MSc thesis

Computer Science

Select-based random access to variable-byte encodings

Jussi Timonen

February 28, 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	February 28, 2020	15 pages, 17 appendice pages

Tiivistelmä — Referat — Abstract

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

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

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

Avainsanat — Nyckelord — Keywords

ulkoasu, tiivistelmä, lähdeluettelo

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms specialisation line

Contents

1 Introduction

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

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

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

An alternative solution with *select* data structure is proposed and explained in detail later. Comparison to *rank* by Brisaboa et al is shown with different implementations of *rank* and *select* and several different data sets are used. 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, reading the data straight from the memory is much faster than reassembling the word from blocks.

(TODO: refer to results)

2 Variable-byte encoding

Variable-byte (VB) encoding (Williams and Zobel, 1999) is a method for compressing integers via omitting leading zero bits that would be present in a longer fixed width word. In normal data sets, VB encoding loses in compression performance to generic algorithms like Huffman encoding or Lempel-Ziv encoding, but is generally faster to decode (Brisaboa et al., 2009) and sometimes preferred due to its simplicity. As later shown in Chapter 3, constant time random access is possible with VB encoding.

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

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

Elias Delta and Gamma codes (Elias, 1975) are popular encoding methods for forementioned data sets. They assign short bit array codes to small numbers, which allows them to outperform VB encoding on small number datasets (Williams and Zobel, 1999). Elias Delta and Gamma codes can't support fast random access efficiently because of their changing code length.

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 extra bit is set to 1 only on the block containing the least significant bits of the integer. This information is used in decoding to signal if next chunk continues the current integer. For example, let's assume $b = 4$ and let

$n = 42$ be a 16 bit unsigned integer. The standard 8-bit representation of n is `00000000 00101000`. When split to blocks of b bits, it becomes `0000 0000 0010 1000`. Empty blocks are omitted and continuation bits are added to the remaining blocks. The result is `00010 11000`, which is the compressed data.

```
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

Decoding is essentially just reversing the encoding steps: chunks are read until a chunk with 1 as continuation bit is found. Continuation bits are removed from all the chunks and the blocks are concatenated to form the original number. As in the previous example, $b = 4$, encoded message is `00010 11000` and $n = 0$. The block from the first chunk is extracted and added to n, making $n =$`10`. A bitwise shift to the left equal to b is applied to n, changing $n =$`100000`. Then the block is extracted from the next chunk. This block is added n, making $n = 42$. Because the previous continuation bit was 1, decoding for this number is complete. More examples shown in Table 2.1. An example implementation of encode and decode with block length of 7 is shown in Figure 2.1 and Figure 2.2.

Small lengths of c can yield better compression rate at the cost of more bit manipulation, while longer chunks need less bit manipulation and offer less effective compression. Generally block length of 7 has been used because it tends to perform well on average and handling chunks as bytes is convenient (Manning et al., 2008).

```
function 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
            numbers.append(n)
            n ← 0
    return numbers
```

Figure 2.2: VByte decoding

Original number	first block	second block	third block	fourth block
4	14000			
17	00001	10001		
620	01100	00110	10010	
60201	11001	00010	01011	11110

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

VB encoding is a well known compression algorithm. It's origins date back to 1980's and the famous MIDI music file format. It stored some of the numbers in a ”variable-length quantity” form, which was a 7-bit block VB structure (Association, 1996). Similar encoding types have existed for example in Apache Lucene (as vInt) and IBM DB2 database (as Variable Byte). Later, VB encoding was was to be found efficient in compressing integer lists. It was first experimented as a tool for compressing inverted index lists of word locations in documents (Scholer et al., 2002). That yielded excellent records, and since then many different approaches have been introduced.

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

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 strutures. 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, but it adds a very large overhead which removes the benefit of compression.

3.1 Rank and Select

Rank and select are two succinct data structure operations which operate on a bit array B. $Rank_1$(B, i) gives the sum of 1 bits between B[0] and B[i] and $select_1$(b, i) gives the index of i-th 1 bit in B. Both operations can be implemented to work in constant time (Gog et al., 2014). Locations of 1 bits in B should represent locations of the items 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. VB encoding has several advantages with search and rank: it's data is compressed in blocks of equal length which significantly shrinks the length of B. Also the bit array C formed from the continuation bits already stores the locations of items. In this case, $rank_1$(C, i) would give the sum of end bytes up to i-th index and $select_1$(C, i) would give the location of the ending byte of i-th compressed element.

3.2 Rank and Select implementation

The rank and select implementations used in this article are C++ libraries from (Gog et al., 2014). The library has several implementations of both rank and select as well as an implementaion of a bit array. Table 3.1 has *rank* and *select* size requirements of implementations used in this experiment. Both rank implementations have a constant

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

Structure	Extra size taken
Rank	25% of bit array
Rankv5	6.25% of bit array
Select	8.3% of bit array (on average)

space requirement, while select's needed size depends on the number of 1's in the data.

For every 512th bit in the array, rank stores the absolute value of 1's preceding that bit in a superblock. For every 64th bit in a superblock, a relative count is stored. With this setup, an rank can be instantly calculated for every 64th bit. To get rank(i), superblock[i/512] and it's relative value (i%512)/64 are fetched. Then 1 bits from (i-i%64) to i are counted and added to get the correct sum. For Rankv5, superblock size is 2048 and relative count is stored every 384th bit, which offers a tradeoff between performance and memory usage.

Select data structure works similarly to rank. Index location of every 4096th set bit is stored in the superblock and location of every 64th set bit is stored relative to the superblock. *Select*

3.3 DAC via rank

Directly addressable codes in VB was first introduced by (Brisaboa et al., 2009) in 2009. Their solution was to divide chunks in encoded bytes to separate arrays A_1 to A_n and then use $rank_0$ to get direct access. If i is the requested index, then $rank_0(A_k$, i) returns the number of elements preceding i that have continuation bit set at 0. In other words, the number of elements that continue to A_{k+1}. See Figure 3.1 for visualization of data.

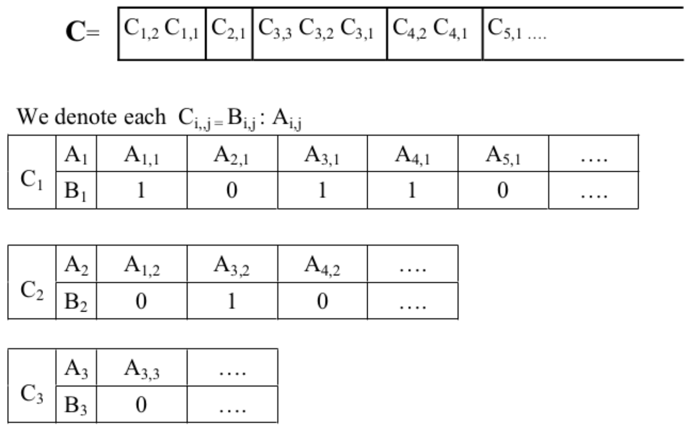

Figure 3.1: Data structure by Brisaboa et al., visualized

(TODO: create this with tikz)

For example, if an element needed 3 chunks when encoded, the least significant bits would go to A_1, second least significant bits to A_2 and the most significant bits would go to A_3. When fetching i-th element, getting the first chunk is just fetching A_1[i]. If the continuation bit is set at 0 (meaning there are more chunks to this element), getting the correct index for A_2 is calculated with $rank_0(A_1, \text{i})$. A pseudo code example of DAC with *rank* is shown in Figure 3.2 with block length of 8.

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

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 a few bit operations.

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

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

If the original element size is 32 bits and the data was compressed to k blocks of length b. The starting byte s of the element is *select*(i) * b div 8 where div is the integer division. Offset o is the remainder of previous division, o = x*b mod 8. The 32-bit word is read from memory from byte location s. Then bit shift left for 32-b*k-o is applied to remove trailing bits and bit shift right 32-b*k to re-align bits. If block size is one byte, offset calculation is not needed and thus the process is slightly faster.

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

$i \leftarrow$ wanted index
$A \leftarrow$ block byte array
$B \leftarrow$ continuation bit array
$begin \leftarrow \textsc{Select}(index)$
$len \leftarrow \textsc{CalculateLength}(begin)$
$mask \leftarrow \textsc{ByteMask}(len)$
$word \leftarrow A[begin]$ $\triangleright$ read a word from the byte array
$word \leftarrow word \mathbin{\&} mask$

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

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

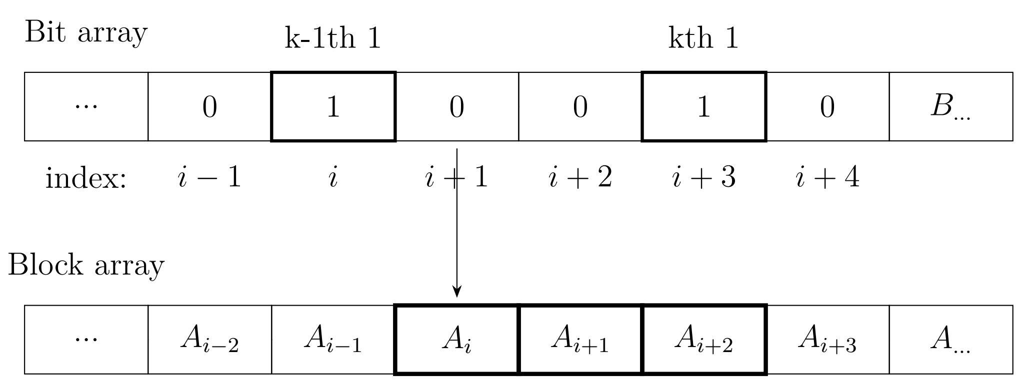

Figure 4.2: Data

TODO: try bitvector SD performance + size (select) (maybe write about different select implementations)

5 Experimental results

The following experiments are run on ¡at the moment my work Lenovo Thinkpad T480s, TODO run on a server and write specs here¿. The algorithms were implemented with C++ (TODO: list compile tags and whatnot here) using data structures and functions from (Gog et al., 2014).

Five different kinds of VB decoding algorithms were used. Bris-implementations are from (Brisaboa et al., 2009).

- Bris4v5 - Bris implementation with block size 4, using rank support v5
- Bris8 - Bris implementation with block size 8, using rank support v
- Bris8v5 - Bris implementation with block size 8, using rank support v5
- Our4 - Our implementation with block size 4, using select support mcl
- Our8 - Our implementation with block size 8, using select support mcl

Eight kinds of datasets were used. For each number in the dataset, p was randomly selected from P and then number was randomly selected from range $[0,2^p]$.

- all5M - 5 million numbers, P = 7,8,15,16,23,24,30,31
- all50M - 50 million numbers, P = 7,8,15,16,23,24,30,31
- byte5M - 5 million numbers, P = 7,7,7,8,8,8,16,31
- byte50M - 50 million numbers, P = 7,7,7,8,8,8,16,31
- small5M - 5 million numbers, P = 3,4,5,6,7,8,16,31
- small50M - 50 million numbers, P = 3,4,5,6,7,8,16,31
- vsmall5M - 5 million numbers, P = 2,2,3,3,3,4,4,15
- vsmall50M - 50 million numbers, P = 2,2,3,3,3,4,4,15

Table 5.1: Results in milliseconds, smaller is better.

Experiment	all5M	all50M	byte5M	byte50M	small5M	small50M	vsmall5M	vsmall50M
Bris4v5	348.41	768.83	345.16	772.84	344.54	768.45	346.86	771.61
Our4	127.92	263.22	127.44	263.5	127.9	262.67	127.88	262.58
Bris8	**97.75**	308.17	**98.47**	309.6	**98.14**	307.81	**97.99**	308.09
Bris8v5	127.83	287.46	127.7	288.63	128.14	291.32	127.99	287.84
Our8	103.85	**230.16**	103.92	**230.14**	103.86	**230.02**	103.95	**230.6**

All algorithms save the data to memory and randomize one million index numbers to an array. The time taken is calculated from looping through the index array and VB decoding the number in the index. The times shown in Table 5.1 are an average of 100 such runs.

- Run with 32 bit values, see what happens (some operations may be 32b?) - do cachegrind - - use array instead of vector¡uint8 t¿ for bris? - comparison to basic implementation + Bri09 - compare with b=2,4,8

- do we need to support search()? - SIMD?

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

6 Future work

- something to improve / research?

7 Conclusions

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

Bibliography

Association, T. M. M. (1996). "The Complete MIDI 1.0 Detailed Specification". In:

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

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

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

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

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.

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

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

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, Heidelberg.

Scholer, F., Williams, H. E., Yiannis, J., and Zobel, J. (2002). "Compression of Inverted Indexes For Fast Query Evaluation". In:

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.