Faster Wavelet Trees with Quad Vectors

Matteo Ceregini
Dipartimento di Informatica, Università di Pisa, Italy

Florian Kurpicz
Karlsruhe Institute of Technology, Germany

Rossano Venturini
Dipartimento di Informatica, Università di Pisa, Italy

Abstract

Given a text, rank and select queries return the number of occurrences of a character up to a position (rank) or the position of a character with a given rank (select). These queries have applications in, e.g., compression, computational geometry, and pattern matching in the form of the backwards search—the backbone of many compressed full-text indices. A wavelet tree is a compact data structure that for a text of length n over an alphabet of size σ requires only $n\lceil\log\sigma\rceil(1+o(1))$ bits of space and can answer rank and select queries in $\Theta(\log\sigma)$ time. Wavelet trees are used in the applications described above.

In this paper, we show how to improve query performance of wavelet trees by using a 4-ary tree instead of a binary tree as basis of the wavelet tree. To this end, we present a space-efficient rank and select data structure for quad vectors. The 4-ary tree layout of a wavelet tree helps to halve the number of cache misses during queries and thus reduces the query latency. Our experimental evaluation shows that our 4-ary wavelet tree can improve the latency of rank and select queries by a factor of ≈ 2 compared to the wavelet tree implementations contained in the widely used Succinct Data Structure Library (SDSL).

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Applied computing → Document management and text processing; Applied computing → Document searching

Keywords and phrases wavelet tree, quad vector, query latency

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material The C++ source code of the data structures described in this paper is available at `https://github.com/MatteoCeregini/quad-wavelet-tree`. A Rust implementation is available at `https://github.com/rossanoventurini/qwt`.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:0–23:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Wavelet trees [21] are a self-indexing rank and select data structure, i.e., they can answer rank (how often does a symbol occur in a prefix of length i) and select (where does a symbol occur for the i-th time) queries, while still allowing to access the text. This makes them an important building block for compressed full-text indices, e.g., the FM-index [15] or the r-index [18], where they are used to answer rank queries during the pattern matching algorithm—the so called backwards search.

Due to the plethora of applications, a lot of research has been focused on the efficient construction of wavelet trees in both practice and theory. We give an overview of the state-of-the-art in Section 3. However, there exists barely any research focusing on the query performance of wavelet trees. While there exist alternative representations of the wavelet tree (namely the wavelet matrix, see Section 2) that provide better practical query performance, the better query performance is more of a byproduct of a space efficient representation for large alphabets.

The main building block of wavelet trees (and wavelet matrices) are bit vectors with binary rank and select support. There exist many different approaches tuning the rank and select support for query time and/or space overhead. Faster binary rank and select queries directly translate to faster queries on wavelet trees. We refer to Section 3 for an overview of binary rank and select support for bit vectors. However, improving only the binary rank and select data structure still not fully utilizes the full range of optimizations when it comes to answering queries using wavelet trees.

Our Contributions. The main result of this paper is a practical wavelet tree implementation that improves the query latency in practice by a factor of ≈ 2 compared to its competitors implemented in the widely used *Succinct Data Structure Library* (SDSL) [19]. To this end, we show that representing a wavelet tree as 4-ary tree instead of as binary tree results in half as many cache misses, see Section 5. We focus on the query latency of the wavelet tree, i.e., the amount of time it takes to answer a single query, because the latency is the most relevant time measurement in practice for many applications, e.g., the backwards search in the FM-index [15], where queries depend on one another.

Instead of using bit vectors, our 4-ary wavelet tree quad vectors, i.e., vectors over the alphabet $[0, 4)$, with rank and select support, see Section 4. Here, our rank and select data structure for quad vectors has a space overhead of only 7.81 % and only 6.25 % if only rank support is required.

2 Preliminaries

A *bit vector* is a text over the binary alphabet $\{0, 1\}$. On text T of length n over an alphabet $\Sigma = [0, \sigma)$, we want to answer *rank*, and *select* queries for $i \in [0, n)$ and $\alpha \in \Sigma$:

- $rank_\alpha(i) = |\{j < i \colon T[j] = \alpha\}|$ and
- $select_\alpha(i) = \min\{j \colon rank_\alpha(j) = i\}$.

Rank and select queries on bit vectors of length n can be answered in $O(1)$ time with $o(n)$ additional bits [7, 23]. The *most significant bit* (MSB) of a character is the bit with the highest value. For simplicity, we assume that the MSB is the leftmost bit. The i-th MSB is the bit with the i-th highest value. A length-ℓ *bit-prefix* of a character are the character's ℓ MSBs.

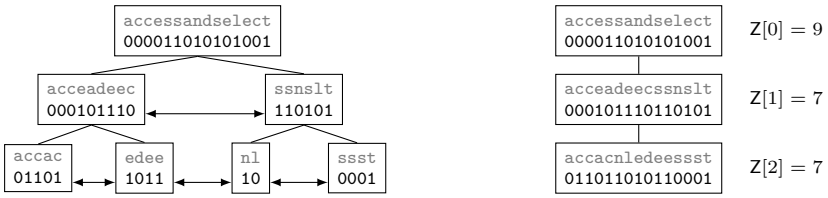

Figure 1 Wavelet tree (left) and a wavelet matrix (right) for the text `accessandselect` over the alphabet {`a` $(000)_2$, `c` $(001)_2$, `d` $(010)_2$, `e` $(011)_2$, `l` $(100)_2$, `n` $(101)_2$, `s` $(110)_2$, `t` $(111)_2$} (bit representation of the characters is given in gray). Note that we depict the text for better readability only; the text is not part of the wavelet tree or wavelet matrix. The wavelet tree can be transformed to a level-wise wavelet tree by concatenating all bit vectors in nodes connected by an arrow. In the wavelet matrix, we can see the same intervals as in the wavelet tree on the same level.

A *wavelet tree* [21] is a binary tree, where each node represents a subsequence of the text. Each node contains character with a specific length-k bit-prefix. The root of a wavelet tree represents all characters with the length-0 bit prefix ϵ, i.e., all characters. Then, whenever we visit a left child of a node that represents characters with bit-prefix α, the child represents character with-bit prefix $\alpha 0$. The right child represents characters with bit-prefix $\alpha 1$. Alternatively, you can say that the left child represents characters that are in the lower half of the alphabet represented in its parent and the right child represents characters in the upper half. Here, the root represents characters in the whole alphabet.

On the ℓ-th level of the tree (the root has level 1) the characters are represented by their ℓ-th MSB. Hence, within a node, all represented characters are stored in a bit vector. If we concatenate the bit vectors of all nodes on the same level, we obtain a *level-wise* wavelet tree. We say that all characters that have been represented in a node of the non-level-wise wavelet tree are in the same interval, see Figure 1 for an example. All intervals in a wavelet tree can be identified by the bit prefix of the characters represented in the interval. In the following, when we mention wavelet trees, we refer to level-wise wavelet trees.

The *wavelet matrix* [8] is an alternative representation of the wavelet tree. The first level of the wavelet matrix are the MSBs of the characters, the same as the first level of the wavelet tree. Then, to compute the next level ℓ, starting with the second, the text is stably sorted using the $(\ell - 1)$-th MSB as key. Just as with the wavelet tree, the characters are represented using their ℓ-th MSB on each level ℓ. The order of the characters on each level is given by the stably sorted text. Sorting the text looses the tree structure of the wavelet tree. However, the same intervals as in the wavelet tree occur on each level, just in a different order.[1] A comparison of the structure of a wavelet tree and a wavelet matrix can be found in Figure 1. In addition to the bit vectors, the number of zero in each level is stored in the array Z, which are needed to answer queries faster using one less binary rank and/or select query per level.

In the following, we use wavelet tree to refer to both wavelet tree and wavelet matrix. Note that we implemented the wavelet matrix to benefit from the fewer binary rank and/or select queries, making them the better choice in practice. Using a wavelet trees for a text of length n over an alphabet of size σ, access, rank, and select queries can be answered in time $O(\log \sigma)$ time. A wavelet tree requires $n\lceil \log \sigma \rceil (1 + o(1))$ bits of space. The sublinear term is needed for the rank and select support of the bit vectors.

[1] They appear in the bit-reversal permutation order, see `https://oeis.org/A030109`, last accessed 2023-01-16.

3 Related Work

The compact and compressed representation of texts with support for access, rank, and select queries (among others) is an active field of research. For example, bit vectors with rank and select support, i.e., binary rank and select data structures, are often a building block for succinct data structures.

Binary Rank and Select Data Structures. For bit vectors of length n, rank and select data structures with constant query time can be constructed in linear time requiring $o(n)$ space [7, 23]. Practical and well-performing implementations of these data structures can be found in the SDSL [19]. The currently most space efficient rank and select support for a size-u bit vector that contains n ones requires only $\log \binom{u}{n} + \frac{u}{\log u} + \tilde{\mathrm{O}}(u^{\frac{3}{4}})$ bits (including the bit vector) [32]. In practice, the currently fastest select data structures are by Vigna [40]. Allowing for multiple configurations using a tuning parameter, they outperform all other select data structures while being space-efficient. However, they still require much more space than the currently most space-efficient data structures by Zhou et al. [41] that have recently been improved w.r.t. query throughput by Kurpicz [26]. There exist many more practical rank and select data structures that are outperformed by the ones mentioned above, e.g., [20, 25, 31]. Another line of research considers compressed [1, 5, 6, 38] and mutable [34, 35] bit vectors with rank and select support.

Wavelet Trees and Wavelet Matrices. Wavelet tree construction is a well studied field. Let T be a text of length n over an alphabet of size σ. The asymptotically best sequential wavelet tree construction algorithms require $O(n \log \sigma / \sqrt{\log n})$ time [2, 29]. These approaches make use of vectorized instructions, i.e., SIMD (single instruction, multiple data), to achieve their running time. There also exist implementations that make use of vectorized instructions available in modern CPUs [24] and are reported to be the fastest in practice. In shared memory, wavelet trees can be computed in $O(\sigma + \log n)$ time requiring only $O(n \log \sigma / \sqrt{\log n})$ work [39]. In practice, the fastest construction algorithms are based on domain decomposition [27, 17], where partial wavelet trees are computed in parallel and are then merged also in parallel, using a bottom-up construction for the partial wavelet tree construction [9]. Wavelet trees can also be computed in other models of computation, e.g., distributed memory [10] and external memory [12]. Wavelet trees can also be compressed. To this end, the wavelet tree is constructed for the Huffman-compressed text.[2] The bit vectors in the Huffman-shaped wavelet tree requires $n\lceil H_0(T) \rceil$ bits of space, where H_0 is the zeroth order entropy of the text. A fully functional wavelet tree requires binary rank and select support on the bit vectors and needs $n\lceil \log \sigma \rceil (1 + o(1))$ bits of space ($n\lceil H_0(T) \rceil (1 + o(1)$ bits of space for the Huffman-shaped wavelet tree). Multi-ary wavelet trees have been considered before for compressed representation of sequences [16]. For more information on wavelet trees, we point to the multiple wavelet tree surveys [14, 22, 28, 30].

Alternative Representations of Sequences. There exist other compressed representations of a text that can answer rank and select queries, while still allowing to access the text. Recently, a practical block tree implementation has been introduced [3]. A block tree is especially useful for highly compressible text, as they require only $O(z \log(n/z))$ words space, where z is the number of Lempel-Ziv factors of the text. Further dictionary-compressed

[2] To be precise, bit-wise negated canonical Huffman codes are required [9].

representations allow for rank and select support in optimal time in compressed space [37] with respect to the size of a string attractor [36] of the text. Rank and select support can also be added to grammar compressed texts of size n over an alphabet of size σ. For a grammar of size g, rank and select support requires $O(\sigma g)$ space [4, 33]. Here, queries can be answered in $O(\log n)$ time.

4 Quad Vectors

At the heart of our 4-ary wavelet trees is a space-efficient and fast rank and select data structure for quad vectors. Our data structure uses a block-based design and follows the popular memory layout for block-based rank and select data structures for bit vectors [26, 41] adapted to quad vectors. In a block-based design, the number of occurrences of different symbols is stored for blocks of different size. The number is stored either for the whole input up to the block or for the input contained in a bigger block. For our quad vector, we store the following information for each symbol $\alpha \in \{\texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}\}$:

- *Super-Blocks* cover 4096 symbols and store the number of occurrences before the start of the super-block.
- *Blocks* cover 512 symbols and store the number of occurrences before the start of the block within the super-block.

For each super-block, we only have to store seven blocks, as there are no occurrences of any symbol before the first block within the super-block, i.e., all counters are zero. The counter within each block can be stored in just 12 bits, as the maximum number of occurrences of a single symbol within one super-block before the last block is 3584 ($\lceil \log 3584 \rceil = 12$). Therefore, the counters of the seven blocks fit into 84 bits and can use 44 bits for the counter of the super-block, when using 128 bits for both super-block and the pertinent blocks.[3] Additionally, storing super-blocks and pertinent blocks interleaved, reduces the number of cache misses and allows for the usage of vectorized instructions [26].

Since we require 128 bits for each super-block including its blocks for each symbol, the space-overhead of the rank data structure is $512/8192 = 6.25\,\%$. This is twice as much as the bit vector rank data structure requires [26]. To answer select queries efficiently, we store every 8192-th occurrence of a symbol. This increases the required space by $1.5625\,\%$.

Answering queries using this approach is similar to the bit vector case. Assume we want to get the rank of the symbol α at position i. We simply have to identify the super-block $(i/4096)$ and the block $((i\%4096)/512)$ where the position occurs in. Adding up these counters, we only have to scan $(i\%512)$ positions within the block and add the number of occurrences of α in the block up to that position to the result. All this can be done in constant time To find the position where the j-th α occurs, we first identify the closest smaller sampled position. Starting from there, we do scan super-blocks until we have identified the super-block containing the position. Then, we continue with scanning block until we have identified the block containing the position. Finally, we scan the quad vector (within the block) until we have found the correct position and return the index. While this may not be a constant time query, it is very fast in practice, see Section 6.

[3] In practice, computer words have size 8, 16, 32, 64, 128, and on modern machines even 256 and 512 bits. Aligning the size of (super-)blocks with computer words improves the performance.

Figure 2 4-ary wavelet tree (left) and a 4-ary wavelet matrix (right) for the text `accessandselect` over the alphabet {`a` $(000)_2$, `c` $(001)_2$, `d` $(010)_2$, `e` $(011)_2$, `l` $(100)_2$, `n` $(101)_2$, `s` $(110)_2$, `t` $(111)_2$} (bit representation of the characters is given in gray), i.e., the same input text as in Figure 1.

Optional Improvements

In the following, we describe optional improvements that help to either reduce cache misses or reduce the space of the data structure. Note that we did not include the space-saving features in our experimental evaluation, as preliminary experiments showed that they heavily impact the query performance outweighing the benefit of the saved space.

Reducing Cache Misses by Doubling the Space. A cache line on nearly all hardware has size 64 bytes. To a super-block and its pertinent blocks fit into one cache line, the number of symbols per super-block and block can be halved. By doing so, we double the number of counters we have to store, but we also guaranteed at most two cache misses per rank query and three per select query. However, by doing so, the space-overhead increases to 12.5 %.

Saving Space by Computing Information. To save space, we can only save the information for three of the four symbols, as we can compute all information for the fourth symbol using the information of the other three symbols. This technique is used for bit vectors, only information for one of the two symbols is stored. Note that the sample index for faster select queries has to be stored for all four symbols. Removing the information for one symbol saves 25 % of memory, hence the memory overhead is now only 4.6875 % (or 9.375 % if we use the technique described above to reduce the number of cache misses).

Saving Space by Encoding Blocks using Elias-Fano. Using the Elias-Fano encoding [11, 13], a monotonic increasing sequence of k integers in a universe of size u can be stored using only $k(2+\log(u/k))$ bits while allowing constant time access to all integers. Since the number of occurrences of symbols withing super-blocks are monotonic increasing sequences, we can use Elias-Fano encoding to store them. To this end, we introduce *mega-blocks* that cover 2^{18} symbols. We store the number of occurrences of each symbol from the beginning of the text to the beginning of each mega-block. In addition to the space-saving measure described above, we now also encode the information for the remaining three symbols. We now can store the following information for each super-block: Three 18-bit counters for three symbols storing the number of occurrences from the beginning of the mega-block and the Elias-Fano encoded sequences that require at most 141 bits. Overall, we require 195 bits for *all three* symbols. Therefore, this variant has a space overhead of 2.41 %.

5 4-Ary Wavelet Tree

When answering queries using a wavelet tree, the query is translated to $O(\log \sigma)$ binary rank and select queries. In practice, most of the time to answer a query on the wavelet tree is spend answering these binary rank and select queries. Additionally, on each level of the

wavelet tree, the binary rank and select queries will result in at least one cache miss, which again is where most of the time for answering a binary rank or select query is used for. To reduce the number of cache misses, we have to reduce the number of levels of the wavelet tree. To this end, we make use of 4-ary wavelet trees. By doubling the number of children, we (roughly) halve the number of levels. If $\lceil \log \sigma \rceil$ is odd, the 4-ary wavelet tree has $\lceil \lceil \log \sigma \rceil / 2 \rceil$ levels.

In a 4-ary wavelet tree, we represent the characters on each level using two bits that we store in a quad vector. If $\lceil \log \sigma \rceil$ is odd, characters on the last level are represented using a single bit in a bit vector. In the first level, each character is now represented by its two MSBs and all characters share a length-0 bit-prefix. When visiting the first child of a node that represents characters with bit prefix α, its first child represents characters with bit-prefix $\alpha\texttt{00}$, the second child represents characters with bit-prefix $\alpha\texttt{01}$, the third child represents characters with bit-prefix $\alpha\texttt{10}$, and the fourth child represents characters with bit-prefix $\alpha\texttt{11}$. Alternatively, you can say that the first child represents characters that are in the lower quarter of the alphabet represented by its parent, the second child represents characters that are in the second quarter, and so on. The root represents the whole alphabet. See Figure 2 for an example.

Querying a 4-ary wavelet tree works similarly to querying a "normal" wavelet tree. Since there are now four children, more book keeping is necessary to keep track of the interval that is visited during the query. This does not result in more rank and select queries on the quad vectors (and possibly bit vector on the last level). Overall, the additional book keeping is less expensive than the cache misses on each level as we can clearly see in our experiments in Section 6.2.

There also exist a "4-ary" wavelet matrix representation of the 4-ary wavelet tree. Here, we also use two bits to represent the characters at each level. Again, the first level of the wavelet matrix is the same as the first level of the 4-ary wavelet tree. Then, to compute the next level ℓ starting with the second, the text is stably sorted using the $(2\ell - 1)$-th and 2ℓ-th MSBs as key. As with the "normal" wavelet tree and wavelet matrix, this results in the same intervals, where characters with the same bit-prefix are represented, just in a different order. Also, queries for the wavelet matrix can be adopted to work with the (4-ary) wavelet matrix in the same way we adopted the queries of the wavelet tree. To do so, we now have to store the exclusive prefix sum of the histogram of all entries of all levels (as a replacement of Z in the "normal" wavelet matrix. Then again, querying the 4-ary wavelet matrix works similarly to querying the "normal" wavelet matrix.

6 Experimental Evaluation

First, we discuss our experimental setup. Then, in Section 6.1, we compare our quad vector implementation with state-of-the-art bit vectors. Finally, in Section 6.2, we show the benefit of using 4-ary wavelet trees instead of wavelet trees based on bit vectors.

Experimental Setup. All the experiments are performed using a single thread on a server machine with 8 Intel i9-9900KF cores with base frequencies of 3.60 GHz running Linux 5.19.0. Each core has a dedicated L1 cache of size 32 KiB, a dedicated L2 cache of size 256 KiB, a shared L3 cache of size 16 MiB, and 64 GiB of RAM. The code is compiled with GCC 12.2.0 using the highest optimization setting (i.e., flags `-O3 -march=native -DNDEBUG -flto`). Our implementation is available at `https://github.com/MatteoCeregini/quad-wavelet-tree`. A Rust implementation is available at `https://github.com/rossanoventurini/qwt`.

Latency vs. Throughput. The *latency* of an operation is the time it takes to complete it, which is the time between the start of the operation and when the result becomes available. The *throughput* of an operation is the number of operations completed in a time span. This is not the same as dividing the time span by the latency since any modern CPU uses pipelining to parallelize the execution of several operations at the same time. This is possible only if these operations are independent, e.g., the input of an operation does not depend on the output of a previous one. The result of this parallelization is that the throughput is always at most the latency, but usually much smaller.

In the case of access, rank, and select queries, the query time in large sequences is dominated by the cost of the cache misses caused by the access of portions of the indexed sequence. Since the CPU can issue several memory requests at the same time, measuring the throughput hides a part of the cost of these cache misses paid by the queries that the CPU can execute in parallel.

In our experiments, we measured the latency of access, rank, and select by forcing the input of each query to depend on the output of the previous one. This is consistent with the use of the queries in real settings. For example, any query on a wavelet tree decomposes into several dependent queries on the underlying binary or 4-ary vector. Similarly, more advanced queries supported by compressed text indexes (e.g., CSA or FM-index) decompose into several dependent queries on the underlying wavelet tree.

6.1 Experimental Evaluation of Quad Vectors

In this section, we compare our quad vectors with existing bit vectors, as there are to the best of our knowledge no other quad vector implementations publicly available.

Data Structures We include the following data structures in our experiments.

- SDSL is the implementation of binary vectors of the popular SDSL library [19].[4] The binary vector is enhanced with the rank_support_v data structure to support rank queries. It also uses select_support_mcl data structure to support select queries, which implements Clark and Munro's approach to compute select queries [7].
- PASTA is the implementation of binary vectors of the PASTA-toolbox library [26].[5] The binary vector is enhanced using the FlatRankSelect data structure to support rank and select queries. Our quad vectors a similar memory layout and vectorized instructions to answer queries.
- QV_{256} and QV_{512} are our implementations of quad vectors with blocks of size 256 and 512 symbols, respectively. Our implementation follows the description in Section 4.

Datasets. In these experiments, we generate random binary sequences with a size that ranges from 32 KiB to 2 GiB, where each bit has a fifty percent chance of being zero or one. We obtain 4-ary sequences just by using consecutive pairs of bits of the generated binary sequences as quad symbols.

The sizes have been selected so that the three smallest datasets of size 32 KiB, 256 KiB, and 16 MiB, respectively, match the sizes of the three levels of cache of our server. For the remaining sizes, we doubled the size of the dataset up to 2 GiB. For the first three datasets, we expect to observe a query time dominated by the cost of performing the bit-wise and

[4] SDSL library available at `https://github.com/simongog/sdsl-lite`.
[5] Pasta-toolbox library available at `https://github.com/pasta-toolbox/bit_vector`.

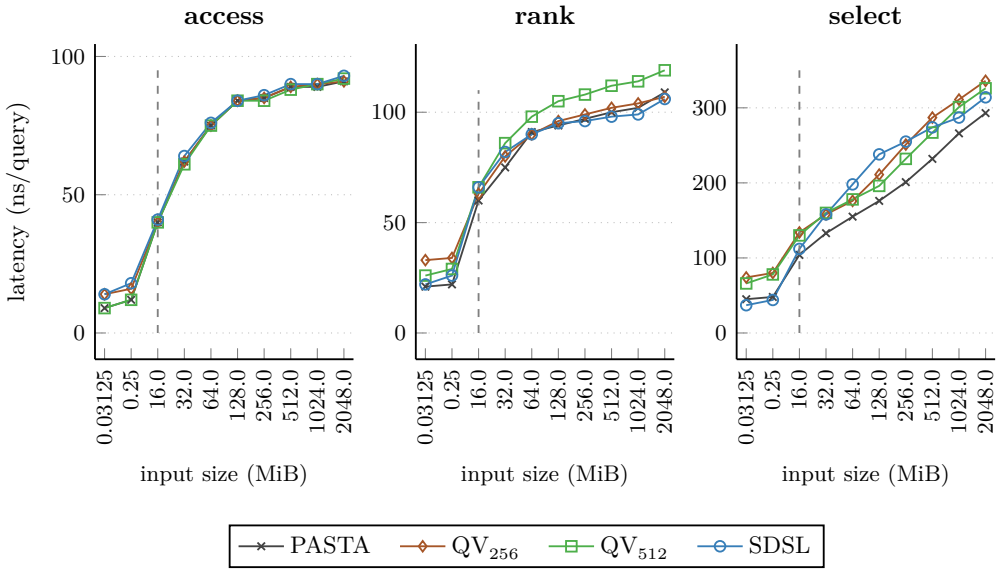

Figure 3 Comparison of access, rank, and select query latency for our *quad vectors* with *bit vectors*. Inputs on the right of the gray dashed line do not fit into the cache any more.

arithmetic operations needed to solve the query. Instead, for the other datasets, the query time will be dominated by the cost of accessing the necessary data from memory.

6.1.1 Access, Rank, and Select Queries

In the experiments, a reported running time is the average time of three runs. For each run, each data structure executes 1 million queries of the target query type. We generate the queries as follows. Let $S[0, n)$ be the indexed sequence.

- access: each query asks to access the symbol at a random position in the sequence.
- rank: we generate a random position $i \in [0, n)$, access that position to retrieve the bit or symbol $S[i]$ and use $\langle i, S[i] \rangle$ as a rank query.
- select: we select a symbol c at random following their distribution in the sequence, i.e., more frequent symbols have a higher probability of being selected. Then, we generate a random value $r \in [1, occ(c)]$, where $occ(c)$ is the number of occurrences of c, and use $\langle r, c \rangle$ as a select query.

As we mentioned before, we want to measure the latency of a query. For this reason, we modify the random positions above by adding the result produced by the preceding query (modulo n for rank and access queries, or modulo $occ(c)$ for select). The results are plotted in Figure 3.

Access Queries. Expect for very small datasets (32 KiB and 256 KiB), all four data structures have roughly the same access time, with differences that are within 10%. This is not surprising because all of them perform few arithmetic and bit-wise operations to extract either one or two bits, which fit within the same cache line. Therefore, for small datasets, the query time is bounded by the time required to perform the bit-wise operations. Instead, for larger datasets, the query time is close to the cost of accessing a cache line from memory.

Rank Queries. For small datasets, PASTA is the fastest being from 1.05 to 1.18 times faster than SDSL, from 1.05 to 1.57 times faster than QV_{256}, and from 1.18 to 1.31 times faster than QV_{512}. We would expected SDSL to be the fastest and QV_{512} to be the slowest, since in this regime the cost is dominated by the CPU and in particular by the number of popcnt instructions[6] needed to compute the rank within a block. On average SDSL needs just one popcnt while PASTA and QV_{256} need 4 popcnts, and QV_{512} need 8 popcnts. For larger datasets, the cost is dominated by the cost of a cache miss. Here the data structures have roughly the same performance being always within 20% of each other. We observe that SDSL, PASTA, and QV_{256} need to access two cache lines to compute a rank: the cache line with counters for the superblock and the blocks and the cache line with data. Instead, QV_{512} needs an extra cache line because its block spans two cache lines. However, this does not induce any major slowdown (that is, paying two full cache misses per query) because these memory operations are independent of each other and the CPU can issue them in parallel, thus reducing the cost of the rank operation to just one cache miss.

Select Queries. For small datasets, PASTA and SDSL are the fastest. They are up to 2 times faster then QV_{256} and QV_{512}. This is expected since searching for a symbol within a basic block in a quad vector is more expensive than bitvectors, since they require more bit-wise operations. For bigger input sizes, PASTA is the fastest and is about $15-35\%$ faster then the other data structures. Instead, SDSL, QV_{256} and QV_{512} have take approximately the same time to answer a select operation. This decrease in performance of SDSL can be explained by the fact that it requires a larger amount of data to compute the result of a selection operation, which results in more cache misses.

6.1.2 Space Overhead

As SDSL is designed to have small blocks of 64 bits, which explains why it is faster for small datasets compared to PASTA, QV_{256}, and QV_{512}. This efficiency for small datasets is paid at a cost of a much larger space overhead. The space required for the rank query support is the following:

- SDSL uses the rank_support_v data structure which logically divides the bit vector into 64-bit basic blocks and 512-bit superblocks. For each superblock, 128 extra bits are used to store the counters relative to the superblock and its basic blocks. Therefore, it requires a space overhead of $\frac{128}{512} = 25\%$.
- PASTA uses the FlatRankSelect data structure which follows a two-layer approach with 512-bit basic blocks and 4096-bit superblocks. For each superblock, it uses 128 bits to store the counters relative to the superblock and its basic blocks. Thus, the overhead caused by the counters is $\frac{128}{4096} = 3.125\%$.
- QV_{256} and QV_{512} require a space overhead of 6.25% and 12.5% respectively to store the superblock and basic block counters.

To support for select queries, additional space overhead is necessary for all tested data structures:

- SDSL uses the select_support_mcl data structure which requires in the worst case 37.5% space overhead.

[6] The popcnt instructions returns the number of one bits in a computer word and is supported by most modern CPUs.

- PASTA uses the FlatRankSelect data structure that stores the position of every 8192-th occurrence of each bit as a 32-bit unsigned integer. Thus, the space overhead caused by the samplings is $\frac{32\times 2}{8192} = 0.78125\%$. Note that the select support requires the rank support to be available.
- QV_{256} and QV_{512} follow the same strategy as PASTA, but now the possible symbols are four: every 8192-th occurrence of each symbol is stored as a 32-bit unsigned integer. Thus, the space overhead caused by the samplings is $\frac{32\times 4}{8192} = 1.5625\%$. As with PASTA, the rank support has to be available.

6.2 Experimental Evaluation 4-Ary Wavelet Trees

In this section, we compare our 4-ary wavelet tree implementation with other wavelet tree implementations, which are all based on bit vectors. Again, to the best of our knowledge there exists no publicly available k-ary wavelet tree implementation for $k > 2$.

Data Structures. In our experiments, we compare the following data structures. Note that in fact we compare wavelet *matrices*, as those are faster in practice as wavelet trees. All implementations mentioned below contain support for both wavelet trees and wavelet matrices. The wavelet matrices are build on top of the bit vectors that we evaluated in Section 6.1.

- SDSL is the implementation of wavelet matrices built on bit vectors of the SDSL library. The data structure is built upon the SDSL's bit vector.
- PASTA is the implementation of wavelet matrices built on bit vectors of the PASTA-toolbox library. The data structure is built upon the PASTA-toolbox's bit vector.
- QWM_{256} and QWM_{512} are our implementations of wavelet matrices built on quad vectors with blocks of size 256 and 512 symbols, respectively, that we describe in Section 4 without the space optimization of using a bit vector as last level when $\lceil \log \sigma \rceil$ is odd.

Datasets. In these experiments, we measure the access, rank and select query time of the data structures on text prefixes between 32 KiB and 2 GiB in size (we use a prefix of the given text for smaller input sizes.), generated from the following datasets.

- English is the concatenation of English text files selected from Gutenberg Project.[7] The alphabet size is 239 and the file size is about 2.06 GiB.
- DNA is a sequence of newline-separated gene DNA sequences obtained from files from Gutenberg Project.[8] The alphabet size is 16 and the file size is about 0.4 GiB. Since the file is too small for our experiments, the prefixes were extracted from a big enough text obtained by concatenating the DNA text with itself several times.
- random16 and random256 are random texts over an alphabet of size 16 and 256, resp. The characters of the text are chosen uniformly at random. During our experiments, all data structures use the same random inputs. We use these texts to show that the query latency does not depend on the structure of the text.

6.2.1 Access, Rank, and Select Queries

The queries are generated in the same way as described in Section 6.1. Again, we measure latency in the experiments. The results are depicted in Figure 4.

[7] The file is available at `http://pizzachili.dcc.uchile.cl/texts/nlang/`
[8] The file is available at `http://pizzachili.dcc.uchile.cl/texts/dna/`

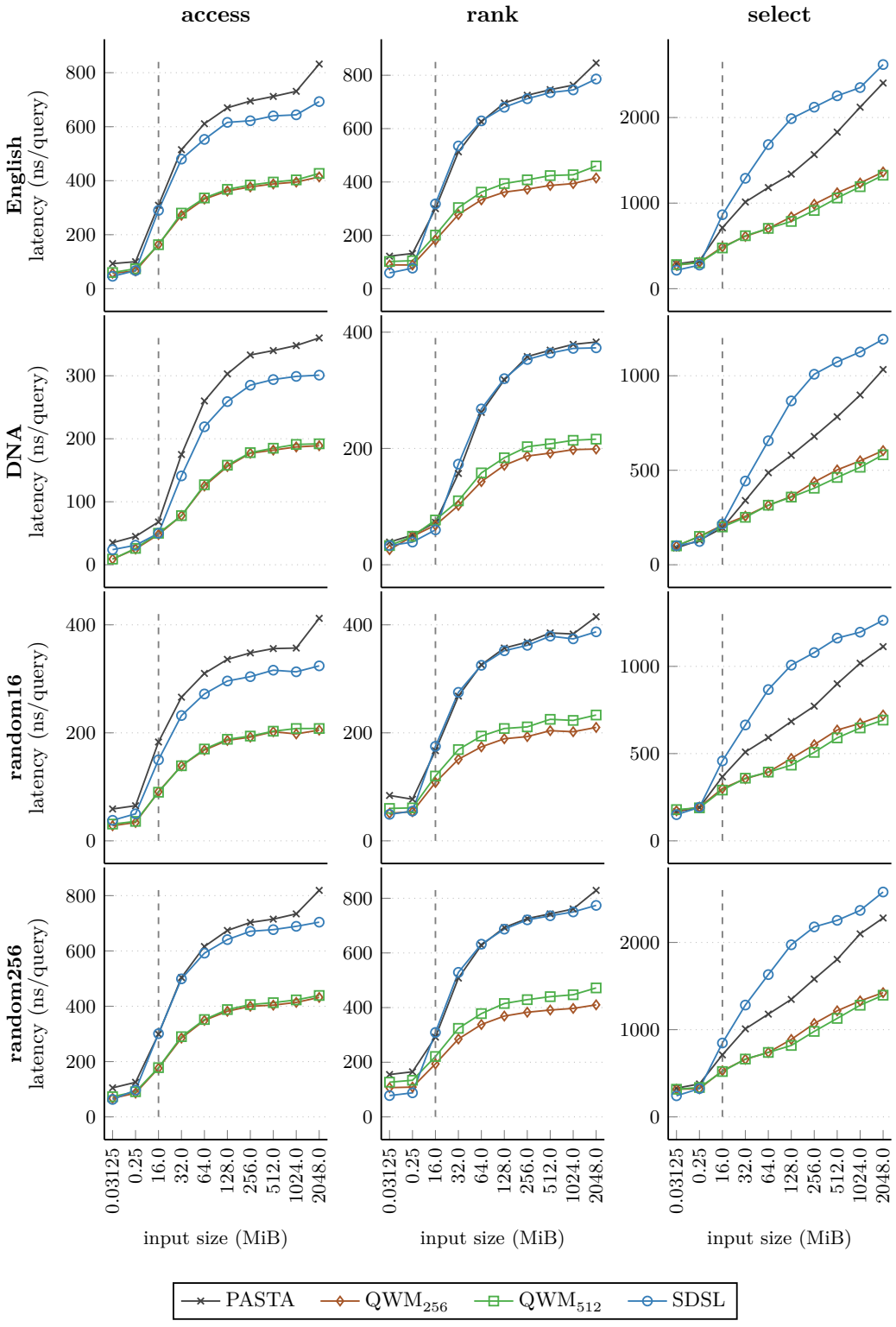

Figure 4 Comparison of access, rank, and select queries on different *wavelet matrix* implementations for on different inputs and input sizes. Inputs on the right of the gray dashed line do not fit into the cache any more.

Access queries. On the English text, we get the following results. For datasets up to 256 KiB SDSL is overall the fastest, being up to 1.3 faster than QWM_{256} and QWM_{512} and up to 2.02 times faster than PASTA. For datasets bigger then 256 KiB, QWM_{256} and QWM_{512} have similar query times and are also the fastest, being 1.6 to 1.78 times faster than SDSL and 1.8 to 2 times faster than PASTA. We obtain similar results for random256, hinting that the results do not depend on the structure of the input. On the DNA text, the overall picture looks the same. However, due to the smaller alphabet size the latencies are overall lower. For small datasets, QWM_{256} and QWM_{512} are the fastest while having similar query times. They are up to 2.67 times faster than SDSL and 3.89 times faster than PASTA. For larger datasets, QWM_{256} and QWM_{512} are still the fastest being 1.57 to 1.81 times faster than SDSL and 1.82 to 2.24 times faster than PASTA. Again, we obtain a similar result for random16.

Rank queries. On prefixes of the English text smaller than or equal to 256 KiB, PASTA is the fastest, being up to 1.51 times faster than QWM_{256}, up to 1.73 times faster then QWM_{512} and up to 1.71 times faster then PASTA. For datasets bigger than 256 KiB, QWM_{256} is the fastest, being roughly 10% faster then QWM_{512} and 1.75 to 1.92 times faster than SDSL and 1.64 to 2.04 times faster than PASTA. SDSL is overall the fastest for smaller inputs of the DNA text, being roughly up to 20–25% faster then the other data structures. For largest datasets, QWM_{256} is the fastest, being 1.7 to 1.9 times faster than SDSL. QWM_{512} is at approximately less than 10% slower than QWM_{256}. As for access queries, the data structures behave the same for random inputs.

Select queries. For English inputs smaller than or equal to 256 KiB, SDSL is the fastest, being about up to 1.3 times faster then QWM_{256} and QWM_{512} and up to 1.34 times faster then SDSL. For datasets bigger than 256 KiB, QWM_{256} and QWM_{512} are the fastest while having similar query access times: they are about 1.77 to 2.4 times faster than SDSL and about 1.46 to 1.81 times faster then PASTA. For smaller DNA inputs, PASTA is overall the fastest, being up to 14% faster than the other data structures. For bigger datasets QWM_{512} and QWM_{256} are the fastest while having similarly query times. They are approximately 1.72 to 2.42 times faster than SDSL and about 1.32 to 1.77 times faster than PASTA. As for the queries before, the overall picture is the same for select queries on random inputs for all data structures tested in this evaluation.

6.2.2 Space Overhead

The space overhead of wavelet matrices is the same given by the sum of the space overheads required to both support rank and select queries on the underlying bit vectors or quad vectors that we described in Section 6.1 (plus a small overhead introduced by each wavelet matrix implementation). Note that this space overhead can be significant depending on the data structure used for rank and select support. For example, on the 1 GiB prefix of the English text, the SDSL wavelet matrix require 48 % additional space. The most space-efficient implementation is PASTA with a space-overhead of only 3.71 %. Our 4-ary wavelet matrices require only 6.44 % (QWM_{256}) and 12.69 % (QWM_{512}) additional space.

7 Conclusion

We have shown that using quad vectors instead of bit vectors as basis for wavelet trees improves the query latency by reducing the number of cache misses. While bit-vector-based

wavelet trees outperform our 4-ary wavelet trees for inputs that fit into cache, on more realistic inputs our new 4-ary wavelet trees heavily outperform all other tested wavelet trees with speedups up to 2.67 (access), 1.9 (rank), and 1.7 (select) compared to currently used wavelet tree implementations contained in the SDSL and PASTA-toolbox.

It remains an open problem to combine the 4-ary Wavelet tree layout with the sublinear construction algorithm based on vectorized instructions. Another interesting line of future research are compressed 4-ary wavelet trees, e.g., Huffman-shaped 4-ary wavelet trees. Also, our space-saving technique using Elias-Fano encoding could be adopted to bit vectors. Here, this approach is promising, as fewer information has to be computed and the information for even more blocks fits into a single cache line.

Acknowledgements. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500), from the University of Pisa - Project no. PRA 2020-2021 26, and from Italian Ministry for Education and Research – PRIN Project no. 2017K7XPAN.

References

1 Diego Arroyuelo and Manuel Weitzman. A hybrid compressed data structure supporting rank and select on bit sequences. In *SCCC*, pages 1–8. IEEE, 2020. doi:10.1109/SCCC51225.2020.9281244.

2 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *SODA*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

3 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021. doi:10.1016/j.jcss.2020.11.002.

4 Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *ESA*, volume 9294 of *Lecture Notes in Computer Science*, pages 142–154. Springer, 2015. doi:10.1007/978-3-662-48350-3_13.

5 Kai Beskers and Johannes Fischer. High-order entropy compressed bit vectors with rank/select. *Algorithms*, 7(4):608–620, 2014. doi:10.3390/a7040608.

6 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Trans. Algorithms*, 2022. doi:10.1145/3524060.

7 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996.

8 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.

9 Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Practical wavelet tree construction. *ACM J. Exp. Algorithmics*, 26:1.8:1–1.8:67, 2021. doi:10.1145/3457197.

10 Patrick Dinklage, Johannes Fischer, and Florian Kurpicz. Constructing the wavelet tree and wavelet matrix in distributed memory. In *ALENEX*, pages 214–228. SIAM, 2020. doi:10.1137/1.9781611976007.17.

11 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.

12 Jonas Ellert and Florian Kurpicz. Parallel external memory wavelet tree and wavelet matrix construction. In *SPIRE*, volume 11811 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2019. doi:10.1007/978-3-030-32686-9_28.

13 Robert Mario Fano. On the number of bits required to implement an associative memory. Technical report, MIT, Computer Structures Group, 1971. Project MAC, Memorandum 61".

14 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.

15 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

16 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007. doi:10.1145/1240233.1240243.

17 José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Parallel construction of wavelet trees on multicore architectures. *Knowl. Inf. Syst.*, 51(3):1043–1066, 2017.

18 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

19 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

20 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, 2005.

21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. ACM/SIAM, 2003.

22 Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *CCP*, pages 210–221. IEEE Computer Society, 2011. doi:10.1109/CCP.2011.16.

23 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

24 Yusaku Kaneta. Fast wavelet tree construction in practice. In *SPIRE*, volume 11147 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2018. doi:10.1007/978-3-030-00479-8_18.

25 Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2005. doi:10.1007/11427186_28.

26 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

27 Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *J. Discrete Algorithms*, 43:2–17, 2017. doi:10.1016/j.jda.2017.04.001.

28 Christos Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.

29 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

30 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

31 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *SEA*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012. doi:10.1007/978-3-642-30850-5_26.

32 Mihai Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

33 Alberto Ordóñez Pereira, Gonzalo Navarro, and Nieves R. Brisaboa. Grammar compressed sequences with rank/select support. *J. Discrete Algorithms*, 43:54–71, 2017. doi:10.1016/j.jda.2016.10.001.

34 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Inf. Syst.*, 99:101756, 2021. doi:10.1016/j.is.2021.101756.

35 Nicola Prezza. A framework of dynamic data structures for string processing. In *SEA*, volume 75 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.11.

36 Nicola Prezza. On string attractors. In *ICTCS*, volume 2243 of *CEUR Workshop Proceedings*, pages 12–16. CEUR-WS.org, 2018.

37 Nicola Prezza. Optimal rank and select queries on dictionary-compressed text. In *CPM*, volume 128 of *LIPIcs*, pages 4:1–4:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.4.

38 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

39 Julian Shun. Improved parallel construction of wavelet trees and rank/select structures. *Inf. Comput.*, 273:104516, 2020. doi:10.1016/j.ic.2020.104516.

40 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

41 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.