A Practical Alphabet-Partitioning Rank/Select Data Structure*

Diego Arroyuelo[1,2] and Erick Sepúlveda[2]
darroyue@inf.utfsm.cl, erick.sepulvedav@gmail.com

[1] Millennium Institute for Foundational Research on Data (IMFD), Chile.
[2] Department of Informatics, Universidad Técnica Federico Santa María, Chile. Vicuña Mackenna 3939, Santiago, Chile

Abstract. This paper proposes a practical implementation of an *alphabet-partitioning* compressed data structure, which represents a string within compressed space and supports the fundamental operations rank and select efficiently. We show experimental results that indicate that our implementation outperforms the current realizations of the alphabet-partitioning approach (which is one of the most efficient approaches in practice). In particular, the time for operation select can be reduced by about 80%, using only 11% more space than current alphabet-partitioning schemes. We also show the impact of our data structure on several applications, like the intersection of inverted lists (where improvements of up to 60% are achieved, using only 2% of extra space), and the distributed-computation processing of rank and select operations. As far as we know, this is the first study about the support of rank/select operations on a distributed-computing environment.

1 Introduction

Given a string $s[1..n]$, over an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$, operation $s.\mathsf{rank}_c(i)$ computes the number of occurrences of symbol $c \in \Sigma$ in $s[1..i]$. Operation $s.\mathsf{select}_c(j)$, on the other hand, yields the position of the j-th occurrence of symbol c in s. Finally, operation $s.\mathsf{access}(i)$ yields symbol $s[i]$.

These operations are fundamental for many applications [?], such as snippet extraction in text databases [?], query processing in information retrieval [?,?], cardinal trees, text search, and graph representation [?], among others.

Since the amount of data managed by these applications is usually large, *space-efficient* data structures to support these operations are vital [?]. *Succinct data structures* use space close to the information theory minimum, while supporting operations efficiently. *Compressed data structures*, on the other hand, take advantage of certain regularities in the data to further reduce the space usage. These are the focus of this paper. In particular, we propose a surprisingly simple and practical implementation of the *alphabet-partition* approach [?] for compressing a string while supporting operations rank, select, and access. We

* Funded by the Millennium Institute for Foundational Research on Data (IMFD).

show that our data structure introduces interesting trade-offs for supporting these operations. Also, we show how our data structure impacts several important applications, and show proof-of-concept experiments on the distributed support of rank and select. This is the first such study we are aware of.

2 Related Work

2.1 Succinct Data Structures for Bit Vectors

In this paper we will need to use a succinct data structure to represent bit vectors $B[1..n]$ with m `1` bits, and support operations rank and select. In particular, we are interested in the `SDarray` data structure from Okanohara and Sadakane [?], which uses $m \lg \frac{n}{m} + 2m + o(m)$ bits of space, and supports select in $O(1)$ time (provided we replace the rank/select bit vector data structure on which `SDarray` is based by a constant-time select data structure [?]). Operations rank and access are supported in $O\left(\lg \frac{n}{m}\right)$ time.

2.2 Compressed Data Structures

A *compressed data structure* uses space proportional to some compression measure of the data, e.g., the 0-th order empirical entropy of a string $s[1..n]$ over an alphabet of size σ, which is denoted by $H_0(s)$ and defined as:

$$H_0(s) = \sum_{c \in \Sigma} \frac{n_c}{n} \lg \frac{n}{n_c}, \tag{1}$$

where n_c is the number of occurrences of symbol c in s. The sum includes only those symbols c that do occur in s, so that $n_c > 0$. The value $H_0(s) \leq \lg \sigma$ is the average number of bits needed to encode each string symbol, provided we encode them using $\lg \frac{n}{n_c}$ bits.

2.3 Rank/Select Data Structures on Strings

Wavelet Trees A *wavelet tree* [?] (WT for short) is a succinct data structure that supports rank and select operations, among many virtues [?]. The space requirement is $n \lg \sigma + o(n \lg \sigma)$ bits [?], while operations rank, select, and access take $O(\lg \sigma)$ time. To achieve compressed space, the WT can be given the shape of the Huffman tree, obtaining $nH_0(s) + o(nH_0(s)) + O(n)$ bits of space. Operations take $O(\lg n)$ worst-case, or $O(1 + H_0(s))$ on average [?]. Alternatively, one can use compressed bit vectors [?] to represent each WT node. The space usage is $nH_0(s) + o(n \lg \sigma)$ bits, and operations take $O(\lg \sigma)$ time.

The approach by Ferragina et al. [?], which is based on multiary WTs, supports the operations in $O(1 + \frac{\lg \sigma}{\lg \lg n})$ worst-case time, and the space usage is $nH_0(S) + o(n \lg \sigma)$ bits. Later, Golynski et al. [?] improved the (lower-order term of the) space usage to $nH_0(s) + o(n)$ bits. Notice that if $\sigma = O(\text{polylog}(n))$, these solutions allow one to compute the operations in $O(1)$ time.

Reducing to Permutations Golynski et al. [?] introduce an approach that is more effective than the previous ones for larger alphabets. Their solution requires $n(\lg \sigma + o(\lg \sigma))$ bits of space, supporting operation rank in $O(\lg\lg\sigma)$ time, whereas operations select and access are supported in $O(1)$ time (among other trade-offs, see the original paper for details). Later, Grossi et al. [?] achieve higher-order compression, that is $nH_k(s) + o(n \lg \sigma)$ bits. Operations rank and select are supported in $O(\lg\lg\sigma)$, whereas access is supported in $O(1)$ time.

Alphabet Partitioning We are particularly interested in this paper in the alphabet-partitioning approach [?]. Given an alphabet $\Sigma = \{0, \ldots, \sigma-1\}$, the aim of *alphabet partitioning* is to divide Σ into p subalphabets $\Sigma_0, \Sigma_1, \ldots, \Sigma_{p-1}$, such that $\bigcup_{i=0}^{p-1} \Sigma_i = \Sigma$, and $\Sigma_i \cap \Sigma_j = \emptyset$ for all $i \neq j$.

The Mapping from Alphabet to Subalphabet. The data structure [?] consists of an alphabet mapping $m[1..\sigma]$ such that $m[i] = j$ iff i has been mapped to subalphabet Σ_j. Within Σ_j, symbols are re-enumerated from 0 to $|\Sigma_j| - 1$ as follows: if there are k symbols smaller than i that have been mapped to Σ_j, then i is encoded as k in Σ_j. Formally, $k = m.\mathsf{rank}_j(i)$. Let $n_j = |\{i,\ \ m[s[i]] = j\}|$ be the number of symbols of string s that have been mapped to subalphabet Σ_j. A way of defining the partitioning (which is called `sparse` [?]) is:

$$m[\alpha] = \left\lceil \lg\left(\frac{n}{n_\alpha}\right) \lg n \right\rceil, \tag{2}$$

where symbol $\alpha \in \Sigma$ occurs n_α times in s. Notice that $m[\alpha] \leq \lceil \lg^2 n \rceil$.

The Subalphabet Strings. For each subalphabet Σ_ℓ, we store the subsequence $s_\ell[1..n_\ell]$, with the symbols of the original string s that have been mapped to subalphabet Σ_ℓ.

The Mapping from String Symbols to Subalphabets. In order to retain the original string, we store a sequence $t[1..n]$, which maps every symbol $s[i]$ into the corresponding subalphabet. That is, $t[i] = m[s[i]]$. If $\ell = t[i]$, then the corresponding symbol $s[i]$ has been mapped to subalphabet Σ_ℓ, and has been stored at position $t.\mathsf{rank}_\ell(i)$ in s_ℓ. Also, symbol $s[i]$ in Σ corresponds to symbol $m.\mathsf{rank}_\ell(s[i])$ in Σ_ℓ. Thus, we have $s_\ell[t.\mathsf{rank}_\ell(i)] = m.\mathsf{rank}_\ell(s[i])$.

Notice that t has alphabet of size p. Also, there are n_0 occurrences of symbol 0 in t, n_1 occurrences of symbol 1, and so on. Hence, we define:

$$H_0(t) = \sum_{i=0}^{p-1} \frac{n_i}{n} \lg \frac{n}{n_i}. \tag{3}$$

Computing the Operations. One can compute the desired operations as follows, assuming that m, t, and the sequences s_ℓ have been represented using appropriate rank/select data structures (details about this later). For $\alpha \in \Sigma$, let $\ell = m.\mathsf{access}(\alpha)$ and $c = m.\mathsf{rank}_\ell(\alpha)$. Hence,

$$s.\mathsf{rank}_\alpha(i) \equiv s_\ell.\mathsf{rank}_c(t.\mathsf{rank}_\ell(i)),$$

and

$$s.\mathsf{select}_{\alpha}(j) \equiv t.\mathsf{select}_{\ell}(s_{\ell}.\mathsf{select}_{c}(j)).$$

If we now define $\ell = t[i]$, then we have

$$s.\mathsf{access}(i) \equiv m.\mathsf{select}_{\ell}(s_{\ell}.\mathsf{access}(t.\mathsf{rank}_{\ell}(i))).$$

Space Usage and Operation Times. Barbay et al. [?] have shown that $nH_0(t) + \sum_{i=0}^{p-1} n_\ell \lg \sigma_\ell \leq nH_0(s) + o(n)$. This means that if we use a zero-order compressed rank/select data structure for t, and then represent every s_ℓ even in uncompressed form, we obtain zero-order compression for the input string s. Recall that $p \leq \lceil \lg^2 n \rceil$, hence the alphabets of t and m are poly-logarithmic. Thus, a multi-ary wavelet tree [?] is used for t and m, obtaining $O(1)$ time for rank, select, and access. The space usage is $nH_0(t) + o(n)$ bits for t, and $O\left(\frac{n \lg\lg n}{\lg n}\right) H_0(s) = o(n)H_0(s)$ bits for m. For s_ℓ, if we use Golynski et al. data structure [?] we obtain a space usage of $n_\ell \lg \sigma_\ell + O\left(\frac{n_\ell \lg \sigma_\ell}{\lg\lg\lg n}\right)$ bits per partition, and support operation select in $O(1)$ time for s_ℓ, whereas rank and access are supported in $O(\lg\lg\sigma)$ time for s_ℓ.

Overall, the space is $nH_0(s)+o(n)(H_0(s)+1)$ bits, operation select is supported in $O(1)$ time, whereas operations rank and access on the input string s are supported in $O(\lg\lg\sigma)$ time (see [?] for details and further trade-offs).

Practical Considerations. In practice, the `sparse` partitioning defined in Equation (2) is replaced by an scheme such that for any $\alpha \in \Sigma$, $m[\alpha] = \lfloor \lg r(\alpha) \rfloor$. Here $r(\alpha)$ denotes the ranking of symbol α according to its frequency (that is, the most-frequent symbol has ranking 1, and the least-frequent one has ranking σ). Thus, the first partition contains only one symbol (the most-frequent one), the second partition contains two symbols, the third contains four symbols, and so on. Hence, there are $p = \lfloor \lg \sigma \rfloor$ partitions. This approach is called `dense` [?]. Another practical consideration is to have a parameter ℓ_{min} for `dense`, such that the top-$2^{\ell_{min}}$ symbols in the ranking are represented directly in t. That is, they are not represented in any partition. Notice that the original `dense` partitioning can be achieved by setting $\ell_{min} = 1$.

3 A Practical Alphabet-Partitioning Rank/Select Data Structure

The alphabet-partitioning approach was originally devised to speed-up decompression [?]. Barbay et al. [?] showed that alphabet partitioning is also effective for supporting operations rank and select on strings, being also one of the most competitive approaches in practice. Next we introduce an alternative implementation of alphabet partitioning, which is able to trade operation-access efficiency for rank/select efficiency.

The main idea is as follows: in the original proposal, mapping t (introduced in Section 2.3) is represented with a multiary wavelet tree [?], supporting rank, select, and access in $O(1)$ time, since t has alphabet of size $O(\text{polylog}(n))$. However, as far as we know, there is no efficient implementation of multiary wavelet trees in practice. Indeed, Barbay et al. [?] use a WT in their experiments, whereas the `sdsl` library [?] uses a Huffman-shaped WT by default for t. We propose an implementation of the alphabet-partitioning approach that is faster in practice: rather than having a global mapping t, we distribute the workload among partitions. This shall allow us to use a simpler and faster approach (for instance, a single bit vector per partition) that replaces t.

3.1 Data Structure Definition

Our scheme consists of the mapping m and the subalphabets subsequences s_ℓ for each partition ℓ, just as originally defined in Section 2.3. In our case, however, we disregard mapping t, and replace it by a bit vector $B_\ell[1..n]$ per partition ℓ of the original alphabet. We set $B_\ell[i] = \mathtt{1}$ iff $s[i] \in \Sigma_\ell$ (or, equivalently, it holds that $m.\mathsf{access}(s[i]) = \ell$). Notice that B_j has n_j $\mathtt{1}$s.

Given a symbol $\alpha \in \Sigma$ mapped to subalphabet $\ell = m.\mathsf{access}(\alpha)$, let $c = m.\mathsf{rank}_\ell(\alpha)$ be its representation in Σ_ℓ. Hence, we define:

$$s.\mathsf{rank}_\alpha(i) \equiv s_\ell.\mathsf{rank}_c(B_\ell.\mathsf{rank}_\mathsf{1}(i)).$$

Also,

$$s.\mathsf{select}_\alpha(j) \equiv B_\ell.\mathsf{select}_\mathsf{1}(s_\ell.\mathsf{select}_c(j)).$$

Unfortunately, operation $s.\mathsf{access}(i)$ cannot be supported efficiently by our approach: since we do not know symbol $s[i]$, we do not know the partition j such that $B_j[i] = \mathtt{1}$. The alternative is to check every partition, until for a given ℓ it holds that $B_\ell[i] = \mathtt{1}$. Once this partition ℓ has been determined, we compute

$$s.\mathsf{access}(i) \equiv m.\mathsf{select}_\ell(s_\ell.\mathsf{access}(B_\ell.\mathsf{rank}_\mathsf{1}(i))).$$

Although in general our implementation does not support access efficiently, there are still relevant applications where this operation is not needed, such as computing the intersection of inverted lists [?,?,?], or computing the term positions for phrase searching and positional ranking functions [?].

Besides, many applications need operation access to obtain not just a single symbol, but a snippet $s[i..i+L-1]$ —e.g., snippet-generation tasks [?,?]. In this case, one needs operation access to obtain not just a single symbol, but a snippet $s[i..i+L-1]$ of L consecutive symbols in s. Let us call $s.\mathsf{snippet}(i, L)$ the corresponding operation. Rather than using operation access L times to obtain the desired symbols, we define Algorithm 1. The idea is that for each partition $j = 0, \ldots, p-1$, we obtain the symbols contained in $s[i..i+L-1]$ that correspond to this partition. Line 4 of the algorithm computes the number of symbols to be extracted from this partition. Operation select on B_j is used to determine the position of each symbol within the snippet, as it can be seen in Line 7.

Algorithm 1: $\mathsf{snippet}(i, L)$

```
1: Let $S[1..l]$ be an array of symbols in $\Sigma$.
2: for $j = 0$ to $p - 1$ do
3:    $cur \leftarrow B_j.\mathsf{rank}_1(i - 1)$
4:    $count \leftarrow B_j.\mathsf{rank}_1(i + L - 1) - cur$
5:    for $k = 1$ to $count$ do
6:       $cur \leftarrow cur + 1$
7:       $S[B_j.\mathsf{select}(cur) - i + 1] \leftarrow m.\mathsf{select}_j(s_j.\mathsf{access}(cur))$
8:    end for
9: end for
10: return $S$
```

3.2 Analysis of Space Usage and Query Time

If we use the `SDarray` representation of Okanohara and Sadakane [**?**] to represent the bit vectors B_ℓ, their total space usage is $\sum_{i=0}^{p-1}(n_i \lg \frac{n}{n_i} + 2n_i + o(n_i))$. Notice that $\sum_{i=0}^{p-1} n_i \lg \frac{n}{n_i} = nH_0(t)$, according to Equations (1) and (3). Also, we have that $2\sum_{i=0}^{p-1} n_i = 2n$. Finally, for $\sum_{i=0}^{p-1} o(n_i)$ we have that each term in the sum is actually $O(n_i/\lg n_i)$ [**?**]. In the worst case, we have that every partition has $n_i = n/p$ symbols. Hence, $n_i/\lg n_i = n/(p \lg \frac{n}{p})$, which for p partitions yields a total space of $O(n/\lg \frac{n}{p})$ bits. This is $o(n)$ since $\lg \frac{n}{p} \in w(1)$. In our case, $p \leq \lg^2 n$, hence $\sum_{i=0}^{p-1} o(n_i) \in o(n)$.

Summarizing, bit vectors B_ℓ require $n(H_0(t) + 2 + o(1))$ bits of space. This is 2 extra bits per symbol when compared to mapping t from Barbay et al.'s original approach [**?**]. The whole data structure uses $nH_0(s) + 2n + o(n)(H_0(s) + 1)$ bits.

Regarding operation times, $s.\mathsf{select}$ can be supported in $O(1)$ time (by using the `SDarray` from Section 2.1). Operation $s.\mathsf{rank}$ can be supported in $O(\lg n)$ worst-case time: if $n_i = O(\sqrt{n})$, operation $B_i.\mathsf{rank}$ takes $O(\lg \frac{n}{n_i}) = O(\lg n)$ time. Algorithm $\mathsf{snippet}$ takes $O(\sum_{i=0}^{p-1} \lg \frac{n}{n_i} + L \lg\lg\sigma)$ time. The sum $\sum_{i=0}^{p-1} \lg \frac{n}{n_i}$ is maximized when $n_i = n/p = n/\lg^2 n$. Hence, $\sum_{i=0}^{p-1} \lg \frac{n}{n_i} = O(\lg^2 n \cdot \lg\lg n)$, thus the total time for $\mathsf{snippet}$ is $O(\lg^2 n \cdot \lg\lg n + L \lg\lg\sigma) = O((L + \lg^2 n)\lg\lg n)$. As a comparison, using operation access to extract the snippet would yield time $O(pL) = O(L \lg^2 n)$. When $L = \Theta(\lg\lg n)$, both approaches are asymptotically similar. However, as soon as $L = \omega(\lg\lg n)$, the time for algorithm $\mathsf{snippet}$ is $O((\omega(\lg\lg n) + \lg^2 n)\lg\lg n)$, versus $O(\omega(\lg\lg n) \cdot \lg^2 n)$ of operation access. Thus, for sufficiently long snippets, operation $\mathsf{snippet}$ is faster than using access.

Regarding construction time, bit vectors B_i can be constructed in linear time: we traverse string s from left to right; for each symbol $s[j]$, determine its partition i and push-back the corresponding symbol in s_i, and position j into an extendible array [**?**]. Afterwards, the `SDarray` for B_i is constructed from this array. Extendible arrays can be implemented to obtain good performance in practice [**?**], so this imposes no restrictions to our approach.

4 Experimental Results and Applications

4.1 Experimental Setup

We implemented our data structure following the `sdsl` library [?]. Our source codes were compiled using `g++` with flags `-std=c++11` and `-O3`. Our source code can be downloaded from `https://github.com/ericksepulveda/asap`. We run our experiments on an HP Proliant server running an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores, 15 MB of cache, and 48 GB of RAM.

We used a 3.0 GB prefix of the Wikipedia (dump from August 2016). We removed the `XML` tags, leaving just the text. The text has 8,468,328 distinct words. We represent every word using a 32-bit unsigned integer, resulting in 1.9 GB of space. The zero-order empirical entropy of this string is 12.45 bits.

We tested `sparse` and `dense` partitioning, the latter with parameter $\ell_{min} = 1$ (i.e., the original `dense` partitioning), and $\ell_{min} = \lg\lg\sigma = \lg 23$ (which corresponds to the partitioning scheme currently implemented in `sdsl`). The number of partitions generated is 476 for `sparse`, 24 for `dense` $\ell_{min} = 1$, and 46 for `dense` $\ell_{min} = \lg 23$.

4.2 Experimental Results for Basic Operations

For operations rank and select, we tested two alternatives for choosing the symbols on which to base the queries:

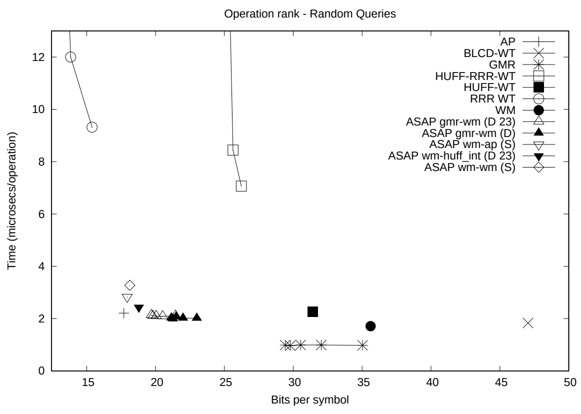
Random Symbols: 30,000 alphabet symbols generated uniformly at random.
Query-log Symbols: we use the query log from the TREC 2007 Million Query Track [3]. We removed stopwords, and used only the words that exist in the alphabet. Overall we obtained 29,711 words (not necessarily unique).

For rank operation, we generate uniformly at random the positions where the query is issued. For select, we search for the j-th occurrence of a given symbol, with j generated at random (we are sure that there are at least j occurrences of the queried symbol). For operation access, we generate text positions at random.

Figures 1 and 2 show the experimental results for operations rank and select, comparing with the most efficient approaches from the `sdsl`. We name ASAP our approach, after agile and succinct alphabet partitioning. We show several combinations for mapping m and sequences s_ℓ, as well as several ways to carry out the alphabet partitioning. For instance, the label ASAP gmr-wm (D 23) corresponds to the scheme using Golynski et al. data structure [?] (`gmr_wt<>` in `sdsl`) for s_ℓ, a wavelet matrix for mapping m, and `dense` $\ell_{min} = \lg 23$ partitioning. Label ASAP gmr-wm (D) is the same approach as before, this time using the original `dense` partitioning. The `sparse` partitioning is indicated with "(S)" in the labels. Bit vectors B_ℓ are implemented using `sd_vector<>` from `sdsl`, which corresponds to the `SDArray` data structure [?]. We show only the most competitive combinations. The original alphabet partitioning scheme is labeled AP in the

[3] `https://trec.nist.gov/data/million.query07.html`

plots. We used the default scheme from `sdsl`, which implements mappings t and m using Huffman-shaped WTs, and the sequences s_ℓ using wavelet matrices. The alphabet partitioning used is `dense` $\ell_{min} = \lg 23$. This was the most competitive combination for AP in our tests.



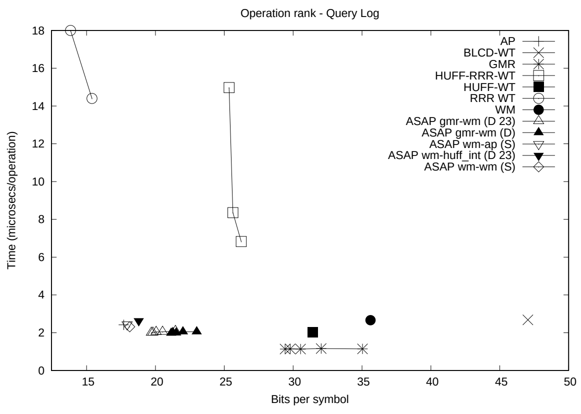
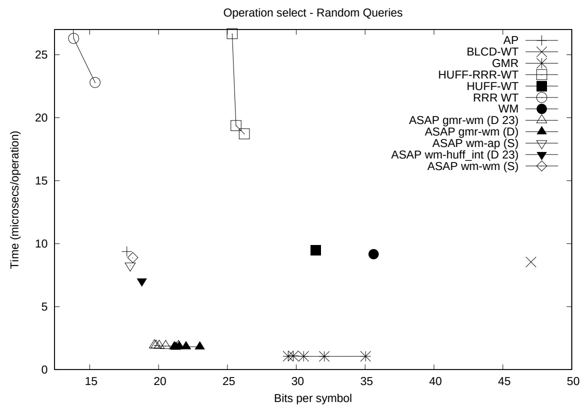


Fig. 1. Experimental results for rank. The x axis starts at $H_0(s) = 12.45$ bits.

As it can be seen, ASAP yields interesting trade-offs. In particular, for select on random symbols, alternative ASAP gmr-wm (D 23) uses 1.11 times the space of AP, and reduces the average time per select by 79.50% (from 9.37 to 1.92 microseconds per select). For query-log symbols, we obtain similar results. However, in this case there is another interesting alternative: ASAP wm-ap (S) uses only 1.01 times the space of AP, and reduces the average select time by 38.80%. For rank queries we improve query time between 4.78% (ASAP wm-wm (S)) to 17.34% (ASAP gmr-wm (D 23)). In this case the improvements are smaller compared to select. This is because operation rank on bit vectors `sd_vector<>` is not as efficient as



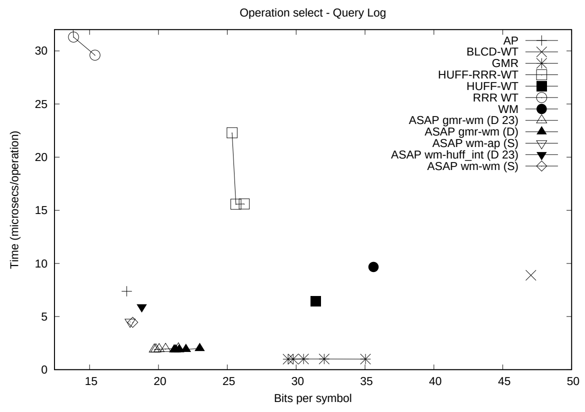


Fig. 2. Experimental results for select. The x axis starts at $H_0(s) = 12.45$ bits.

select [?]. Overall, ASAP gmr-wm (D 23) improves ASAP by 79.50% for operation select, and 17.34% for operation rank, using 1.11 times the space of ASAP.

Figure 3 shows experimental results for operation access. As expected, we cannot compete with the original AP scheme. However, we are still faster than RRR WT, and competitive with GMR [?] (yet using much less space).

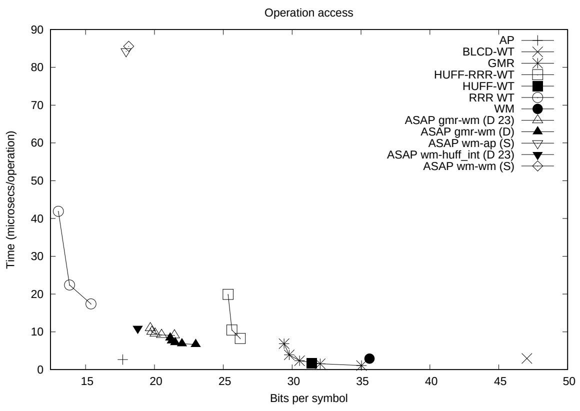


Fig. 3. Experimental results for access. The x axis starts at $H_0(s) = 12.45$ bits.

4.3 Application 1: Snippet Extraction

We study next the snippet extraction task, common in text search engines [?,?]. In our experiments we tested with $L = 100$ (see Figure 4). As it can be seen, we are able to reduce the time per symbol considerably (approximately by 75%) when compared with operation access, making our approach more competitive for snippet extraction. It is important to note that operation B_j.select in line 7 of Algorithm 1 is implemented using the select operation provided by the sd_vector<> implementation. A more efficient approach in practice would be to have an iterator that allows one to obtain the desired 1 bits in segment $[i..i+L-1]$ of the bit vectors, without repeatedly using operation select. This iterator is still not provided by the sdsl library and would not be effective for sd_vector<>, it could be a good idea for another kind of bit vectors.

4.4 Application 2: Intersection of Inverted Lists

Another relevant application of rank/select data structures is that of intersecting inverted lists. A previous work [?] has shown that one can carry out the intersection of inverted lists by representing the document collection (seen as a single string) with a rank/select data structure. Figure 5 shows experimental results for intersecting inverted lists. We implemented the variant of intersection algorithm tested by Barbay et al. [?]. As it can be seen, ASAP yields important improvements in this application: using only 2% extra space, ASAP wm-wm (S) is

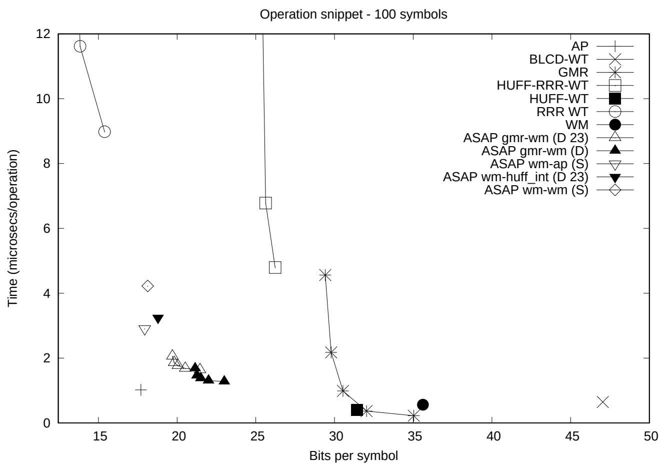


Fig. 4. Experimental results for extracting snippets of length $L = 100$. The x axis starts at $H_0(s) = 12.45$ bits.

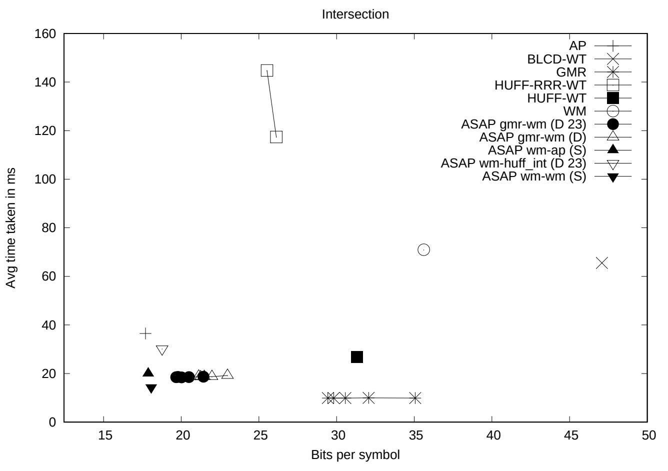


Fig. 5. Experimental results for inverted list intersection. Times are in milliseconds. The x axis starts at $H_0(s) = 12.45$ bits.

able to reduce the intersection time of AP by 60.67%. This is a promising result: our query time (of around 16 milliseconds per query) is competitive with that of inverted indexes for BM25 query processing in IR (around 12 milliseconds per query is the usual time reported in the literature [?,?]). There are, also, faster and smaller compression approaches for inverted indexes for the case of re-enumerated document collections [?], like the highly efficient Partitioned Elias-Fano (PEF) [?] . In this particular case, 16 milliseconds per query is not competitive with PEF. Neither is the space usage. However, it is worth to remind that within the space of the H_0-compressed text, we are also implicitly storing the inverted index. This can be used not only to extract snippets, but also to look for positional information, substituting positional inverted indexes (or full inversions) [?], all within the same space. This would make our data structure more competitive.

4.5 Application 3: Distributed Computation of rank and select

The partitions generated by the alphabet partitioning approach are amenable for the distributed computation of batches of rank and select operations. In a distributed query processing system, a specialized node is in charge of receiving the query requests (this is called the *broker*), and then distributes the computation among the computation nodes (or simply *processors*). We study how to support the fast computation of batches of rank and select queries using the original AP approach and our proposal (ASAP).

A simple approach for operation rank would be to partition the input string into equal-size chunks, and let each processor deal with one chunk. To support the distributed computation of rank, the processor storing the ith chunk, from the left, also stores the rank of each symbol up to chunk $i-1$. The space needed to store these ranks is $O(\sigma \lg n)$ bits per processor, which is impractical for big alphabets (as the ones we are testing in this paper). Also, this works only for operation rank, yet not for select. Next, we consider more efficient approaches in general.

A Distributed Query-Processing System Based on AP. The subalphabet sequences s_ℓ are distributed among the computation nodes, hence we have p processors in the system. We also have a specialized broker, storing mappings m and t. This is a drawback of this approach, as these mappings become a bottleneck for the distributed computation.

A Distributed Query-Processing System Based on ASAP. In this case, the subalphabet sequences s_ℓ and the bit vectors B_ℓ are distributed among the computation nodes. Unlike AP, now each computation node acts as a broker: we only need to replicate mapping m on them. The overall space usage is $O(p\sigma \lg\lg p)$ if we use an uncompressed WT for m. This is only $O(\sigma \lg\lg p) = o(n)H_0(s)$ bits per processor [?]. In this simple way, we avoid having a specialized broker, but distribute the broker task among the computation nodes. This avoids bottlenecks at the broker, and can make the system more fault tolerant.

Queries arrive at a computation node, which uses mapping m to distribute it to the corresponding computation node. For operation $s.\mathsf{access}(i)$, we carry out a broadcast operation, in order to determine for which processor ℓ, $B_\ell[i] = \mathtt{1}$; this is the one that must answer the query. For extracting snippets, on the other hand, we also broadcast the operation to all processors, which collaborate to construct the desired snippet using the symbols stored at each partition.

Comparison. The main drawback of the scheme based on AP is that it needs a specialized broker for m and t. Thus, the computation on these mappings is not distributed, lowering the performance of the system. The scheme based on ASAP, on the other hand, allows a better distribution: we only need to replicate mapping m in each processor, with a small space penalty in practice. To achieve a similar distribution with AP, we would need to replicate m and t in each processor, increasing the space usage considerably (mainly because of t). Thus, given a fixed amount of main memory for the system, the scheme based on ASAP would be likely able to represent a bigger string than AP. Table 1 shows experimental results on a simulation of these schemes. We only consider computation time, disregarding communication time. As it can be seen, ASAP uses the distributed

Table 1. Experimental results for the distributed computation of operations on a string. Times are in microseconds per operation, on average (for extracting snippets, it is microseconds per symbol). For rank and select, the symbols used are from our query log. Scheme ASAP implements the sequences s_ℓ using wavelet matrices, whereas mapping m is implemented using a Huffman-shaped WT. The partitioning is `dense` $\ell_{min} = \lg 23$. The number of partitions (i.e., computation nodes in the distributed system) is 46.

Operation	ASAP		AP		AP/ASAP
	Time	Speedup	Time	Speedup	
rank	0,373	8.03	1.310	1.91	3.51
select	0.706	8.41	2.970	2.55	4.21
access	1.390	8.11	2.130	1.45	1.53
snippet	0.466	6.96	0.939	1.25	2.02

system in a better way. The average time per operation for rank and select are reduced by about 71% and 76%, respectively, when compared with AP. For extracting snippets, the time per symbol is reduced by about 50%. Although the speedup for 46 nodes might seem not too impressive (around 7–8), it is important to note that our experiments are just a proof of concept. For instance, the symbols could be distributed in such a way that the load balance is improved.

5 Conclusions

Our alphabet-partitioning rank/select data structure offers interesting trade-offs in practice. Using slightly more space than the original alphabet-partitioning data structure from [?], we are able to reduce the time for operation select by about 80%. The performance for rank can be improved between 4% and 17%. For the inverted-list intersection problem, we showed improvements of about 60% for query processing time, using only 2% extra space when compared to the original alphabet-partitioning data structure. This makes this kind of data structures more attractive for this relevant application in information retrieval tasks. We also studied how the alphabet-partitioning data structures can be used for the distributed computation of batches of rank, select, access, and snippet operations. As far as we know, this is the first study about the support of these operation on a distributed-computing environment. In our experiments, we obtained speedups from 6.96 to 8.41, for 46 processors. This compared to 1.25–2.55 for the original alphabet-partitioning data structure. Our results were obtained simulating the distributed computation, hence considering only computation time (and disregarding communication time). The good performance observed in the experiments allows us to think about a real distributed-computing implementation. This is left for future work, as well as a more in-depth study that includes aspects like load balance and total communication time, among others. As another interesting future work, it would be interesting to study how our data structure behaves with different alphabet sizes, as well as how it compares with approaches like the one used by Gog et al. [?].