Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 236 (2009) 131–145
www.elsevier.com/locate/entcs

# Rank and Select for Succinct Data Structures*

Antonio Fariña[1] Susana Ladra[2] Oscar Pedreira[3]
Ángeles S. Places[4]

*Database Lab., University of A Coruña, 15071 A Coruña, Spain*

**Abstract**

In this paper, we study different approaches for rank and select on sequences of bytes and propose new implementation strategies. Extensive experimental evaluation comparing the efficiency of the different alternatives are provided.
Given a sequence of bits, a rank query counts the number of occurrences of the bit 1 up to a given position, and a select query returns the position of the ith occurrence of the bit 1. These operations are widely used in information retrieval and management, being the base of several data structures and algorithms for text collections, graphs, etc.
There exist solutions for computing these operations on sequences of bits in constant time using additional information. However, new applications require rank and select to be computed on sequences of bytes instead of bits. The solutions for the binary case are not directly applicable to sequences of bytes. The existing solutions for the byte case vary in their space-time trade-off which can still be improved.

*Keywords:* information retrieval, algorithms, succinct data structures, rank, select

## 1 Introduction

Information management generally involves working with large collections of data from a variety of data types. An important issue in these applications is obtaining compact representations of information that also make possible its efficient processing. Succinct data structures aim at representing data (e.g., sets, trees, hash tables, graphs or texts) using as little space as possible while still being able to efficiently solve the required operations on the data. Self-indexes for text collections [12] and

* This work has been partially supported by "Ministerio de Educación y Ciencia" (PGE and FEDER) ref. TIN2006-16071-C03-03 and (FPU Program) ref. AP-2006-03214 (for O. Pedreira), and "Xunta de Galicia" ref. PGIDIT05SIN10502PR and ref. 2006/4.

[1] Email: fari@udc.es
[2] Email: sladra@udc.es
[3] Email: opedreira@udc.es
[4] Email: asplaces@udc.es


doi:10.1016/j.entcs.2009.03.019

compressed web graphs [1] are two representative examples of applications of succinct data structures. Binary sequences and the operations *rank* and *select* defined on them are the base of many succinct data structures:

> Given an offset inside a sequence of bits, *rank* counts the number of times the bit 0 (resp. 1) appears up to that position. *select* returns the position in that sequence where the $i$-th occurrence of bit 0 (resp. 1) takes place.

Full-text indexes are a good example in which the performance of these two operations is specially relevant [12]. The importance of these complementary operations for the performance of succinct data structures has motivated extensive research in this field [10]. Several strategies have been developed to efficiently compute *rank* and *select* when dealing with binary sequences. They are usually based on building auxiliary structures that lead to a more efficient management of the sequence. The strategies proposed in [9] and [5] compute *rank* and *select* in constant time. Some years later, [13] and [14] exploited the compressibility of binary sequences obtaining constant time *rank* and *select* implementations too, with smaller representations of the sequence. The goal pursued by these developments is the optimization of the trade-off between the efficiency of the *rank* and *select* operations and the space needed by the representation of the sequence.

New problems and applications require *rank* and *select* to be generalized to sequences of an arbitrary number of symbols instead of bits [6,10]. In this case, given a sequence of symbols $S = s_1 s_2 \dots s_n$, $\boldsymbol{rank}_s(S, i)$ returns the number of times the symbol $s$ appears in $S[1, i]$, and $\boldsymbol{select}_s(S, j)$ returns the position of $S$ containing the $j^{th}$ occurrence of the symbol $s$. The most typical example is the computation of *rank* and *select* in sequences of bytes instead of bits, needed, for example, in recent developments in text indexing.

The strategies used with binary sequences cannot be directly applied to the general case or, if applied, they may require a significant amount of memory. Thus, rather than directly applying those techniques, most of the approaches for the general case try to adapt them or to transform the problem in such a way that it can be reduced to using *rank* and *select* in binary sequences. This is the case of wavelet trees [6]. In this paper we present implementation issues of *rank* and *select* operations in byte sequences, showing that in some cases, using a straightforward sequential scan implementation and some implementation optimizations can improve the space/time trade-off obtained by other techniques that use additional structures. We also show that some of them can obtain good performance in *rank* but not in *select*, in which a direct implementation can have a better performance.

The rest of the paper is organized as follows. Section 2 reviews the strategies developed for the *rank* and *select* operations in binary sequences. In Section 3, the main proposals for the implementation of *rank* and *select* in the general case using additional structures and reducing the problem to the case of binary sequences are presented. Section 4 describes the different sequential implementation issues we have explored. Sections 6 and 7 present the experimental results and the conclusions of the paper respectively.

## 2 Bit-oriented Rank and Select

The *rank* and *select* operations were defined in [9], one of the first research works devoted to the development of succinct data structures. In [9], an implementation of *rank* and *select* that was able to compute *rank* in constant time was proposed. The author of [9] used that implementation as the basis of a compact and efficient implementation of binary trees. Given a binary sequence $B[1,n]$ of size $n$, a two-level directory structure is built. The first level stores $rank(i)$ for every $i$ multiple of $\lceil \log n \rceil^2$. The second level stores $rank'(j)$ for every $j$ multiple of $\lceil \log n \rceil$, where $rank'(j)$ computes *rank* within subsequences of size $\lceil \log n \rceil^2$. To compute $rank_1(B,i)$ we can use these two directory levels to obtain the number of times the bit 1 appears before the subsequence of size $\lceil \log n \rceil^2$ containing the position $i$. The same happens in the second level of the directory structure. The final result is obtained using *table lookups*. The bits of the subsequence of size $\lceil \log n \rceil$ containing the position $i$ that could not be processed with the information of the directories, are used as the index for a table which tells us the number of times bit 1 or 0 appears in them. Therefore *rank* can be computed in constant time. However, with this approach *select* is computed in $O(\log \log n)$, since it has to be implemented using binary searches. The space needed by these additional directory structures is $o(n)$.

Later works improved these results obtaining constant time implementations for *rank* and *select* [5,11]. A new directory structure organized in three levels was proposed in [5]. In this case, the first directory level stores the positions of every $\lceil \log n \rceil \lceil \log \log n \rceil$'th 1 bit. The second level stores the positions of bits set to 1 in the subranges corresponding to the first level, and the same happens with the third directory level with respect to the second one. With this more complex structure, [5] is able to compute the two operations in constant time requiring $O(n)$ additional space. Take into account that this implementation works for the operation $select_1$, and we would have to create the analogous for $select_0$ if needed (the representation of the sequence proposed by [13] is not complete [10]).

The solutions proposed by [9,5,11] are based on the idea of using additional data structures for efficiently computing *rank* and *select* without taking into account the content of the binary sequence and its statistical properties (number of 1 bits and their positions in the sequence). [13] and [14] explored a new approach working with compressed binary sequences which are also able to efficiently compute *rank* and *select*. [13] first explored the possibility of representing the sequence as a set of compressed blocks of the same size, each of them represented by the number of 1 bits it contains and the number corresponding to that particular subsequence. Since with this scheme the number of blocks grows almost linearly with the size of the sequence, [13] also proposed an interval compression scheme that clusters suitable adjacent blocks together into intervals of varying length.

The compressed representation of binary sequences proposed by [14] is based on a numbering scheme. The sequence is divided into a set of blocks, each of them represented by the number of 1 bits it contains and an identifier, in such a way those blocks with few 1 bits require shorter identifiers. This approximation obtains zero-order compression and is currently the best complete representation of binary

sequences [10] (that is, it supports access, *rank* and *select* in constant time for both 0 and 1 bits). [14] also shows how this binary sequence data structure can be used for the optimal representation of $k$-ary trees and multisets.

Another research line aims at compression of binary sequences when the number of 1 bits is small. The approach known as *gap encoding* obtains compressed representations of binary sequences encoding the gaps between consecutive 1 bits in the sequence. [15,8,10] present several developments and improvements for this approach, although we have to take into account that it is supposed that the number of 1 bits in the sequence is small. In other case, the proposals previously described usually perform better.

## 3 Byte-oriented Rank and Select Based on Bit-oriented Solutions

Although *rank* and *select* operations were initially defined over binary sequences, new developments and applications require these operations to be defined over sequences of symbols from an arbitrary alphabet. In this paper we focus on the problem of computing *rank* and *select* over sequences of bytes, which, for example, is a topic of interest in text indexing. In this more general case, the solutions described in the previous section for binary sequences cannot be valid or directly applied. This section describes the most important approximations to this problem based on the use of binary sequences: the use of *bitmaps*, and the use of Wavelet Trees.

*Constant time* rank *and* select *using* bitmaps

The easiest way to efficiently compute *rank* and *select* in byte sequences consists in using indicator *bitmaps* (binary sequences) for each byte [12]. For each position of the original byte sequence, only the bitmap corresponding to its byte has a 1 bit in that position. Therefore, as we can compute *rank* and *select* in binary sequences in constant time, we can also do it in the case of sequences of bytes. The price to pay for this efficient implementation is the space used by the bitmap for each byte and the necessary additional data structures for computing *rank* and *select* in constant time in each one of them. We will refer as 256-BM to this approach in the rest of the paper.

*Wavelet trees*

The wavelet tree was proposed in [6,7] and permits to efficiently compute *rank* and *select* in sequences of symbols from an arbitrary alphabet $\Sigma$ of size $n$. The wavelet tree is a balanced binary tree in which each node stores a bitmap. The tree is built as follows. The root is given a bitmap of the same size as the sequence of symbols. For each position, the bitmap is set to 0 if the symbol corresponding to that position belongs to the first half of the alphabet, and 1 in other case. The symbols labeled with a 0 are processed in the left child of the node, and those labeled with 1 are processed in the right child. Therefore, the child node has associated with the

first half of the alphabet and the right one with the second half. This process is repeated until the alphabet cannot be divided again and we reach the leaves of the tree. Figure 1 shows a simple example with a sequence of symbols from the alphabet $\Sigma = \{a, b, c, d\}$ (text is shown only for clarity, but it is not actually stored).

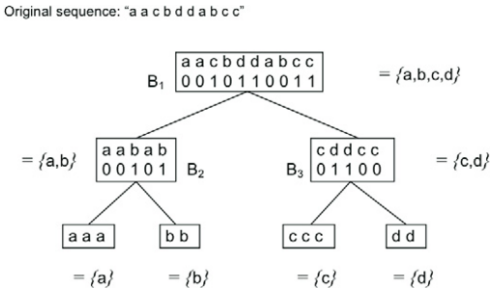


Fig. 1. Example of wavelet tree

Generalizing this definition of binary wavelet tree to the case of byte sequences ($\Sigma = \{0, \ldots, 255\}$) results in a balanced binary tree with eight levels. In the level $i$ of the tree, the bit in the bitmap for each byte is the $i$ bit of its binary representation, so in this case the tree is even easier to implement.

Suppose we want to compute $rank_c$ at the position $i$ in the sequence of bytes. We traverse the tree from the root to the leaf corresponding to $c$. By applying *rank* in the bitmap of each node we obtain the position in which *rank* is applied for the next level of the tree. For example, in the wavelet tree shown in Figure 1, as $a$ belongs to the first half of the alphabet, we compute $rank_a(B, 5)$ and move to the left child of the root. In the next level we compute $rank_0(B_2, 3) = 2$, so the answer to $rank_a(B, 5)$ is 2. To answer *select*, the tree is traversed from the leaf corresponding to the character to the root following the same idea. For example, if we want locate the $2^{nd}$ $d$, $select_d(B, 2)$, then we start in the leaf corresponding to $d$, then compute $select_1(B_3, 2) = 3$ and then $select_1(B_1, 3) = 6$, so the answer is 6.

# 4 Direct Implementation of Byte-oriented Rank and Select

The solutions described in the previous section for *rank* and *select* in byte sequences try to efficiently compute these operations avoiding a sequential scan of the sequence. However, we have identified some implementation issues that can make a sequential scan of the whole sequence, or only a part of it (if auxiliary structures are built), to be competitive. In some cases this *straightforward* or direct implementation can perform better in time, and in the space needed. In this section we describe this implementation issues when computing *rank* and *select* with a sequential scan.

*Straightforward implementation*

When computing *rank* and *select* with a sequential scan we have to compare each byte of the sequence and increment a counter if needed. The first issue we explored is

how to efficiently compute this comparison (pseudocode for $rank_c(B, i)$ is provided. *select* is implemented in the same way):

- Comparison using $if$. The first option for the comparison is the use of an $if$ sentence for the comparison.

  **Algorithm 1** *Using if*

  *FOR* $j \leftarrow 1$ *to* $i$
  *IF* $B_j = c$
  $count \leftarrow count + 1$
  *ENDIF*
  *ENDFOR*

- If with *skip loop*. We can replace the $if$ sentence with a *skip loop* condition. Experimental results show that this optimization improves the performance of the loop, when the number of occurrences of the selected byte $c$ is not very high.

  **Algorithm 2** *Skip loop*

  *FOR* $j \leftarrow 1$ *to* $i$
  *WHILE* $(B_j \neq c)$ *and* $(j \leq i)$
  $j \leftarrow j + 1$
  *ENDWHILE*
  $count \leftarrow count + 1$
  *ENDFOR*
  *IF* $B_j \neq c$
  $count \leftarrow count - 1$

- XOR. We can remove the $if$ sentence by adding to the counter the result of an XOR operation.

  **Algorithm 3** *XOR*

  *FOR* $j \leftarrow 1$ *to* $i$
  $count \leftarrow count + \neg(B_j \oplus c)$
  *ENDFOR*

- Table lookup. We can avoid the comparison by using a table with 256 entries, where for each byte $j$, we add to the counter the value of $table[B_j]$. Only the position corresponding to the byte $c$ has a value 1, the others 0.

  **Algorithm 4** *Table lookup*

  $table[c] \leftarrow 1$
  *FOR* $j \leftarrow 1$ *to* $i$
  $count \leftarrow count + table[B_j]$
  *ENDFOR*
  $table[c] \leftarrow 0$

A comparison of those four implementations is given in Table 1. The time (in $\mu secs.$) needed to count all the occurrences of a given byte value in a byte-array is shown. We take into account three groups of byte-values depending on their

frequency (Low, Med, and High). As it is explained in Section 6, RT and NLT are two sequences of bytes with around 249MB. They contain random bytes (RT) and ASCII data from natural language text (NLT). *IF skip-loop* seems to be the best choice in NLT when low frequency values are searched for, whereas *table-lookup* approach behaves better in other cases, outstanding also the most stable times.

| *count* | If (without skip-loop) | | If (skip-loop) | | xor | | Table lookup | |
|---|---|---|---|---|---|---|---|---|
| | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| Low Freq | 0.553 | 0.548 | 0.318 | 0.307 | 0.560 | 0.560 | 0.309 | 0.311 |
| Med Freq | 0.560 | 0.558 | 0.325 | 0.307 | 0.557 | 0.560 | 0.310 | 0.313 |
| High Freq | 0.566 | 0.660 | 0.325 | 0.538 | 0.560 | 0.562 | 0.314 | 0.312 |

Table 1
Comparison of the presented implementations to count the occurrences of a given byte value.

*Parallel implementation*

The previous strategies consider a sequential scan of the sequence one byte at a time. When computing *rank* and *select* in byte sequences we can read an integer in each iteration of the loop and process its four bytes. With this approach, the computational cost introduced by the loop is divided by four. We can do the comparison of each byte of the integer with the same strategies previously described. For example, if we want to use the "table" approach, the sentence to update the counter in the loop would be:

$$count \leftarrow count + table[byte_1] + table[byte_2] + table[byte_3] + table[byte_4]$$

and the same can be applied to the other alternative implementations.

Table 2 presents a comparison of the most efficient byte-parallel implementations against their simple counterparts. IF-based approach is now the best choice in most cases.

| *count* | byte-parallel | | | | | | simple | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | If | | Table lookup | | xor | | Table lookup | | If (skip-loop) | |
| | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| Low Freq | 0.182 | 0.171 | 0.222 | 0.222 | 0.235 | 0.226 | 0.309 | 0.311 | 0.318 | 0.307 |
| Med Freq | 0.189 | 0.168 | 0.222 | 0.222 | 0.234 | 0.225 | 0.310 | 0.313 | 0.325 | 0.307 |
| High Freq | 0.206 | 0.436 | 0.222 | 0.222 | 0.246 | 0.432 | 0.314 | 0.312 | 0.325 | 0.538 |

Table 2
Comparison of the byte-parallel implementations.

## 5 Byte-oriented Rank and Select using Block Structures

All the previous techniques described in this section have a linear complexity with the size of the sequence since they are based in a sequential scan. As we explained

in Section 2, constant time in the case of binary sequences can be obtained using a two level directory structure (blocks and superblocks). It can be adapted to the case of sequences of bytes. First, we use only a directory level, described in Section 5.1. In Section 5.2, we optimize the first approach using a two-level structure.

### *5.1 Single Block Structure*

Straightforward implementations can be improved by storing at given intervals absolute counters of the number of times each byte appears before that position. With this approach we do not compute *rank* and *select* in constant time, but we have to perform a sequential scan only in a small portion of the sequence. This permits us to easily adjust the space/time trade-off by just changing the size of the intervals.

Given a sequence of bytes $B[1,n]$, we use a one-level directory structure, dividing the sequence into $b$ blocks. Each block stores the number of occurrences of each byte from the beginning of the sequence to the start of that block.

With this approach, $rank_{b_i}(B,j)$ is obtained by counting the number of occurrences of $b_i$ from the beginning of the last block before $j$ up to the position $j$, and adding to that the value stored in the corresponding block for byte $b_i$. Instead of $O(n)$, this structure answers *rank* in time $O(n/b)$. To compute $select_{b_i}(B,j)$ we binary search the stored values in the blocks for the first value $x$ such that $rank_{b_i}(B,x) = j$, and complete the search with a sequential scanning in that block. The time is $O(\log b + n/b)$.

### *5.2 Two-level Block Structure*

Given a sequence of bytes $B[1,n]$, we use a two-level directory structure, dividing the sequence into $sb$ superblocks and each superblock into $b$ blocks of size $n/(sb*b)$. The first level stores the number of occurrences of each byte from the beginning of the sequence to the start of each superblock. The second level stores the number of occurrences of each byte up to the start of each block from the beginning of the superblock it belongs to. The second-level values cannot be larger than $sb*b$, and hence can be represented with fewer bits.

With this approach, $rank_{b_i}(B,j)$ is obtained by counting the number of occurrences of $b_i$ from the beginning of the last block before $j$ up to the position $j$, and adding to that the values stored in the corresponding block and superblock for byte $b_i$. Instead of $O(n)$, this structure answers *rank* in time $O(n/(sb*b))$. To compute $select_{b_i}(B,j)$ we binary search for the first value $x$ such that $rank_{b_i}(B,x) = j$. We first binary search the stored values in the superblocks, then those in the blocks inside the right superblock, and finally complete the search with a sequential scanning in the right block. The time is $O(\log sb + \log b + n/(sb*b))$.

An interesting property is that this structure is parameterizable. That is, there is a space/time tradeoff associated to parameters $sb$ and $b$. The shorter the blocks, the faster the sequential counting of occurrences of byte $b_i$.

# 6 Empirical Results

We have tested our developments over two large byte-arrays with both real and synthetic data. As real data we chose the AP Newswire 1998 corpus (AP) from TREC-2 [5] collection. AP corpus consists of many news in XML form, and contains 250,634,186 bytes which are mainly natural language text (NLT). As expected, those bytes from AP corpus follow a very biased distribution of frequency, as some of them appear many times (i.e. the BLANK) and others (around 100 byte values) do not appear at all. As synthetic data, we also generated a byte-array of the same size as AP corpus that consists of random bytes following a uniform distribution. We will refer to this data as *RT* in advance.

Our results compare the efficiency of six different approaches to compute byte-oriented rank and select operations:

(i) *base**: which traverses the byte-array sequentially (using the IF-based approach).

(ii) *WT**: which uses a binary wavelet tree without any kind of super-blocks to rapidly compute binary rank and select operations.

(iii) *WT(sb)*: similar to the previous technique but using super-blocks and blocks following the idea in [9].

(iv) *base(b)*: the optimization of *base* technique that is based on keeping 256-counters for given sampled offsets of the byte-array (blocks).

(v) *base(sb)*: which is an optimization of the previous technique, using a two-level structure of counters (blocks and superblocks).

(vi) *256-BM*: which aims at performing byte-oriented rank and select directly by handling 256 bitmaps (one indexing each type of byte).

As expected, the more memory available to keep blocks the more efficient *base(b)* and *base(sb)* become. In practice, we used for *base(b)* and *base(sb)* as much additional memory as *WT(sb)* needs. However, results showing the effects of the amount of memory available to hold blocks on the efficiency of *base(b)* and *base(sb)* approaches are given at the end of Section 6.

For each technique, we present the time needed to answer *count*, *rank*, *select* and *access* operations on the NLT and RT byte-arrays, for three different groups of bytes: *Low frequency bytes* (LFq), *Medium frecuency bytes* (MFq) and *High frequency bytes* (HFq). *Count* operation was performed over the whole sample byte-arrays for LFq, MFq, and HFq groups, whereas *rank* operation was applied for each group over three fixed offsets of the byte-arrays. Those offsets correspond to the bytes in positions $\frac{1}{4}x$, $\frac{1}{2}x$, and $\frac{3}{4}x$, where $x$ is the size of the whole byte-array. In the case of *select* operations we focused in the cost of performing both $select_c(1)$, and $select_c(\frac{3}{4}y)$, where $y$ is the number of occurrences of byte $c$. We also compute *access* average times for random positions of the byte-array.

An isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache),

[5] http://trec.nist.gov.

with 4 GB dual-channel DDR-400Mhz RAM was used in our tests. It ran Debian GNU/Linux (kernel v2.4.27). The compiler used was gcc v3.3.5 and `-O9` compiler optimizations were set. Time results measure CPU user time in microseconds.

*Experimental results for* count *operation*

Table 3 presents *count* results. As expected, the first two techniques obtain very poor results. Specially in the case of LFq byte-values, using a wavelet tree as in *WT** is still a better idea than performing a sequential count through the whole byte-array as in *base**. It is also noticeable that even though in our RT byte-array measured times seem to change only slightly depending on their frequency, in NLT the least frequent byte-values occur so rarely (only once), that they can be found very rapidly in *WT**. Among the four more optimized techniques, *256-BM* takes advantage of using a large amount of memory and becomes around 8 times faster than *WT(sb)*. Anyway, *WT(sb)* is still very fast as it only has to perform a binary *rank* on the leaf containing the searched byte-value. Finally, *base(b)* and *base(sb)* are also able to count the occurrences of any byte-value in less than 2 $\mu secs$ and 1 $\mu secs$ respectively, but they are slower than *WT(sb)* technique. Notice also that *base(b)* and *base(sb)* worsen as the frequency of the searched byte-value increases, whereas *256-BM* and *WT(sb)* remain constant in practice.

| *count* | base* | | WT* | | WT (sb) | | 256-BM | | base (b) | | base (sb) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| LFq | 186472 | 172674 | 17319 | 0.020 | 0.120 | 0.120 | 0.015 | 0.015 | 0.904 | 0.873 | 0.429 | 0.409 |
| MFq | 192871 | 175773 | 17231 | 45.743 | 0.120 | 0.120 | 0.015 | 0.015 | 1.028 | 0.875 | 0.488 | 0.409 |
| HFq | 193671 | 301054 | 16831 | 138749.907 | 0.120 | 0.110 | 0.015 | 0.015 | 1.118 | 2.012 | 0.527 | 0.948 |

Table 3
Time for count operation (in $\mu secs$).

*Experimental results for* rank *operation*

Results regarding *rank operation* are given in Table 4. The *base** approach becomes faster than WT*. Better results are obtained when low frequency byte values are searched for and when rank is applied to a smaller offset of the byte-array (less bytes have to be traversed). *256-BM* and *WT(sb)* obtain exactly the same constant times shown for *count* scenario. In the case of *base(b)* and *base(sb)*, results are still worse than those of the two previous techniques. Assuming that $rank_c(i)$ is being computed, these results depend basically on the number of bytes that have to be traversed from the previous block before $i$; that is, they depend on the gap from the previous block ($i \bmod samplePeriod$). More precisely the *samplePeriod* is the size of each block. For example for base(b), the *samplePeriod* was 2788, those gaps are 1036, 2072, and 320 bytes respectively for the percentages 25, 50, and 75 shown in Table 4 for *base(b)*.

| *rank* | % | base* | | WT* | | WT (sb) | | 256-BM | | base (b) | | base (sb) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| LFq | 25 | 46093 | 42694 | 12558490 | 6790967 | 0.120 | 0.120 | 0.015 | 0.015 | 0.713 | 0.685 | 0.147 | 0.283 |
| | 50 | 91586 | 85287 | 25896063 | 13527943 | 0.120 | 0.120 | 0.015 | 0.015 | 1.355 | 1.328 | 0.234 | 0.516 |
| | 75 | 138779 | 128381 | 39129052 | 20841831 | 0.120 | 0.120 | 0.015 | 0.015 | 0.263 | 0.247 | 0.379 | 0.198 |
| MFq | 25 | 47993 | 44393 | 12585087 | 6953943 | 0.120 | 0.130 | 0.015 | 0.015 | 0.795 | 0.687 | 0.164 | 0.266 |
| | 50 | 96385 | 85387 | 25143178 | 14173846 | 0.120 | 0.120 | 0.015 | 0.015 | 1.491 | 1.323 | 0.258 | 0.508 |
| | 75 | 143978 | 128481 | 37792255 | 21900671 | 0.130 | 0.120 | 0.015 | 0.015 | 0.264 | 0.251 | 0.464 | 0.207 |
| HFq | 25 | 47993 | 208368 | 12554092 | 7552851 | 0.120 | 0.130 | 0.015 | 0.015 | 0.728 | 2.560 | 0.146 | 1.058 |
| | 50 | 95685 | 419436 | 24964205 | 15257680 | 0.120 | 0.120 | 0.015 | 0.015 | 1.509 | 5.716 | 0.261 | 2.219 |
| | 75 | 143778 | 629504 | 37499300 | 22826529 | 0.120 | 0.120 | 0.015 | 0.015 | 0.250 | 0.328 | 0.713 | 0.608 |

Table 4
Time for rank operation (in $\mu secs$).

*Experimental results for* select *operation*

Things change when $select_x(1)$ is computed; that is, when we aim at obtaining the offset where the first occurrence of $x$ appears. Results are shown in Table 5. Except for MFq byte-values in NLT, both the simple *base** and *base(b)* or *base(sb)* become faster than *WT** and *WT(sb)* respectively. This occurs because in *base* approach $select_c(1)$ needs only a fast binary search, that is much faster than performing a down-top traversal of the wavelet-tree (and computing 8 *binary selects*). The main advantage of *WT(sb)* is that it obtains almost constant times that are independent of the frequency of $c$ value, and also independent of the offset where the first occurrence of $c$ appears.

| $select_c(1)$ | base* | | WT* | | WT (sb) | | 256-BM | | base (b) | | base (sb) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| LFq | 0.489 | 114453.400 | 294.095 | 9449563.002 | 2.208 | 2.151 | 0.069 | 0.098 | 0.531 | 1.348 | 0.528 | 0.533 |
| MFq | 0.443 | 354.111 | 47.283 | 38344.170 | 2.192 | 2.190 | 0.077 | 0.075 | 0.475 | 1.836 | 0.467 | 0.613 |
| HFq | 0.303 | 1.780 | 34.725 | 12.778 | 2.243 | 2.111 | 0.078 | 0.079 | 0.329 | 0.212 | 0.325 | 0.224 |

Table 5
Time for $select_c(1)$ operation (in $\mu secs$).

Focusing on $select_c(y)$ operation, *WT(sb)* put up again a good show (see Table 6), obtaining practically the same results as in $select_c(1)$. However, the *base(sb)* improves also its performance in all cases. It is also noticeable that *base** becomes a much faster choice than *WT** when $select_c(y)$ is to be performed. In this scenario *WT** works too inefficiently to be chosen as a useful alternative. As expected, the *256-BM* technique is again the faster choice to obtain $select_c(1)$ and $select_c(y)$.

*Experimental results for* access *operation*

We have only evaluated *access* operation using *WT(sb)* and *base(sb)* techniques. *WT(sb)* returns the byte at a given position in 194.2 ns, whereas *base(sb)* averages 4.2 ns to compute the same operation. *WT(sb)* is clearly slower than *base(sb)* because it performs a down-top traversal of the wavelet-tree (computing *binary*

| $select_c(y)$ | base* | | WT* | | WT (sb) | | 256-BM | | base (b) | | base (sb) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 75% | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT | RT | NLT |
| LFq | 2541779 | 1064092 | 42633518 | 9518553 | 2.301 | 2.148 | 0.101 | 0.084 | 1.876 | 1.348 | 1.087 | 0.534 |
| MFq | 2115839 | 1200780 | 38960077 | 23557419 | 2.219 | 2.236 | 0.079 | 0.068 | 1.618 | 1.370 | 0.548 | 0.613 |
| HFq | 2697361 | 1392936 | 38980075 | 24170326 | 2.275 | 2.185 | 0.082 | 0.059 | 1.324 | 1.928 | 0.785 | 0.649 |

Table 6
Time for $select_c(y)$ operation (in $\mu secs$).

*selects*). *base(sb)* stores the original sequence as an array, so *access* operation can be trivially implementated.

*A brief recap of the experimental evaluation*

Results seem to show up *base(b)* and specially *base(sb)* as two interesting alternatives to the use of binary wavelet-trees *WT(sb)* when rank, select and access operations need to be computed over sequences of bytes. Even though *base(sb)* obtains worse results than *WT(sb)* for rank operation it is faster than *WT(sb)* for computing *select* and *access*. Even though *256-BM* obtained the best results, it is important to take into account that, in very large byte-sequences, the amount of memory needed to keep those 256 bitmaps could be so huge that *256-BM* might have been penalized because of swapping. In such case the results obtained by the *256-BM* approach would still obtain good CPU user-time, but elapsed-time would worsen a lot.

### *6.1 Memory Usage and Efficiency using Block Structures*

*Analysis for base(b) technique*

Since each block contains 256 counters (4 bytes each), so each block wastes 1024 bytes. It is possible to easily modify the number of blocks used for *base(b)* technique. As expected, the more blocks are used the more efficient *base(b)* becomes.

In Table 7 we set different numbers of blocks such that *base(b)* uses around 1%, 10%, 20%, 50%, and 100% the amount of memory allocated for the blocks and super-blocks needed by the *WT(sb)* technique. The exact amount of memory used is shown in the second row. The third row in that table gives the number of blocks used, and the fourth the number of bytes that are covered by each of such blocks. Rows from the fifth to the eighth show respectively the time (in $\mu secs$.) needed to compute *count*, *rank*, $select_c(1)$ and $select_c(y)$ in RT sequence of bytes. Results for *count* and *rank* depend respectively, on the distance from the last block to end of the byte-array, and on the distance between the previous block and the ranked offset. Those gaps are shown in the last two rows of Table 7. Results for $select_c(1)$ are almost constant if medium frequency byte values are searched for. However, results for $select_c(y)$ show that trading space for efficiency is possible and leads us to an interesting speed-up as more memory is available.

| % mem used by WT(sb) | 1% | 10% | 20% | 50% | 100% |
|---|---|---|---|---|---|
| Mem. usage (KB) | 901 | 9171 | 18350 | 45973 | 90943 |
| Number of blocks | 900 | 9170 | 18349 | 45972 | 89898 |
| bytes covered by block | 278484 | 27332 | 13660 | 5452 | 2788 |
| count all | 198.723 | 19.906 | 0.423 | 0.248 | 0.956 |
| rank | 98.475 | 9.441 | 3.909 | 1.567 | 0.812 |
| $select_c(1)$ MFq, | 0.193 | 0.192 | 0.203 | 0.192 | 1.836 |
| $select_c(y)$ MFq | 106.455 | 17.276 | 7.401 | 0.224 | 1.370 |
| gap count (bytes) | 277072 | 27080 | 508 | 296 | 1352 |
| gap rank (bytes) | 139242 | 13666 | 6830 | 2726 | 1394 |

Table 7
Trade-off efficiency vs memory usage for *base(b)* technique in RT byte-array.

*Analysis for base(sb) technique*

Analogously, we can choose values for *sb* (number of superblocks) and for *b* (number of blocks in a superblock) in *base(sb)* technique, in order to increase the speed of *rank* and *select* operations, or to reduce the amount of memory allocated.

Figure 2 shows different memory requirements for *select* operations using *base(sb)* technique. As the amount of memory grows, it becomes faster. Different values for *b* are chosen. Notice that it some values are not possible, because the block counters, represented with few bits, can be overflowed.

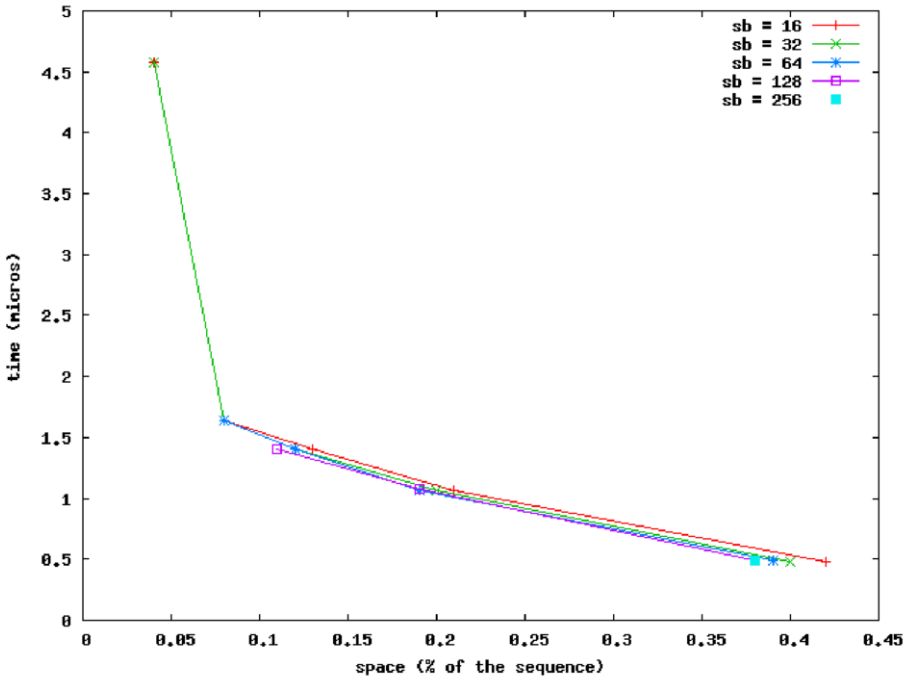


Fig. 2. Trade-off efficiency vs memory usage for *base(sb)* technique in *select* operations over NLT byte-array

# 7 Conclusions and Future Work

In this paper, we have targeted at different possible choices to tackle the problem of obtaining byte-oriented rank and select operations over sequences of bytes. We presented our experiences on developing and implementing six different alternative approaches. Firstly, we showed the simplest choice, which consists on sequentially processing the sequence of bytes from the beginning and counting the number of occurrences of a byte-value until a given offset (rank) or until a given number of occurrences is reached (select). Several alternatives to count those occurrences where presented in Section 4. Although IF-alternative seemed to be the most efficient approach, obtains results that do not depend on the number of occurrences of the byte-value searched for. It was shown that just by representing a byte-array as an integer-array (or more generally, a machine-word-array), permits us to use faster byte-parallel rank and select operations. More precisely, processing times can be reduced to the half.

Traditional approaches such as *WT(sb)* and *256-BM* were discussed and implemented. *256-BM* obtained the best performance for rank and select operations, but it requires a huge amount of memory. *WT(sb)* showed up also as a fast alternative to perform rank and select. However, our simple *base(sb)* alternative overcome the results obtained by *WT(sb)* (using the same amount of memory) for computing select and access operations. Given those results we also showed the interesting trade-off between space and efficiency that can be obtained depending on the number of blocks and superblocks used to index a byte-sequence.

We applied the two-level directory structure presented in this paper in the implementation of the self-index presented in [2]. This self-index is based on the construction of a byte-oriented wavelet-tree that is applied to index text compressed with either any semistatic byte-oriented word-based compressor [4,3]. Being byte-oriented, this new self-index requires the use of byte-oriented rank and select operations. The use of the directory structures for computing *rank* and *select* have an important impact on the index efficiency.

# References

[1] Boldi P., Vigna S.: The webgraph framework I: compression techniques In: Proc. of the 13th international conference on World Wide Web (WWW). (2004) 595-602.

[2] Brisaboa, N., Fariña, A., Ladra, S., Navarro, G.: Reorganizing Compressed Text In: Proc. of International ACM SIGIR Conference 2008 (SIGIR 2008). To appear.

[3] Brisaboa, N., Fariña, A., Navarro, G., Paramá, J.: Lightweight natural language text compression. Information Retrieval **10**(1) (2007) 1–33

[4] Brisaboa, N., Iglesias, E., Navarro, G., Paramá, J.: An efficient compression code for text databases. In: Proc. of the 25th European Conference on IR Research (ECIR'03). Number 2633 in Lecture Notes in Computer Science, Springer (2003) 468–481

[5] Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo (1996)

[6] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA'03), ACM Press (2003)

[7] Grossi, R., Gupta, A., Vitter, J.S.: When indexing equals compression: Experiments with compressing suffix arrays and applications. In: Proc. 15th Annual ACM Symposium on Discrete Algorithms (SODA). (2004) 636–645

[8] Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: Compressed data structures: Dictionaries and data-aware measures. In: Proc. of the 2006 IEEE Data Compression Conference (DCC '06). (2006)

[9] Jacobson, G.: Succinct static data structures. PhD thesis, Carnegie Mellon University (1989)

[10] Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science (2006) Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.

[11] Munro, I.: Tables. In: Proc. 16th Foundations of Software Technology and Theoretical Computer Science. Number 1180 in Lecture Notes in Computer Science, Springer (1996)

[12] Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys **39**(1) (2007)

[13] Pagh, R.: Low redundancy in static dictionaries with o(1) worst case lookup time. In: Proc. of 26-th International Colloquium on Automata, Languages, and Programming (ICALP'99). Number 1644 in Lecture Notes in Computer Science, Springer (1999)

[14] Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA'02), ACM Press (2002)

[15] Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms **48** (2003) 294–313