lenghts vary. VB encoding has several advantages with search and rank: the data is compressed in blocks of equal length which significantly shrinks the length of B. Moreover the bit array C formed from the continuation bits already stores the locations of items and works as the needed indexing array. In this case, $rank_1$(C, i) would give the sum of end blocks up to i-th index and $select_1$(C, i) would give the location of the ending block of i-th compressed element.

Table 3.1: Memory requirements of SDSL rank and select data structures

Structure	Extra size taken
Rank	25% of bit array
Rankv5	6.25% of bit array
Select	8-12% of bit array

3.2 Rank and Select implementation

The rank and select implementations used in this thesis are from C++ library 'SDSL-lite' by (Gog et al., 2014). The library has an implementation of a bit array and several implementations of both rank and select to support the bit array. Also a few useful functions were used during the implementation phase. Table 3.1 has *rank* and *select* size requirements of implementations used in this experiment. Both rank implementations have a constant space requirement over the bit array, while select's needed size depends on the number of 1's in the data.

The data structure of rank has two layers. First layer is the superblock array. For every 512th bit, the number of 1's from the beginning of the array is stored to the superblock array. The second layer is the relative count block. It stores a relative count of 1's for every 64th bit. To calculate rank(i), the superblock index is s = i/512 and relative count block index is r = i-s / 64, both being integer divisions. Then number of 1 bits from index r to i are calculated. All three values added together add up to rank(i). Rankv5 is a lighter structure: it's superblock size is 2048 and relative counts are taken for every 384th bit. This causes the final bit calculation to be more costly, but reduces memory requirement to a quarter.

Select data structure works similarly to rank. The index location of every 4096th set bit is stored in the superblock and location of every 64th set bit is stored relative to the superblock. With similar calculations, the location of closest 64th set bit is calculated and then bit array is iterated until required amount of bits is reached.

In both cases, one function call always gets a value from the superblock and then from the superblock's relative block. The only differiating factor is the manual bit count from relative block's index to wanted index. This is at most the size of one relative block and thus both of the functions work in constant time (González et al., 2005).

3.3 DAC via rank

Directly addressable codes in VB was first introduced by (Brisaboa et al., 2009). In their solution, the chunks are sorted in separate arrays by their significance. For each integer, the block of the least significant chunk is stored in the first array A_1, and its bit to the first bit array B_1. Then if the number was stored in multiple chunks, the block of the next chunk is stored to A_2 and the bit to B_2 and so on. After the data is set, rank data structure is built for each bit array. The data structure is implicit and does not require additional space apart from the *rank* structure. Figure 3.1 contains a visualization of the data structure.

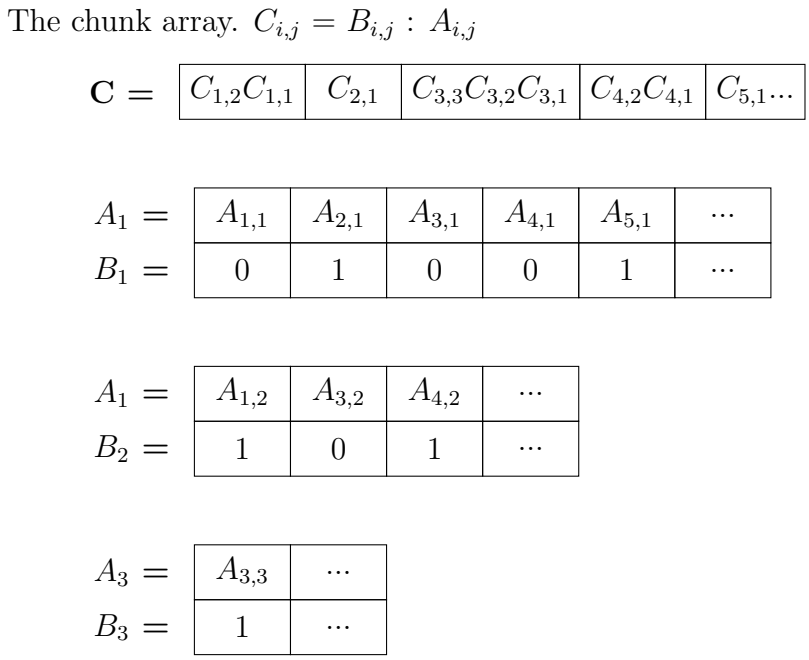

Figure 3.1: Data structure by Brisaboa et al., visualized

To decode an element from index i, first block is fetched from A_1[i]. Because each encoded element is composed of at least one block, the first block is obtainable via straight indexing. With small integers, this is all that is required. If the data has a lot of small integers, the *rank* method has a huge advantage.

However if the element was stored in multiple blocks, the index of the next block in A_2 is obtained from $i \leftarrow rank_0(A_1, \text{i})$. $Rank_0$ returns the number of zeros in the bit array preceding index i. In other words, the number of elements before i that continue to the next array. This in turn means that i now has the index of the element's next block in array A_2. This process is repeated until the i-th bit of the bit array of current level is set.

The resulting integer is constructed from the blocks of the fetched chunks. A pseudo code example of DAC with *rank* is shown in Figure 3.2 with block length of 8.

$index \leftarrow$ wanted index
$A \leftarrow$ block arrays
$B \leftarrow$ continuation bit arrays
$level \leftarrow 0$
$number \leftarrow 0$
while $B[level][index] = 0$ **do**
    $block \leftarrow A[level][index]$
    $number \leftarrow number \ll 8$
    $number \leftarrow number + block$
    $index \leftarrow \textsc{Rank}(B[level], index)$
    $level \leftarrow level + 1$
$block \leftarrow A[level][index]$
$number \leftarrow number \ll 8$
$number \leftarrow number + block$

Figure 3.2: Example pseudo code of DAC with rank by Brisaboa et al.

–TODO: create another version: 7bit bris. Store bits in both bit array and chunk (to see if it's faster to get the continuation bit from the chunk than from the bit array) (also needs 7bit compliant dataset)

4 DAC with select query

Using $select_1$ on the continuation bit array to achieve direct access is more intuitive than using $rank_1$. The element locations are already marked with 1's in B and a single $select_1$ query gets the desired starting point, while the forementioned version (Brisaboa et al., 2009) used one *rank* query for each chunk beyond the first. Minimizing the amount of *select* and *rank* queries is important. They run in constant time but their impact is huge, because rest of the VB decoding consists of a few bit operations.

To use $select_1$ with VB, continuation bits need to be separated from chunks to their own bit array and a $select_1$ structure built over it. Because every compressed element has 1 only on it's last block, $Select_1$(i) returns the location of the end byte of i-th element. Therefore the start of j-th element in the block array is at block $b_s = select_1$(j-1) + 1. Implementation was simplified by using only block sizes 4 and 8 to prevent block splitting between bytes.

Unlike the standard VB encoding, the continuation bits are removed from the chunks and stored in their own array, which leaves the blocks in their own array. This allows the compressed number to be read from the memory block and removes the need of block concatenation. The data in blocks is written into memory as they appear in the original integer, so that when reading a word from the block byte array, the bits and bytes are already in correct order.

For example, let the integer bit length be 32 and the original data be compressed to 5 blocks of 4 bits. First, the byte location of the first block is needed. $Select_1$(i) gets the index location of needed block. Because the block length differs from byte length, the the two values need to be calculated. First, the starting byte location s needs to be calculated ($s = select_1(i) \times 4) \div 8$, the latter operand being the integer division. Second, the first block might not be at the start of the byte s. This is why offset $o = select_1(i) \times 4) mod 8$ is needed (in the example case, o equals to either 0 or 4). Then a 32-bit word w is read from memory from byte location s.

At this point, w contains the wanted bits, but also has extra bits from the previous and next compressed integers. If offset was not zero, there are trailing o bits in w from the previous integer. These can be conveniently removed by bit-shifting w right for o bits. Then a pre-calculated bitmask (0xFFFFF in this case) is applied and w contains the

requested value. Because most systems are able to read memory only from byte addresses, there is a corner case where the value is stored in too many bytes for a word. In this setup, this happens if compressed length was 8 blocks and offset was 4. Then another step of reading another byte and storing the remaining bits to w is required. This is entirely avoidable if integer length is constrained. In this case maximum integer length would be 28 bits.

The most intuitive way to calculate block length of i-th element is from $select_1$(i+1) and subtract the previously calculated start block index. This however causes a second $select_1$ query, which is costly. A much faster option is to iterate forward in bit array until the next 1. Alternatively, the block length can be calculated by reading an integer from the bit array, aligning it's offset and counting the trailing zeros. Figure 4.1 contains an example of VB decoding with DAC with *select* and block size 8. Different block sizes need extra calculation to get the block location from the byte array. In the example, $CalculateLength$ returns the length of the number in blocks and $ByteMask(k)$ returns a bit mask for k bytes.

$i \leftarrow$ wanted index
$A \leftarrow$ block byte array
$B \leftarrow$ continuation bit array
$begin \leftarrow$ SELECT($index$)
$len \leftarrow$ CALCULATELENGTH($begin$)
$mask \leftarrow$ BYTEMASK(len)
$word \leftarrow A[begin]$ ▷ read a word from the byte array
$word \leftarrow word \,\&\, mask$

Figure 4.1: Pseudo code of DAC with select with block length 8

Figure 4.2 portrays an example of the bit array and the block array. $Select_1(k-1)$+1 returns the index of the starting byte of ith compressed element. Length of the compressed element is obtainable e.g. via $select_1(k)$. Then a word is read from the block array, starting from block A_i. Because element length in this case is 3, blocks not belonging to the element are removed either with a bitmask or by double bit shifting, resulting in decoded k-th element.

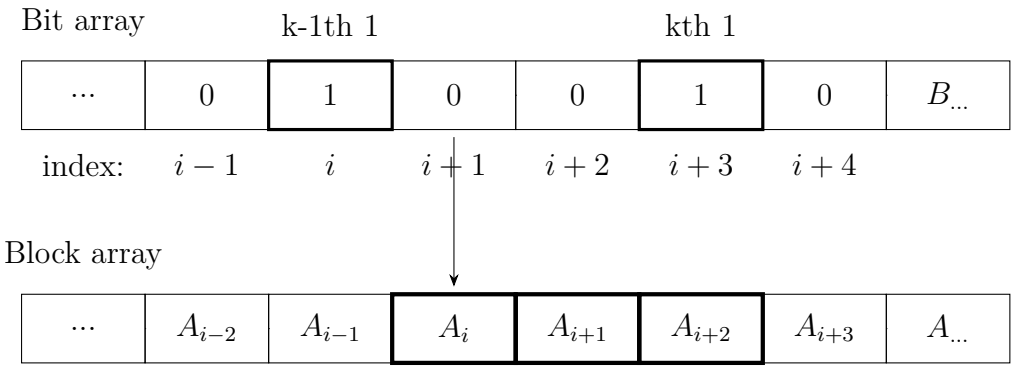

Figure 4.2: Data

5 Experimental results

The following experiments were run on a AMD Phenom(tm) II X6 1055T Processor@2.8Ghz with 64kB+64kB L1 cache, 512kB L2 cache, 6144kB L3 cache and 32GB of DDR3 1333MHz. The computer runs Ubuntu 14.04.4 (3.13.0-91-generic x86_64). The code was compiled with g++-7 -std=c++11 -DNDEBUG and with libraries -lsdsl -ldivsufsort -ldivsufsort64.

Five different VB decoding algorithms were compared. The Bris-algorithms are from (Brisaboa et al., 2009). All algorithms are implemented using data structures and functions from the SDSL library (Gog et al., 2014). Each implementation is compressing and decoding uint64_t integers.

- Bris4v5 - Bris implementation with block size 4, using rank support v5
- Bris8 - Bris implementation with block size 8, using rank support v
- Bris8v5 - Bris implementation with block size 8, using rank support v5
- Select4 - Proposed implementation with block size 4, using select support mcl
- Select8 - Proposed implementation with block size 8, using select support mcl

Four differently constructed datasets were used and three different sizes of dataset were used (5M, 50M, 500M).

- all - all integers randomly 1-4 bytes long
- twolarge - one eighth of integers 4 bytes long, one eighth 2 bytes long, rest 1 byte
- onelarge - one eighth of integers 2 bytes long, rest 4 bits or less
- onlysmall - all integers 4 bits or less

All implementations read the data set from a file and compress it to memory. Then one million index integers are randomized and stored to an array. The times shown in Table 5.1 are times taken from looping through the index array and VB decoding the number in the

Table 5.1: Results in milliseconds, smaller is better.

	bris8	bris4	bris8v5	bris4v5	select8	select4
all (5M)	191.80	471.50	234.00	569.40	149.20	175.90
all (50M)	269.20	573.80	323.00	785.00	245.40	275.00
all (500M)	298.50	677.80	368.20	941.80	355.90	382.20
twolarge (5M)	96.70	275.50	110.70	329.00	125.60	151.90
twolarge (50M)	140.40	379.70	159.80	456.80	227.70	242.20
twolarge (500M)	168.20	441.00	191.50	584.70	330.20	341.00
onelarge (5M)	25.80	40.10	28.30	49.20	95.90	100.40
onelarge (50M)	49.80	91.30	50.90	105.70	212.70	213.20
onelarge (500M)	55.00	100.00	58.50	124.00	315.80	307.70
onlysmall (5M)	16.50	20.20	16.50	23.80	84.70	88.40
onlysmall (50M)	31.20	51.80	32.10	57.00	201.20	198.80
onlysmall (500M)	34.10	56.00	35.00	63.10	297.90	293.80

index. Additionally, each run is iterated multiple times and an average of the runtimes is taken.

The results show that the proposed *select* method outperforms *rank* when the data set contains several integers that are encoded into more than one block. The difference is explained with how *rank* and *select* use the data structure. *Select* is always called once, while *rank* is called once for each block beyond the first. Bris8 performs better than Bris8v5, but also requires more memory. See Figure 3.1 for memory consumption.

8-bit versions perform better than 4-bit versions timewise, because they need less operations overall. However 4-bit versions have better compression in some data sets, because they are able to compress some integers better.

- see how data length percentage matters (do datasets with k/8 long numbers). - see how cache matters

- 4bit is naturally slower, because of smaller block size - select is better with longer numbers, because it only needs 1 select call and reads straight from memory

- See Ahn & Moffat 2005; Lemire et al. 2018 - subarray access(?)

- compare to elias-fano?

- get compression sizes of 4 and 8 bit versions

- figure out why data size matters

- do we need to support search()?

- *rank* vs rank? should formulas have a specific style?

6 Future work

- something to improve / research?

7 Conclusions

It is good to conclude with a summary of findings. You can also use separate chapter for discussion and future work. These details you can negotiate with your supervisor.

Bibliography

Bhattacharjee, B., Lim, L., Malkemus, T., Mihaila, G., Ross, K., Lau, S., McArthur, C., Toth, Z., and Sherkat, R. (2009). "Efficient index compression in DB2 LUW". In: *Proceedings of the VLDB Endowment* 2.2, pp. 1462–1473.

Brisaboa, N. R., Ladra, S., and Navarro, G. (2009). "Directly Addressable Variable-Length Codes". In: *String Processing and Information Retrieval*. Ed. by J. Karlgren, J. Tarhio, and H. Hyyrö. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 122–130. ISBN: 978-3-642-03784-9.

Elias, P. (Sept. 1975). "Universal Codeword Sets and Representations of the Integers". In: *IEEE Trans. Inf. Theor.* 21.2, 194–203. ISSN: 0018-9448. DOI: 10.1109/TIT.1975.1055349. URL: https://doi.org/10.1109/TIT.1975.1055349.

Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pp. 326–337.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. (2005). "Practical implementation of rank and select queries". In: *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pp. 27–38.

Google (2019). *Protocol Buffers Encoding*. URL: https://developers.google.com/protocol-buffers/docs/encoding#varints (visited on 02/26/2020).

Lemire, D., Kurz, N., and Rupp, C. (2018). "Stream VByte: Faster byte-oriented integer compression". In: *Information Processing Letters* 130, 1–6. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2017.09.011. URL: http://dx.doi.org/10.1016/j.ipl.2017.09.011.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*.

MIDI Manufacturers Association and others (1996). "The complete MIDI 1.0 detailed specification". In: *Los Angeles, CA, The MIDI Manufacturers Association*.

Plaisance, J., Kurz, N., and Lemire, D. (2015). *Vectorized VByte Decoding*. arXiv: 1503.07387 [cs.IR].

Salomon, D. (2007). *Variable-length codes for data compression*. Springer Science & Business Media.

Scholer, F., Williams, H. E., Yiannis, J., and Zobel, J. (2002). "Compression of inverted indexes for fast query evaluation". In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 222–229.

Williams, H. E. and Zobel, J. (1999). "Compressing Integers for Fast File Access". In: *COMPJ: The Computer Journal* 42.3, pp. 193–201.

Zobel, J. and Moffat, A. (1995). "Adding compression to a full-text retrieval system". In: *Software: Practice and Experience* 25.8, pp. 891–903.