MSc thesis

Computer Science

Select-based random access to variable-byte encodings

Jussi Timonen

May 24, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Dr. Simon Puglisi, Dr. Juha Kärkkäinen

Examiner(s)

Dr. Simon Puglisi, Dr. Juha Kärkkäinen

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

Dr. Simon Puglisi, Dr. Juha Kärkkäinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	May 24, 2020	29 pages

Tiivistelmä — Referat — Abstract

Enormous datasets are a common occurence today and compressing them is often beneficial. Fast direct access to any element in the compressed data is a requirement in the field of compressed data structures, which is not easily supported with traditional compression methods.

Variable-byte encoding is a method for compressing integers of different byte lengths. It removes unused leading bytes and adds an additional *continuation bit* to each byte to denote whether the compressed integer continues to the next byte or not. An existing solution using a *rank* data structure performs well in this given task. This thesis introduces an alternative solution using a *select* data structure and compares the two implementations. An experimentation is also done on retrieving a subarray from the compressed data structure.

The *rank* implementation performs better on data containing mostly small integers. The *select* implementation benefits on larger integers. The *select* implementation has significant advantages on subarray fetching due to how the data is compressed.

ACM Computing Classification System (CCS)
Information systems → Data management systems → Data structures → Data layout → Data compression

Avainsanat — Nyckelord — Keywords

compression, data structure

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. Some datasets are too big to handle in memory and thus must be stored in larger, slower storage (Hon et al., 2010). Compressing these datasets is often beneficial, because doing so naturally decreases memory requirements, but also can make it faster to transfer data from disk to memory (Zobel and Moffat, 1995; Williams and Zobel, 1999).

A fundamental method for data compression is variable-length coding (Salomon, 2007). The main idea of variable-length encoding is that 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 within the compressed data. If this is required, the usual data compression algorithms are inefficient, because data must be decompressed from the beginning to the point where the the sought element is located. Fortunately, such random access to elements is not a requirement compression algorithms usually need to fulfill.

However, efficient random access to compressed data is very useful in compressed data structures (Venturini, 2013). In addition to saving storage space and bandwidth, the compression increases the likelihood of data already being in cache (Scholer et al., 2002).

Variable-byte encoding is a method for compressing integers of different byte length. An additional *continuation bit* is added to the data to denote whether the compressed integer is continued to the next byte or not. To facilitate random access, two different light-weight data structures are built over the continuation bit array. $Rank_1(i)$ returns the number of 1 bits in the bit array between indexes 0 and i-1. $Select_1(i)$ returns the index of the i-th 1 bit in the array. The fact that the compressed data blocks are of the same size is used with the data structures to allow direct access to the compressed variable-byte data.

A variable-byte encoding based integer compression method with fast random access was first introduced by (Brisaboa et al., 2009). They used a clever block reorganizing and a *rank* data structure to achieve random access. Their solution is currently the only published solution for the problem and it has been widely adopted (see, e.g., Konow et al., 2017; Shareghi et al., 2016).

In this thesis, a novel alternative solution that instead makes use of a *select* data structure

is proposed and explained in detail. The underlying question is if a *select* based method is viable for random access. Compared to Brisaboa et al., proposed method uses a very simple data reorganizing, and capitalizes on the assumption that data is encoded the same way it is in the source. Data can then be read straight from the memory and reassembling the integer from its encoded blocks is not needed.

Comparison of the proposed method and the *rank*-based implementation by Brisaboa et al. is provided with different implementations of *rank* and *select*. The *rank* method is expected to perform better on data sets containing mostly small integers. It is assumed that larger numbers lead to slower random access for *rank*-based methods. The performance of the proposed *select* method is indifferent of the size of the elements in the data. Different kinds of data sets are used to assess how each approach performs. The assumption is that *select* works better with data sets with very large integers. It remains to be seen how large the integers need to be for the *select* method to outperform the *rank* method.

The performance of the algorithms when decompressing a long subarray is also compared. Because of the way the data is stored, the proposed *select* method offers fast access to the next or previous element and is expected to perform well with any given data set or subarray length. Solutions for subarray fetching with both *rank* and *select* methods are proposed and compared.

2 Variable-byte encoding of integers

Variable-byte (VB) encoding (MIDI Manufacturers Association and others, 1996; 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 such as text files or images, leading zero bits are usually not present in large quantitites and thus VB encoding loses in compression performance to generic algorithms like Huffman encoding (Huffman, 1952) or Lempel-Ziv encoding (Ziv and Lempel, 1977).

Elias Delta and Gamma codes (Elias, 1975) are popular encoding methods for integer data sets. Their encoding process assigns short bit sequences to small integers, which is why they outperform, in terms of compression, VB encoding on datasets containing lots of small integers (Williams and Zobel, 1999), such as inverted indexes (Anh and Moffat, 2005; Pibiri and Venturini, 2019). VB is generally faster to decode (Williams and Zobel, 1999; Trotman, 2003) and so is sometimes preferred, also due to its simplicity of implementation. In this thesis, the data sets used for experimentation will have the focus on slightly larger, unordered integers.

Variable-byte encoding originates from and is used in the MIDI music file format (MIDI Manufacturers Association and others, 1996) and several applications have a similar implementation of VB. Apache Lucene has the vInt datatype (Apache, 2013) which works well with short inverted index lists (Wan and Pan, 2009). The 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 uses variable byte encoding to store record identifier lists (Bhattacharjee et al., 2009).

VB encoding was first experimented as a tool for compressing inverted index lists of word locations in documents by Scholer et al. In that setting, VB codes yielded excellent results, and since then many different approaches have been introduced. 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 the document for advanced search purposes. Usually these lists are preprocessed to form an inverted index, storing each number as its difference to the previous number instead of the actual number (Manning et al., 2008). Common words have a lot of entries in these lists, which makes

the inverted index encoded integers small. In contrast, rare words have only a few entries but the integers stored are larger. These lists are mostly small integers with a need to support larger ones and thus are excellent candidates for VB encoding.

More recent studies have taken a look into the machine code level for VB and applied SIMD (Single instruction, multiple data) instructions for VB decoding (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.

2.1 Variable-byte encoding

The name *variable-byte* reflects the way in which the integers are stored. When the integers are stored in an array, each element requires the same fixed amount of space. This is inefficient if a lot of the values are very small but the array also has to support very large integers. VB encoding attempts to eliminate the unneeded leading zeros. The bits in an integer are split into blocks of length b and empty blocks from the beginning are discarded. Because the lengths of the integers (i.e. the number of bytes used to represent them) may now be different, the data cannot be stored as it is. Instead a *continuation bit* is added in front of each block to form a *chunk*. This bit is set to 0 on all chunks except for the last one which contains the least significant bits. The continuation bit is used in decoding to signal whether or not the current integer continues in the next chunk.

For example, the standard 16-bit representation of an unsigned integer 42 is `00000000 00101010`. Assuming the block length is 4, it is split to `0000 0000 0010 1010`. The empty blocks are removed from the beginning and then the continuation bits are added. 1 is added in front of the last block (containing the least significant bits) and 0 to the other blocks, resulting in `00010 11010`. Table 2.1 contains examples of VB encoded integers with block length 4.

A pseudo code for VB encoding with block length 7 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. The block length can be changed by replacing 128 with 2^b, where b is the desired block length. Since the block length is 7, the continuation bit is added in row 8 by adding $2^7 = 128$ to the last block.

Smaller block lengths can yield a better compression rate at the cost of more bit manip-

Original number	first block	second block	third block	fourth block
4	10100			
17	00001	10001		
620	01100	00110	10010	
60201	01001	00010	01011	11110

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

```
1:  function VBEncodeNumber(n)
2:      bytes ← list
3:      while true do
4:          prepend(bytes,n mod 128)
5:          if n < 128 then
6:              break
7:          n ← n div 128
8:      bytes[len(bytes)-1] += 128
9:      return bytes

10: function VBEncode(numbers)
11:     bytestream ← list
12:     for each n ∈ numbers do
13:         bytes ← VBEncodeNumber(n)
14:         extend(bytestream,bytes)
15:     return bytestream
```

Figure 2.1: VByte encoding

ulation and therefore possibly slower decompression, while bigger block lengths need less bit manipulation and offer less effective compression. On the other hand, a bigger block length means a smaller percentage of added continuation bits. Generally, a block length of 7 has been popular because it tends to perform well on average and handling chunks as bytes is convenient (Manning et al., 2008).

2.2 Variable-byte 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 resulting blocks are concatenated to form the original integer. A pseudo code implementation of VB decoding with a block length of 7 is shown in Figure 2.2. APPEND adds an element to the end of the list. If the block length is changed, additional bit operation steps when reading the data may be needed.

Continuing from where 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 Rank and Select

Rank and *select* are two array operations that are widely used in compressed data structures. $Rank_x(i)$ counts the number of occurences of x before index i in the data. $Select_x(i)$ returns the index of the i-th occurence of x in the data. In this thesis, these two operations are assumed to work with a bit array. $Rank_1(i)$ counts the number of bits set to 1 before i and $select_1(i)$ returns the index of the i-th bit set to 1. Both operations can be implemented to work in constant time (see, e.g., Gog et al., 2014). The two operations are related to each other: when used on a bit array $B =$ `0100 1101 1011`, $select(5)$ returns 8 (underlined) and therefore $rank(8)$ returns 4.

To use *rank* or *select* in indexing, the set 1 bits in the bit array should reflect the element locations in the encoded data. For most compression algorithms, this requires the bit array to be created in addition to the existing data. The length of the bit array in these cases usually has to be close to the length of the data, because encoded element lengths vary. This spatial increment makes the compression ineffective and therefore *rank* and *select* cannot be applied to any given compression algorithm.

VB encoding has several advantages with *select* 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 number of end blocks before the i-th index and $select_1$(C, i) would give the location of the ending block of i-th compressed element.

3.1 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. A few useful functions from the library were also used during the implementation phase. Table 3.1 has *rank* and *select* size requirements of the implementations used. 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. This chapter describes a way to achieve fast random access to

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

Structure	Extra size taken
$Rank_v$	25% of bit array
$Rank_{v5}$	6.25% of bit array
Select	8-23% of bit array (depends on the data)

variable length codes. The described method also has a low space overhead, increasing the size of the compressed sequence by much less than one bit per element. Two versions of *rank* from the 'SDSL-lite' library are used in this thesis, $rank_v$ and $rank_{v5}$ and the *select* version is $select_{MCL}$. These implementations were chosen because they are fast and can be created over a regular bit array.

The data structure to support $rank_v$ has two layers. The 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. For every 64th bit inside a superblock, it stores the number of 1's since the start of this superblock.

For $rank(i)$, the superblock index $s = i$ / 512 and relative count block index $r = (i - s)$ / 64 need to be calculated, with both division operations being integer divisions. Then the number of 1 bits from index r to i are calculated from the original bit array. These three values add up to $rank(i)$. $Rank_{v5}$ is a lighter structure: its 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 requirements to a quarter.

The *rank* data structures are based on (Vigna, 2008). A $rank_v$ superblock covers 512 bits and is divided to $512/64 = 8$ relative blocks. This means a relative block needs to be able to store a number up to 512 in itself, which takes $\log_2 512 = 9$ bits. The first relative block always equals 0, because a relative block counts the number of 1 bits from the start of the superblock. Therefore the remaining 7 relative blocks take 63 bits and fit into one 64-bit word. For every 512 bits, this implementation requires 64 bits for the superblock and another 64 blocks for the relative blocks, resulting in additional $128/512 = 25\%$ space.

The $rank_{v5}$ superblock covers 2048 bits and is divided to 6 relative blocks of 384 bits, requiring $\log_2 2048 = 11$ bits each. The first relative block again equals 0, so the rest of the blocks fit to a 64-bit word. The $rank_{v5}$ implementation requires an additional $(64+64)/2048 = 6.25\%$ space.

The *select* data structure works in a similar way to rank. The index location of every

4096th set bit is stored in the superblock and the location of every 64th set bit is stored relative to the superblock. With similar calculations, the location of the closest 64th set bit is calculated and then the bit array is iterated until the required amount of bits is reached. The $select_{MCL}$ is a practical variant of a PAT tree, the data structure described by Clark, 1997. The memory requirement of the *select* implementation depends on the portion of 1 bits in the bit array.

For both rank and select, one function call always gets a value from the superblock and then from the superblock's relative block. The only variable 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 can be made work in constant time using modern popcount instructions (González et al., 2005). The implementations are explained in greater detail in Gog et al., 2014.

4 Directly addressable codes

The ability to process large amounts of data fast is one of the key challenges in the field of search engines. Compressed data structures are applied to reduce the size of data so that it fits into cache, memory or even hard drive while still allowing it to be accessed easily. Directly addressable codes (DAC), a direct access to any element in a compressed list or array, is one of the basic building blocks in compressed data structures. For example, it is needed in inverted index compression (Culpepper and Moffat, 2007) and compressed text search (Moura et al., 2000).

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 would be to store the location of each element to an array, but this adds a very large overhead which removes the benefit of compression.

4.1 DAC via rank

Directly addressable VB codes were first introduced by (Brisaboa et al., 2009). In their solution, the chunks are stored 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 continuation 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 its continuation bit to B_2 and so on.

After the data has been arranged this way, a *rank* data structure is built for each bit array. The data structure does not require additional space on top of that used by the normal VB encoding apart from that used by the *rank* structure. Figure 4.1 contains a visualization of the data structure. The C array is the original VB encoded data, which is split into separate block arrays A_k and contiuation bit arrays B_k.

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 directly indexing A_1. With small integers, this is all that is required. If the data has a lot of small integers, the *rank* method thus has a huge advantage.

The chunk array. $C_{i,j} = B_{i,j} : A_{i,j}$

$\mathbf{C} =$

$C_{1,2}C_{1,1}$	$C_{2,1}$	$C_{3,3}C_{3,2}C_{3,1}$	$C_{4,2}C_{4,1}$	$C_{5,1}$	$\cdots$

$A_1 =$ / $B_1 =$

$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$	$A_{5,1}$	$\cdots$
0	1	0	0	1	$\cdots$

$A_2 =$ / $B_2 =$

$A_{1,2}$	$A_{3,2}$	$A_{4,2}$	$\cdots$
1	0	1	$\cdots$

$A_3 =$ / $B_3 =$

$A_{3,3}$	$\cdots$
1	$\cdots$

Figure 4.1: Data structure by Brisaboa et al., visualized

However if the element was stored in multiple blocks, the index of the next block in A_2 is obtained from $i \leftarrow rank_0(A_1, i)$. $Rank_0$ returns the number of zeros in the bit array preceding index i. In other words, it returns the number of elements before i that continue to the next array. This in turn means that i is the index of desired element's next block in array A_2. This process is repeated until the i-th bit of the bit array of the current level (the continuation bit for the current block) is 1. The resulting integer is constructed from the blocks of the fetched chunks. A pseudo code example of DAC with *rank* is shown in Figure 4.2 with block length of 8.

As an example, the fourth element from the data structure shown in Figure 4.1 is decoded. The first block of data is fetched by directly accessing $A_1[3]$. Then the continuation bit $B_1[3]$ is checked. It is not set, which indicates that the element continues to the next block. The index of the next block is obtained via a $\textsc{Rank}(B_1, 3)$ call. There are two zero bits in B_1 before $B_1[3]$, which means the next block is in $A_2[2]$. The continuation bit at $B_2[1]$ is 1 so the decompression is completed.

Interestingly, the block length of this method can be allowed to change within different array levels which opens a possibility for further compression optimization. Brisaboa et al. experiments with a few different block length setups and leaves open the problem of finding optimal block lengths. While the problem of finding the optimal block length for each data set in regular VB encoding is interesting, the possibility to optimize it for each

block level separately is surely more challenging.

$index \leftarrow$ 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 4.2: Example pseudo code of DAC with rank by Brisaboa et al.

5 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 aforementioned 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 just 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 its 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 $b_s = select_1(j\text{-}1) + 1$. In this thesis, the 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 bit 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 to do 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, but since the block length differs from byte length, the starting byte location s needs to be calculated $s = (select_1(i) \times 4) \div 8$, the latter operand being for integer division. 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 is equal 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 the offset was not zero, there are o trailing 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 desired value. Most systems are able to read memory only from byte addresses, which leads

to a corner case where the compressed integer is stored in more bytes than are contained in a word. In the aforementioned situation, this happens if the compressed length was 8 blocks and offset was 4. In this case, another step of reading another byte and storing the remaining bits to w is required. This is entirely avoidable if the integer length can be constrained. If the maximum integer length was set to 28 bits, the bits would be within a single word in memory and reading the extra byte would not be needed.

The most intuitive way to calculate the block length of the i-th element is to subtract $select_1$(i) from $select_1$(i+1). This however causes an additional $select_1$ query, which is costly. A much faster option is to calculate the block length from the continuation bit array. This can be done by reading a word from the bit array and bit-shifting the word until the bit representing the start of the i-th element is on the rightmost end and then counting the trailing zeros (for example, by using the GCC function `__builtin_ctz`). The trailing zeros equal to the number of blocks in the i-th element that do not have the continuation bit set. This number is then incremented by one to also count the ending block which has the continuation bit set to 1.

Figure 5.1 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 pseudo code, CalculateLength returns the length of the number in blocks and ByteMask(k) returns a bit mask for k bytes. In line 8, a long word is read from memory starting at memory address $A[begin]$.

```
1: i ← wanted index
2: A ← block byte array
3: B ← continuation bit array
4: begin ← Select(B, index)
5: len ← CalculateLength(begin)
6: mask ← ByteMask(len)
7: word ← A[begin]
8: word ← word & mask
```

Figure 5.1: Pseudo code of DAC with select with block length 8

Figure 5.2 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. The length of the compressed element is 3, which is obtainable, e.g., via $select_1(k)$. Then a word is read from the block array, starting from block A_i and blocks not belonging to the element are removed for example with a bit mask, resulting in k-th element being decoded.

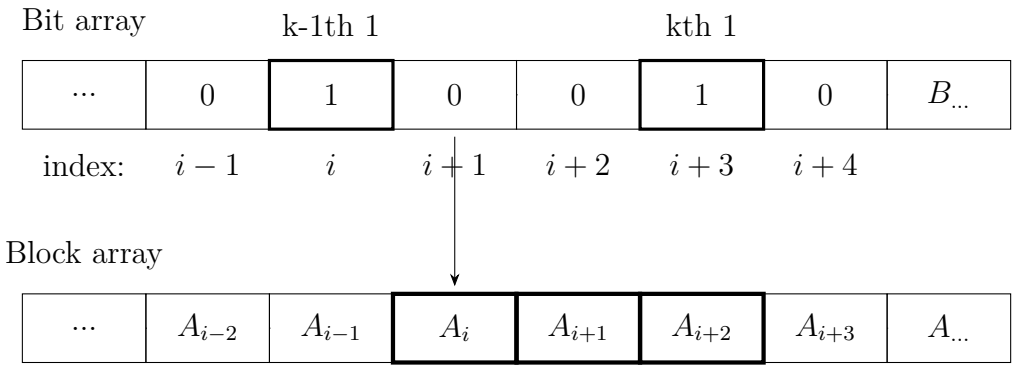

Figure 5.2: Relation of continuation bits and data blocks in VB encoding.

5.1 Elias-Fano encoding

Direct access via select can also be implemented with Elias-Fano encoding, but the data has to be sorted. This is usually a given in inverted index lists of search engines. In Elias-Fano encoding (Elias, 1974; Vigna, 2013), the data is at first split into low and high bits. The low bits, the k least significant bits, are stored to their own array. Direct access to the lower bit array is trivial. The higher bits are gap-encoded and then encoded in unary. For example, [1,3,5,8,11] becomes [1,2,2,3,3] and then is encoded as a unary bit array B = `0100100100010001`.

$Select_1$ has an interesting interaction with gap encoded unary arrays. $k = Select_1(B, i)$ returns the location of i-th 1 in B. Up to index k, the bit array has had i ones and $i - k$ zeros. The i corresponds to the number of compressed integers in B and every zero in the array means an increment of 1 in the numbers. In other words, $i - k$ equals to the integer at position i before gap-encoding. Therefore $select_1(i)$ - i returns the original higher bit number. The split between bits is made to reduce the gap sizes.

Elias-Fano encoding is an excellent tool if the elements are sorted and the higher bits are reasonably small, because large integers cause a very inefficient unary encoding. This can be further optimized by finding clusters in the integers and partitioning the indexes (Ottaviano and Venturini, 2014). Elias-Fano encoding is not compared in this thesis, because the data considered is not necessarily sorted.

6 Subarray access

After locating and decompressing one value from a compressed data structure, decompressing the subsequent elements cam become significantly easier. Calculating the memory location of the next element may not be needed if the next compressed value is located next to the previously fetched element. This is especially beneficial to the select-based VByte decoding, where calculating the start byte location takes majority of the runtime. In addition to this, the length of the next compressed integer is very fast to calculate because the required data is already in memory and immediately local to the last point of access.

6.1 Subarray access with select

A single direct access with *select* needs to calculate the length of the desired element. This is done by reading a word from the continuation bit array, bit-shifting until the bit representing the first block is on the rightmost end, and then counting the trailing zeros in the word. Subsequent length calculations are done by bit-shifting the word equal to the length of the previous element and then counting trailing zeros. Once the word is shifted through, another word is read and the process continues.

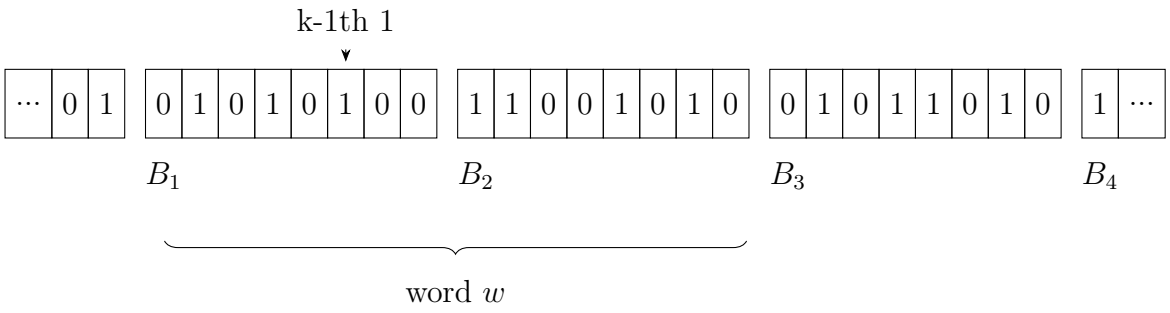

Figure 6.1: Continuation bit array, the bits are grouped by the bytes they are in

Figure 6.1 shows an example of a continuation bit array. The index of the 1 bit returned by the *select* query issued for the starting index is marked, and the desired starting index $i = select(k-1)+1$ is the following bit on the right. The byte where the i-th bit is located can be calculated by doing an integer division $i/8$. In the example, a 16-bit word

w is read from the byte location B_1. It is beneficial to read as large word as possible, but 16-bits is used in this example for the sake of simplicity. Next the starting bit is shifted to the start of the word, the offset being 6 in this case. After the operation, w contains bits `00110010 10000000`, with the least significant bit being on the left. The length of the current element is $l = 1 + \textsc{TrailingZeros}(w)$. The number of trailing zeros can be efficiently calculated with the GCC function `__builtin_ctz` (or `bits::lo` from the 'SDSL-lite' library).

The length of the desired element l and the location i can now be used to decompress the desired element. For the next element, i is incremented by l and w is bit shifted l bits and the process is repeated. When w has no 1 bits, another word is read from the following bytes. A few extra steps are required, because the old w might have continuation 0 bits left for the current word. The number of leftover zeros is added to the first length calculation after reading a new word for w. This number can be calculated either by adding bit shift amounts together and subtracting the result from word-length. Another option is to count leading zeros of the old w before the first switch. In the example, one zero is left over from the byte B_2.

6.2 Subarray access with rank

The shown *rank* method by Brisaboa et al. described earlier can also effectively decompress the next value with slight modifications. When fetching a subarray, the goal becomes to minimize *rank* calls. When a new block level is reached for the first time during a subarray query, the index is fetched with *rank* as normal, and the result of the query is stored. When a new index is needed from a previously used level, the desired block is known to be right next to the previously fetched one. Therefore the stored index value is incremented by one and that value is used instead of issuing another *rank* query. This way the number of *rank* calls needed is equal to the number of *rank* calls needed to decompress the largest number in the subarray.

An example situation from a previous chapter is pictured in Figure 6.2. A subarray of four elements will be decoded from the start of the array. The starting location could very well be anywhere, but the start was chosen for simplicity. The array P is used to store the block indices. Initially the values in P are set to -1 to denote that the value is unset and that level has not been reached yet. When a block level is queried for the first time, the index value from the *rank* call is stored in P and used instead of further *rank* calls.

The subarray is in this case started from index 0. The value of P for the current level is checked at the start of every block fetch. $P[1]$ is unset, so a value needs to be fetched and stored there. Because the *rank* method does not need an actual *rank* call in the first block array, $P[1]$ is set to the starting index 0 and that value is used to fetch the first block. $B_1[0]$ shows that the element is continued to the next block level. $P[2]$ is -1 (because this is the first time the second level is reached), so a $P[2] = rank_0(B_1,0)$ call is needed to fetch the second block. $B_2[0]$ indicates that the element is not continued anymore.

For the location of the first block of the second element, $P[1]$ is checked. It is set, so the value it contains is increased by one and used as the index. The bit array shows that this element is only one block long. The first two blocks of the third element are fetched the same way. $P[2]$ is unset and a $P[2] = rank_0(B_2,0)$ is used to fetch the index. The fourth element does not reach to a new block level, so each block location is fetched from P.

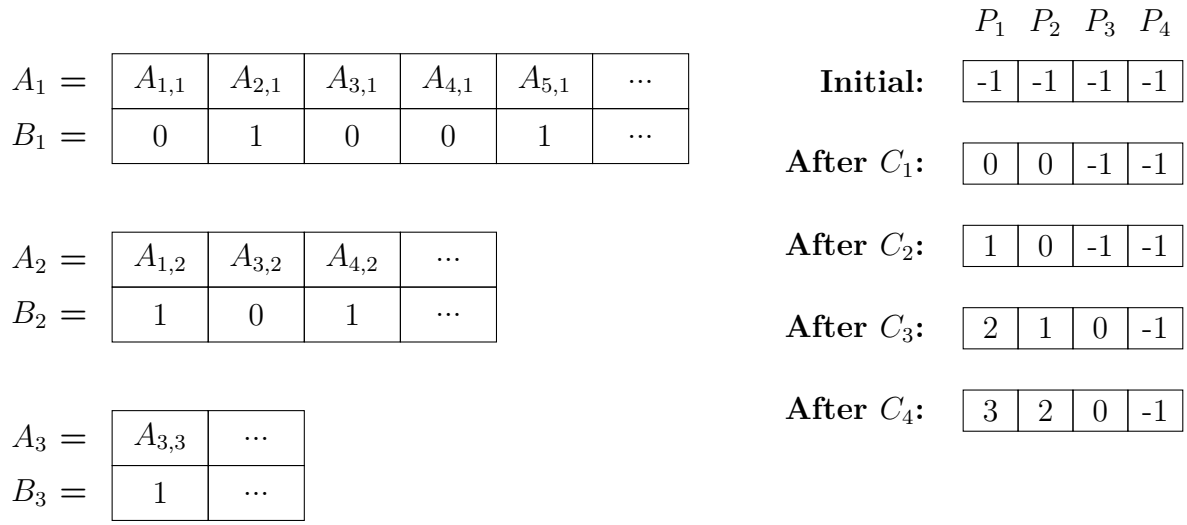

Figure 6.2: Index lookup array demonstration in *rank* based subarray query.

7 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) with minimal programs running in the background. The implementations were written in C++ and compiled with `g++-7 -std=c++11 -DNDEBUG -O3` and with libraries `-lsdsl -ldivsufsort -ldivsufsort64`.

The implementations for these results can be found online at https://github.com/mozzie/vbyte.

7.1 VB decoding comparison

The list of implementations is depicted in Table 7.1. Three different VB decoding implementations were compared with two different block lengths. Two *rank*-based implementations of VB used are based on (Brisaboa et al., 2009) and use $rank_v$ and $rank_{v5}$ data structures from the SDSL-library (Gog et al., 2014). The different *rank* data structures achieve different trade-offs between memory and speed. The *select* algorithm is the one proposed in this thesis and it is implemented using the $select_{MCL}$ data structure from the SDSL-library. Different block lengths were used to illustrate how block size affects the runtime and memory requirement. Lengths 8 and 4 were chosen because they are relatively easy to implement.

Each implementation compresses and decompresses 64-bit unsigned integers. The *rank*

Table 7.1: Generated datasets

Algorithm	Explanation
Bris4	Bris implementation with block size 4, using $rank_v$
Bris4v5	Bris implementation with block size 4, using $rank_{v5}$
Bris8	Bris implementation with block size 8, using $rank_v$
Bris8v5	Bris implementation with block size 8, using $rank_{v5}$
Select4	Proposed implementation with block size 4, using $select_{MCL}$
Select8	Proposed implementation with block size 8, using $select_{MCL}$

Table 7.2: Generated datasets

Dataset	Explanation
all (5M)	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

Table 7.3: Results in milliseconds, smaller is better. Smallest time is bolded

	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

and *select* implementations are explained in detail in Section 4.1 and Chapter 5, respectively.

Table 7.2 describes four different types of synthetic datasets that were used for the experiments. Each dataset was generated to three different sizes (5M, 50M, 500M). The content of the datasets were chosen in order to see how different implementations work with different sized data and different magnitudes of integers.

All implementations read the data set from a file and compress it in memory, randomize one million query indexes and store these indexes to an array. The times shown in Table 7.3 are times taken from looping through the index array and VB decoding the number at each index. Additionally, each run is iterated 10 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. Figure 7.1 shows average times of one million *rank* and *select* calls with different average block lengths. The effect of multiple *rank* calls is visible, as is the single *select* call behavior, which leads to a much flatter curve.

The 8-bit versions tend to perform better than the 4-bit versions. This is because the 8-bit version stores more data in one block and so less block readings are required. However, the 4-bit versions achieve better compression in data sets containing small integers. The select versions do not depend as much on the block size, because all the blocks in the compressed element are read at the same time. Interestingly the 4-bit version of *select* performs better than the 8-bit version with some data sets, even though the 4-bit version should execute at least the same commands as the 8-bit version in every scenario. This is most likely an effect of caching, because the stored data set is smaller in the 4-bit version, and thus more of it fits in higher levels of memory.

The $rank_{v5}$ naturally has worse performance because more calculations are needed in the last phase of calculating the rank sum. As a trade-off, it requires significantly less memory than $rank_v$, as seen on Table 3.1.

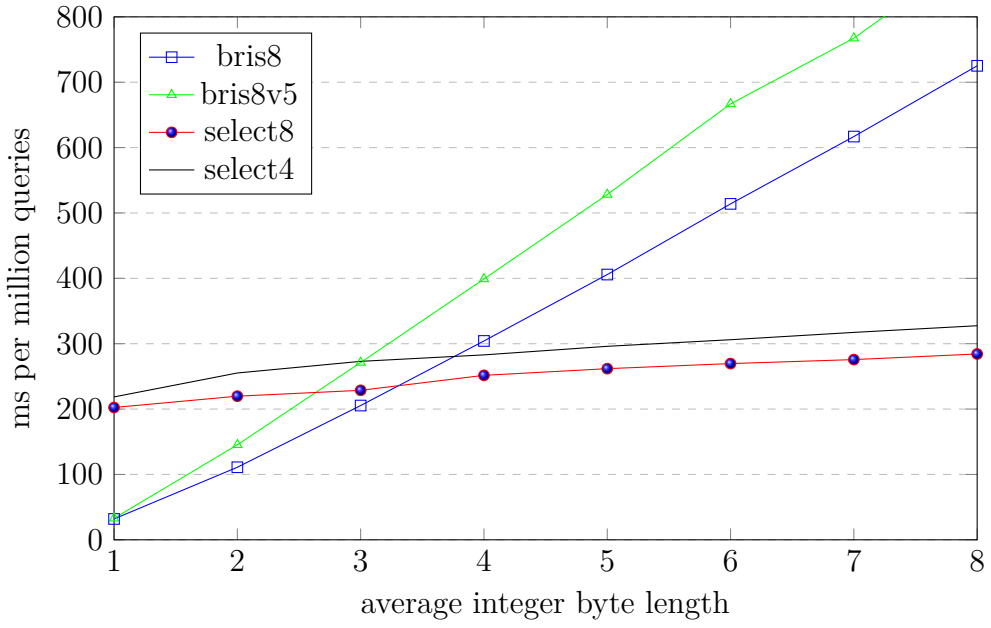

Figure 7.1: Rank and select performances

dataset	compressed size	bris8	bris8v5	select8
all	168.4MB	4.67MB	1.17MB	1.54MB
twolarge	112.4MB	3.12MB	781kB	1.48MB
onelarge	63.2MB	1.76MB	439kB	1.43MB
onlysmall	56.3MB	1.56MB	391kB	1.43MB

Table 7.4: Memory requirement for 50M 64-bit numbers 8bit blocks, smaller is better.

dataset	compressed size	bris4	bris4v5	select4
all	177.7MB	7.65MB	1.91MB	1.63MB
twolarge	116.1MB	4.66MB	1.14MB	1.54MB
onelarge	42.5MB	2.12MB	531kB	1.44MB
onlysmall	31.3MB	1.56MB	391kB	1.43MB

Table 7.5: Memory requirement for 50M 64-bit numbers 4bit blocks, smaller is better.

7.2 Memory usage

Tables 7.4 and 7.5 show memory size requirements for *rank* and *select* data structures with data sets consisting of 50 million 64-bit numbers. The compressed size column is the sum of the sizes of the continuation bit array and the data blocks. The tables show that the static requirement of $rank_{v5}$ is superior with these datasets. If the dataset had even larger integers, *select* would be dominant. For *rank* and *select* methods used in this thesis, the total space taken by the extra data structures is only a small fraction of the overall size.

The static memory requirement of *select* is easily noticeable, as is the relation between the memory requirement of *rank* and the total number of blocks. Data structures in versions with smaller block size naturally take more space, but smaller block size allows better compression in some data sets and therefore a smaller total size. This can be seen when comparing the two forementioned figures: All 4-bit versions of the algorithms require more space than the 8-bit versions, but the compressed size of the data is significantly smaller in the bottom two data sets. The data structure memory usage for the bottom row is similar in both versions due to the fact that the continuation bit array is exactly the same in both cases.

7.3 Subarray access results

The subarray access experiments were run similarly to the experiments of previous sections. The data sets used contained 50 million numbers each. Most of the numbers were 4 bits long, with a varying amount of 32-bit integers randomly among them. The numbers in the x-axis correspond to the number of 32-bit integers per 1000 integers. One million indexes were randomized and both the index numbers and the data sets were preloaded into memory. A subarray of length 50 was decompressed from each randomized index location. Some precaution was used in the index randomizing to prevent reading from an out of bounds location. The time shown in the figure is milliseconds taken to decompress one million subarrays. Each result is an average of 10 runs.

Figure 7.2 shows results for subarray fetching. The probability of having a long integer within the subarray range is visible from the results of the *rank* based methods. When the density of the long integers increase, so does the need to access multiple block layers and thus the average time increases. The results of the *select* based methods visualize the static need of singular *select* query. The 4-bit version of *select* requires a few more instructions during the block location phase and is therefore slightly slower overall.

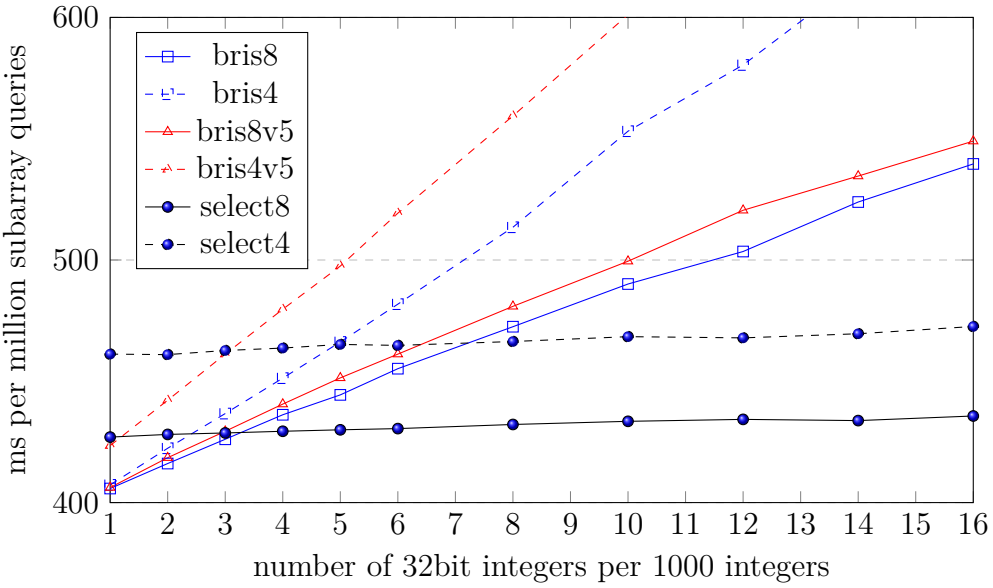

Figure 7.2: *Rank* and *select* performance with subarray fetching. The scale of the y-axis was selected to clarify the difference of the results.

8 Conclusions and Future work

This thesis explained variable-byte encoding, a compression method that stores the data in fixed length blocks and omits empty leading blocks. A continuation bit is added to each block to signal where an element ends and a new one begins. *Rank* and *select*, two array operations were explained and efficient implementations over bit arrays were introduced in detail.

An efficient solution to direct access to any element in a compressed data structure was previously introduced by Brisaboa et al. Their solution uses VB encoding and a *rank* data structure. An alternative solution using *select* was proposed in this thesis. Both solutions were experimented with a few different data structure implementations. As expected, the *rank* version performed better with data sets that contain small integers. This is largely due to the clever block reorganizing technique that allows the first block to be fetched with just an array lookup. The *rank* query is also slightly faster than a *select* query, and therefore the dataset needs to have a large portion of longer elements before *select* performs better.

All implementations were slightly modified to support subarray fetching. The order of how data is stored with *select* compression enables an efficient fetching of a subarray. Only one *select* query is needed in the beginning, and the decoding of the elements past the first one consists of just a few bit operations. The amount of *rank* queries depend on the length of the subarray and the average size of the elements. Therefore even a small amount of large elements in an array otherwise filled with small elements causes the runtime to increase significantly. This leads to the conclusion that the *select* method, if not strictly superior, is a strong contender and a good option in particular when decompressing subarrays is required.

8.1 Future work

There has already been a few successful attempts to decompress VB encoded integers sequentially with SIMD instructions (Lemire et al., 2018; Plaisance et al., 2015). It is worthwhile to check if the *select* based approach presented in this thesis benefits from SIMD. The data needs less manipulation because it is already stored in the correct order,

but calculating the integer block lengths might benefit from SIMD.

Another improvement to VB encoding is to see whether the compressed numbers can be changed. This is trivial until the block length of the compressed number changes. Then an additional data structure or cache is needed and the supporting rank/select data structure needs to be recalculated.

TODO: a few paragraphs about haswell instructions

Bibliography

Anh, V. N. and Moffat, A. (2005). "Inverted index compression using word-aligned binary codes". In: *Information Retrieval* 8.1, pp. 151–166.

Apache (2013). *Apache Lucene - Index File Formats*. URL: https://lucene.apache.org/core/3_5_0/fileformats.html#VInt (visited on 04/10/2020).

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.

Clark, D. (1997). "Compact pat trees". PhD thesis.

Culpepper, J. S. and Moffat, A. (2007). "Compact set representation for information retrieval". In: *International Symposium on String Processing and Information Retrieval*. Springer, pp. 137–148.

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.

Elias, P. (1974). "Efficient storage and retrieval by content and address of static files". In: *Journal of the ACM (JACM)* 21.2, pp. 246–260.

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).

Hon, W.-K., Shah, R., and Vitter, J. S. (2010). "Compression, indexing, and retrieval for massive string data". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer, pp. 260–274.

Huffman, D. A. (1952). "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9, pp. 1098–1101.

Konow, R., Navarro, G., Clarke, C. L., and López-Ortíz, A. (2017). "Inverted treaps". In: *ACM Transactions on Information Systems (TOIS)* 35.3, pp. 1–45.

Lemire, D., Kurz, N., and Rupp, C. (Feb. 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*.

Moura, E. Silva de, Navarro, G., Ziviani, N., and Baeza-Yates, R. (2000). "Fast and flexible word searching on compressed text". In: *ACM Transactions on Information Systems (TOIS)* 18.2, pp. 113–139.

Ottaviano, G. and Venturini, R. (2014). "Partitioned elias-fano indexes". In: *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 273–282.

Pibiri, G. E. and Venturini, R. (2019). *Inverted Index Compression*.

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.

Shareghi, E., Petri, M., Haffari, G., and Cohn, T. (2016). "Fast, small and exact: Infinite-order language modelling with compressed suffix trees". In: *Transactions of the Association for Computational Linguistics* 4, pp. 477–490.

Trotman, A. (2003). "Compressing inverted files". In: *Information Retrieval* 6.1, pp. 5–19.

Venturini, R. (2013). *Compressed Data Structures for Strings: On Searching and Extracting Strings from Compressed Textual Data*. Vol. 4. Springer Science & Business Media.

Vigna, S. (2008). "Broadword implementation of rank/select queries". In: *International Workshop on Experimental and Efficient Algorithms*. Springer, pp. 154–168.

– (2013). “Quasi-succinct indices”. In: *Proceedings of the sixth ACM international conference on Web search and data mining*, pp. 83–92.

Wan, J. and Pan, S. (2009). “Performance evaluation of compressed inverted index in lucene”. In: *2009 International Conference on Research Challenges in Computer Science*. IEEE, pp. 178–181.

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

Ziv, J. and Lempel, A. (1977). “A universal algorithm for sequential data compression”. In: *IEEE Transactions on information theory* 23.3, pp. 337–343.

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