Faster Wavelet Tree Queries

Matteo Ceregini*, Florian Kurpicz†, and Rossano Venturini*

*University of Pisa
ceregini@studenti.unipi.it
rossano.venturini@unipi.it

†Karlsruhe Institute of Technology
kurpicz@kit.edu

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—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. Our improved wavelet tree representation and predictive model allows us to speed up queries by a factor of 2–3.

1 Introduction

Wavelet trees [19] 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 to access the text. This makes them an important building block for compressed full-text indices, e.g., the FM-index [12] or the r-index [16], where they are used to answer rank queries during the pattern matching algorithm—the backwards search. More applications are discussed in multiple surveys [14, 20, 25, 27]. A lot of research has been focused on the efficient construction of wavelet trees. But, there exists barely any research focusing on the query performance of wavelet trees. The wavelet matrix, 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. The main building block of wavelet trees are bit vectors with binary rank and select support. There exist multiple data structures for rank and select support introducing different time-space trade-offs. Faster binary rank and select queries directly translate to faster queries on wavelet trees. Still, improving only binary rank and select data structures does not fully utilizes the full range of optimizations.

Our Contributions. We show that using a 4-ary wavelet tree instead of the usual binary wavelet tree 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* [17]. Furthermore, we introduce the *Rank with Additive Approximation* problem and show how 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.

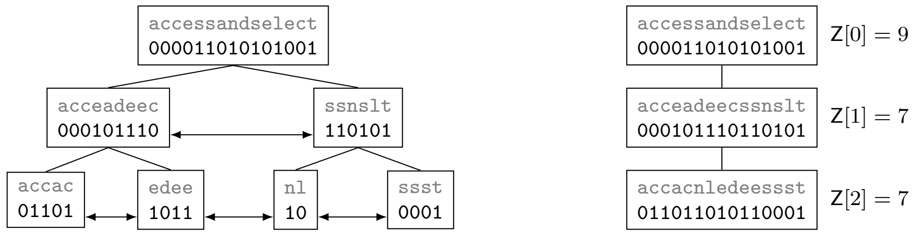

Figure 1: Wavelet tree (left) and a wavelet matrix (right) for the text `accessandselect` over the alphabet $\{\mathtt{a}\ (000)_2, \mathtt{c}\ (001)_2, \mathtt{d}\ (010)_2, \mathtt{e}\ (011)_2, \mathtt{l}\ (100)_2, \mathtt{n}\ (101)_2, \mathtt{s}\ (110)_2, \mathtt{t}\ (111)_2\}$ (bit representation 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.

Preliminaries. 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:

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

On bit vectors of length n, both queries can be answered in $O(1)$ time requiring $o(n)$ additional bits[6, 21]. The *most significant bit* (MSB) of a character is the bit with the highest value. 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 its ℓ MSBs.

A *wavelet tree* [19] is a binary tree, where each node represents a subsequence of the text. Each node contains character sharing a length-k bit-prefix. The root of a wavelet tree represents all characters, i.e., sharing the empty length-0 bit-prefix. Then, whenever we visit the left child of a node that represents characters with bit-prefix α, it represents character with-bit prefix $\alpha\mathtt{0}$. The right child represents all characters sharing bit-prefix $\alpha\mathtt{1}$. 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. In the following, we consider level-wise wavelet trees.

The *wavelet matrix* [7] 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. This removes the tree structure of the wavelet tree. However, the same intervals as in the wavelet tree occur on each level, just 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. In the following, we use wavelet tree to refer to both wavelet tree and wavelet matrix.

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

Related Work. Practical and well-performing implementations of rank and select structures can be found in the SDSL [17]. The currently most space efficient rank and select support for a size-u bit vector that contains n ones requires only $\log\binom{u}{n} + \frac{u}{\log u} + \tilde{O}(u^{\frac{3}{4}})$ bits (including the bit vector) [29]. In practice, the currently fastest select data structures are by Vigna [34]. However, they still require much more space than the currently most space-efficient data structures [23, 35].

Let T be a text of length n over an alphabet of size σ. The best sequential wavelet tree construction algorithms require $O(n \log \sigma / \sqrt{\log n})$ time [1, 26]. These approaches make use of vectorized instructions and also have been implemented [10, 22]. In shared memory, wavelet trees can be computed in $O(\sigma + \log n)$ time requiring only $O(n \log \sigma / \sqrt{\log n})$ work [33]. In practice, the fastest construction algorithms are based on domain decomposition [24, 15] and utilize a bottom-up construction as base-case [8]. Wavelet trees can also be computed in other distributed [9] and external memory [11]. To compress a wavelet tree, it is constructed for the Huffman-compressed text. A fully functional Huffman-shaped wavelet tree requires $n\lceil H_0(T)\rceil(1 + o(1))$ bits of space, where $H_0(T)$ is the zeroth order entropy of T. In theoretical work, multi-ary wavelet trees are used to reduce query time in the RAM model to $\Theta(\log_{\log n} \sigma)$ [13].

Recently, a practical block tree implementation has been introduced [2]. 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. Further dictionary-compressed representations allow for rank and select support in optimal time in compressed space [31] with respect to the size of a string attractor [32] of the text. For a grammar of size g and an alphabet of size σ, rank and select support requires $O(\sigma g)$ space [3, 30]. Here, queries can be answered in $O(\log n)$ time.

2 4-Ary Wavelet Trees and Quad Vectors

When answering queries using a wavelet tree in practice, the query is translated to $O(\log \sigma)$ binary rank and select queries. On each level of the wavelet tree, the binary rank and select queries will result in at least one cache miss, which 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.

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. 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 α, its four children represent characters with bit-prefix $\alpha\texttt{00}$, $\alpha\texttt{01}$, $\alpha\texttt{10}$, and $\alpha\texttt{11}$. Similarly to the binary case, there exist 4-ary wavelet matrices.

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 [23, 35].

Function $Rank_\alpha(i)$
  $r_0 = i,\ b_0 = 0$
  for $k = 1, \dots, \ell + 1$ **do**
    $\alpha_k = (\alpha \ >> \ 2 * (\ell - 1 - k))\ \&\ 3$, $\texttt{offset} = C_k[\alpha_k]$
    $b_k = Q[k].rank_{\alpha_k}(b_{k-1}) + \texttt{offset}$
    $r_k = Q[k].rank_{\alpha_k}(r_{k-1}) + \texttt{offset}$
  return $r_\ell - b_\ell$

Algorithm 3.1. 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.

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 $\alpha \in \{\texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}\}$: *Superblocks* cover 2048 symbols and store the number of occurrences before the start of the super block. *Blocks* cover 256 symbols and store the number of occurrences before the start of the block within the super block. The smaller size (super)blocks result in double the space-overhead but halve the cache misses, as the pertinent information fits into one cache line. We refer to the full paper [5] for a more detailed description.

3 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, prefecthing helps hiding memory latency and reducing the impact of memory access delays on the CPU's execution pipeline.

There is *hardware* and *software* prefetching. Hardware prefetching is implemented within the CPU's microarchitecture and is driven by the hardware itself. Software prefetching is controlled by the programmer or the compiler through explicit instructions and requires a deep understanding of memory access patterns and the memory hierarchy, as incorrect 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 3.1. A $rank_\alpha(i)$ query on a wavelet tree has to 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. 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 i-th character. These two cache lines can be requested in parallel as they only depend on position i.

3.1 Predicting Cache Lines in a Quad Vector

This challenge led us to the definition of the *Rank with Additive Approximation* problem.

Definition 3.1 (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 and vice versa.

Lemma 3.1. *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. **TODO**: Changed to $< \epsilon$ and $> \epsilon$, because 2ϵ was too much. 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. □

Lemma 3.2. *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.*

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 of α, for some j which is a multiple of $\epsilon/2$.

We add *rank* support to the bit vectors B_{α}. A query $rank^{\approx}_{\alpha}(i)$ 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 $\tilde{r} = k \cdot \epsilon/2 - 1$ □

3.2 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 3.1. 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 [28]. 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 3.2 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 ϵ. The set $D_{k,\alpha}$ consists of the position in the quad vector corresponding to those occurrences. The positions in these sets can be stored within $\Theta(\log \epsilon)$ bits per position, e.g., by associating 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 3.3. *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 3.2. 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 $[r_{k-1}, d_{k-1}]$. Now, we have $r_k - \tilde{r}_k = rank_{\alpha_k}(r_{k-1}) - rank^{\approx}_{\alpha_k}(\tilde{r}_{k-1}) = rank_{\alpha_k}(d_{k-1}) - z - (rank_{\alpha_k}(d_{k-1}) - \Delta) = \Delta - z \leq \epsilon$. 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 $[d_{k-1}, r_{k-1}]$. Now, we have $r_k - \tilde{r}_k = rank_{\alpha_k}(r_{k-1}) - rank^{\approx}_{\alpha_k}(\tilde{r}_{k-1}) = rank_{\alpha_k}(d_{k-1}) + z - (rank_{\alpha_k}(d_{k-1}) - \Delta) = z + \Delta \leq r_{k-1} - \tilde{r}_{k-1} \leq \epsilon$.

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.

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 3.2 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 3.2 with error $\epsilon = 256$. 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. Cache lines needed by access and select queries cannot be predicted with our solution, as for each level there is a double dependency (position and symbol) on the result of the previous level.

4 Experimental Evaluation

For our experiments, we used a machine equipped with two AMD EPYC 7713 and 2 TB DDR4 RAM running Ubuntu 20.04.3 LTS kernel version 5.4.0-155. 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`. Our Rust implementation is available at `https://github.com/rossanoventurini/qwt`. We ran each experiment ten times (10M queries for each run) and report the average running time.

Note that we compare wavelet *matrices* if available, as those are faster in practice than wavelet trees. In the following, *sdsl_wm* denotes wavelet matrices built on bit vec-

Table 1: *Latency* of access, rank, and select queries (row 1–3) given in μs and the *space* (row 4) is given in GiB. The small number in parentheses is the *speedup of* QWM^{pfs}_{256} over the method represented by the column. All results for 8 GiB input files.

	input	sdsl_wm		sdsl_fbb		pasta_wm		sucds		$\text{QWM}^{\text{pfs}}_{256}$	
access	English	1270	(1.7×)	1506	(2.1×)	1618	(2.2×)	1122	(1.5×)	731	(1.0×)
	CC	1185	(1.7×)	1897	(2.7×)	1511	(2.2×)	1210	(1.7×)	700	(1.0×)
	DNA	239	(1.5×)	665	(4.2×)	353	(2.2×)	316	(2.0×)	157	(1.0×)
	Wiki	1216	(1.7×)	1712	(2.4×)	1681	(2.4×)	1198	(1.7×)	712	(1.0×)
rank	English	1498	(3.2×)	1474	(3.1×)	1797	(3.8×)	1408	(3.0×)	472	(1.0×)
	CC	1350	(2.8×)	1913	(3.9×)	1725	(3.5×)	1424	(2.9×)	490	(1.0×)
	DNA	321	(1.8×)	665	(3.8×)	394	(2.2×)	503	(2.9×)	176	(1.0×)
	Wiki	1402	(2.9×)	1649	(3.4×)	1855	(3.8×)	1442	(3.0×)	488	(1.0×)
select	English	4849	(2.2×)	—	—	4882	(2.2×)	4245	(1.9×)	2229	(1.0×)
	CC	4483	(2.0×)	—	—	4646	(2.1×)	4396	(1.9×)	2260	(1.0×)
	DNA	1032	(2.0×)	—	—	910	(1.7×)	1440	(2.8×)	521	(1.0×)
	Wiki	4546	(2.1×)	—	—	4956	(2.3×)	4349	(2.0×)	2185	(1.0×)
space	English		11.9		5.0		8.0		10.5		9.0
	CC		11.9		5.8		8.0		10.5		9.0
	DNA		3.0		2.2		2.0		3.9		2.3
	Wiki		11.9		5.8		8.0		10.5		9.0

tors of the SDSL library (`wm_int`) [17]. We also included the fastest compressed wavelet tree implementation in the SDSL—*sdsl_fbb* [18]. A wavelet matrix implementation built on bit vectors of the PASTA-toolbox library, using the most space-efficient rank and select data structures [23], is denoted by *pasta_wm*. Additionally, *sucds* is the wavelet matrix implementation in the sucds library[2]. QWM^{pfs}_{256} is our new wavelet matrix using quad vectors with block size 256 and the predictive model. We wanted to include a wavelet matrix based on learned compressed rank and select data structures [4], however, the experiments for inputs > 1 GiB did not finish in reasonable time.

As inputs, we use text prefixes between 16 KiB and 8 GiB in size, generated from the following datasets. *English* is the concatenation of all 35 750 English text files from the Gutenberg Project without project related headers. *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`. *CC* is a concatenation of the WET files of Common Crawl corpus, without project related headers. *Wiki* is a concatenation of XML data of the English Wikipedia from June 2023. We did not use inputs with larger alphabets as most competitors do not support those.

Experimental Results. Due to space constraints, we only consider the latency of access, rank, and select by forcing the input of each query to depend on the output

[2] `https://github.com/kampersanda/sucds`, last accessed 2023-11-08.

of the previous one. This is consistent with the in real settings, e.g., the backwards search. For a very thorough evaluation, we refer to the full paper [5], where we also give the throughput and show results for different input sizes.

We report a summary of our experimental results for inputs of size 8 GiB in Table 1. There, we can see that our new wavelet tree is always the fastest. For access and select queries, we achieve a speedup of 1.5–2.2 compared to sdsl_wm, the second fastest wavelet tree. When using our predictive model for rank queries, we can improve this speedup up to 3.2. For small alphabets, e.g., DNA, the predictive model provides no advantage, as there is only one level in our 4-ary wavelet tree rendering prefetching useless. The reported speedups are in line with other implementations, e.g., sucds, which provides slightly slower rank queries than sdsl_wm.

The space requirements of all wavelet trees are also unsurprising. The compressed wavelet tree sdsl_fbb requires the least space, the space efficient implementation pasta_wm requires just a little bit more than the input size, and our new solution is also very space efficient. Both, sdsl_wm and sucds require slightly more space due to the underlying rank and select data structures.

Overall, our new wavelet tree provides impressive speedups compared to all other available wavelet tree implementations. It is also very space-efficient, i.e., only compressed wavelet trees require significantly less space. In the future, we want to integrate our predictive model in compressed wavelet trees.

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), by the PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 "Precision medicine & personalized healthcare", by the "Algorithms, Data Structures and Combinatorics for Machine Learning" (MIUR-PRIN 2017), and by the "Algorithmic Problems and Machine Learning" (MIUR-PRIN 2022).

References

[1] M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *SODA*, 2015.

[2] D. Belazzougui, M. Cáceres, T. Gagie, P. Gawrychowski, J. Kärkkäinen, G. Navarro, A. Ordóñez Pereira, S. J. Puglisi, and Y. Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021.

[3] D. Belazzougui, P. H. Cording, S. J. Puglisi, and Y. Tabei. Access, rank, and select in grammar-compressed strings. In *ESA*, 2015.

[4] A. Boffa, P. Ferragina, and G. Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Trans. Algorithms*, 2022.

[5] M. Ceregini, F. Kurpicz, and R. Venturini. Faster wavelet trees with quad vectors. *CoRR*, abs/2302.09239, 2023.

[6] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, 1996.

[7] F. Claude, G. Navarro, and A. Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015.

[8] P. Dinklage, Jonas Ellert, J. Fischer, F. Kurpicz, and M. Löbel. Practical wavelet tree construction. *ACM J. Exp. Algorithmics*, 26:1.8:1–1.8:67, 2021.

[9] P. Dinklage, J. Fischer, and F. Kurpicz. Constructing the wavelet tree and wavelet matrix in distributed memory. In *ALENEX*, 2020.

[10] P. Dinklage, J. Fischer, F. Kurpicz, and J. Tarnowski. Bit-parallel (compressed) wavelet tree construction. In *DCC*, 2023.

[11] J. Ellert and F. Kurpicz. Parallel external memory wavelet tree and wavelet matrix construction. In *SPIRE*, 2019.

[12] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, 2000.

[13] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007.

[14] P. Ferragina, R.Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009.

[15] J. Fuentes-Sepúlveda, E. Elejalde, L. Ferres, and D. Seco. Parallel construction of wavelet trees on multicore architectures. *Knowl. Inf. Syst.*, 51(3):1043–1066, 2017.

[16] T. Gagie, G. Navarro, and N. Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[17] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, 2014.

[18] S. Gog, J. Kärkkäinen, D. Kempa, M. Petri, and S. J. Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019.

[19] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, 2003.

[20] R. Grossi, J. S. Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *CCP*, 2011.

[21] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, 1989.

[22] Y. Kaneta. Fast wavelet tree construction in practice. In *SPIRE*, 2018.

[23] F. Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, 2022.

[24] J. Labeit, J. Shun, and G. E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *J. Discrete Algorithms*, 43:2–17, 2017.

[25] C. Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012.

[26] J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. *Theor. Comput. Sci.*, 638:91–97, 2016.

[27] G. Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[28] A. Orlandi and R. Venturini. Space-efficient substring occurrence estimation. *Algorithmica*, 74(1):65–90, 2016.

[29] M. Patrascu. Succincter. In *FOCS*, 2008.

[30] A. Ordóñez Pereira, G. Navarro, and N. R. Brisaboa. Grammar compressed sequences with rank/select support. *J. Discrete Algorithms*, 43:54–71, 2017.

[31] N. Prezza. Optimal rank and select queries on dictionary-compressed text. In *CPM*, 2019.

[32] Nicola Prezza. On string attractors. In *ICTCS*, 2018.

[33] J. Shun. Improved parallel construction of wavelet trees and rank/select structures. *Inf. Comput.*, 273:104516, 2020.

[34] S. Vigna. Broadword implementation of rank/select queries. In *WEA*, 2008.

[35] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, 2013.