Faster Wavelet Tree Queries

Florian Kurpicz[†], Angelo Savino[*], and Rossano Venturini[*]

[*]University of Pisa
a.savino6@studenti.unipi.it
rossano.venturini@unipi.it

[†]Karlsruhe Institute of Technology
kurpicz@kit.edu

September 30, 2025

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 compression, computational geometry, and most notably pattern matching in the form of the backward search, which is the backbone of many compressed full-text indices. Currently, in practice, for text over non-binary alphabets, the wavelet tree is probably the most used data structure for rank and select queries. In this paper, we present techniques to speed up queries by a factor of two (access and select) up to three (rank), compared to the wavelet tree implementation contained in the widely used Succinct Data Structure Library (SDSL). To this end, we change the underlying tree structure from a binary tree to a 4-ary tree and reduce cache misses by approximating rank queries using a predictive model to prefetch all data required for the actual rank query. We further extend our approach to Huffman-shaped wavelet trees, allowing us to obtain a data structure that is able to process queries up to three times faster for access and select, and 3.8 times faster for the rank operation (compared to the SDSL), while also occupying compressed space (up to 30% less space for compressible datasets).

Keywords: Wavelet Trees, Rank/Select Queries, Succinct Data Structures, Compressed Data Structures, Algorithm Optimization, Cache Efficiency

1 Introduction

Wavelet trees [2] are a compressible self-indexing rank and select data structure, i.e., they can answer rank (number of occurrences of symbol up to position i) and select (position of i-th occurrence of symbol) queries, while still allowing access to the text. They were introduced in the context of the compressed suffix array, but their properties make them also an important building block for other compressed full-text indices, e.g., the FM-index [3] or the r-index [4], where they are used to answer rank queries during the pattern matching algorithm—the backwards search—.

Wavelet trees have many applications, which are discussed in multiple surveys [5, 6, 7, 8]. Wavelet trees are also known to be highly adaptable to different compression schemes [2], and an important line of work is finding ways to employ this in different applications[9]. A lot of research has been focused on the efficient construction of wavelet trees in both practice and theory. In Section 2, we present the wavelet tree data structures and give an overview of the state-of-the-art. But, there exists barely any research focusing on the query performance of wavelet trees. The wavelet matrix [10], an alternative representation of the wavelet tree, provides better practical query performance, however, this is more of a byproduct of a space efficient representation for large alphabets, thanks to its pointer-less layout. 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 2 for an overview

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), by the PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 "Precision medicine & personalized healthcare", and by the MIUR (grant agreement No. 20229BCXNW)
A preliminary version of this article appeared in *DCC 2024* [1]

of binary rank and select support for bit vectors. Still, improving only the binary rank and select data structure still does not fully utilize the full range of optimizations when it comes to answering queries using wavelet trees.

Our Contributions

Wavelet trees usually utilize binary trees as underlying tree structure. We show that using a 4-ary tree as underlying tree structure (see Section 3) results in a query speedup of up to 2 for all queries compared to its competitor implemented in the widely used *Succinct Data Structure Library* (SDSL) [11]. Furthermore, we introduce the *Rank with Additive Approximation* problem and show how to utilize a small prediction model to locate data necessary during rank queries. We use this information to prefetch this data, achieving a total speedup of up to 3. In Section 6 we extend the approach for Huffman-shaped wavelet trees by employing a variant of Huffman coding based on the construction of a quaternary Huffman tree, which gives us faster query times for compressible datasets that stack up with the previously introduced optimizations. This allows us to obtain a wavelet tree that is twice as fast as the fastest compressed wavelet tree in the SDSL, while also requiring up to 30% less space. In our experimental evaluation (see Section 7), we not only show these impressive speedups for such a well-researched data structure but also that our data structures requires less space and are faster to construct—making them strictly superior to their relative competitors.

2 Preliminaries and Related Work

In this section we will present some background information needed to understand Wavelet Trees and our optimizations.

We will start by presenting the solutions for performing rank and select queries over a binary alphabet, which is the simplest version of the problem, then we will present wavelet trees and explain how are they built using the solution we found for the binary alphabet version of the problem. Finally, we talk about wavelet tree construction and alternative representation of sequences that are used to solve the rank and select problem.

Binary Rank and Select Data Structures

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

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

On bit vectors of length n, rank and select queries can be answered in $O(1)$ time requiring $o(n)$ additional bits [12, 13]. 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.

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 [12, 13]. Practical and well-performing implementations of these data structures can be found in the SDSL library [11]. 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) [14]. In practice, the currently fastest select data structures are by Vigna [15]. 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. [16] that have recently been improved w.r.t. query throughput by Kurpicz [17]. There exist many more practical rank and select data structures that are outperformed by the ones mentioned above [18, 19, 20]. Another line of research considers compressed [21, 22, 23, 24, 25] and mutable [26, 27] bit vectors with rank and select support.

Wavelet Trees

A *wavelet tree* [2] is a binary tree, where each node represents a subsequence of the text. Each node contains characters sharing a length-k bit-prefix. If we consider a given node that represents all characters sharing the bit-prefix β, then its left child will represent characters sharing bit-prefix β0, while its right one will represent all characters sharing bit-prefix β1. The root of a wavelet tree represents all characters in the alphabet, i.e., sharing the empty length-0 bit-prefix.

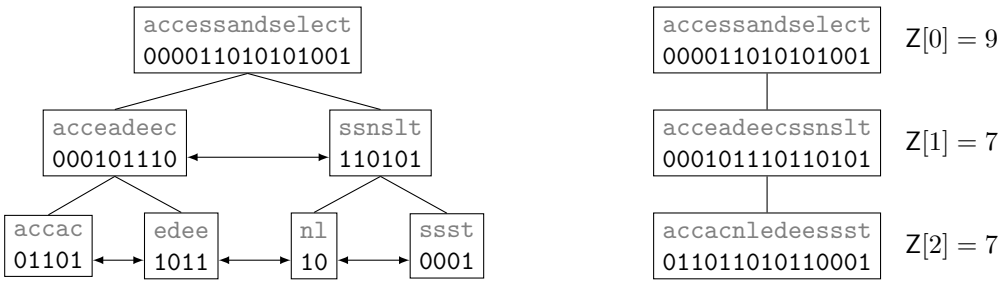

Figure 1: Wavelet tree (left) and a wavelet matrix (right) for the text `accessandselect` over the alphabet {`a` $(000)_2$, `c` $(001)_2$, `d` $(010)_2$, `e` $(011)_2$, `l` $(100)_2$, `n` $(101)_2$, `s` $(110)_2$, `t` $(111)_2$} (bit representation of characters given in gray). Note that we depict the text for better readability only; the text is not part of the wavelet tree or wavelet matrix. By concatenating all bit vectors in nodes of the wavelet tree that are connected by arrows, we obtain a level-wise wavelet tree. Each level in the wavelet matrix contains the same intervals.

The characters are not stored explicitly. On the ℓ-th level of the tree (the root has level 1), characters are represented by their ℓ-th MSB, which 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 a 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 by that interval. In the following, we consider level-wise wavelet trees.

The *wavelet matrix* [10] 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 following levels ℓ, 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. This removes the tree structure of the wavelet tree. However, the same intervals as in the wavelet tree occur on each level, except in a bit-reversal permutation[1] order. A comparison of the structure of a wavelet tree and a wavelet matrix can be found in Figure 1. The number of zeros in each level is stored in the array Z, which is needed to answer queries using one less binary rank and/or select query per level compared to wavelet trees. We give the rank query algorithms for wavelet trees and matrices in Algorithms 2.1 and 2.2, resp. In the following, we use wavelet tree to refer to both wavelet tree and wavelet matrix. A wavelet trees for a text over an alphabet of size σ can answer access, rank, and select queries in $O(\log \sigma)$ time.

Huffman-shaped wavelet trees are constructed in much the same way as classical (balanced) wavelet trees, with the difference that the symbols are not all the same length, and are translated through Huffman coding. This means that the wavelet tree has the same shape of the Huffman tree generated from the text we are representing. This approach is compatible with all prefix free codes. Building a Huffman-shaped wavelet matrix requires first building a permutation of the codes, where the reversed codes are also lexicographically sorted.

[1]See `https://oeis.org/A030109`, last accessed 2023-11-08.

```
1  Function WaveletTreeRank_α(i)
2      start = 0, size = n
3      for level = 0, ..., ⌈log σ⌉ and i > 0 do
4          before = bv.rank_1(start)
5          position = bv.rank_1(start + i) − before
6          in = bv.rank_1(start + size) − before
7          if level-th MSB of α is 1 then
8              start = start + (size − in)
9              size = in
10             i = position
11         else
12             size = size − in
13             i = i − position
14         start+ = n
15     return i
```

Algorithm 2.1. Rank query for a *wavelet tree* over a text of length n over an alphabet of size σ where all levels are stored in one consecutive bit vector bv.

```
1  Function WaveletMatrixRank_α(i)
2      start = 0, size = n
3      for level = 0, ..., ⌈log σ⌉ and i > 0 do
4          before = bv.rank_1(start)
5          position = bv.rank_1(start + i) − before
6          in = before − Z_1[level]
7          if level-th MSB of α is 1 then
8              i = before
9              start = ((level+1)n) + Z_0[level] + in
10         else
11             i = i − before
12             start =
                ((level+1)n)+(start−(level·n))−in
13         start+ = n
14     return i
```

Algorithm 2.2. Rank query for a *wavelet matrix* over a text of length n over an alphabet of size σ where all levels are stored in one consecutive bit vector bv. Additionally, $Z_\alpha[\ell]$ denotes the number of occurrences of α on level ℓ.

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.

Wavelet Tree Construction

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 [28, 29]. These approaches make use of vectorized instructions, i.e., SIMD (single instruction, multiple data), to achieve their running time and also have been implemented [30, 31]. In shared memory, wavelet trees can be computed in $O(\sigma + \log n)$ time requiring only $O(n \log \sigma / \sqrt{\log n})$ work [32]. In practice, the fastest construction algorithms are based on domain decomposition [33, 34], 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 [35]. Wavelet trees can also be computed in other distributed [36] and external memory [37]. To compress a wavelet tree, it is constructed for the Huffman-compressed text.[2] The bit vectors in the Huffman-shaped wavelet tree require $n\lceil H_0(T) \rceil$ 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). There are also wavelet trees for degenerate strings [38]. In theoretical work, multi-ary wavelet trees are used to reduce query time in the RAM model to $\Theta(\log_{\log n} \sigma)$ [39].

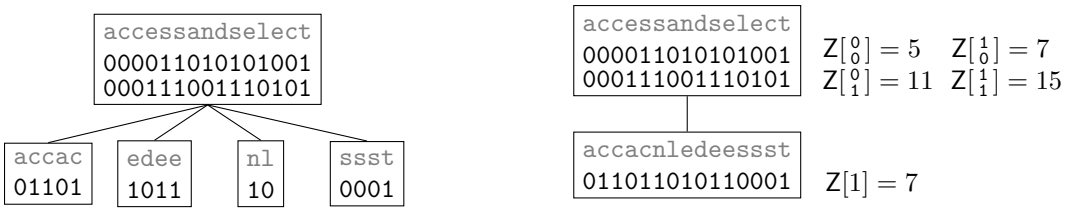

Figure 2: 4-ary wavelet tree (left) and a 4-ary wavelet matrix (right) for the text `accessandselect` over the alphabet {`a` $(000)_2$, `c` $(001)_2$, `d` $(010)_2$, `e` $(011)_2$, `l` $(100)_2$, `n` $(101)_2$, `s` $(110)_2$, `t` $(111)_2$} (bit representation of the characters is given in gray), i.e.,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

[2] Bit-wise negated canonical Huffman codes are required [35].

Alternative Representations of Sequences

There exist other compressed self-indices that can answer rank and select queries. Recently, a practical block tree implementation has been introduced [40]. Block trees are 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. Unfortunately, even highly tuned implementations are slow to compute [41]. Further dictionary-compressed representations allow for rank and select support in optimal time in compressed space [42] with respect to the size of a string attractor [43] of the text. For a grammar of size g and an alphabet of size σ, rank and select support requires $O(\sigma g)$ space [44, 45]. Here, queries can be answered in $O(\log n)$ time.

3 4-Ary Wavelet Trees

When answering queries using a wavelet tree in practice, the query is translated to $O(\log \sigma)$ binary rank and select queries, see Algorithms 2.1 and 2.2. Most of the time to answer a query on the wavelet tree is spent 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. 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. We opted to focus on quaternary wavelet trees and not other higher degree variants as our preliminary experiments showed a significant loss in space for both the uncompressed and compressed versions

In a 4-ary wavelet tree, we represent the characters on each level using two bits that we store in a quad vector, i.e., a vector over the alphabet $\{0, 1, 2, 3\}$ with access, rank, and select support, see Section 4. If $\lceil \log \sigma \rceil$ is odd, characters on the last level are represented using a 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 a node that represents characters with bit prefix β, its four children represent characters with bit-prefix $\beta\texttt{00}$, $\beta\texttt{01}$, $\beta\texttt{10}$, and $\beta\texttt{11}$ (first to fourth child).

There also exists 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 binary 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 binary wavelet matrix).

Queries in 4-Ary Wavelet Trees

Querying a 4-ary wavelet tree works similarly to querying a binary wavelet tree. Since there are now four children, more bookkeeping is required to track the interval visited during the query, which is handled using a larger vector C, which stores the histogram of the frequencies of symbols in a given level. 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 bookkeeping is less expensive than the cache misses on each level as we can clearly see in our experiments in Section 7.1. Then again, querying the 4-ary wavelet matrix works following the same logic of querying the binary wavelet matrix, except that we take couples of bits instead. See Algorithm 5.1 for an example.

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. We use a block-based design and follows the popular memory layout for block-based rank and select data structures for bit vectors [17, 16]. 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 (which are sufficient to support texts in the order of terabytes), 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 enables the use of vectorized instructions [17].

Lemma 4.1. *A quad vector with rank support has a space-overhead of 6.25 %. Adding select support introduces an additional overhead of* $4\lceil \log n \rceil / 8192$.

Proof. We require 128 bits for each super block including its blocks for each symbol. Resulting in a space-overhead of $4 \cdot 128/8192 = 6.25\,\%$. Since we store the block of every 8192-nd occurrences of a symbol to answer select queries more efficiently, this introduces another $4 \cdot 32/8192 = 1.5625\,\%$ space-overhead (allowing select support for quad vectors of length up to 2^{41} quads). □

We can reduce the number of cache misses by doubling the required space. A cache line on nearly all hardware has size 64 bytes. To make a super block and its pertinent blocks fit into one cache line, the number of symbols per block can be halved. This doubles 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 %. Figure 3 shows the structure of such solution, which will be the focus of our experimental evaluation in Section 7.

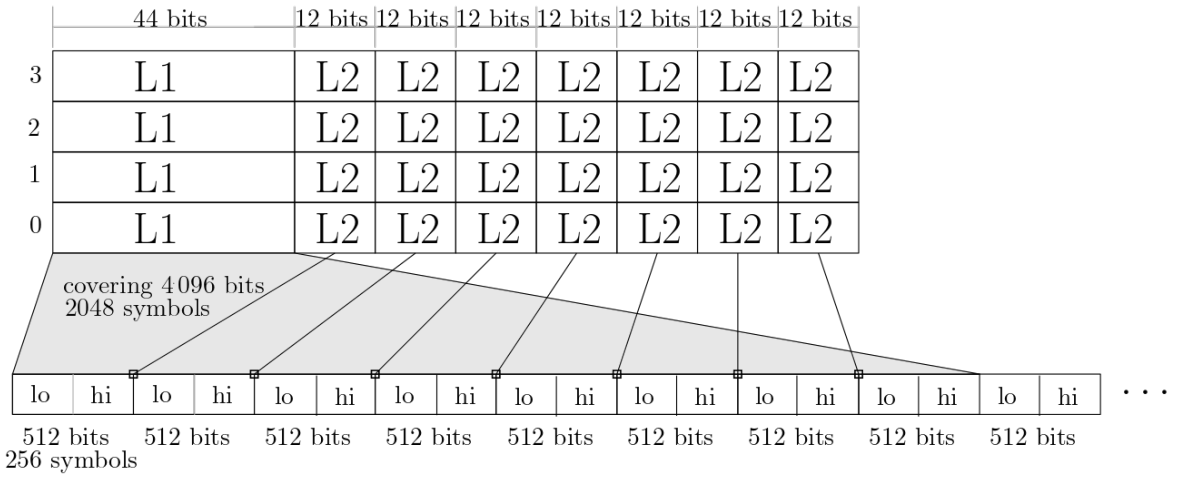

Figure 3: Data layout for our quad vector with blocks of 256 symbols.

In theory, we can save space by storing only information for three symbols and using sophisticated data structures to represent the information. However, we did not implement the following variant, as preliminary experiments showed that the space-saving features heavily impacted the query performance.

[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.

Lemma 4.2. *A quad vector with rank support requires only up to 2.41 % space-overhead.*

Proof. We 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. Removing the information for one symbol saves 25 % of space, hence the space-overhead is now only 4.6875 %.

Using the Elias-Fano encoding [46, 47], 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 and encode only 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 up to 2.41 %. □

4.1 Answering Queries

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 (($i \mod 4096$)/512) where the position occurs in. Adding up these counters, we only have to scan ($i \mod 512$) 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 7.

5 Faster Rank Queries with Prefetching

Modern CPUs can issue multiple memory requests concurrently, paving the way for proactive prefetching of cache lines predicted to be accessed in the near future. By issuing the memory requests for the accessed cache line and the anticipated ones simultaneously, prefetching helps hiding memory latency and reducing the impact of memory access delays on the CPU's execution pipeline.

Prefetching manifests in two forms:*hardware* and *software* prefetching. Hardware prefetching is implemented within the CPU's microarchitecture and is driven by the hardware itself. Modern CPUs come equipped with dedicated prefetcher units that monitor memory access patterns and automatically issue prefetch requests based on these patterns. These units analyze the memory addresses being accessed and attempt to predict future memory accesses. They then fetch the predicted data into the cache in advance. For example, the *sequential prefetching* predicts that the next memory location to be accessed will be contiguous to the current one. It fetches additional cache lines in advance to take advantage of spatial locality. This is particularly effective for array traversal where data tends to be stored sequentially. The *strided prefetching* instead looks at the delta between the addresses of the memory accesses and looks for patterns within it. If a consistent pattern in the stride is detected, the CPU fetches cache lines based on this pattern, assuming that the algorithm will continue accessing memory addresses with the same stride.

Software prefetching is controlled by the programmer or the compiler through explicit instructions. Programmers can insert prefetch instructions (e.g., `_mm_prefetch` intrinsic for x86 CPUs) into their code to indicate which data should be prefetched and when. Software prefetching requires a deep understanding of the algorithm's memory access patterns and the underlying memory hierarchy, because incorrect or excessive prefetching can lead to performance degradation.

The goal of this section is to show how to introduce software prefetching in the algorithm of the rank query. For the following discussion, we give the pseudo code for $rank_\alpha(i)$ query on a 4-ary wavelet matrix (which is the preferred choice in practice) in Algorithm 5.1. A $rank_\alpha(i)$ query on a wavelet tree has to

```
1 Function Rank_α(i)
2 |  r_0 = i, b_0 = 0
3 |  for k = 1, ..., ℓ + 1 do
4 |  |  α_k = (α >> 2 * (ℓ − 1 − k)) & 3, offset = C_k[α_k]
5 |  |  b_k = Q[k].rank_{α_k}(b_{k−1}) + offset
6 |  |  r_k = Q[k].rank_{α_k}(r_{k−1}) + offset
7 |  return r_ℓ − b_ℓ
```

Algorithm 5.1. Rank query for a *4-ary wavelet matrix* with ℓ levels. For level k, $Q[k]$ is the quad vector and $C_k[\alpha_k]$ is the number of characters $< \alpha_k$ on level k. The value b_k represents the number of characters that appear before position i and whose prefix is lexicographically smaller than α, while r_k tracks the rank of position i.

traverse each of the $\ell = \lceil \lceil \log \sigma \rceil / 2 \rceil$ levels. At each level k, we perform two rank queries on the quad vectors of that level for the character $\alpha_k \in [0, 3]$ to compute b_k and r_k. These two rank queries require the results b_{k-1} and r_{k-1} of the two rank queries computed at the previous level. Below we discuss only how to perform the prefetching for r_k as the prefetching for b_k can be done in a similar way.

Every rank query in a quad vector for a given position i needs to access only two cache lines: one containing counters for the superblock and block of that position and one containing the data block with the i-th character. These two cache lines can be requested in parallel as they only depend on position i. Hence, we observe just the latency of at most one cache miss per level. The challenge in eliminating this cache miss is that both the cache lines we need to access at a certain level k depend on the position r_{k-1}, which is known only when the rank query at level $k-1$ has been computed. Solving this challenge could be possible with a predictive model capable of anticipating the cache lines required for ranking at position r_{k-1} across all levels k, way before position r_{k-1} is computed. Such an advanced prediction would enable us to initiate simultaneous requests for all these cache lines.

5.1 Predicting Cache Lines in a Quad Vector

This challenge motivates the introduction of the *Rank with Additive Approximation* problem, which consists in estimating the rank of a character α for a given position such that its actual rank is bounded by a given error rate ϵ. This can be done by using a bit vector as will be shown in Lemma 5.2

Definition 5.1 (Rank with Additive Approximation)**.** *Let $Q[1, n]$ be a quad vector and $\epsilon \in \mathbb{N}$ a fixed additive error. Given a position i and a symbol $\alpha \in [0, 3]$, the Rank with Additive Approximation $rank^{\approx}_{\alpha}(i)$ approximates the correct rank query by returning an arbitrary value $r \in [\tilde{r}, \tilde{r} + \epsilon)$, where $r = rank_{\alpha}(i)$.*

A prediction model that correctly predicts the needed cache lines of a certain level, is actually solving the Rank with Additive Approximation problem on the quad vector of the previous level with ϵ equal to the cache line size, e.g., 512 bits (256 quads). Vice versa, if we have a solution for the problem with the same ϵ, we have a way to predict the required cache lines.

Lemma 5.1. *A data structure for the Rank with Additive Approximation problem on a quad vector $Q[1, n]$ with additive error $\epsilon \in \mathbb{N}$ needs at least $\Omega(n/\epsilon)$ bits of space.*

Proof. Assume by contradiction that there exists a solution for the problem that uses $o(n/\epsilon)$ bits of space for any quad vector of length n. Given any $Q[1, n]$, we obtain an expanded version $\hat{Q}[1, 3\epsilon n]$ by replacing each character with a run of 3ϵ of its copies. We use the above data structure to index $\hat{Q}$ using $o(n)$ bits of space. Now, we reconstruct Q by querying the data structure for all characters at the beginning and the end of each run. The correct character in Q can be identified because the results of the two queries differ by more than ϵ, while the results for the other characters differ by less than ϵ. Therefore, we could use this data structure to represent any quad vector with less than $2n$ bits, which is impossible because of an information-theoretical lower bound. □

The following lemma finally shows how to perform the approximated rank in constant time in such a way that we can bound the number k of actual occurrences of a character α in the range with size depending on the chosen ϵ.

Lemma 5.2. *The Rank with Additive Approximation problem on a quad vector $Q[1,n]$ with additive error $\epsilon \in \mathbb{N}$ can be solved in constant time using $\Theta(n/\epsilon)$ bits, i.e., matching the space lower bound.*

Proof. We use a bit vector $B_\alpha[1, \lceil 2n/\epsilon \rceil]$ for each of the character $\alpha \in [0,3]$. We split $Q[1,n]$ into blocks of size $\epsilon/2$. The ith bit in B_α is set to 1 if and only if the ith block of Q contains the jth occurrence of α, for any j which is a multiple of $\epsilon/2$.

We add *rank* support to the bit vectors B_α. A query $rank^{\approx}_\alpha(i)$ for B_α is solved as follows. Let $j = \lfloor 2i/\epsilon \rfloor$ be the block in Q that contains position i. Let $k = rank_1(j)$. We know that the number of occurrences of α up to block j is at least $k \cdot \epsilon/2 - 1$ and at most $k \cdot \epsilon/2 + \epsilon/2 - 1$. Since each block has size $\epsilon/2$, we know that $rank_\alpha(i) \in [k \cdot \epsilon/2 - 1, k \cdot \epsilon/2 + \epsilon - 1)$. Therefore, we can use $\tilde{r} = k \cdot \epsilon/2 - 1$ □

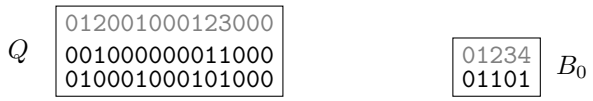

Figure 4: Example of the approximation of rank. On the right the bit vector B_0 using $\epsilon = 6$ for the quad vector Q on the left. If we want to approximate the rank for position $i = 10$, we compute the rank of the bit vector at position $j = \lfloor 2 \cdot 10/6 \rfloor = 3$. This results in $k = rank_1(3) = 2$. The rank of the quad vector at position $i = 10$ is then approximated as $\tilde{r} = k \cdot \epsilon/2 - 1 = 2 \cdot 3 - 1 = 5$. The correct rank is $r = rank_0(10) = 6$, which is within the range $[5, 5 + \epsilon)$.

5.2 Predicting Cache Lines in a Wavelet Tree

Consider the rank query $rank_\alpha(i)$. For addressing this query through a 4-ary wavelet tree, we divide the character α into its quaternary components $\alpha_1, \alpha_2, \ldots, \alpha_\ell$. Then, at level k, we compute $r_k = rank_{\alpha_k}(r_{k-1})$, see Algorithm 5.1. As we mentioned above we focus on prefetching for r_ks (line 5), as we can deal with b_ks in a similar way.

The prefetching is possible if can approximate each r_k with $\tilde{r}_k$, such that $r_k \in [\tilde{r}_k, \tilde{r}_k + \epsilon)$ with $\epsilon = 256$. Indeed, each cache line has size 512 bits and, thus, spans 256 positions of the quad vector at level k. The value $\tilde{r}_k$ introduces uncertainty only within the span of two consecutive cache lines. Note that prefetching is effective only if we compute the approximated ranks $\tilde{r}_k$ for all the levels and issue the requests for the required cache lines in parallel before we use these cache lines to compute the exact ranks r_k.

Unfortunately, solving the Rank with Additive Approximation problem with error ϵ for the quad vector at each level of the wavelet tree is not enough to guarantee that $r_k \in [\tilde{r}_k, \tilde{r}_k + \epsilon)$, for all the levels k. This is because the value $\tilde{r}_k$ is computed with an approximated rank at position $\tilde{r}_{k-1}$ because the exact position r_{k-1} is unknown, i.e., we can compute $rank^{\approx}_{\alpha_k}(\tilde{r}_{k-1})$ and not $rank^{\approx}_{\alpha_k}(r_{k-1})$. As the position $\tilde{r}_{k-1}$ is already affected by some error, the errors of our approximations sum up level by level. Thus, at level k the error could be up to $(k-1)\epsilon$.

We can solve this issue by correcting the approximations at each level. This approach is inspired by a solution for the substring occurrence estimation on texts with compressed indexes [48]. The main idea is to refine the estimates at each level k with a correction term Δ. To compute Δ we need to store a set of *discriminant* positions $D_{k,\alpha}$ for each character $\alpha \in [0,3]$ at level k.

In the solution of Lemma 5.2 we store a bit vector B_α for each character $\alpha \in [0,3]$. A bit was set to one for each position corresponding to an occurrence of α which is a multiple of $\epsilon/2$. The set $D_{k,\alpha}$ consists of the position in the quad vector corresponding to those occurrences. We note the positions in these sets can be stored within $\Theta(\log \epsilon)$ bits per position in several ways. The most suitable one for our purposes is to associate each position with its corresponding bit set to one in B_α and store its offset within the corresponding block.

At query time, given r_{k-1} and the character α_k, we want to compute the discriminant position d_{k-1} which is the successor of r_{k-1} in the set D_{k,α_k}. This discriminant position can be computed in constant time with a rank and a select query on the bit vector of α. Once we computed d_{k-1}, the correction term Δ is

$\min(d_{k-1} - \tilde{r}_{k-1}, \epsilon - 1)$ and the approximated rank is computed as $\tilde{r}_k = rank_{\alpha_k}(d_{k-1}) - \Delta$. This correction is enough to guarantee that our approximations always remain at a distance at most ϵ from the correct ones over all the levels k of the wavelet tree.

Lemma 5.3. *At any level k, we have $r_k \in [\tilde{r}_k, \tilde{r}_k + \epsilon)$.*

Proof. The proof is by induction on k. For the first level $k = 1$, as at the beginning $\tilde{r_0} = r_0$, we have $r_1 \in [\tilde{r}_1, \tilde{r}_1 + \epsilon)$ by Lemma 5.2. For general k, we assume that $r_{k-1} \in [\tilde{r}_{k-1}, \tilde{r}_{k-1} + \epsilon)$, and we prove $r_k \in [\tilde{r}_k, \tilde{r}_k + \epsilon)$. We want to prove that $\tilde{r}_k \leq r_k$ and $r_k - \tilde{r}_k \leq \epsilon$. There are two cases based on the relationship between d_{k-1} and r_{k-1}. By definition we know that $\tilde{r}_{k-1} \leq d_{k-1}$ and by inductive hyphotesis $\tilde{r}_{k-1} \leq r_{k-1}$.

The first case is $r_{k-1} \leq d_{k-1}$. Thus, we have $\tilde{r}_{k-1} \leq r_{k-1} \leq d_{k-1}$. Let z be the number of occurrences of the ranked characters α_k in the interval $[r_{k-1}, d_{k-1}]$.

$$\begin{aligned} r_k - \tilde{r}_k &= rank_{\alpha_k}(r_{k-1}) - rank^{\approx}_{\alpha_k}(\tilde{r}_{k-1}) \\ &= rank_{\alpha_k}(d_{k-1}) - z - (rank_{\alpha_k}(d_{k-1}) - \Delta) \\ &= \Delta - z \leq \epsilon \end{aligned}$$

The last inequality follows by $[r_{k-1}, d_{k-1}]$ being contained in $[\tilde{r}_{k-1}, d_{k-1}]$, bounding z by the minimum of the length of $[\tilde{r}_{k-1}, d_{k-1}]$ and $\epsilon - 1$. If the interval is larger than $\epsilon - 1$, there cannot be more than $\epsilon - 1$ of α_k since we sampled a discriminant position every ϵ occurrences of α_k. It also follows that $z \leq \Delta$ and, thus, $\tilde{r}_k \leq r_k$.

The second case is $d_{k-1} < r_{k-1}$. Thus, $\tilde{r}_{k-1} \leq d_{k-1} \leq r_{k-1}$. Let z be the number of occurrences of α_k in the interval $[d_{k-1}, r_{k-1}]$.

$$\begin{aligned} r_k - \tilde{r}_k &= rank_{\alpha_k}(r_{k-1}) - rank^{\approx}_{\alpha_k}(\tilde{r}_{k-1}) \\ &= rank_{\alpha_k}(d_{k-1}) + z - (rank_{\alpha_k}(d_{k-1}) - \Delta) \\ &= z + \Delta \leq r_{k-1} - \tilde{r}_{k-1} \leq \epsilon \end{aligned}$$

The first inequality follows by observing that Δ is at most the distance between $\tilde{r}_{k-1}$ and d_{k-1} and z is at most the distance between d_{k-1} and r_{k-1}. So, their sum is at most $r_{k-1} - \tilde{r}_{k-1}$. The last inequality is by inductive hypothesis. □

The space required by this predicting data structure is $\Theta((n/\epsilon)\log\epsilon)$ for each level of the wavelet tree. So, the overall space usage is $\Theta((n\log\sigma/\epsilon)\log\epsilon)$ bits. As we mentioned above, prefetching with the above data structure can be done by setting $\epsilon = 256$. However, we are left with an issue. If the indexed sequence is too large, the predicting data structure itself does not fit in the cache and, thus, to avoid cache misses in the wavelet tree we would pay cache misses in the predicting data structure. This issue could be solved by introducing a hierarchy of predictors in which a predictor at a specific level takes on the responsibility of prefetching the necessary cache lines for the subsequent-level predictor. Each predictor allows an error that is roughly ϵ times less than the one at the next level, until the predictor at the head of the hierarchy fits in cache. Unfortunately, a larger hierarchy becomes impractical quite soon for two reasons. First, to fully exploit prefetching we would have to request all the predicted cache lines in parallel, and current CPUs can issue only 5–10 memory requests in parallel. Second, each level of the hierarchy introduces a cost of $\Theta(\log\sigma)$ to the query.

5.3 Practical Implementations

In our implementation, we relaxed the previous solution in several respects. First, we do not use the correcting term Δ and the discriminant positions. This is because in our tests we used sequences with an alphabet size σ up to 256, which requires a wavelet tree of at most 4 levels. Thus, the error growth here is very limited and it can be afforded by prefetching more cache lines. Second, we limit the hierarchy to just two levels of predictors. The first one implements the solution of Lemma 5.2 with error $\epsilon = 2048$. For the second level, we observe that super block and block counters can be used as a variant of the solution of Lemma 5.2 with error $\epsilon = 256$. Even if it is less space-efficient, it is preferred because that space is already required by the wavelet tree implementation. This way, we can use the first level to predict the super block containing r_k for

each level and prefetch the cache lines containing the counters of those super blocks and their blocks. Then, we use these counters to refine the predictions to prefetch the cache lines with the correct blocks of data in the quad vectors.

The first level of the hierarchy has to store a bit vector of size $n' = n/2048$ bits for each character $\alpha \in [0, 3]$ in each level of the wavelet tree. We enhance this bit vector with rank support with a space overhead of $n'/4$ bits [15]. So, the overhead of the predicting data structure is $\ell(4n/2048 + n/2048) = 5\ell n/2048$ bits, where ℓ is the number of levels in the 4-ary wavelet tree. The predictor at the second level of the hierarchy does not introduce any extra space overhead. Observe that the first predictor fits in a 32 MB L3 cache for sequences up to ≈ 100 Billions of characters over an alphabet of size 256.

We conclude by observing that cache lines needed by access and select queries cannot be predicted with proposed solutions. The reason is that each query on a certain level has a double dependency on the result of the previous one. Indeed, both position and symbol are known only when the previous query is solved.

6 Compressing Quad Wavelet Trees

Wavelet trees can be compressed in 2 different ways: by compressing the single bit vectors or by changing the shape of the wavelet tree, by giving it a Huffman-shape. Most of the time compressing bit vectors does not guarantee fast query times, as it can happen when using RLE and even constant time solution (such as the RRR bit vector[25]) are usually practically slower due to the higher complexity of the algorithms that can slow down operations. Another solution is to compress the original string using prefix-free codes (such as Huffman-codes) and then representing the compressed string itself, therefore, giving the wavelet tree a Huffman-shape. This representation for binary wavelet trees requires $n\lceil H_0(T)\rceil(1 + o(1))$ bits of space. In practice, especially for certain repetitive texts (such as BWTs), both solutions are often combined to obtain a wavelet tree that never requires more space than only changing shape, but at the cost of slower query time performance. For this reason, we will focus on changing the wavelet tree shape without affecting our quad vector implementation. Using this representation also means that more frequent symbols are represented using less bits, meaning that less levels are accessed when querying for such symbols, causing the queries to get resolved in less time compared to the uncompressed approach. This means that the average time for queries on random position is $O(H_0(T))$.

To translate this approach to quaternary trees, the first idea would be to simply handle two bits of code at a time, in the same fashion as it's done for uncompressed quad wavelet trees, but the main problem for this implementation is to handle odd length codes, that would be represented in the quad vectors as padded, reducing the effectiveness of the compression.

The better solution is to create a quaternary wavelet tree based on a quaternary Huffman-tree. This can be achieved using a similar procedure as the one for binary Huffman-codes and was already proven optimal by Huffman for every sized alphabet [49]. The use of higher degree Huffman trees doesn't guarantee an optimal encoding for integer codes (as the binary one does), but still guarantees the optimal encoding for the chosen alphabet and as such allows us to fully take advantage of every quad vector at each level. This solution also uses only quad vectors and as such can take advantage of all the optimizations that are used in the uncompressed version of the quad wavelet tree, while also using a compression scheme that can introduce meaningful compression rates. The space of this implementation requires at most $n(H_0(T) + 2) + o(n(H_0(T) + 2)) + O(\sigma \log n)$ bits of space, due to the fact that using a bigger alphabet implies the need to use more bits to represent each symbol. This space upper bound can be further reduced if we know the distribution of the symbols in the text T, in particular if we know the probability of the most frequent symbol P_1 we can reduce the upper bound to $n(H_0(T) + 2r) + o(n(H_0(T) + 2r)) + O(\sigma \log(n))$, where r is the redundancy of the code (in number of symbols, which require 2 bits each) that can be bounded by $r \leq P_1 \cdot \frac{4}{\ln(4)} + \log_4(3) + \log_4(\log_4 e) - \log_4(e) + \frac{1}{3} \approx 2.885 \cdot P_1 + 0.168$ [50].

Table 1 shows the compression rates obtained by using the different Huffman compressions on the datasets we used in our experimental evaluation, that are described in Section 7.1. As we can see, the quaternary Huffman coding is always better than the naive approach (binary + padding), and in the worst case for these datasets each symbol requires 0.11 bits more compared to the optimal solution (binary).

	$\lceil \log \sigma \rceil$	Entropy	Huffman	Huffman + pad	4-ary Huffman
English	8	4.57	4.61	5.07	4.69
DNA	2	2.00	2.00	2.00	2.00
CC	8	6.20	6.23	6.88	6.34
Sources	8	5.67	5.70	6.40	5.81

Table 1: Number of bits/symbol used in the different approaches for Huffman coding

7 Experimental Evaluation

We first discuss our experimental setup. Then, in Section 7.1, we show the benefit of using 4-ary wavelet trees and approximate rank queries. Finally, in Section 7.2, we compare quad and bit vectors.

Experimental Setup

For our experiments, we used a machine equipped with an Intel Core i9-9900KF (8 cores, 2 threads per core, 3 60GHz base clock, and cache sizes: 64 KB L1I and 64 KB L1D per core, 256 KB L2I+D per core, and 16 MB L3I+D, shared between all cores) and 64 GB DDR4 RAM running Ubuntu 23.10 LTS kernel version6.5.0-35-generic. All experiments were performed using a single thread, with hyperthreading and turbo boost disabled. C++ code of competitors was compiled with GCC 11.1.0 with flags `O3` and `march=native` and Rust code was compiled using `cargo build --release`. [4] We ran each experiment ten times (10M queries for each run) and report the average running time. We generate all queries in advance as follows. Let $S[0, n)$ be the indexed sequence. Each access query asks to access the symbol at a random position in the sequence. For rank queries, we generate a random position $i \in [0, n)$ and use $\langle i, S[i] \rangle$ as a rank query. For select queries, 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. Our code to reproduce the experiments is available at `https://github.com/AngeloSav/qwt-experiments`.

	Dataset	sdsl_wm	sdsl_huffwt	pasta_wm	simple-sds_wm	sucds_wm	WT	HWT	QWT	HQWT
Access	English	1172	610	1391	1108	1177	1293	692	646	**390**
	Sources	1156	803	1352	1295	1212	1263	878	648	**489**
	CC	1180	906	1394	1148	1213	1271	973	645	**535**
	DNA	187	191	293	157	283	250	248	**113**	141
Rank	English	1301	661	1492	1260	1272	1309	693	495	**334**
	Sources	1220	835	1433	1228	1265	1242	874	523	**399**
	CC	1261	946	1506	1250	1260	1260	978	523	**435**
	DNA	230	230	280	230	386	256	255	133	**129**
Select	English	3856	1835	2954	3102	3303	2989	1704	1707	**1017**
	Sources	3754	2334	2884	3163	3336	2952	2092	1719	**1244**
	CC	3752	2643	3024	3154	3336	3014	2320	1728	**1366**
	DNA	810	733	644	633	1050	700	698	401	**399**
Space	English	11.9	6.9	8.0	12.7	10.7	8.3	**4.8**	9.0	5.3
	Sources	11.9	8.5	8.0	13.0	10.7	8.3	**5.9**	9.0	6.6
	CC	11.9	9.3	8.0	12.9	10.7	8.3	**6.5**	9.0	7.2
	DNA	3.0	3.0	**2.0**	3.0	4.0	2.1	2.1	2.3	2.3

Table 2: *Latency* of access, rank, and select queries (row 1–3) given in ns and the *space* (row 4) is given in GiB. In bold the best performance for each row. All results for 8 GiB input files.

7.1 Evaluation of 4-Ary Wavelet Trees

We compare our 4-ary wavelet trees with other wavelet trees. Note that we compare wavelet *matrices* if available, as those are faster in practice than wavelet trees, however, all implementations mentioned below contain also support for both wavelet trees. To the best of our knowledge, there exists no other k-ary wavelet

[4] Our Rust implementation is freely available at `https://github.com/rossanoventurini/qwt`.

tree implementation for $k > 2$ and no other (uncompressed) wavelet tree implementation with access, rank, and select support. In the following, sdsl_wm denotes wavelet matrices built on bit vectors of the SDSL library (`wm_int`) [11]. We also included sdsl_huffwt, a Huffman shaped wavelet *tree* implementation from the SDSL library that uses the same bit vectors as sdsl_wm. A wavelet matrix implementation built on bit vectors of the PASTA-toolbox library, using the most space-efficient rank and select data structures [17], is denoted by pasta_wm. Additionally, we also implemented in Rust our own version of binary balanced and Huffman-shaped wavelet matrices (respectively WT and HWT), that use the same bit vector layout as pasta_wm. QWT256Pfs and QWT512 are our implementations of wavelet matrices built on quad vectors with blocks of size 256 and 512 symbols per block, cf. Section 3. QWT∗Pfs denotes the usage of our predictive model (see Section 5). We wanted to include a wavelet matrix based on learned compressed rank and select data structures [23], however, here query times are significantly greater, making the plots harder to read. Furthermore, the experiments for inputs > 1 GiB did not finish in reasonable time. Therefore, we excluded this implementation from the results. As specified in Section 5.3, we do not implement the correction of our estimate using discriminant positions, as for ASCII texts we can simply afford to prefetch more cache lines.

As inputs, we use text prefixes with sizes ranging from 16 KiB to 8 GiB, generated from the following datasets. English is the concatenation of all 35 750 English text files from the Gutenberg Project. We removed the headers related to the project, leaving just the real text. DNA are FASTQ files from the 1000 Genomes Project, where we considered only the raw sequence and kept only the characters `A`, `C`, `G`, and `T`, to obtain a very small alphabet. CC is a concatenation of the WET files of Common Crawl corpus, i.e., a web crawl without HTML tags. Here, we removed all meta information added by the corpus. Sources is a concatenation of plaintext files from some of the biggest open source projects available on GitHub (such as the Linux kernel, the GCC compiler and the LLVM project). Note that we did not use the famous Pizza&Chili corpus, as we needed inputs larger than 2 GiB. We did not use inputs with larger alphabets as most competitors do not support those.

Comparison Between Quad Wavelet Trees

We first compare our different implementations of quad wavelet trees, using blocks of either 256 or 512 symbols and with and without our prefetching model. In figure Figure 5 we show the result of our tests on the English dataset, as the result is similar for the other datasets too. As we can notice, for both rank and access operations, the version using 256-symbol blocks is never slower than the 512 one, while being only marginally slower when answering select queries. Furthermore, when the text is bigger than the L3 cache, the introduction of prefetching allows our wavelet trees to suffer less cache misses. In the case of QWT256Pfs, it results in a speedup of up to 1.55 compared to our wavelet tree without prediction model with the same block size. The same results are also reflected on our compressed implementations, where we can see that the data structure with 256 block size is still better than the 512 one, with only a small loss in performance for the select operation. Here too, the introduction of our prefetching model allow for faster rank queries, resulting in a speedup of up to 1.6 for HQWT256Pfs compared to HQWT256.

From now on we will use QWT and HQWT to refer to our wavelet trees that use a block size of 256 symbols and exploit our prefetching model.

Access, Rank, and Select Queries

Figure 6 shows the query result of our experimental evaluation for the uncompressed data structures. For *access* and *select* queries, the behavior of all algorithms is very similar. Here, pasta_wm is always the slowest. For inputs for which the wavelet tree fits into the L2 cache, WT, sdsl_wm and QWT have a similar query time. This is to be expected, as there are still not many cache misses during querying. There is a steep increase in query time as soon as the wavelet tree does not fit into the L3 cache. However, QWT's query time does not increase as fast, resulting in a speedup of up to 1.73 (access) and 2.17 (select) compared to sdsl_wm.

Finally, for *rank* queries the behavior of the three previously discussed algorithms is similar. We now also include the wavelet tree QWT using our prediction model. We can actually see it being effective as soon as the wavelet trees does not fit into the L3 cache of the core complex, i.e., as soon as we expect more and more cache misses during querying. Here, we can notice an even greater difference in query times, resulting in a speedup of up to 3.17 compared to sdsl_wm.

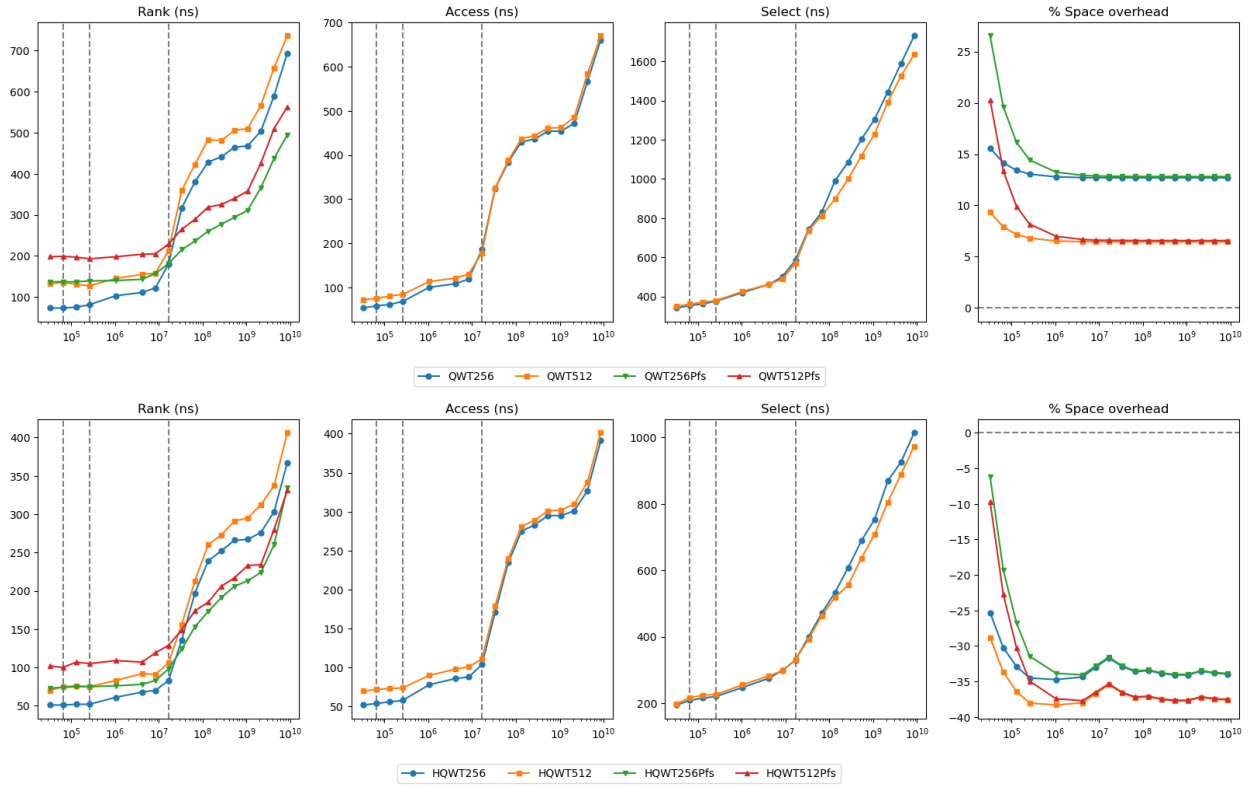

Figure 5: Comparison of all our new wavelet trees implementations. The first row shows a comparison between our four uncompressed implementations, while the second one compares our compressed implementations. On the x-axis the size of the original text. The first three columns represent the latency of the rank, access and select operations and the last column shows the space overhead of the wavelet tree compared to the input (storing one character per byte). The vertical dashed gray lines indicate the L1, L2, and L3 cache sizes of the CPU used for these experiments.

In Figure 7 we show a comparison of the different compressed implementations of the wavelet trees. Note that we did not include the results for the DNA dataset, as it is not compressible using Huffman coding because it is composed of 4 equally distributed symbols. As we can notice, in all compressible datasets, the shift to a Huffman shape greatly lowers the query latency by a factor that is proportional to the entropy of the text, as expected. We can also notice the gains from the two can be combined in order to obtain an even faster wavelet tree that uses the quaternary Huffman structure (HQWT), resulting in a speedup of up to 3.0x (access) and 3.7x (select) compared to sdsl_wm and up to 1.6x (both access and select) compared to QWT. The HQWT also exploits our prefetching model, that allows us to obtain a speedup proportional to the average number of levels accessed, making it the fastest implementation for our tests. This last approach, allows us to achieve even faster query times for large sequences, that can result in a speedup of up to 3.8x compared to sdsl_wm (2.1x compared to sdsl_huffwt).

A summary of the results for the largest inputs can be found in Table 2, where we also highlighted the best-performing implementation for each test. We also included two other popular rust implementations of wavelet matrices: simple-sds_wm, from the simple succinct data structure library[5], and sucds_wm, from the sucds library[6], both employing efficient bit vector implementations based on Vigna's data structures[15].

[5]available at `https://github.com/jltsiren/simple-sds`
[6]available at `https://github.com/kampersanda/sucds`

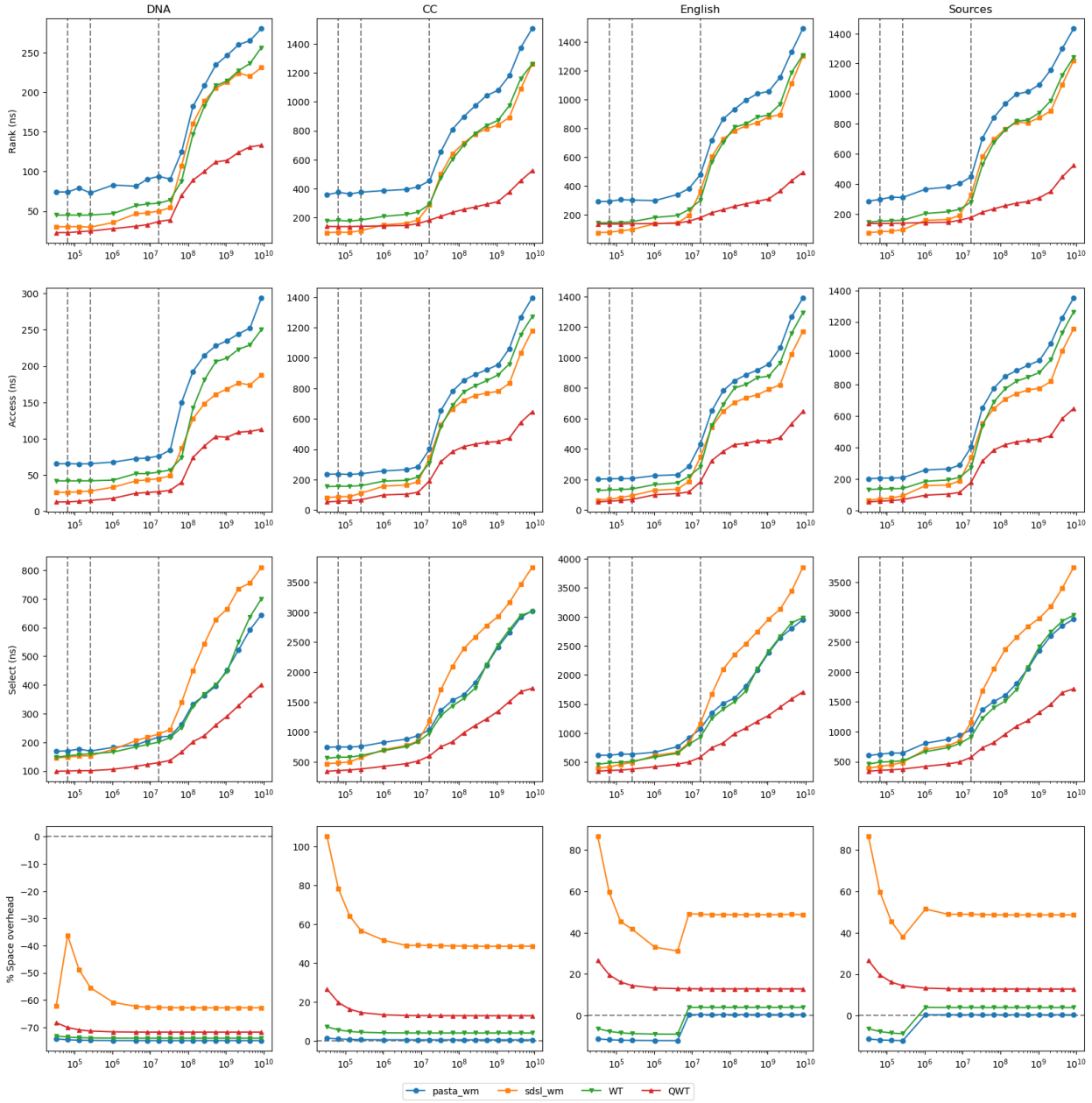

Figure 6: Comparison of our new uncompressed wavelet trees with competitors. On the x-axis the size of the original text. The first three rows give the access, rank, and select query latency of the implementations, and the last row shows the space overhead of the wavelet tree compared to the input (storing one character per byte). The vertical dashed gray lines indicate the L1, L2, and L3 cache sizes of the CPU used for these experiments.

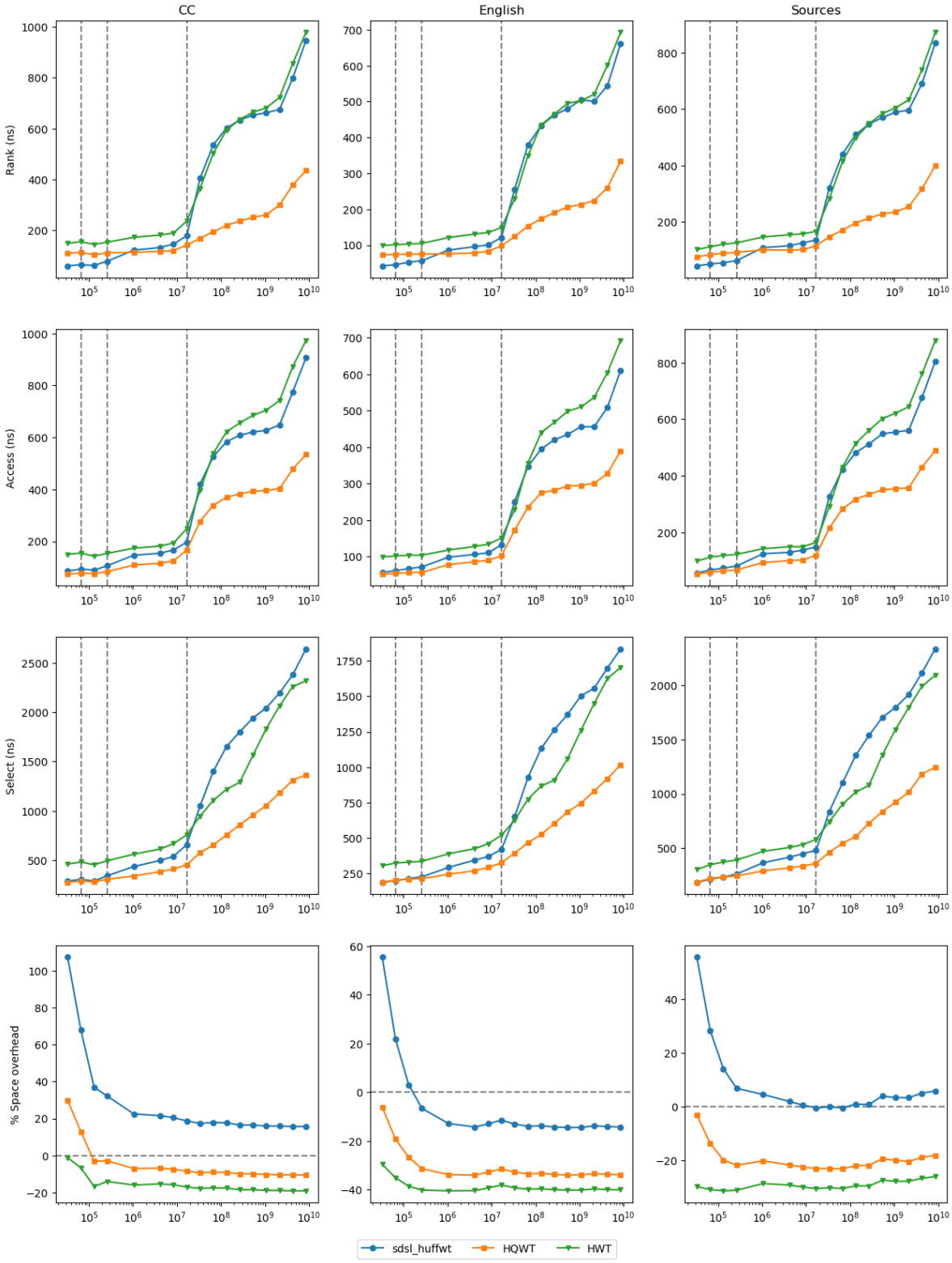

Figure 7: Comparison of our new compressed wavelet trees with competitors. On the x-axis the size of the original text. The first three rows give the access, rank, and select query latency of the implementations, and the last row shows the space overhead of the wavelet tree compared to the input (storing one character per byte). The vertical dashed gray lines indicate the L1, L2, and L3 cache sizes of the CPU used for these experiments.

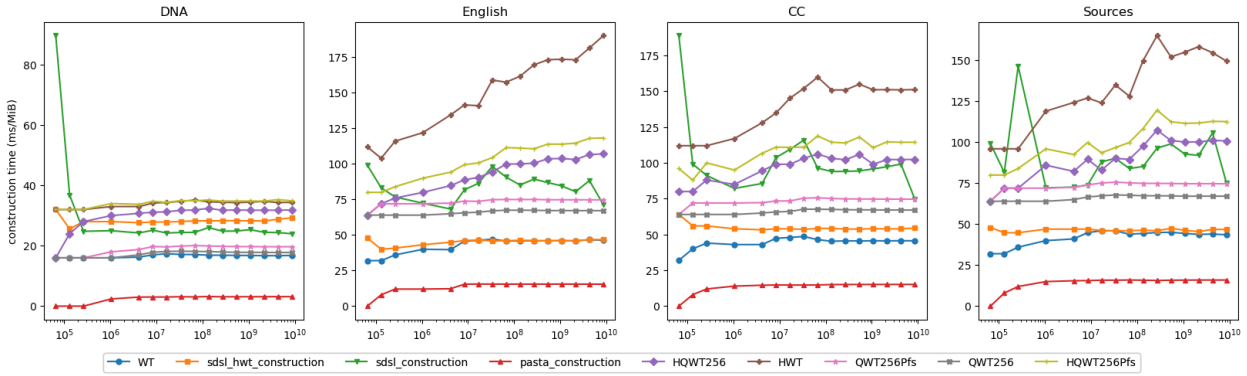

Figure 8: Wavelet tree construction times normalized by input size. On the x-axis the size of the original text. The vertical dashed gray lines indicate the L1, L2, and L3 cache sizes of the CPU used for these experiments.

Space-Overhead and Construction

The last row in Figure 6 shows the additional space required by the wavelet tree compared to the input. Additional space $< 0\,\%$ for uncompressed wavelet trees is due to the compression of the alphabet, where the wavelet tree requires less than 8 levels and the input requires one byte per character. Similarly, stark increases are due to a new level being required, which is not the case for our wavelet trees, as we do not use a bit vector for the last level, even if it would suffice, initially wasting some space in the progress. The spikes in the beginning are due to the general overhead of the data structures. Overall, the space-overhead is mostly based on the used rank and select data structures. Since our quad vectors support rank and select using less memory than the bit vectors in the SDSL, our wavelet trees are also more memory efficient—around $75\,\%$ less space-overhead. Without surprise, pasta_wm is the most space-efficient wavelet tree, however, this comes with slower (wavelet tree) queries, with WTbeing a close second due to using the same memory layout.

These considerations are also applicable to the compressed wavelet trees (last row of Figure 7), where we can see that sdsl_huffwt is the worst-performing data structure, mainly due to the use of less space-efficient bit vectors, while the HWTand HQWT implementations can take better advantage of the compression scheme, achieving lower space overheads. The higher space required for HQWT compared to HWTis mainly due to the use of quad vector, as it is about the same as the one between QWT and WT.

The construction time of the wavelet trees is shown in Figure 8. Here, pasta_wm is the fastest to construct, followed by WT. Our QWT256 and QWT256Pfs require similar time, but the additional time required for the prediction model is visible, and sdsl_wm is the slowest to construct. As per the compressed data structures, our Rust implementation for both the binary (HWT) and quaternary (HQWT256) wavelet matrix take more time to construct than their uncompressed counterpart, with the time growing as the file size get larger, especially when the alphabet size is large: this is due to the fact that we need to lookup the code for each symbol every time from a codebook, that when it's too big to be fully kept in cache can cause slowdowns. The HQWT256 and HQWT256Pfs result faster to construct than HWT because we handle two bits of the codes at the time, meaning that we conduct half of the codebook lookups. Notice that sdsl_huffwt is faster to construct than most of the wavelet matrices, because, as it is implemented as a wavelet *tree*, it can better exploit the shape of the Huffman tree to achieve faster build times. Note that we are not focusing on the construction and highly tuned construction algorithms for wavelet trees exist [30].

7.2 Evaluation of Quad Vectors

We now compare our quad vectors with the bit vectors used in the wavelet trees in Section 7.1 to show that the speedup is actually due to the improvements presented in this paper and not only due to better rank and select support for quad vectors, in that a single query is not faster than that of a simple bit vector.

sdsl_bv is the implementation of bit vectors of the SDSL library [11]. For rank queries, the bit vector is enhanced with rank_support_v, and it uses select_support_mcl to support select queries, i.e., Clark and Munro's

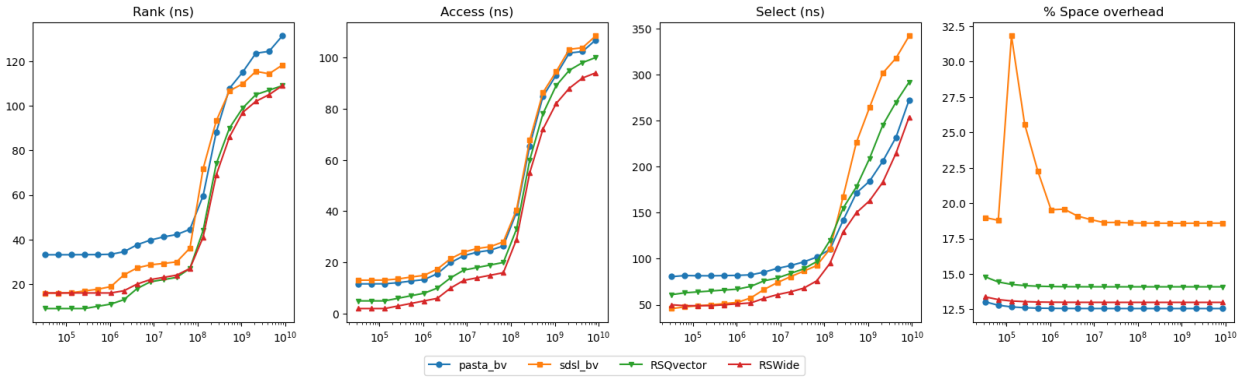

Figure 9: Comparison of access, rank, and select query latency for our *quad* vectors with *bit* vectors.

approach to compute select queries [12]. pasta_bv is the implementation of bit vectors of the PASTA library [17].

The bit vector is enhanced using the FlatRankSelect data structure to support rank and select queries. RSWideis the bit vector we used for our rust implementation of both binary wavelet trees WT and HWT. RSQvector is our implementation of quad vectors with blocks of size 256 (see Section 4), as it is used as the basis for the wavelet trees we tested against our competitors.

In these experiments, we generate random bit/quad sequences containing from 16 K to 8 G symbols, i.e., the number of symbols contained in each level of the wavelet trees. Each bit has a 50 % chance of being set and each quad has a 25 % percent chance of appearing.

The running time of access, rank, and select queries of the bit vectors is depicted in Figure 9. There, we can see that access queries as well as rank and select queries for larger inputs require roughly the same time. In particular, RSWiderequires roughly the same time (rank) or is faster (select) than RSQvector.

The additional space usage is as expected and similar to the wavelet trees. Again, sdsl_bv has a spike for smaller inputs, which can be explained by the general overhead of the data structure starting at this input size.

Overall, the evaluation of the bit and quad vectors show that the improvements of three wavelet tree queries considered here are solely due to the algorithmic ideas presented in this paper.

8 Conclusion

We have presented two improvements for wavelet trees that achieve a speedup for access and select queries of up to 2 and for rank queries—which are the most important queries in many applications—up to 3. To this end, we first changed the underlying tree structure from a binary tree to a 4-ary tree, reducing the number of cache misses in the process. Furthermore, we introduced the Rank with Additive Approximation problem and showed a small predictive model that solved this problem in practice. This predictive models allow us to prefetch all data necessary for rank queries, resulting in a better speedup compared to access and select queries. These approaches can be also applied to Huffman-shaped wavelet trees, allowing, for compressible sequences, to achieve a speedup of up to 3 for access and select queries and up to 3.8 for rank queries, while also residing in compressed space.

It remains an open problem to combine the 4-ary wavelet tree layout with the sublinear construction algorithm based on vectorized instructions. Additionally, we want to explore bit vectors for even larger alphabets, as our experiments indicate that the techniques proposed in this paper benefit from wavelet trees with more levels.

Acknowledgements

Thanks to Matteo Ceregini for the preliminary work on the original C++ implementation of Quad Vectors and the Quad Wavelet Tree and his work on the preliminary version of the paper.

References

[1] Ceregini Matteo, Kurpicz Florian, Venturini Rossano. Faster Wavelet Tree Queries in *Proceedings of the Data Compression Conference (DCC)*:223-232 2024.

[2] Grossi Roberto, Gupta Ankur, Vitter Jeffrey Scott. High-Order Entropy-Compressed Text Indexes in *ACM-SIAM Symposium on Discrete Algorithms*:841–850ACM/SIAM 2003.

[3] Ferragina Paolo, Manzini Giovanni. Opportunistic Data Structures with Applications in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*:390–398IEEE Computer Society 2000.

[4] Gagie Travis, Navarro Gonzalo, Prezza Nicola. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space *Journal of the ACM.* 2020;67:2:1–2:54.

[5] Ferragina Paolo, Giancarlo Raffaele, Manzini Giovanni. The Myriad Virtues of Wavelet Trees *Information and Computation.* 2009;207:849–866.

[6] Grossi Roberto, Vitter Jeffrey Scott, Xu Bojian. Wavelet Trees: From Theory to Practice in *Proceedings of the First International Conference on Data Compression, Communications and Processing*:210–221IEEE Computer Society 2011.

[7] Makris Christos. Wavelet Trees: A Survey *Computer Science and Information Systems.* 2012;9:585–625.

[8] Navarro Gonzalo. Wavelet Trees for All *Journal of Discrete Algorithms.* 2014;25:2–20.

[9] Gog Simon, Kärkkäinen Juha, Kempa Dominik, Petri Matthias, Puglisi Simon J.. Fixed Block Compression Boosting in FM-Indexes: Theory and Practice *Algorithmica.* 2019;81:1370–1391.

[10] Claude Francisco, Navarro Gonzalo, Pereira Alberto Ordóñez. The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets *Information Systems.* 2015;47:15–32.

[11] Gog Simon, Beller Timo, Moffat Alistair, Petri Matthias. From Theory to Practice: Plug and Play with Succinct Data Structures in *Experimental Algorithms*;8504 of *Lecture Notes in Computer Science*:326–337Springer 2014.

[12] Clark David R., Munro J. Ian. Efficient Suffix Trees on Secondary Storage (extended Abstract) in *ACM-SIAM Symposium on Discrete Algorithms*:383–391ACM/SIAM 1996.

[13] Jacobson Guy. Space-Efficient Static Trees and Graphs in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*:549–554IEEE Computer Society 1989.

[14] Patrascu Mihai. Succincter in *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*:305–313IEEE Computer Society 2008.

[15] Vigna Sebastiano. Broadword Implementation of Rank/Select Queries in *Experimental Algorithms*;5038 of *Lecture Notes in Computer Science*:154–168Springer 2008.

[16] Zhou Dong, Andersen David G., Kaminsky Michael. Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences in *Experimental Algorithms*;7933 of *Lecture Notes in Computer Science*:151–163Springer 2013.

[17] Kurpicz Florian. Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors in *String Processing and Information Retrieval*;13617 of *Lecture Notes in Computer Science*:257–272Springer 2022.

[18] González Rodrigo, Grabowski Szymon, Mäkinen Veli, Navarro Gonzalo. Practical Implementation of Rank and Select Queries in *Experimental Algorithms*:27–38 2005.

[19] Kim Dong Kyue, Na Joong Chae, Kim Ji Eun, Park Kunsoo. Efficient Implementation of Rank and Select Functions for Succinct Representation in *Experimental Algorithms*;3503 of *Lecture Notes in Computer Science*:315–327Springer 2005.

[20] Navarro Gonzalo, Providel Eliana. Fast, Small, Simple Rank/Select on Bitmaps in *Experimental Algorithms*;7276 of *Lecture Notes in Computer Science*:295–306Springer 2012.

[21] Arroyuelo Diego, Weitzman Manuel. A Hybrid Compressed Data Structure Supporting Rank and Select on Bit Sequences in *Proceedings of the 39th International Conference of the Chilean Computer Science Society (SCCC)*:1–8IEEE 2020.

[22] Beskers Kai, Fischer Johannes. High-Order Entropy Compressed Bit Vectors with Rank/Select *Algorithms.* 2014;7:608–620.

[23] Boffa Antonio, Ferragina Paolo, Vinciguerra Giorgio. A Learned Approach to Design Compressed Rank/Select Data Structures *ACM Transactions on Algorithms.* 2022.

[24] Kärkkäinen Juha, Kempa Dominik, Puglisi Simon J.. Hybrid Compression of Bitvectors for the FM-Index in *Proceedings of the Data Compression Conference*:302–311IEEE 2014.

[25] Raman Rajeev, Raman Venkatesh, Satti Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-Ary Trees, Prefix Sums and Multisets *ACM Transactions on Algorithms.* 2007;3:43.

[26] Pibiri Giulio Ermanno, Kanda Shunsuke. Rank/Select Queries over Mutable Bitmaps *Information Systems.* 2021;99:101756.

[27] Prezza Nicola. A Framework of Dynamic Data Structures for String Processing in *Experimental Algorithms*;75 of *LIPIcs*:11:1–11:15Schloss Dagstuhl - Leibniz-Zentrum für Informatik 2017.

[28] Babenko Maxim A., Gawrychowski Pawel, Kociumaka Tomasz, Starikovskaya Tatiana. Wavelet Trees Meet Suffix Trees in *Proceedings of the 2015 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*:572–591SIAM 2015.

[29] Munro J. Ian, Nekrich Yakov, Vitter Jeffrey Scott. Fast construction of wavelet trees *Theoretical Computer Science.* 2016;638:91–97.

[30] Dinklage Patrick, Fischer Johannes, Kurpicz Florian, Tarnowski Jan-Philipp. Bit-Parallel (Compressed) Wavelet Tree Construction in *Proceedings of the Data Compression Conference (DCC)*:81–90IEEE 2023.

[31] Kaneta Yusaku. Fast Wavelet Tree Construction in Practice in *String Processing and Information Retrieval*;11147 of *Lecture Notes in Computer Science*:218–232Springer 2018.

[32] Shun Julian. Improved parallel construction of wavelet trees and rank/select structures *Information and Computation.* 2020;273:104516.

[33] Labeit Julian, Shun Julian, Blelloch Guy E.. Parallel lightweight wavelet tree, suffix array and FM-index construction *Journal of Discrete Algorithms.* 2017;43:2–17.

[34] Fuentes-Sepúlveda José, Elejalde Erick, Ferres Leo, Seco Diego. Parallel construction of wavelet trees on multicore architectures *Knowledge and Information Systems.* 2017;51:1043–1066.

[35] Dinklage Patrick, Ellert Jonas, Fischer Johannes, Kurpicz Florian, Löbel Marvin. Practical Wavelet Tree Construction *ACM Journal of Experimental Algorithmics (JEA).* 2021;26:1.8:1–1.8:67.

[36] Dinklage Patrick, Fischer Johannes, Kurpicz Florian. Constructing the Wavelet Tree and Wavelet Matrix in Distributed Memory in *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*:214–228SIAM 2020.

[37] Ellert Jonas, Kurpicz Florian. Parallel External Memory Wavelet Tree and Wavelet Matrix Construction in *String Processing and Information Retrieval*;11811 of *Lecture Notes in Computer Science*:392–406Springer 2019.

[38] Alanko Jarno N., Biagi Elena, Puglisi Simon J., Vuohtoniemi Jaakko. Subset Wavelet Trees in *Experimental Algorithms*;265 of *LIPIcs*:4:1–4:14Schloss Dagstuhl - Leibniz-Zentrum für Informatik 2023.

[39] Ferragina Paolo, Manzini Giovanni, Mäkinen Veli, Navarro Gonzalo. Compressed representations of sequences and full-text indexes *ACM Transactions on Algorithms.* 2007;3:20.

[40] Belazzougui Djamal, Cáceres Manuel, Gagie Travis, et al. Block Trees *Journal of Computer and System Sciences.* 2021;117:1–22.

[41] Köppl Dominik, Kurpicz Florian, Meyer Daniel. Faster Block Tree Construction in *Proceedings of the 31st Annual European Symposium on Algorithms (ESA 2023)*;274 of *LIPIcs*:74:1–74:20Schloss Dagstuhl - Leibniz-Zentrum für Informatik 2023.

[42] Prezza Nicola. Optimal Rank and Select Queries on Dictionary-Compressed Text in *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching*;128 of *LIPIcs*:4:1–4:12Schloss Dagstuhl - Leibniz-Zentrum für Informatik 2019.

[43] Prezza Nicola. On String Attractors in *Proceedings of the 19th Italian Conference on Theoretical Computer Science (ICTCS 2018)*;2243 of *CEUR Workshop Proceedings*:12–16CEUR-WS.org 2018.

[44] Belazzougui Djamal, Cording Patrick Hagge, Puglisi Simon J., Tabei Yasuo. Access, Rank, and Select in Grammar-compressed Strings in *Algorithms - ESA 2015*;9294 of *Lecture Notes in Computer Science*:142–154Springer 2015.

[45] Pereira Alberto Ordóñez, Navarro Gonzalo, Brisaboa Nieves R.. Grammar compressed sequences with rank/select support *Journal of Discrete Algorithms.* 2017;43:54–71.

[46] Elias Peter. Efficient Storage and Retrieval by Content and Address of Static Files *Journal of the ACM.* 1974;21:246–260.

[47] Fano Robert Mario. On the number of bits required to implement an associative memory tech. rep.MIT, Computer Structures Group 1971. Project MAC, Memorandum 61".

[48] Orlandi Alessio, Venturini Rossano. Space-efficient substring occurrence estimation *Algorithmica.* 2016;74:65–90.

[49] Huffman David A.. A Method for the Construction of Minimum-Redundancy Codes *Proceedings of the Institute of Radio Engineers (IRE).* 1952;40:1098-1101.

[50] Gallager R.. Variations on a theme by Huffman *IEEE Transactions on Information Theory.* 1978;24:668-674.