MSc thesis

Computer Science

# Select-based random access to variable-byte encodings

Jussi Timonen

April 14, 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 | April 14, 2020 | 24 pages, 25 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** **1**

**2 Variable-byte encoding of integers** **3**

2.1 VB encoding . . . . . 4

2.2 VB decoding . . . . . 4

**3 Directly addressable codes** **7**

3.1 Rank and Select . . . . . 7

3.2 Rank and Select implementation . . . . . 8

3.3 DAC via rank . . . . . 9

**4 DAC with select query** **12**

4.1 Elias-Fano encoding . . . . . 14

**5 Subarray access** **15**

**6 Experimental results** **17**

6.1 VB decoding comparison . . . . . 17

6.2 Memory usage . . . . . 19

6.3 Subarray access results . . . . . 20

**7 Conclusions and Future work** **22**

**Bibliography** **23**

# 1 Introduction

Enormous datasets are a common case in today's applications. 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).

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 in the compressed data. If this is required, the usual data compression algorithms are inefficient. Fortunately, such random access to elements is not a requirement compression algorithms usually need to fulfill. However, random access to compressed data is very 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 fast random access was first introduced by (Brisaboa et al., 2009). They used a clever block reorganizing and a fast *rank* data structure to achieve random access. Their solution is currently the only published solution for the problem and it has been widely adopted (e.g. Konow et al., 2017).

In this thesis, an alternative solution that instead makes use of a *select* data structure is proposed and explained in detail. Comparison to the *rank*-based implementation by Brisaboa et al is provided with different implementations of *rank* and *select* and several different data sets. The 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.

The *rank* method is found superior when dataset contains mostly small integers. With larger integers, the decoding needs multiple *rank* calls and the new *select*-based method performs better. Additionally, due to how the data is stored, the proposed *select* method offers fast access to the next or previous element. This makes a subarray fetch very

convenient and fast. A solution for subarray fetching and comparison is proposed in this thesis.

# 2 Variable-byte encoding of integers

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 so is sometimes preferred, also due to its simplicity of implementation.

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 to 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, but inverted index encoding makes the integers are small. In contrast, rare words have only a few entries but the integers stored are larger. These lists are thus excellent candidates for VB encoding.

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). 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).

Later, VB encoding was found to be efficient for compressing integer lists. It was first experimented as a tool for compressing inverted index lists of word locations in documents by (Scholer et al., 2002). In that setting, VB codes yielded excellent results, 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 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.

Elias Delta and Gamma codes (Elias, 1975) are popular encoding methods for the aforementioned data sets. Their encoding process assigns short bit sequences to small integers, which is why they outperform in trems of compression VB encoding on datasets containing a lot of small integers (Williams and Zobel, 1999), such as inverted indexes (Anh and Moffat, 2005; Pibiri and Venturini, 2019). In this thesis, the data sets used for experimentation will have the focus on slightly larger, unordered numbers.

## 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 whether or not the current integer continues in the next chunk.

For example, the standard bit representation of an unsigned integer 42 is `00000000 00101010`. When split to blocks of length 4, it becomes `0000 0000 0010 1010`. The empty blocks are removed and then continuation bits are added. 1 is prepended to the least significant block and 0 to the other blocks, resulting in `00010 11010`. A pseudo code for VB encoding 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 lengths can yield a better compression rate at the cost of more bit manipulation and therefore possibly slower decompression, 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, 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 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 resulting blocks are concatenated to form the original integer. A pseudo code implemen-

```
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

| 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.

tation 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.

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 operations on a bit array B. $Rank_1$(B, i) gives the number of 1 bits between B[0] and B[i-1], inclusive, 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 (see, e.g., 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 lengths 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.

## 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 from the library 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. This chapter describes a way to achieve fast random access to 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 $select_{MCL}$. All these were chosen, because they are faste 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. 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 from the original bit array. All three values added together add up to rank(i). $Rank_{v5}$ 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.

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 to 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. 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 to 0, so the rest of the blocks fit to a 64-bit word.

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 location of every 64th set bit is stored

**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 | 6-15% of bit array (TODO: check this |

relative to the superblock. With similar calculations, the location of closest 64th set bit is calculated and then bit array is iterated until the required amount of bits is reached. The $select_{MCL}$ is a practical variant of a PAT tree (Clark, 1997).

In both cases, 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).

## 3.3 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 is set, a *rank* data structure is built for each bit array. The data structure does not require additional space apart from the *rank* structure. Figure 3.1 contains a visualization of the data structure.

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. In addition, the block length of this method can change within different array levels. This opens a possibility for further compression optimization.

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

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 | $\cdots$ |

$A_1 =$ $B_2 =$

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

$A_3 =$ $B_3 =$

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

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

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 the 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 ← wanted index
A ← block arrays
B ← continuation bit arrays
level ← 0
number ← 0
while B[level][index] = 0 do
    block ← A[level][index]
    number ← number ≪ 8
    number ← number + block
    index ← RANK(B[level], index)
    level ← level + 1
block ← A[level][index]
number ← number ≪ 8
number ← number + block
```

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

# 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 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 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, 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 desire 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 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 the integer length is constrained. In this case the maximum integer length would be 28 bits.

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 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. This counts 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 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
$begin \leftarrow \textsc{Select}(index)$
$len \leftarrow \textsc{CalculateLength}(begin)$
$mask \leftarrow \textsc{ByteMask}(len)$
$word \leftarrow A[begin]$ ▷ 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. 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 either with a bitmask or by double bit shifting, resulting in k-th element being decoded.

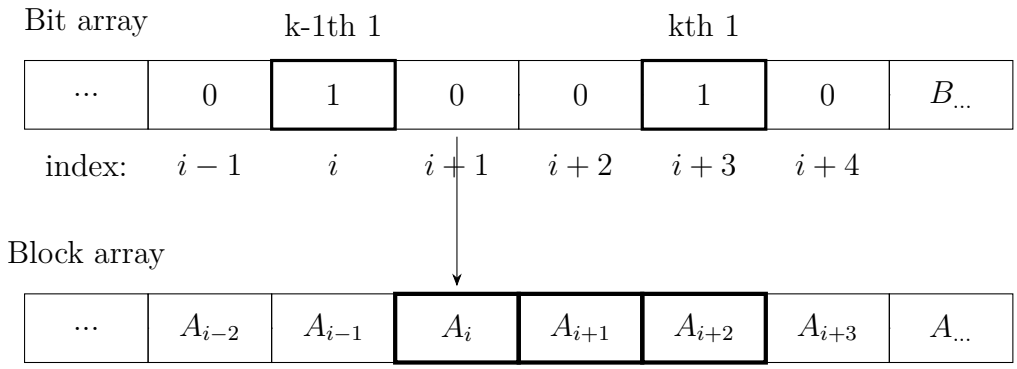


**Figure 4.2:** Relation of continuation bits and data blocks in VB encoding

## 4.1 Elias-Fano encoding

Direct access via select can also be implemented with Elias-Fano encoding, but the data has to be sorted, which 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. 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 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. This approach can be further optimized by finding clusters in the integers and partitioning the indexes (Ottaviano and Venturini, 2014).

# 5 Subarray access

After locating and decompressing one value from a compressed data structure, decompressing the subsequent elements becomes significantly easier. Calculating the memory location of the next element is not needed because the next compressed value is usually 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.

## 5.1 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 process continues.

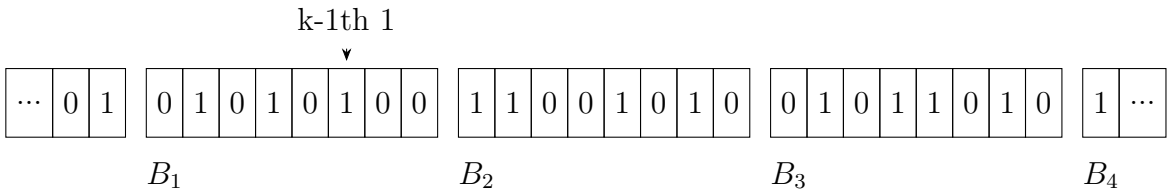


**Figure 5.1:** Continuation bit array, the bits are grouped by the bytes they are in

Figure 5.1 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 $i$-th bit is located can be calculated by doing an integer division $i/8$. 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`, least significant bit being on the left. The length of the current element is $l =$

1 + TrailingZeros($w$). The number of trailing zeros can be efficiently calculated with 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 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 word switch. 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 $B_2$.

## 5.2 Subarray access with *rank*

The shown *rank* method by Brisaboa et al. can also effectively decompress the next value with slight modifications. When fetching a longer 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 $rank_1$ as normal, and the result of the query is stored. The next time the same level is reached, the previously stored result is incremented by one and that value is used instead of issuing another $rank_1$ query. This way the number of $rank_1$ calls needed is equal to the number of $rank_1$ calls needed to decompress the largest number in the subarray.

- examples / visualizaton

# 6 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 codes were compiled with `g++-7 -std=c++11 -DNDEBUG -O3` and with libraries `-lsdsl -ldivsufsort -ldivsufsort64`.

## 6.1 VB decoding comparison

Six 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 type integers. The *rank* and *select* implementations are explained in section 3.2.

- 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 $select_{MCL}$

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

**Table 6.1:** 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 |

- onlysmall - all integers 4 bits or less

All implementations read the data set from a file and compress it to memory, randomize one million query indexes and store results to an array. The times shown in Table 6.1 are times taken from looping through the index array and VB decoding the number in the 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. Figure 6.1 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.

The 8-bit versions tend to perform better than the 4-bit versions. This is because 8-bit version stores more data in one block and less block readings are required. However, 4-bit versions achieve better compression in data sets containing small integers. 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 4-bit version should execute at least the

same commands as the 8-bit version in every scenario. This is most likely due to cache, because the stored data set is smaller in the 4-bit version.

The v5 version of rank 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, as seen on Figure 3.1.

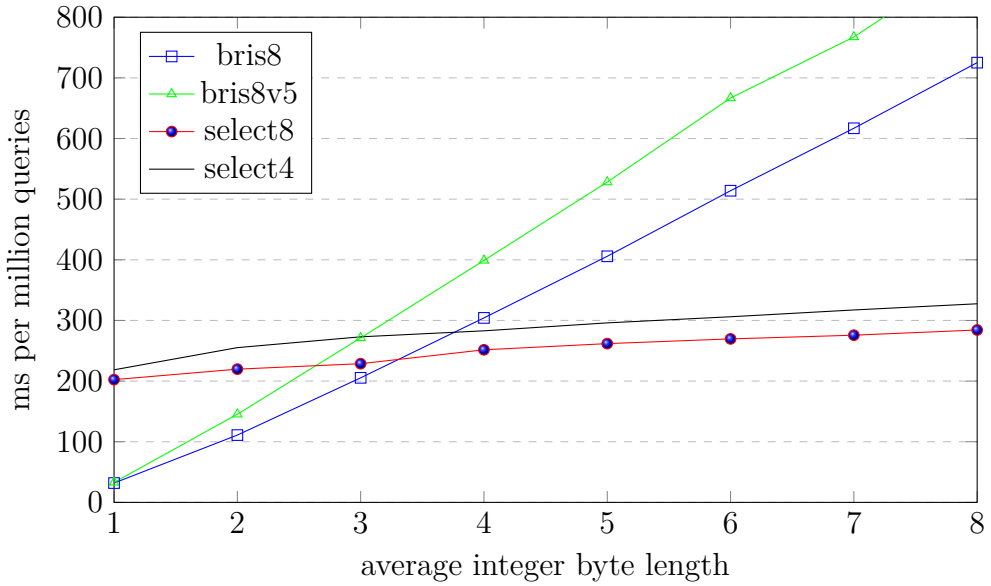


**Figure 6.1:** Rank and select performances

## 6.2 Memory usage

Table 6.2 and Table 6.3 show memory size requirements for *rank* and *select* data structures with data sets consisting of 50 million 64-bit numbers. The compressed size column has a sum of continuation bit array and the data blocks. 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

| 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 6.2:** 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 6.3:** Memory requirement for 50M 64-bit numbers 4bit blocks, smaller is better.

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.

## 6.3 Subarray access results

Figure 6.2 shows results for subarray fetching. The data sets used contained mostly 4-bit integers and a few 32-bit integers randomly among them. The number in the x-axis corresponds to the number of large integers within 1000 integers and the y-axis has the average time of fetching 50 subsequent numbers from an initially random location.

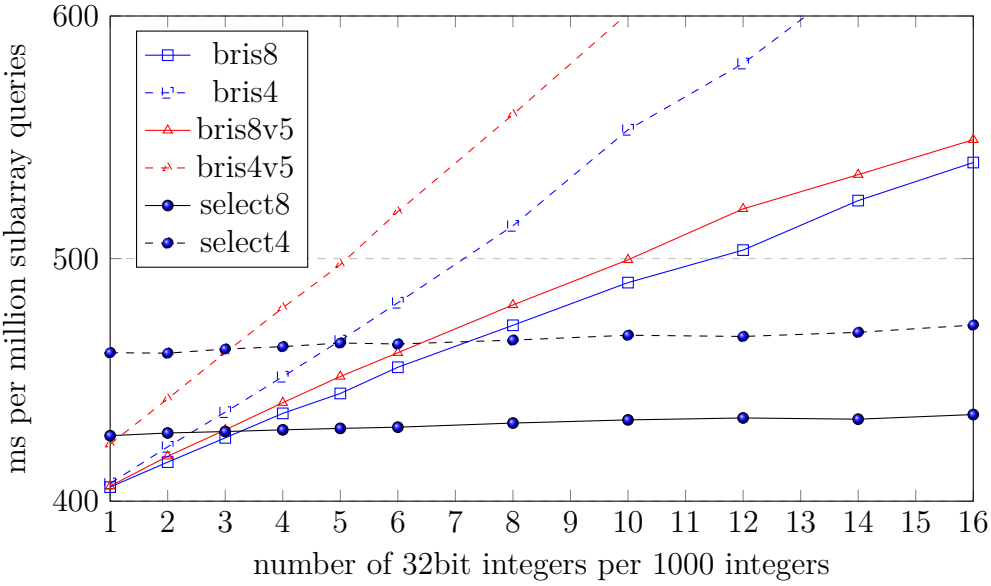


**Figure 6.2:** Rank and select performance with subarray fetching

# 7 Conclusions and Future work

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.

There has already been a few successful attempts to decompress VB encoded integers 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 the data 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: - see how cache matters

- figure out why data size matters

- add subarray examples / visualization

- write more about subarray access in results

- more text / references

- write conclusions

- write introduction

- write abstract

# 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.

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).

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. (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*.

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.

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.

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.