


Wavelet Tree, Part I: A Brief History

Paolo Ferragina 

Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
Department of Computer Science, University of Pisa, Italy

Raffaele Giancarlo 

Department of Mathematics and Computer Science, University of Palermo, Italy

Giovanni Manzini 

Department of Computer Science, University of Pisa, Italy

Giovanna Rosone 

Department of Computer Science, University of Pisa, Italy

Rossano Venturini 

Department of Computer Science, University of Pisa, Italy

Jeffrey Scott Vitter 

Department of Computer Science, Tulane University, New Orleans, LA, USA
Department of Computer and Information Science, The University of Mississippi, MS, USA

Abstract

The *Wavelet Tree* data structure introduced in Grossi, Gupta, and Vitter [16] is a space-efficient technique for rank and select queries that generalizes from binary symbols to an arbitrary multisymbol alphabet. Over the last two decades, it has become a pivotal tool in modern full-text indexing and data compression because of its properties and capabilities in compressing and indexing data, with many applications to information retrieval, genome analysis, data mining, and web search. In this paper, we survey the fascinating history and impact of Wavelet Trees; no doubt many more developments are yet to come. Our survey borrows some content from the authors' earlier works.

This paper is divided into two parts: one (this one) giving a brief history of Wavelet Trees, including its varieties and practical implementations, dedicated to this Festschrift's honoree Roberto Grossi; the second part deals with Wavelet Tree-based text indexing and is included in the Festschrift dedicated to Giovanni Manzini [10].

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Wavelet tree, data compression, text indexing, compressed suffix array, Burrows-Wheeler transform, rank and select

Digital Object Identifier 10.4230/OASICS.Grossi.2025.15

Category Research

Acknowledgements The authors would like to thank Hongwei Huo for helpful comments and figures.

1 Introduction

The field of compressed full-text indexing [29] involves the design of data structures (aka, indexes) that support fast substring matching using small amounts of space. For a text string $\mathcal{T}[1, n]$ over an arbitrary alphabet Σ of size σ and a pattern $\mathcal{P}[1, m]$, the goal of text indexing is to preprocess \mathcal{T} using succinct space so that queries like the following can be quickly answered: (1) count the number *occ* of occurrences of \mathcal{P} in \mathcal{T} ; (2) locate the *occ* positions in \mathcal{T} where \mathcal{P} occurs; and (3) starting at text position *start*, extract the length- ℓ substring $\mathcal{T}[start, start + \ell - 1]$.



© Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter;
licensed under Creative Commons License CC-BY 4.0

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.

Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter; Article No. 15; pp. 15:1–15:11

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A main goal is to create an index whose size is roughly equal to the size of the text in compressed format, with search performance comparable to the well-known indexes on uncompressed text, such as suffix trees and suffix arrays. Some compressed data structures are in addition *self-indexes* in that the data structure encapsulates the original text and, thus, can quickly recreate any portion of it. As a result, the original text is not needed, it can be discarded and replaced by only the (self-)index.

Most compressed indexing techniques developed in the last two decades make use of the powerful *Wavelet Tree* data structure, developed by Grossi, Gupta, and Vitter [16], with many applications to information retrieval, genome analysis, data mining, and web search. A Wavelet Tree supports fast rank and select queries on any string from a general multisymbol alphabet Σ . As such, it provides an elegant extension of rank-select data structures for binary sequences [31, 29] to the general case of alphabets of arbitrary size.

The Wavelet Tree represents a string of symbols from an arbitrary alphabet Σ as a hierarchical set (or tree) of binary strings. It has the following key compression property: If each such binary string is compressed according to its information-theoretic minimum size, then the original string is compressed to its information-theoretic minimum size.

The Wavelet Tree tree structure can be thought of as a generalization in the context of computational geometry of the two-dimensional range search data structure of Chazelle [4] in which the two-dimensional points stored in a binary tree node are initially sorted in one dimension and then recursively partitioned into children based upon the second dimension. Kärkkäinen [21] used a similar data structure in a very different way and purpose in the context of highly repetitive texts.

Why the name “Wavelet Tree”? One of the codevelopers suggested the name based upon his research interests in the fields of image compression and database query optimization, where he used wavelet transforms and their hierarchical decompositions to approximate multiresolution data [33].

In this paper (PART I), we give a brief history of Wavelet Trees, including its varieties and practical implementations. We refer the reader to PART II, included in the Festschrift dedicated to Giovanni Manzini, for a discussion about the important role of Wavelet Trees in text indexing, particularly for compressed suffix arrays and the FM-index, which can produce self-indexes with size related to the higher-order entropy of the text.

2 Preliminaries

Let $\Sigma = \{0, 1, \dots, \sigma - 1\}$ be a finite ordered alphabet of size σ . Let $\mathcal{T} = \mathcal{T}[0, n - 1]$ be a text string consisting of n symbols from alphabet Σ . For simplicity, we assume that $\mathcal{T}[n - 1]$ is a special end-of-text symbol $\#$ that is the lexicographically smallest symbol in Σ . We use $\mathcal{T}[i, j]$ to denote the substring of \mathcal{T} consisting of the symbols $\mathcal{T}[i]\mathcal{T}[i + 1] \dots \mathcal{T}[j]$. A prefix of \mathcal{T} is a substring of the form $\mathcal{T}[0, i]$, and a suffix is a substring of the form $\mathcal{T}[j, n - 1]$.

Two key operations we will need for accessing and traversing Wavelet Trees are *rank* and *select*:

► **Definition 1** (rank and select). *Let $\mathcal{B}[1, m]$ be a bit array of length m and let bit $b \in \{0, 1\}$. We define the rank and select operations as follows:*

- $\text{rank}_b(\mathcal{B}, i)$: *return the number of occurrences of bit b in $\mathcal{B}[1, i]$ for any $1 \leq i \leq m$;*
- $\text{select}_b(\mathcal{B}, j)$: *return the position in \mathcal{B} of the j th occurrence of bit b for any $1 \leq j \leq \text{rank}_b(\mathcal{B}, m)$.*

► **Definition 2** ((0th-order) empirical entropy). Let \mathcal{T} be a text string of length n over an alphabet $\Sigma = \{0, 1, \dots, \sigma - 1\}$ of size σ . The 0th-order empirical entropy of \mathcal{T} is defined as

$$\mathcal{H}_0 = \mathcal{H}_0(\mathcal{T}) = \frac{1}{n} \sum_{\omega \in \Sigma} n_{\omega} \log \frac{n}{n_{\omega}}$$

where n_{ω} is the number of occurrences in \mathcal{T} of symbol $\omega \in \Sigma$.

A more stringent notion of entropy is the *information-theoretic minimum* (or *0th-order finite-set entropy*), given by

$$\frac{1}{n} \log \binom{n}{n_1, n_2, \dots, n_{\sigma}}.$$

It is always less than or equal to the 0th-order empirical entropy.

Higher orders of entropy can be defined readily if we encode symbols based upon their frequencies of occurrence when grouped into common contexts.

► **Definition 3** (k th-order empirical entropy). Let \mathcal{T} be a text string of length n over an alphabet $\Sigma = \{0, 1, \dots, \sigma - 1\}$ of size σ . The k th-order empirical entropy of \mathcal{T} is defined as

$$\mathcal{H}_k = \mathcal{H}_k(\mathcal{T}) = \frac{1}{n} \sum_{x \in \Sigma^k} n_x \mathcal{H}_0(\mathcal{T}_x)$$

where $x \in \Sigma^k$ designates a string (context) of length k and \mathcal{T}_x denotes the string of length n_x formed by taking the symbol immediately preceding each occurrence of x in \mathcal{T} and concatenating all these symbols together.

In a like manner, we can define the k th-order finite-set entropy by

$$\frac{1}{n} \sum_{x \in \Sigma^k} \log \binom{n_x}{n_{1,x}, n_{2,x}, \dots, n_{\sigma,x}},$$

where $n_{y,x}$ for $y \in \Sigma$ is the number of occurrences of yx as a substring in \mathcal{T} . The k th-order finite-set entropy is always less than or equal to the k th-order empirical entropy.

In other words, the key point of the k th-order entropy definition is that we can achieve it by encoding the relevant parts of \mathcal{T} using 0th-order entropy. (This property was later appropriately dubbed “boosting.”) In particular, if we form the substrings \mathcal{T}_x for each relevant k -context x and compress each \mathcal{T}_x to achieve space of n_x times its 0th-order empirical entropy $\mathcal{H}_0(\mathcal{T}_x)$, then the resulting compression for the full text string \mathcal{T} will be n times the k th-order empirical entropy $\mathcal{H}_k(\mathcal{T})$. Wavelet Trees are a natural and elegant way to compress each substring \mathcal{T}_x to $n_x \mathcal{H}_0(\mathcal{T}_x)$ bits plus lower-order terms, and thus the resulting cumulative encoding of \mathcal{T} has size in bits bounded by $n \mathcal{H}_k(\mathcal{T})$ plus lower-order terms. This fact is exploited in PART II to show how Wavelet Trees can be used to create text indexes that use space related to the high-order entropy of \mathcal{T} .

For two strings S and T , we define the *lexicographic order* $S \prec T$ (and say that S is smaller than T) if S is a proper prefix of T , or if there exists an index $1 \leq i \leq \min\{|S|, |T|\}$ such that $S[i] < T[i]$ and for all $j < i$, $S[j] = T[j]$.

We use $SA[0, n - 1]$ to denote the suffix array (SA) of \mathcal{T} . It consists of the positions of all the suffixes of \mathcal{T} in lexicographical order. For example, in the text string $\mathcal{T} = \text{tcaaaatatatgcaacatatagattagattgtat\#}$ in Table 1, the smallest suffix begins at position $SA[0] = 35$, and the next smallest suffix begins at position $SA[1] = 2$. The ordered suffixes

15:4 Wavelet Tree, Part I: A Brief History

■ **Table 1** For text $\mathcal{T} = \text{tcaaaatatatgcaacatatagttattagattgtat\#}$ containing symbols from the alphabet $\Sigma = \{\#, \mathbf{a}, \mathbf{c}, \mathbf{g}, \mathbf{t}\}$ shown in the first column, the subsequent columns show the suffix array SA , its neighbor function Ψ , and the LF function of the BWT. The (wide) second-to-last column shows the suffixes in sorted order; each suffix starts at the symbol under F (first) and ends at the symbol under L (last). The BWT of \mathcal{T} is the string of symbols $L = \text{tcacaatttttcatttgtgaattaatagaag\#ataa}$. The last column \mathcal{B} is the bit array for the root node of the Wavelet Tree of L ; symbols $\#$, \mathbf{a} , and \mathbf{c} (for the left subtree) are designated by a 0, and symbols \mathbf{g} and \mathbf{t} (for the right subtree) are designated by a 1. The Wavelet Tree for L is pictured in Figure 1.

i	\mathcal{T}	SA	Ψ	LF	F	L	\mathcal{B}
0	t	35	31	23	# t c a ... a t		1
1	c	2	2	16	a a a a ... t c		0
2	a	3	4	1	a a a t ... c a		0
3	a	13	5	17	a a c a ... g c		0
4	a	4	11	2	a a t a ... a a		0
5	a	14	18	3	a c a t ... c a		0
6	t	26	19	24	a g a t ... t t		1
7	a	20	22	25	a g t a ... a t		1
8	t	33	23	26	a t # t ... g t		1
9	a	18	25	27	a t a g ... a t		1
10	t	16	27	18	a t a t ... a c		0
11	g	5	28	4	a t a t ... a a		0
12	c	7	29	28	a t a t ... a t		1
13	a	9	32	29	a t g c ... a t		1
14	a	23	34	30	a t t a ... g t		1
15	c	28	35	19	a t t g ... a g		1
16	a	1	1	31	c a a a ... # t		1
17	t	12	3	20	c a a c ... t g		1
18	a	15	10	5	c a t a ... a a		0
19	t	27	15	6	g a t t ... t a		0
20	a	11	17	32	g c a a ... a t		1
21	g	31	26	33	g t a t ... t t		1
22	t	21	30	7	g t a t ... t a		0
23	a	34	0	8	t # t c ... t a		0
24	t	25	6	34	t a g a ... a t		1
25	t	19	7	9	t a g t ... t a		0
26	a	32	8	21	t a t # ... t g		1
27	g	17	9	10	t a t a ... c a		0
28	a	6	12	11	t a t a ... a a		0
29	t	8	13	12	t a t g ... t a		0
30	t	22	14	22	t a t t ... a g		1
31	g	0	16	0	t c a a ... t #		0
32	t	10	20	13	t g c a ... t a		0
33	a	30	21	35	t g t a ... a t		1
34	t	24	24	14	t t a g ... t a		0
35	#	29	33	15	t t g t ... g a		0

are pictured in the (wide) second-to-last column of the table. For convenience, we regard text \mathcal{T} as a circular string and implicitly use modular arithmetic base n for the indices of \mathcal{T} . In other words, references to a symbol $\mathcal{T}[i]$ actually refer to $\mathcal{T}[i \bmod n]$.

Further development of suffix arrays that leads to the notion of compressed suffix arrays is discussed in PART II. Compressed suffix arrays are self-indexes that use space proportional to the high-order entropy of the text and provide fast pattern-matching queries. The key

notion that leads to the compression of suffix arrays is the important *neighbor function*, which for each $i \in [0, n - 1]$ maps i to the lexicographic order of the suffix formed by taking the i th smallest suffix of \mathcal{T} and rotating it to the left by one symbol.

The Burrows-Wheeler Transform (BWT) is another reversible transformation of a text string \mathcal{T} that rearranges the symbols of \mathcal{T} in a way that groups together the same symbols from similar contexts, thus resulting in high compressibility [2, 26]. In terms of Table 1, the BWT is represented by the last subcolumn (designated by L). PART II provides further development of the BWT that leads to the FM-index, which like the compressed suffix array, uses space proportional to the high-order entropy of the text and provides fast pattern-matching queries. The key notion behind the FM-index and its compression is the LF function, which is the inverse of the neighbor function for compressed suffix arrays mentioned above. That is, the LF function maps each i to the lexicographic order of the suffix formed by taking the i th smallest suffix of \mathcal{T} and rotating it to the right (rather than to the left) by one symbol. The symbol under L in row i is the same symbol under F in row $LF(i)$.

3 Wavelet trees

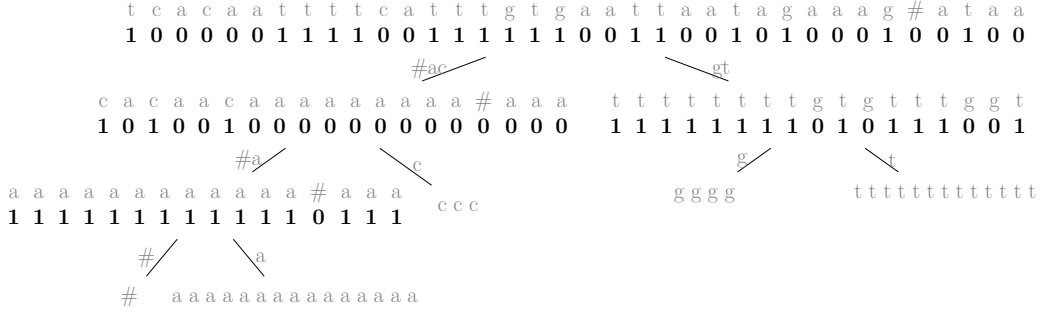
Wavelet Trees provide an elegant and efficient implementation of rank-select data structures for strings of symbols from an arbitrary multisymbol alphabet. They generalize the well-known rank-select data structures such as RRR [31] that handle binary strings.

Conceptually, the Wavelet Tree is most simply described as a full (and often balanced) binary tree (see the example in Figure 1). The root node is a bit array \mathcal{B} of the same length as the input text \mathcal{T} and partitions the text's alphabet into two sets, where a 0-bit indicates that the corresponding text symbol is in the left set and a 1-bit indicates that the text symbol is in the right set. (See Table 1 for the bit array \mathcal{B} applied to the string L .) Such a bit array representation happens recursively at each internal node, where the text at the internal node is of the symbols dispatched from the parent node with their order in the text preserved. Each leaf node represents a distinct symbol of the alphabet of the input text, with its multiplicity preserved.

Operations of *rank* and *select* on binary strings, as defined in Definition 1, are used to navigate up and down the Wavelet Tree. For example, in the root node of the Wavelet Tree of Figure 1, consider the symbol a in the sixth position of L , which is represented by a 0, which means that its corresponding entry on the next level is in the left subtree. To find that entry, we compute $rank_0(L, 6) = 5$ using a data structure such as RRR; we thus find that the corresponding entry is entry 5 in the left subtree. To go from that entry on level 2 back to the position of the corresponding entry in root node, we compute $select_0(L, 5) = 6$.

Of course, extra data structures are needed in order to support the needed query operations and functionality in an efficient way. Those details are not given here in this survey. Instead we focus on the main ideas that make wavelet trees such an elegant and powerful method.

The key aspect of Wavelet Trees is that they cleverly decompose the text into a hierarchy of bit arrays without introducing any redundancy, so that the total size of the raw bit arrays in the Wavelet Tree is exactly the same as the size (in bits) of the input text, regardless of the shape of the Wavelet Tree. If the bit array at each node is 0th-order entropy compressed, the resulting cumulative space usage of the Wavelet Tree is equal to n times the 0th-order entropy-compressed size of the input text, again regardless of the shape of the tree [16, 11].



■ **Figure 1** The Wavelet Tree for string $L = \text{tcacaattttcatttgtgaattaatagaaag\#ataa}$ of Table 1.

4 Properties of Wavelet Trees

The Wavelet Tree pictured in Figure 1 is a balanced binary tree, in which the number of symbols at each internal node are partitioned as evenly as possible for the level below. As such it has at most $\log \sigma$ levels, and each level (in raw uncompressed form) consists of a bit array of length n . Using RRR [31] to encode each internal node allows fast rank and select queries for general alphabets on the overall tree.

For an internal node of m bits with ℓ 1-bits and $m - \ell$ 0-bits, the RRR encoding [31] uses $\log \binom{m}{\ell} + o(\ell) + \mathcal{O}(\log \log m)$ bits, where $\log \binom{m}{\ell}$ is the information theoretic minimum. This form of entropy is called 0th-order finite-set entropy; it is always less than or equal to the empirical entropy of Definition 2, since it removes dependence from the statistical model. Summing the $\log \binom{m}{\ell}$ terms over all the internal nodes of the Wavelet Tree cascades into the multinomial coefficient $\binom{n}{n_1, n_2, \dots, n_\sigma}$ [17], which is less than or equal n times the 0th-order empirical entropy of the text \mathcal{T} .

There is much interesting work on efficient ways to construct Wavelet Trees. For example, Munro *et al.* [27] show that construction can be done in $\mathcal{O}(n(\log \sigma)/\sqrt{\log n})$ time.

5 Varieties of Wavelet Trees and Their Characteristics

There are several variants of Wavelet Trees. In this section, we focus on Huffman Wavelet Trees, Pruned Wavelet Trees, and Wavelet Trees with multiway branching.

In [24, 28] the authors propose giving the Wavelet Tree the Huffman shape of the frequencies with which the symbols appear in the text string \mathcal{T} . It has been shown that the total number of bits stored in the Huffman Wavelet Tree is exactly the number of bits output by a Huffman compressor that takes the symbol frequencies in \mathcal{T} , which is upper-bounded by $n(\mathcal{H}_0(\mathcal{T}) + 1)$. Moreover, the accesses to the leaves in the Huffman Wavelet Tree can be obtained with frequency proportional to their number of occurrences in \mathcal{T} . Huffman shapes can be combined with multiway Wavelet Trees and entropy compression of the bitmaps [1]. In this case, one can achieve space $n\mathcal{H}_0(\mathcal{T}) + o(n)$ bits, worst-case time $\mathcal{O}(1 + (\log \sigma)/\log \log n)$, and average case time $\mathcal{O}(1 + \mathcal{H}_0(\mathcal{T})/\log \log n)$ for both rank and select queries.

Ferragina, Giancarlo, and Manzini [11] addressed the impact of the Wavelet Tree data structure on data compression by providing a theoretical analysis of a wide class of compression algorithms based upon Wavelet Trees, and proposing a novel framework, called *pruned Wavelet Trees*, that aims for the best combination of Wavelet Trees of properly-designed shapes and compressors either binary (like, run-length encoders) or non-binary (like, Huffman and arithmetic encoders). There were three main contributions in that paper.

First, they proposed a thorough analysis of Wavelet Trees as *stand-alone* general-purpose compressor, by considering two specific cases (extending [13]) in which either the binary strings associated to their nodes are compressed via RLE (referred to as the Rle Wavelet Tree) or via Gap Encoding (referred to as the Ge Wavelet Tree). These analyses provided compression bounds that depended on the features of these prefix-free encoders and the 0th-order entropy of the input string. As a result, the authors were able to prove that Rle Wavelet Trees can achieve a compression bound in terms of the k th order empirical entropy, whereas Ge Wavelet Trees *cannot*. This result has been then strengthened in [25] where the authors showed that Rle can be replaced by the succinct dictionaries of [31].

Ferragina *et al.* [11] introduced the *pruned Wavelet Trees* that generalize and improve Wavelet Trees by working on (C.1) the shape (or topology) of the binary tree underlying their structure; and on (C.2) the assignment of alphabet symbols to the binary-tree leaves. The authors showed that it is possible to exhibit an infinite family of strings over an alphabet Σ for which changing the Wavelet Tree shape influences the coding cost by a factor $\Theta(\log \sigma)$, and changing the assignment of symbols to leaves influences the coding cost by a factor $\Theta(\sigma)$. Moreover, they showed that (C.3) Wavelet Trees commit to binary compressors, losing the potential advantage that might come from a mixed strategy in which only some strings are binary and the others are defined on an arbitrary alphabet (and can be compressed via general-purpose 0th-order compressors, such as arithmetic and Huffman coding). Again, that paper showed that it is possible to exhibit an infinite family of strings for which a mixed strategy yields a *constant* multiplicative factor improvement over standard Wavelet Trees.

It is exactly to address these questions that Ferragina *et al.* [11] introduced the pruned Wavelet Trees where only a subset of their nodes is binary, and then developed a combinatorial optimization strategy addressing (C1)–(C3) simultaneously. As a corollary, they specialized that strategy to design a polynomial-time algorithm for finding the optimal pruned Wavelet Tree of fixed shape and assignment of alphabet symbols to its leaves; or, for selecting the optimal tree shape when only the assignment of symbols to the leaves of the tree is fixed.

Ferragina *et al.* [12] showed that increasing the branching factor in the wavelet tree nodes can reduce the time for rank and select queries to constant time, assuming that the alphabet size satisfies $\sigma = \mathcal{O}(\text{polylog } n)$.

6 Practical Implementations of Wavelet Trees

In this section, we review some approaches for designing fast and compact implementations of Wavelet Trees. Grossi, Vitter, and Xu [18] provided a full generic package of Wavelet Trees for a wide range of options on the dimensions of coding schemes and tree shapes. Their experimental study reveals the practical performance of the various modifications. It provides potential users a rationale for how to implement a Wavelet Tree based upon the particular characteristics of the underlying datat.

The authors[18] considered three styles of Wavelet Trees:

- Normal: Each node is partitioned by alphabetically dividing the symbols represented in the node into two nearly *equal-sized* halves;
- Alphabetic weight-balanced: Each node is partitioned by alphabetically dividing the symbols represented in the node into two nearly *equal-weighted* halves, where the weights are designated by the frequencies of the symbols; and
- Huffman: Each node is partitioned as in Huffman compression.

They also considered several styles of encoding the conceptual bit array in each node, the main ones being

- None: The raw bit array is used at each node; support for member, rank, and select queries is provided by $o(n)$ -space auxiliary structures using superblocks and lookup-table based popcount operations [15];
- RRR: The practical implementation of the RRR method [31] given in [6];
- RLE+ γ : Run-length encoding with Elias γ encoding [8] for the runs; queries are supported using the $o(n)$ -space auxiliary structures [15];
- RLE+ δ : Same as RLE+ γ , but using Elias δ encoding [8] instead of γ ;
- SC: Small-integer t -subset encoding [22];
- AC: Arithmetic coding [19]; and
- LP/*: Bit arrays are compressed by a specified method iff there are at most three distinct symbols represented in the node, since nodes close to the leaf level are typically highly compressible.

The overall results of experiments on 33 Wavelet Tree options on texts from English, Protein, and Genome sources (several with very low entropy) suggest that RLE-encoded Wavelet Trees are the best in space efficiency when coding BWTs or sequences with very low 0th-order entropy, but are slower in query performance. RLE+ δ is especially space-efficient for low-entropy sequences. AAWT and Huffman-shaped Wavelet Trees have similar performance, with the latter more space efficient for low-entropy sequences. When used with RRR encoding, both offer good all-around space and query time performance. AAWT+None is generally the fastest in all the experiments for queries and construction time.

Many existing implementations construct the Wavelet Matrix [7], which is an alternative representation for the Wavelet Tree, dropping its structure and thus achieving faster performance in practice. It is especially effective for large alphabets.

At the first level of the Wavelet Matrix, the most significant bits (MSBs) of the symbols are stored, analogous to the first level of the Wavelet Tree. Then, to construct 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 symbols are represented using their ℓ th MSB on each level ℓ . However, the order of the symbols 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 order¹. The number of 0s in each level is stored in an array. This information 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, and refer the interested reader to [7] for more details.

Practical Binary Rank and Select Data Structures

As we mentioned before, rank and select data structures with constant query time can be constructed in linear time requiring $o(n)$ extra space for a binary vector of size n [5, 20]. The currently most space-efficient rank and select support for a size- n binary array that contains m ones requires only $\log \binom{n}{m} + n/\log n + \tilde{O}(n^{3/4})$ bits (including the bit vector) [30], which however is not of practical interest.

Almost all practical data structures to support rank queries follow the same layout. The main idea is to split the binary array into blocks of b bits each and superblocks of B blocks each. This way, every bit belongs to a block and every block belongs to a superblock. Then

¹ See <https://oeis.org/A030109>, last accessed 2025-02-24.

rank queries are supported by storing a counter for each superblock and block. For each superblock, we store the number of ones in the binary array up to the beginning of the superblock. Similarly, for each block, we store the number of ones from the beginning of its superblock to the beginning of the block. This hierarchy of counters is used to limit the number of bits to be used for each counter. The counter of a superblock is stored in $\Theta(\log n)$ bits, while the counter of a block is stored in $\Theta(\log b + \log B)$ bits.

A rank query at a certain position is computed by summing up the counter of the superblock and the counter of the block that contains the queried position, and the rank intra-block. The latter is computed with a popcount operation, which is supported by any modern CPU (see e.g. [9]).

Different superblock and block sizes are chosen by different implementations. For example, many implementations [32, 14] use superblocks of size $B = 8$ and blocks of size $b = 64$ bits. Wider block sizes $B = 4$ and $b = 512$ are chosen by the most recent implementations [34, 23] and computations intra-block with such a large block size are done with SIMD operations. These implementations have a smaller space overhead than the ones with smaller block sizes, but they require more time to answer rank queries. However, when the binary vector is large, the query time is dominated by the cost of accessing the block content from memory rather than the CPU time to perform the operations within the block. Thus, the disadvantage of having larger block sizes becomes almost negligible, and these solutions should be preferred because of their smaller space overhead.

The support for select queries is more involved. There exist two possible approaches: *rank-based* and *position-based*. Rank-based select structures rely on the rank data structure that is augmented by a lightweight index storing sampled positions for every k th occurrence of a 1 or 0. To answer a select query, the first step is to locate the nearest block using the sampled positions. Subsequently, consecutive blocks are examined until the basic block containing the target bit (with the specified rank) is found. The exact position within the basic block can then be determined directly. Although this method typically does not guarantee constant query time, it is efficient in practice. On the other hand, position-based select structures split the binary vector into blocks and sample positions of the block based on its density [32]. For example, if the block is sparse, we can store the answer of every select query directly. The main advantage of position-based select data structures is their constant worst-case query time. However, this comes at the cost of higher space usage.

Faster Wavelet Tree Implementations

When answering queries using a Wavelet Tree, the query is translated to $\Theta(\log \sigma)$ binary rank or select queries as described above. 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 the dominant cost of these binary queries. The currently fastest implementation [3] reduces the number of cache misses by reducing the number of levels. This is done by making use of a 4-ary Wavelet Tree. By doubling the number of children, it (roughly) halves the number of levels. A 4-ary Wavelet Tree represents the symbols on each level using two bits stored in a quad vector, i.e., a vector over the alphabet $\{0, 1, 2, 3\}$ with access, rank, and select support. The rank and select support for the quad vector is implemented using strategies similar to the ones of rank and select data structures for binary vectors.

The speed-up of this implementation over the other existing ones is approximately a factor 2 for all the queries. The rank queries can be further improved using a small prediction model designed to anticipate and pre-fetch the cache lines required for rank queries. This could give a further improvement up to a factor of 1.6 for rank query [3].

References

- 1 Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. doi:10.1016/j.tcs.2013.10.019.
- 2 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994.
- 3 Matteo Ceregin, Florian Kurpicz, and Rossano Venturini. Faster wavelet tree queries. In *Data Compression Conference*, pages 223–232. IEEE, 2024. doi:10.1109/DCC58796.2024.00030.
- 4 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. doi:10.1137/0217026.
- 5 David Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA’96)*, pages 383–391, New York, NY, 1996. Association for Computing Machinery.
- 6 Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE ’08)*, pages 176–187, November 2008. doi:10.1007/978-3-540-89097-3_18.
- 7 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.
- 8 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.
- 9 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
- 10 Paolo Ferragina, Raffaele Giancarlo, Roberto Grossi, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter. Wavelet Tree, Part II: Text indexing. submitted to Festschrift’s honoree Giovanni Manzini.
- 11 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.
- 12 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article 20, 2007.
- 13 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. This work combined earlier versions from *SIAM Symposium on Discrete Algorithms (SODA)*, January 2004, and “Fast compression with a static model in high-order entropy,” *Data Compression Conference (DCC)*, March 2004. doi:10.1145/1198513.1198521.
- 14 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.
- 15 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, 2005.
- 16 Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA’03)*, pages 841–850, New York, NY, 2003. Association for Computing Machinery.
- 17 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC’00)*, May 2000. doi:10.1137/S0097539702402354.
- 18 Roberto Grossi, Jeffrey S. Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 210–221, 2011. doi:10.1109/CCP.2011.16.
- 19 P. G. Howard and Jeffrey S. Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6):857–865, June 1994. doi:10.1109/5.286189.

- 20 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS'89)*, pages 549–554, Washington, DC, 1989. IEEE Computer Society. doi:10.1109/SFCS.1989.63533.
- 21 Juha Kärkkäinen. *Repetition-based Text Indexing*. Ph.d., University of Helsinki, Finland, 1999.
- 22 Donald E. Knuth. *The Art of Computer Programming*, volume Volumg 3: Sorting and Searching. Addison-Welsey, 2nd edition edition, 1998.
- 23 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.
- 24 Veli Mäkinen and Gonzalo Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report Tech. Rep. C-2004-20, University of Helsinki, April 2004.
- 25 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, pages 229–241, Heidelberg, Germany, 2007. Springer-Verlag Berlin. doi:10.1007/978-3-540-75530-2_21.
- 26 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. An earlier version appeared in *Proceedings of the 10th Symposium on Discrete Algorithms (SODA '99)*, January 1999, 669–677. doi:10.1145/382780.382782.
- 27 J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.
- 28 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.
- 29 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
- 30 Mihai Patrascu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.
- 31 Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007. doi:10.1145/1290672.1290680.
- 32 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.
- 33 J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204, Philadelphia, PA, June 1999. Awarded the 2009 SIGMOD Test of Time Award for the most impactful paper from the SIGMOD conference 10 years earlier.
- 34 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.