Wavelet Forests Revisited

Eric Chiu
Department of Computer Science, Stony Brook University, NY, USA

Dominik Kempa
Department of Computer Science, Stony Brook University, NY, USA

Abstract

Rank and select queries are basic operations on sequences, with applications in compressed text indexes and other space-efficient data structures. One of the standard data structures supporting these queries is the *wavelet tree*. In this paper, we study *wavelet forests*, that is, wavelet-tree structures based on the *fixed-block compression boosting* technique. Such structures partition the input sequence into fixed-size blocks and build a separate wavelet tree for each block. Previous work showed that this approach yields strong practical performance for rank queries.

We extend wavelet forests to support select queries. We show that select support can be added with little additional space overhead and that the resulting structures remain practically efficient. In experiments on a range of non-repetitive and repetitive inputs, wavelet forests are competitive with, and in most cases outperform, standalone wavelet-tree implementations. We also study the effect of internal parameters, including superblock size and navigational data, on select-query performance.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Data compression; Information systems → Information retrieval

Keywords and phrases wavelet tree, wavelet forest, select queries

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.11

Related Version *Full Version*: `https://arxiv.org/abs/2604.11338`

Supplementary Material *Software*: `https://github.com/echiu12/wavelet-forest`

Funding Partially funded by the NSF CAREER Award 2337891.

1 Introduction

Consider a string $S \in [0\,..\,\sigma)^n$ of length n over the integer alphabet $[0\,..\,\sigma)$.

- Given any $i \in [0\,..\,n]$ and any symbol $c \in [0\,..\,\sigma)$, the *rank query* returns the value $\mathrm{rank}_S(i, c) := |\{j \in [1\,..\,i] : S[j] = c\}|$, i.e., the number of occurrences of c in $S[1\,..\,i]$;
- Given any $r \in \mathbb{Z}_{\geq 1}$ and any symbol $c \in [0\,..\,\sigma)$, the *select query* returns the value $\mathrm{select}_S(r, c)$, defined as the rth smallest element of the set $\{i \in [1\,..\,n] : S[i] = c\}$. If the size of this set is less than r, we set $\mathrm{select}_S(r, c) = \infty$.

Rank and select queries are among the most basic queries on sequences. They form the backbone of many compressed data structures for strings [12, 21, 35, 14, 24, 26, 25]. They are also core components of space-efficient representations of more complex data types, including permutations, parentheses, document collections, trees, graphs, and grids [32, 31, 11, 19, 15].

One of the most efficient data structures supporting rank and select queries is the *wavelet tree*, invented by Grossi, Gupta, and Vitter [20]. For a string $S \in [0\,..\,\sigma)^n$, wavelet trees use $\mathcal{O}(n \log \sigma)$ bits and support rank and select queries in $\mathcal{O}(\log \sigma)$ time.[1] Efficient implementations of wavelet trees have been the subject of intense study, and multiple highly

[1] Space of wavelet trees can also be bounded in terms of the kth-order empirical entropy $H_k(S)$ of S [20].

© Eric Chiu and Dominik Kempa;
licensed under Creative Commons License CC-BY 4.0

efficient solutions have been designed, optimizing many aspects of wavelet trees, including space usage, query time, construction time, and working space, both in theory and in practice [5, 16, 22, 38, 1, 30, 17, 28, 36, 7, 8, 9, 3, 10, 27].

In this paper, we focus on a particular wavelet-tree implementation called a *wavelet forest* [18].[2] The basic idea behind wavelet forests is to partition the input sequence into blocks and construct a wavelet tree for each block, while also maintaining global rank values at block boundaries; these boundaries are then grouped into larger hierarchies such as superblocks and hyperblocks. Simply splitting the sequence and constructing a *standalone* wavelet tree for every block – even using state-of-the-art wavelet-tree implementations – does not yield a significant improvement. However, it is shown in [18] that, when this idea is combined with several algorithmic improvements (including alphabet mapping at the block and superblock levels, merging the bitvectors of all wavelet trees into one, sharing lookup tables, dynamic selection of blocks in each superblock, pointerless tree navigation, etc.), wavelet forests achieve excellent performance, leading to new state-of-the-art results for rank queries. A unique advantage of wavelet forests is their *adaptability*: they achieve strong practical performance regardless of the underlying sequence type. In particular, they work well on both *non-repetitive* and *highly repetitive* input texts, making them an excellent off-the-shelf choice in many applications. The implementation of the wavelet forest from [18] is part of the `SDSL` library – a powerful and flexible library of succinct data structures [17].[3] However, the implementation of the wavelet forest in [18] supports only rank queries and lacks support for select queries. Until now, it has not been known whether the strong practical performance of wavelet forests also extends to select queries and, if so, at what cost.

Our Results. The contribution of this paper is twofold:

- We propose the first efficient implementation of select queries on wavelet forests. With only minor additions, wavelet forests support select queries with little extra cost in practice and, in most cases, outperform standalone optimized wavelet trees. For example, with the RRR bitvector implementation, they use the same or less space while improving query time by up to a factor of two.
- In addition to demonstrating the performance of our structure on a range of inputs (Section 5.1), we also explore the effect of internal parameters, including the superblock size and the impact of navigational headers (Sections 5.2 and 5.3).

Organization of the Paper. In Section 2, we introduce the notation and definitions used throughout the paper. In Section 3, we present a basic overview of the components of wavelet forests. In Section 4, we describe the details of our implementation. Finally, in Section 5, we present experiments demonstrating the performance of wavelet forests for select queries.

2 Preliminaries

Strings. A *string* is a finite sequence of symbols from a given set Σ, called the *alphabet*. For any $i \in [1\,..\,|S|]$, we denote the ith leftmost symbol of S by $S[i]$. Strings of the form $S[i\,..\,j)$, where $1 \leq i \leq j \leq |S|+1$, are called *substrings* or *factors* of S.[4] When $i = 1$ (resp. $j = |S|+1$), the substring $S[i\,..\,j)$ is a *prefix* (resp. *suffix*) of S. The concatenation of strings

[2] The technique introduced in [18] is also called *fixed-block compression boosting*.
[3] Available at `https://github.com/simongog/sdsl-lite`.
[4] We use $[i\,..\,j)$, $(i\,..\,j)$, and $(i\,..\,j]$ as shorthand for $[i\,..\,j-1]$, $[i+1\,..\,j-1]$, and $[i+1\,..\,j]$, respectively.

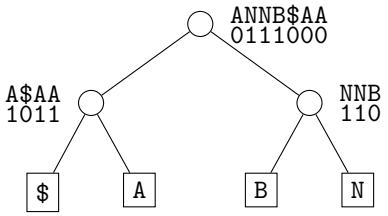

Figure 1 An illustration of a wavelet tree for the string `ANNB$AA`.

S_1 and S_2 is denoted by S_1S_2 or $S_1 \cdot S_2$. We assume that the set Σ is equipped with an order denoted by $\preceq$. The *lexicographic order* on strings over Σ is the extension of the order $\preceq$ on Σ defined as follows: for strings S_1 and S_2, it holds that $S_1 \preceq S_2$ if either S_1 is a prefix of S_2, or there exists $\ell \in [1\,..\,\min(|S_1|, |S_2|)]$ such that $S_1[1\,..\,\ell) = S_2[1\,..\,\ell)$ and $S_1[\ell] \prec S_2[\ell]$.

Burrows–Wheeler Transform. Consider a string $S \in \Sigma^n$ such that $S[n]$ is a unique symbol in S, denoted by $S[n] = \$$, and $\$$ is the smallest symbol in Σ. Let $\mathcal{M}$ denote the $n \times n$ matrix obtained by lexicographically sorting all rotations of S, that is, all strings in the set $\{S[i\,..\,n] \cdot S[1\,..\,i{-}1] : i \in [1\,..\,n]\}$. The *Burrows–Wheeler Transform (BWT)* of S is the string formed by taking the last column of $\mathcal{M}$ [2].

▶ **Example 1.** The BWT of the string $S =$ `BANANA$` is `ANNB$AA`.

Wavelet Trees. The wavelet tree $\mathcal{T}$ of $S \in \Sigma^n$ is a binary tree with $\sigma = |\Sigma|$ leaves, each representing a symbol in the alphabet [20]. Every internal node v of $\mathcal{T}$ represents a subset $\Sigma_v \subseteq \Sigma$ of symbols corresponding to the leaves in the subtree rooted at v. Each node v has an associated string S_v, which is a subsequence of S containing only symbols in Σ_v. Lastly, each internal node has an associated bitvector B_v containing, for every position $j \in [1\,..\,|S_v|]$, information indicating whether $S_v[j]$ is represented by a leaf in the subtree rooted at the left or right child of v. Each bitvector B_v is augmented with a data structure supporting rank and select queries over the binary alphabet. Assuming these queries take $\mathcal{O}(1)$ time, wavelet trees support rank and select queries over S in $\mathcal{O}(\log \sigma)$ time [20].

One of the central applications of wavelet trees is supporting rank queries over the BWT of a given text. Such functionality is at the core of the FM-index [12] and enables efficient pattern matching over the underlying text. In Figure 1, we show the wavelet tree for the BWT of the text from Example 1.

3 Wavelet Forest

In this section, we describe the basic idea behind the wavelet forest [18] and introduce the notation used in the subsequent sections. At a high level, a wavelet forest represents the input text by partitioning it into blocks and building a separate wavelet tree for each block. This approach is justified theoretically by the *fixed-block boosting* theorem proved in [18].

The input text is partitioned hierarchically into *blocks*, *superblocks*, and *hyperblocks*. Blocks have size b, superblocks have size b_s, and hyperblocks have size b_h, where b divides b_s and b_s divides b_h. Thus, each superblock contains b_s/b blocks, and each hyperblock contains b_h/b_s superblocks. In the default wavelet-forest implementation, the superblock size is set to 2^{20}. Technically, wavelet forests also allow variable-sized blocks, which can improve space efficiency. For clarity, however, we restrict the discussion here to fixed-size blocks.

For each block, superblock, and hyperblock, we store rank information at its left boundary, that is, at the first position of the corresponding range. We refer to this value as the *rank at the block*, *rank at the superblock*, and *rank at the hyperblock*, respectively. These values are represented by arrays A_b, A_s, and A_h. More specifically, let c be a symbol. Then the rank of c at hyperblock i_h is given by $A_h[c, i_h]$. If superblock i_s is contained in hyperblock i_h, then the rank of c at superblock i_s is $A_h[c, i_h] + A_s[c, i_s]$. Similarly, if block i_b is contained in superblock i_s and hyperblock i_h, then the rank of c at block i_b is $A_h[c, i_h] + A_s[c, i_s] + A_b[c, i_b]$. In other words, these arrays store rank values in a hierarchical, relative form.

Each block is represented by its own wavelet tree, which can be located via superblock and block headers. These wavelet trees are pointerless and Huffman-shaped, and encode their blocks independently of the rest of the text. Moreover, because each wavelet tree is built only for the local alphabet of its block, its height can be smaller than $\log \sigma$.

To further improve query time, the wavelet forest may also use *navigational block headers*. When enabled, each block stores prefix-rank information for the nodes at each level of its wavelet tree. This reduces the number of bitvector rank queries required during traversal and can speed up traversals used by both rank and select queries.

4 Answering Select Queries

In this section, we describe how we implemented select queries on wavelet forests. We first provide an outline of the procedure and then describe each step in more detail. To compute $\text{select}_S(j, c)$, i.e., the position of the jth occurrence of c in $S \in \Sigma^n$, we proceed as follows:

1. Identify i_h, the index of the hyperblock containing the jth occurrence of c in S.
2. Identify i_s, the index of the superblock containing the jth occurrence of c in S.
3. Identify i_b, the index of the block containing the jth occurrence of c in S. Let j' denote the localized select query argument relative to the block. In other words, let $j' = j - \text{rank}_S((i_b - 1)b, c)$.
4. Navigate the wavelet tree of that block to reach the leaf that corresponds to c while maintaining a stack of nodes on the path to that leaf.
5. Navigate back up the same wavelet tree using the standard wavelet-tree select algorithm with j' as the query argument. Let k denote the result of this localized select query.
6. Return $(i_b - 1)b + k$.

In the first step, we determine the hyperblock index i_h by binary searching the array of hyperblock ranks. More precisely, i_h is the largest hyperblock index such that $A_h[c, i_h] < j$, where $A_h[c, i]$ stores the rank of c at the beginning of the ith hyperblock.

In the second step, we determine the superblock index i_s by binary searching only among the superblocks contained in hyperblock i_h. Thus, i_s is the largest index in the range $((i_h - 1)b_h/b_s \,..\, i_h b_h/b_s]$ such that $A_h[c, i_h] + A_s[c, i_s] < j$.

In the third step, we determine the block index i_b by scanning the block headers of superblock i_s from right to left. Here, a linear scan is necessary because not every block stores the rank of every symbol. A right-to-left scan is preferable to a left-to-right scan because it stops as soon as it reaches the first block whose rank of c is smaller than j. Accordingly, i_b is the largest block index in the range $((i_s - 1)b_s/b \,..\, i_s b_s/b]$ such that $A_h[c, i_h] + A_s[c, i_s] + A_b[c, i_b] < j$. Equivalently, i_b is the first block encountered by the right-to-left scan whose rank of c is smaller than j. Once i_b is known, we compute the localized query value $j' = j - \text{rank}_S((i_b - 1)b, c)$.

In the fourth step, we access the wavelet tree of block i_b and descend to the leaf corresponding to c. This step is needed because the wavelet forest is pointerless, so before starting the localized select query, we must explicitly walk from the root to the correct leaf.

During this traversal, we compute the relevant node information on the fly. The traversal is similar to the one used for rank queries, except that it does not perform a bitvector-rank operation at each node. Because the next step backtracks from the leaf to the root, we store the necessary node information in a stack as we descend. The stack can be stored in a small array. Recall that the individual wavelet trees are Huffman-shaped. The minimum total weight of a Huffman tree of height h is attained by leaf weights $1, F_1, F_2, \ldots, F_h$, where F_i is the ith Fibonacci number and $F_1 = F_2 = 1$. Hence, the minimum block length that can induce height h is $1 + \sum_{i=1}^{h} F_i = F_{h+2}$. Therefore, for blocks of size at most 2^{16}, the height is at most 22, because $F_{24} = 46368 \leq 2^{16} < 75025 = F_{25}$.

In the fifth step, we perform a localized select query with argument j' on the wavelet tree of block i_b. This is exactly the standard wavelet-tree select algorithm [20], applied to the canonical Huffman-shaped wavelet tree of a single block. The algorithm uses the stack built in the previous step to recover the required node information while backtracking from the leaf to the root. We denote the resulting local position by k.

Finally, in the sixth step, we return $(i_b - 1)b + k$. Here, $(i_b - 1)b$ is the number of positions preceding block i_b, and k is the position of the desired occurrence within that block.

5 Experimental Results

Setup. Our experiments were conducted using an Intel Core i7-1355U CPU with 12 MiB L3 cache, 32 GiB of RAM, and a 64-bit Linux Ubuntu 22.04.5 (kernel 6.8.0-60). The frequency scaling for the CPU was set to `performance`. Our code was compiled with `g++` 11.4.0 using the flags `-DNDEBUG -msse4.2 -funroll-loops -O3`. We measured time using `std::chrono::high_resolution_clock` and the size of data structures via serialization.

Implementations. For a fair comparison between wavelet forests and other wavelet-tree variants, we used the implementations provided by the `SDSL` library [17].[5] To clearly distinguish the different implementations in our experiments, we use the following naming convention:

- *WF:* the wavelet forest [18], extended with our implementation of select queries.[6]
- *WT-HF:* the Huffman-shaped wavelet tree [20] provided by SDSL [17].
- *WT-RL:* the run-length-compressed wavelet tree available in SDSL [17, 29].

Both WF and WT-HF store their internal data in bitvectors, and each of them can be instantiated with any of the following bitvector representations:

- *BV:* the uncompressed bitvector implementation in SDSL.
- *HYB:* the hybrid bitvector [23], using superblock rates 8, 16, 32, and 64 (default: 16).[7]
- *RRR:* the RRR bitvector [34], with block sizes 15, 31, 63, 127 (default: 63).

We refer to a particular combination of wavelet-tree variant and bitvector type by concatenating their aliases with a dash. For example, WF-HYB denotes the wavelet forest instantiated with hybrid bitvectors.

[5] In preliminary experiments, we also examined wavelet matrices [5]. However, wavelet matrices are designed to perform well on large alphabets, and we observed that, for our datasets, which have relatively small alphabets, wavelet trees and wavelet forests consistently provided better time–space trade-offs.
[6] The default superblock size is 2^{20}.
[7] We use the hybrid bitvector augmented with support for select queries [4].

Table 1 Statistics of the datasets used in our experiments, including alphabet size σ, size in MiB ($n/2^{20}$), and average run length in the BWT (n/r).

Name	σ	$n/2^{20}$	n/r
dna	16	200	1.63
english	225	200	2.91
sources	230	200	4.40
dblp.xml	96	200	7.09
para	5	410	27
world_leaders	89	44	82
kernel	160	246	93
einstein.en.txt	139	446	1611

Name	σ	$n/2^{20}$	n/r
1000genomes	51	8000	2.29
commoncrawl	234	8000	5.73
enwiki	212	8000	3.82
gutenberg	216	8000	2.69

Datasets. Our dataset consists of two parts. The first part contains four non-repetitive and four repetitive texts from the Pizza&Chili corpus [13]. The second part consists of four 8 GiB texts from various sources: the 1000 Genomes Project [37], Common Crawl [6], Wikipedia, and Project Gutenberg [33]. Statistics for all texts are given in Table 1.

5.1 Experiment 1: Performance of Select Queries

In the first experiment, we compare the performance of wavelet forests for select queries against other wavelet-tree variants. We also vary the bitvector representation and its parameters. For each text in our dataset, we compute its BWT [2] and generate 10^5 random select queries. For each variant, we measure average query time and total index size.

The results for the Pizza&Chili texts are shown in Figure 2. In the majority of cases, wavelet forests are both faster and more space-efficient than standalone wavelet trees. When combined with RRR bitvectors, wavelet forests improve query time by up to a factor of two while using the same or less space. Notable exceptions occur for the `para` and `world_leaders` texts, whose small alphabets limit the ability of wavelet forests to exploit the difference between local and global alphabet sizes. A particularly effective combination is obtained by pairing wavelet forests with hybrid bitvectors. This variant achieves strong overall performance on both non-repetitive and highly repetitive texts, making it a good off-the-shelf choice in practice. The results for the larger texts, shown in Figure 3, exhibit similar trends.

5.2 Experiment 2: Effect of Superblock Size

In the second experiment, we study how the superblock size affects the space usage and select-query time of the wavelet forest. We used two representative texts from the Pizza&Chili corpus: `english`, representing non-repetitive data, and `kernel`, representing repetitive data. As in the previous experiment, we first compute the BWT of each text and then run random select queries. For HYB and RRR, we used the default parameter settings.

The results are shown in Figure 4. They indicate that the superblock size can have a substantial effect on performance. If the superblock size is too large, query time can deteriorate. On the other hand, if the superblock size is too small, space usage increases significantly. Overall, our experiments confirm that choosing a superblock size around 2^{20} is a robust practical default. The same value was previously observed to provide a good trade-off for rank queries in wavelet forests [18].

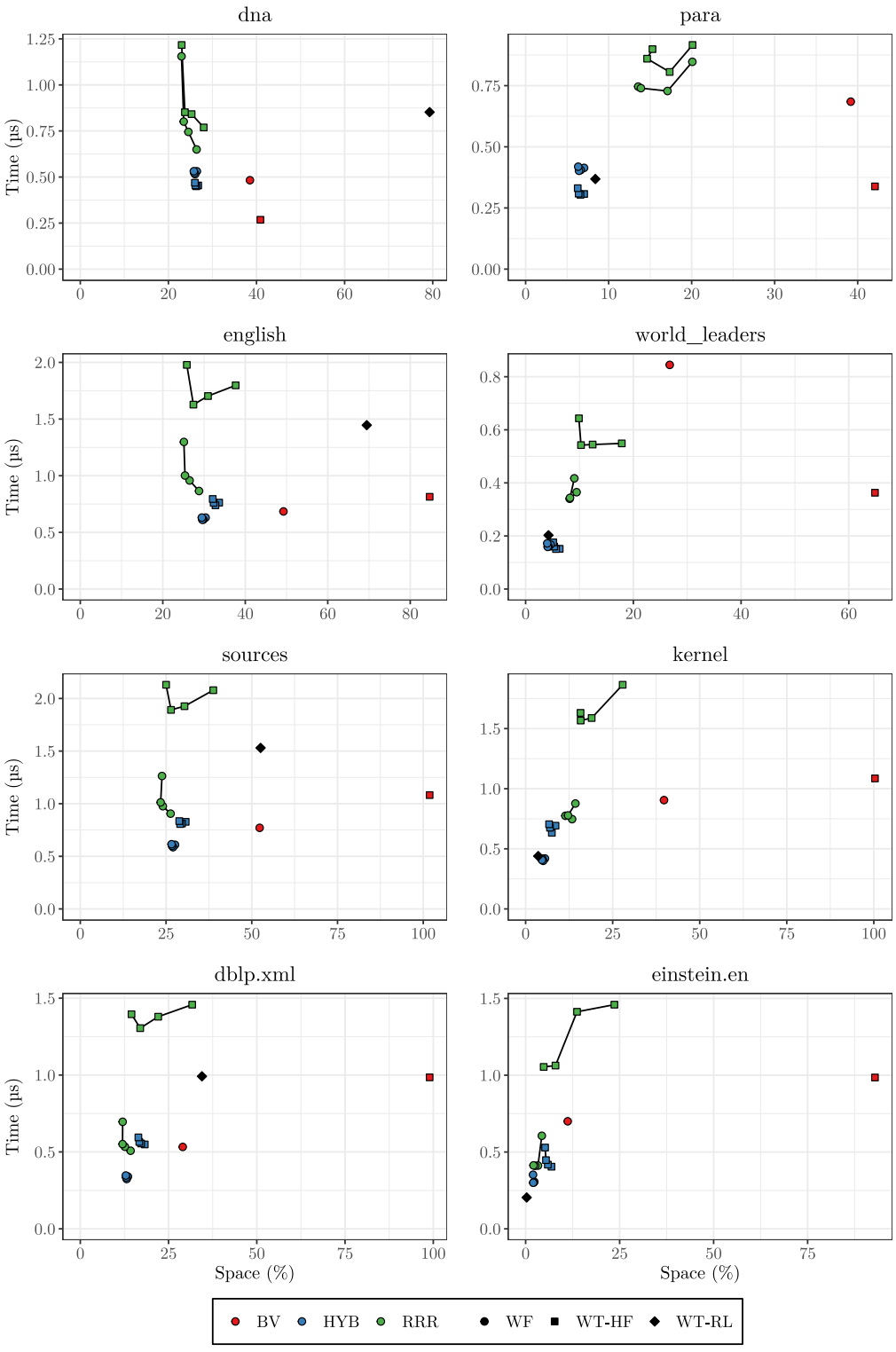

Figure 2 Time-space trade-offs for select queries on texts from the Pizza&Chili corpus. Time is measured in microseconds, and space is expressed as a percentage of the original text size. Non-repetitive and repetitive texts are shown in the left and right columns, respectively.

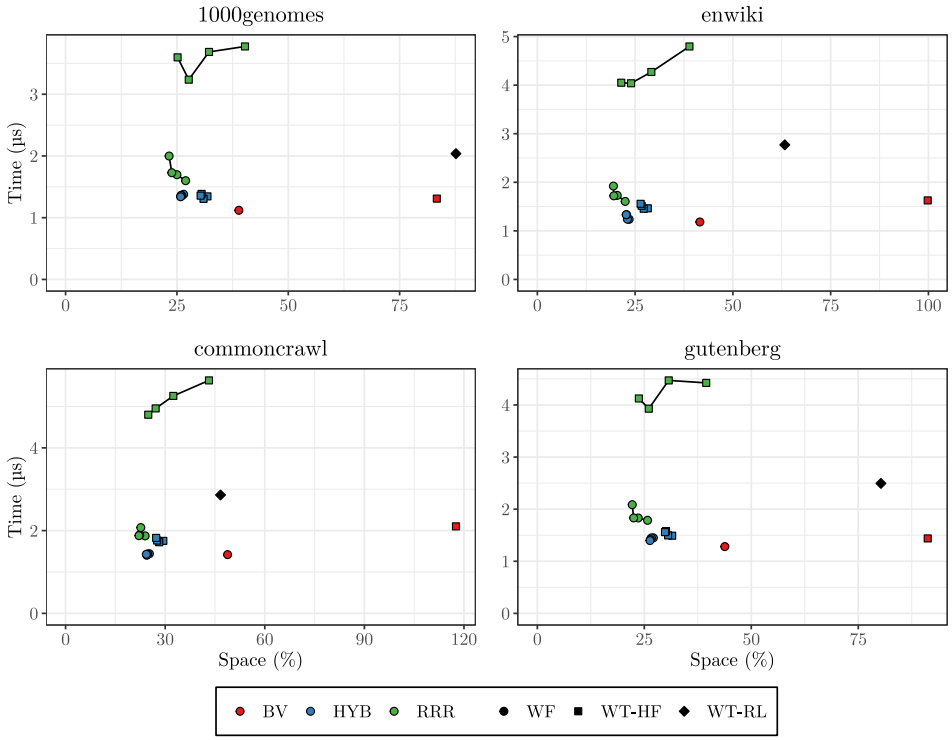

Figure 3 Time-space trade-offs for select queries on 8 GiB texts. Time is measured in microseconds, and space is expressed as a percentage of the original text size.

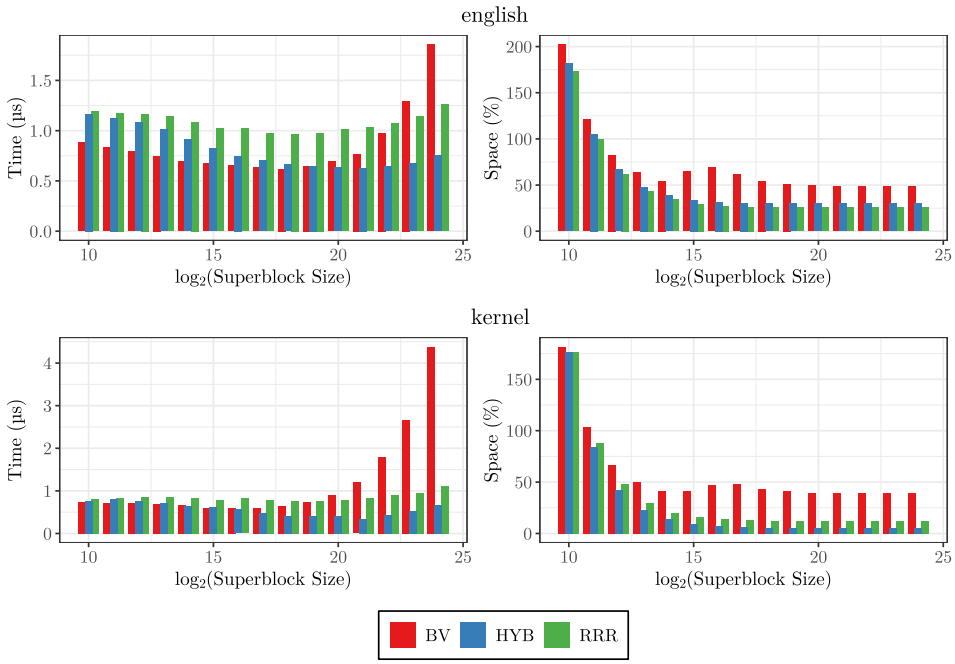

Figure 4 Effect of superblock size on select-query performance in wavelet forests. Time is measured in microseconds, and space is expressed as a percentage of the original text size.

Table 2 Effect of navigational block headers on wavelet-forest select-query performance. Time is measured in microseconds, and space is expressed as a percentage of the original text size.

Text	Bitvector	Space (%)		Time (μs)	
		With	Without	With	Without
english	BV	49.2547	48.7405	0.6913	0.7658
	HYB	29.8543	29.7226	0.6215	0.7518
	RRR	25.3844	25.2495	0.9965	1.7224
kernel	BV	39.7453	38.8623	0.9047	0.9223
	HYB	5.0468	4.9298	0.4018	0.5021
	RRR	12.1714	12.0439	0.7804	1.1842

5.3 Experiment 3: Effect of Navigational Block Headers

In the third experiment, we evaluate how navigational block headers affect the space usage and select-query performance of wavelet forests. Navigational headers are optional components of the block headers in the wavelet-forest implementation that speed up traversal within the wavelet tree of the current block; see [18]. They are enabled by default, and the goal of this experiment is to determine how much they help for select queries. As in the previous experiment, we used the BWTs of two representative texts from the Pizza&Chili corpus, namely `english` and `kernel`.

The results are shown in Table 2. For `english`, across all bitvector types, navigational block headers improve query time by about 1.1× to 1.72×, while increasing space by at most 1.1%. For `kernel`, the corresponding speedups range from about 1.02× to 1.52×, with space overhead up to 2.4%. These results show that navigational block headers substantially improve query time at the cost of only a very small increase in space.

References

1 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In Piotr Indyk, editor, *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

2 Michael Burrows. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

3 Matteo Ceregini, Florian Kurpicz, and Rossano Venturini. Faster wavelet tree queries. In Ali Bilgin, James E. Fowler, Joan Serra-Sagristà, Yan Ye, and James A. Storer, editors, *Proceedings of the 2024 Data Compression Conference (DCC 2024)*, pages 223–232. IEEE, 2024. doi:10.1109/DCC58796.2024.00030.

4 Eric Chiu and Dominik Kempa. Fast select queries using hybrid bitvectors. To appear in Proceedings of the 24th Symposium on Experimental Algorithms (SEA 2026). arXiv:2509.06900.

5 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015. doi:10.1016/J.IS.2014.06.002.

6 Common Crawl Foundation. Common crawl. URL: https://commoncrawl.org.

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

8 Patrick Dinklage, Johannes Fischer, and Florian Kurpicz. Constructing the wavelet tree and wavelet matrix in distributed memory. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the 2020 Symposium on Algorithm Engineering and Experiments (ALENEX 2020)*, pages 214–228. SIAM, 2020. doi:10.1137/1.9781611976007.17.

9 Patrick Dinklage, Johannes Fischer, Florian Kurpicz, and Jan-Philipp Tarnowski. Bit-parallel (compressed) wavelet tree construction. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2023 Data Compression Conference (DCC 2023)*, pages 81–90. IEEE, 2023. doi:10.1109/DCC55655.2023.00016.

10 Paolo Ferragina, Raffaele Giancarlo, Roberto Grossi, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter. Wavelet tree, part II: text indexing. In Paolo Ferragina, Travis Gagie, and Gonzalo Navarro, editors, *The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday*, pages 4:1–4:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/OASIcs.MANZINI.4.

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 and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

13 Paolo Ferragina and Gonzalo Navarro. Pizza&Chili Corpus: Compressed Indexes and their Testbeds. Accessed: 2025-11-03. URL: https://pizzachili.dcc.uchile.cl/.

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

15 Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012. doi:10.1016/J.TCS.2011.12.002.

16 Simon Gog. *Compressed suffix trees: design, construction, and applications*. PhD thesis, University of Ulm, 2011. URL: http://vts.uni-ulm.de/docs/2011/7786/vts_7786_11228.pdf.

17 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

18 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019. doi:10.1007/S00453-018-0475-9.

19 Roberto Grossi. Wavelet trees. In *Encyclopedia of Algorithms*, pages 2355–2359. Springer, 2016. doi:10.1007/978-1-4939-2864-4_642.

20 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

21 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

22 Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *Proceedings of the 1st International Conference on Data Compression, Communications and Processing (CCP 2011)*, pages 210–221. IEEE Computer Society, 2011. doi:10.1109/CCP.2011.16.

23 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Proceedings of the 2014 Data Compression Conference (DCC 2014)*, pages 302–311. IEEE, 2014. doi:10.1109/DCC.2014.87.

24 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

25 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In Sandy Irani, editor, *Proceedings of the 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1002–1013. IEEE, 2020. doi:10.1109/FOCS46700.2020.00097.

26 Dominik Kempa and Tomasz Kociumaka. Breaking the $O(n)$-barrier in the construction of compressed suffix arrays and suffix trees. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, pages 5122–5202. SIAM, 2023. doi:10.1137/1.9781611977554.CH187.

27 Florian Kurpicz, Angelo Savino, and Rossano Venturini. Faster wavelet tree queries. *Software: Practice and Experience*, 55(12):1931–1946, 2025. doi:10.1002/spe.70013.

28 Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017. doi:10.1016/J.JDA.2017.04.001.

29 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

30 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. doi:10.1016/J.TCS.2015.11.011.

31 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/J.JDA.2013.07.004.

32 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, Cambridge, UK, 2016. doi:10.1017/cbo9781316588284.

33 Project Gutenberg. Project Gutenberg. Accessed: 2025-11-03. URL: https://www.gutenberg.org/.

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

35 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

36 Julian Shun. Improved parallel construction of wavelet trees and rank/select structures. *Information and Computation*, 273:104516, 2020. doi:10.1016/J.IC.2020.104516.

37 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015. doi:10.1038/nature15393.

38 German Tischler. On wavelet tree construction. In Raffaele Giancarlo and Giovanni Manzini, editors, *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, pages 208–218. Springer, 2011. doi:10.1007/978-3-642-21458-5_19.