Simple Runs-Bounded FM-Index Designs Are Fast

Diego Díaz-Domínguez
Department of Computer Science, University of Helsinki, Finland

Saska Dönges
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Helsinki Institute for Information Technology (HIIT), Finland
Department of Computer Science, University of Helsinki, Finland

Leena Salmela
Department of Computer Science, University of Helsinki, Finland

Abstract

Given a string X of length n on alphabet σ, the FM-index data structure allows counting all occurrences of a pattern P of length m in $O(m)$ time via an algorithm called *backward search*. An important difficulty when searching with an FM-index is to support queries on L, the Burrows-Wheeler transform of X, while L is in compressed form. This problem has been the subject of intense research for 25 years now. Run-length encoding of L is an effective way to reduce index size, in particular when the data being indexed is highly-repetitive, which is the case in many types of modern data, including those arising from versioned document collections and in pangenomics. This paper takes a back-to-basics look at supporting backward search in FM-indexes, exploring and engineering two simple designs. The first divides the BWT string into blocks containing b symbols each and then run-length compresses each block separately, possibly introducing new runs (compared to applying run-length encoding once, to the whole string). Each block stores counts of each symbol that occurs before the block. This method supports the operation $\mathsf{rank}_c(L, i)$ (i.e., count the number of times c occurs in the prefix $L[1..i]$) by first determining the block i/b in which i falls and scanning the block to the appropriate position counting occurrences of c along the way. This partial answer to $\mathsf{rank}_c(L, i)$ is then added to the stored count of c symbols before the block to determine the final answer. Our second design has a similar structure, but instead divides the run-length-encoded version of L into blocks containing an equal number of runs. The trick then is to determine the block in which a query falls, which is achieved via a predecessor query over the block starting positions. We show via extensive experiments on a wide range of repetitive text collections that these FM-indexes are not only easy to implement, but also fast and space efficient in practice.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases data structures, efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.7

Supplementary Material *Software (Source Code)*: `https://github.com/saskeli/block_RLBWT` archived at `swh:1:dir:244fd876cd15bd291d91e444c8c04bb289aed05f`

Funding This work was supported in part by the Academy of Finland via grants 339070, 351150, and 323233.

Acknowledgements The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Given a string $X[0, n-1]$, the suffix array [21] of X, denoted SA_X (or just SA when clear from context) is a permutation of the integers $[0, n-1]$ that tells the lexicographical order of the suffixes of X, i.e., SA is the permutation such that $X[SA[0]..n] < X[SA[1]..n] < \ldots <$

$X[SA[n-1]..n]$. Because of the lexicographical ordering, all the positions of occurrence of any pattern P that is a substring of X lie in a contiguous interval of SA, and thus pattern matching over X reduces to determining the appropriate interval of SA.

An FM-index is a data structure that supports finding such SA intervals. The main common feature of the FM-index and its variants is support for the function $\mathsf{extendLeft}([i,j],c)$, which, given a suffix array interval $SA[i,j]$ containing all the occurrences of some pattern P, and a symbol c, returns the interval $[i',j']$ such that $SA[i',j']$ contains all occurrences of pattern $P'=cP$. Clearly, having support for $\mathsf{extendLeft}$ allows one to perform pattern matching: given a pattern P of length m, we can find all the occurrences of P in X via a sequence of m applications of $\mathsf{extendLeft}$ that return intervals corresponding to increasingly longer suffixes of the pattern.

FM-index implementations (see [19, 23] and references therein) differ primarily on how they support $\mathsf{extendLeft}$. For many years, this was done via rank queries on the Burrows-Wheeler transform (BWT) of the input string. The BWT of string X, which we denote with L_X, or just L when clear from context, is a permutation of the symbols of X defined by SA_X. In particular, $L[i]=X[SA[i]-1]$ except when $SA[i]=0$, in which case $L[i]=X[n]$.

In their groundbreaking article [10], Ferragina and Manzini showed that, somewhat remarkably, $\mathsf{extendLeft}([i,j],c)=[C[c]+\mathsf{rank}_c(L,i), C[c]+\mathsf{rank}_c(L,i)-1]$, where $C[c]$ is the total number of symbols in X less than symbol c, and the query $\mathsf{rank}_c(L,i)$ returns the number of occurrences of symbol c in $L[0..i]$. Most known FM-index variants follow this scheme. The fastest rank-based FM-index we know of, due to Gog et al. [13], divides L into blocks and represents each block with a Huffman-shaped wavelet tree, the bitvectors of which are compressed with different schemes depending on characteristics of each block's entropy.

Another more recent (and less populated) class of FM-index variants avoids rank queries altogether, instead essentially storing space-efficient mappings from (interval, symbol) pairs to intervals in order to implement $\mathsf{extendLeft}$. The first of these methods is due to Belazzougui and Navarro [3], who describe an index that uses minimal monotone perfect hash functions and uses $nH_k+O(n)$ bits of space, where H_k is the kth-order empirical entropy of the input text [22]. More recently, Nishimoto and Tabei [25], describe a structure using $O(r)$ words space, where r is the number of runs[1] in L.

On highly repetitive strings, which are now produced by many sources, and are notably central to the field of computational genomics [28, 8, 9, 23, 20], the BWT string L is composed of relatively few runs – i.e., r is significantly smaller than n, perhaps by 2-3 orders of magnitude depending on the input – and so L can be compressed well with run-length encoding. The problem then becomes supporting $\mathsf{extendLeft}$ (via rank queries or otherwise) in close to $O(r)$ space.

Contribution. This paper is a back-to-basics examination of run-length compressed FM indexes that explores two simple ways to achieve space usage close to $O(r)$ words while maintaining fast query times in practice.

Both designs are rank-based and divide L into blocks. The first approach divides L into n/b blocks of b symbols each (with the last block possibly having less), before applying run-length encoding to each block. Rank can be implemented on this structure in $O(b)$ time and the index takes $O(r+\sigma n/b)$ words of space. The second approach divides L into blocks containing an equal number b' of *runs*. Rank now takes $O(b'+\mathsf{pred}(r/b'))$ time, where

[1] A run $L[i..j]=c^\ell$ has $\ell>1$ consecutive copies of c such that $i=1$, or $L[i-1]\neq c$, and $L[j]=n$ or $L[j+1]\neq c$.

$\mathsf{pred}(m)$ is the time for a predecessor query over m integers, and space is $O(r + \sigma r/b')$ words. We find that both these simple schemes afford very fast practical implementations, and dominate other approaches in practice on many data sets. Along the way we explore practical structures to support predecessor queries, which may be of independent interest.

Our rationale in this study is twofold. Firstly, simple code is easier to maintain and easier to adapt to different application requirements, which is important given the relevance of the FM-index in modern genomics analyses [18, 8]. Secondly, simpler data structures are often easier to optimize, both for the programmer and for the compiler. For the particular problem we consider in this paper, our results suggest that simpler designs may have an inherent speed advantage over more complex ones. In their simplicity and strong performance, the indexes we describe herein represent non-trivial baselines against which the performance of future, possibly more sophisticated, FM indexes can be measured.

Roadmap. The remainder of this paper is organized as follows. In the next section we review related work. Our new FM-indexes are then described in Section 3. In that section we also present a microbenchmark of different predecessor structures, which are essential for our second index. Then, in Section 4, we present results of benchmarks comparing our indexes to the state of the art. Conclusions and avenues for future work are then offered.

2 Background and Related Work

We now briefly describe four FM index designs [19, 13, 2, 25] that represent the state of the art, either in theory or practice. Implementations of all methods described here are included in our experiments. For a more thorough treatment, including earlier and more obscure indexes, we refer the reader to [24, 23].

Mäkinen and Navarro [19] proposed the FM-index variant to encode L in $O(r \log n)$ bits. It supports $\mathsf{rank}_c(L, i)$ (and therefore $\mathsf{extendLeft}$) in $O(\log \log_w(\sigma + n/r))$ time. Our description here follows that given in Gagie et al. [11].

Let $R = (c_1, \ell_1), \ldots, (c_r, \ell_r)$ be the run-length encoding of L, with pair (c_i, ℓ_i), $i \in [1..r]$, representing the ith run in L, where $c_i \in \Sigma$ is the run symbol and ℓ_i is the run length. We maintain a vector $L' = c_1, \ldots, c_r$ with the run heads in the same order as they appear in L. L' is represented with a data structure of $O(r \log n)$ bits that supports $O(1)$-time access and $\mathsf{rank}_c(L', i)$ in $O(\log \log_w \sigma)$ time. We also maintain an array $C'[1, \sigma]$ storing in $C'[c]$ the number of runs in L whose symbol is smaller than $c \in \Sigma$.

A predecessor data structure of $O(r \log n)$ bits encodes the set $E = \{1\} \cup \{1 < i \leq n,\ L[i-1] \neq L[i]\}$ with the positions in L for the run heads (i.e., the leftmost symbol in every run). The query $\mathsf{pred}(E, i)$ returns pair (i', b), where i' is the predecessor of i in E, and b is the rank of i' in E. $\mathsf{pred}(E, i)$ takes $O(\log \log_w(n/r))$ time using [4].

Let R' be a permutation of R in which the runs are stably sorted by their symbols and let (c_i, ℓ_i) be the ith run in the permutation R'. We store an array $D[1..r]$ of $r \log n$ bits storing in $D[i]$ the cumulative length of the runs associated with c_i in $R'[1..i]$.

Answering $\mathsf{rank}_c(L, i)$ in the RLFM requires us to call $(i', b) = \mathsf{pred}(E, i)$ to get the head position i' and rank b for the run where i lies. We can then obtain the symbol $c' = L'[b]$ associated with i's run. Subsequently, we obtain the number $k = \mathsf{rank}_c(L', b-1)$ of runs for c in the prefix $L'[1..b-1]$ and finally compute the number $x = D[C'[c] + k]$ of cs in the prefix $L[1..j'-1]$, returning $x + i - i' + 1$ if $c = c'$, or return x otherwise.

Overall query time is dominated by $\mathsf{pred}(E, i)$ and $\mathsf{rank}_c(L', b-1)$, which combined take $O(\log \log_w(\sigma + (n/r)))$ time.

Gog et al. [13] describe what is currently the fastest general-purpose FM index we know of. Their approach shares some similarity to one of our schemes in that it divides L into blocks of fixed size b. Each block stores, for each symbol of the alphabet, precomputed ranks up to the beginning of the block. Each block is then encoded using a Huffman-shaped wavelet tree [19], which can answer rank queries up to the beginning of each block, and combined with the precomputed ranks, enables general rank queries to be answered in $O(\log \sigma)$ time. Each wavelet tree take space proportional to the entropy of its block, leading to a bound of $nH_k + o(n \log \sigma)$ bits for the whole index.

Although the index size is not directly relatable to r, the number of runs in L, experiments in Gog et al. [13] show the index performs well on inputs with small r. Intuitively, on a L having many long runs, the blocks tend to have a small alphabet with a skewed distribution of symbols and so will have a small representation when Huffman encoded (which is essentially what the Huffman-shaped wavelet tree does). However, a further optimization has been made, that favors runs even further. In particular, the bit vectors of the wavelet trees are represented with the hybrid encoding of Kärkkäinen et al. [16] which run-length encodes bitvectors having long runs. If the block being encoded has long runs then the bitvector at the root of its wavelet tree will have at least as many, and bitvectors at other nodes will tend to preserve runs too. This makes the space usage much closer to r than it was designed to be, even if it does not explicitly encode runs in L.

Prezza et al. [2] store one character per run in a string $H \in \Sigma^r$ and mark with a 1 the beginning of each run in a bitvector $V_{all}[0..n-1]$. For every $c \in \Sigma$ they store the lengths of all runs of character c consecutively in a bitvector V_c. More precisely, every run of symbol c of length k is represented in V_c as 10^{k-1}. This representation allows them to map rank and access queries on L to rank, select and access queries on H, V_{all}, and V_c. By gap-encoding the bitvectors, this representation takes $r(2\log(n/r) + \log \sigma)(1 + o(1))$ bits of space. The multiplicative term $\log(n/r)$ can be reduced by storing in V_{all} just one out of $1/\epsilon$ ones, where $0 < \epsilon \leq 1$ is a constant. One is still able to answer all queries on L, by using the V_c vectors to reconstruct the positions of the missing ones in V_{all}, though this multiplies query time by $1/\epsilon$. In their implementation of this scheme, Prezza et al. [2] represent H as a Huffman-shaped wavelet tree and store the bitvectors in an Elias-Fano structure [29].

Nishimoto and Tabei [25] proposed the first encoding that represents L in $O(r)$ bits and supports $\mathsf{extendLeft}(i, L[i])$[2] in constant time without resorting to rank operations. Let $I = \{(s_1, e_1), \ldots, (s_r, e_r)\}$ be a sequence of r consecutive ranges over $[1, n]$ such that $L[s_j, e_j]$, with $j \in [1, r]$, is the jth run of L (from left to right). Each range $(s_j, e_j) \in I$ has an associated mapping pair $(s'_j = \mathsf{extendLeft}(s_j, L[s_j]), e'_j = \mathsf{extendLeft}(e_j, L[e_j])$ that represents the contiguous range $L[s'_j, e'_j]$ one obtains by performing $\mathsf{extendLeft}$ operations over the symbols in $L[s_j..e_j]$. Note that (s'_j, e'_j) does not necessarily match a range in I, but can be fully contained in one or cover several of them. The key idea in Nishimoto and Tabei's method to obtain constant time is to further break the ranges to produce a new sequence I' of length r', $r \leq r' < 2r$ where every $(s_j, e_j) \in I'$ has a mapping pair (s'_j, e'_j) that covers a constant number c of other ranges in I'. They maintain an array $D[1..r']$ that stores in $D[j]$ the pair (s_j, s'_j) and a vector $D' = [1..r']$ that stores in $D'[j] = y \in [1, r']$ the index y of the pair $(s_y, e_y) \in I'$ where s'_j lies (i.e., $s_y \leq s'_j \leq e_y$). Answering $\mathsf{extendLeft}(i, L[i])$ in this scheme requires first knowing the pair $(s_j, e_j) \in I'$ enclosing i, then scanning the area $D[D'[j]..D'[j]+c-1]$ until the range $(s_y, e_y) \in I'$ that contains $\mathsf{extendLeft}(i, L[i]) = s'_j + (i - s_j)$ is found. This approach was recently implemented by Brown et al. [5].

[2] This constrained version of $\mathsf{extendLeft}$ is also known as $\mathsf{LF}(i)$ in the literature.

3 Simple Runs-Bounded FM-index Designs

Both of our indexing schemes are based on splitting the BWT into run-length encoded blocks along with preceding symbol counts. To answer $\mathsf{rank}_c(L, i)$ queries, it suffices to find the block containing the ith character, scanning the block and adding the result of the scan to the number of preceding occurrences of c.

The number of preceding symbols are stored in at most $7 + \sum_{c\in\Sigma}\lceil\log_2 \mathsf{rank}_c(L, n)\rceil$ bits per block[3]. In our implementations with a constant number of symbols per block, there is an additional "super block" level that stores precomputed answers for rank queries for blocks of size 2^{32}. The super block approach was taken to allow storing large chunks of memory for the blocks instead of just one very large block of memory, and to guarantee that four bytes would be sufficient to store any symbol counts on the block level. Neither of these turned out to be relevant in our experiments, and the additional layer of complexity likely slows down query performance very slightly and has a negligible impact on data structure size.

Blocks with b symbols. If each block contains b symbols. We can store pointers to the $\lceil n/b\rceil$ blocks in an array BP. With this, $\mathsf{rank}_c(L, i)$ reduces to $\mathsf{rank}_c(BP[\lfloor i/b\rfloor], i \bmod b)$ on the block.

The upper bound space complexity, given logarithmic encoding of integers comprises of:

- $\mathcal{O}\left(\left(\frac{n}{b} + r\right)(\log b + \log\sigma)\right)$ bits for run encoding,
- $\mathcal{O}\left(\frac{n}{b}\sigma\log n\right)$ bits for encoding the block headers containing preceding character counts and
- $\mathcal{O}\left(\frac{n}{b}w\right)$ bits for storing block pointers, w being the machine word length,

making

$$\mathcal{O}\left(\left(\frac{n}{b} + r\right)(\log b + \log\sigma) + \frac{n}{b}(w + \sigma\log n)\right)$$

bits an asymptotic upper bound for these Sb (for *symbol block*) structures.

Query time is simply $\mathcal{O}(b)$ under the word RAM model. We expect at most two memory transfers from uncached memory locations per query, meaning most of the work would be done on cached data.

The blocks themselves consist of a run-length-encoded sequence either using two bytes per run with runs split as necessary, or with each run being encoded with a variable number of bytes depending on the length of the run. In both approaches the first $\lceil\log_2\sigma\rceil$ bits encode the run head.

- In two byte encoded blocks, $16 - \lceil\log_2\sigma\rceil$ bits are used to store the run length. Thus the maximum length for one run is $2^{16-\lceil\log_2\sigma\rceil}$, meaning long runs will need to be split into multiple runs within the same block. In the worst case, for a large alphabet ($> 2^7$) and large block sizes ($\geq 2^{13}$) a single run can be split into more than 2^4 blocks, thus increasing the space usage considerably. However, for genomics data and small ($< 2^{13}$) block sizes, the vast majority of runs fit into two bytes. Scanning these two byte encoded blocks is fast since there is no branching or data dependencies in decoding runs and only a single branch miss-prediction to end scanning. We note that for this variant, the bounds for space complexity apply as long as b is considered a bound constant as opposed to being linear in n for example.

[3] The +7 bits is the possible overhead of keeping data byte-aligned.

- With runs encoded into a variable number of bytes, arbitrarily long runs can be efficiently encoded in $\mathcal{O}(\log \sigma + \log m)$ bits where m is the length of the run. With long runs this is more space efficient than the two byte approach, but decoding, especially of short runs is slower as decoding requires data-dependent branching.

In addition to these encodings we experimented with hybrid compression schemes where the encoding is chosen on a per-block basis. However, we found that the overhead of decoding the more rare block type is high enough that our dynamic encodings remain uncompetetive with more simple block encodings. We suspect that in addition to a branch missprediction, the cold code path causes inefficiencies in instruction caching or pipeline unrolling, that dominate the efficiency gains of a superior encoding for some minority of blocks. An improvement in performance of querying the rarer blocks with a separate encoding compared to the majority blocks of ≈ 100 nanoseconds ($\approx 20\%$ improvement) would be sufficient to warrant reconsidering a hybrid approach. This idea may be feasible with some as yet untested block encodings.

Blocks with b' runs. For finding the block containing the ith symbol when blocks contain a variable number of symbols, we use a predecessor search to find m and $\mathsf{arg\,max}_j(j \leq i)$ s.t. block m starts with the jth symbol. Now given m and j, $\mathsf{rank}_c(L, i)$ becomes $\mathsf{rank}_c(BP[m], i - j)$.

For running the predecessor search we tested four different approaches:

- A simple **binary search** over an array of tuples with the number of symbols in L preceding each block and a pointer to the block itself. This approach is very simple to implement and has minimal memory overhead. However, the memory access pattern is not well supported on modern microprocessors, unless the entire array can fit in cache.
- A **heap ordered binary search tree** is a balanced search tree stored in an array A, such that for every internal node $A[i]$, the left child will be found at $A[2i + 1]$ and the right child at $A[2i + 2]$. Internal nodes contain the number of symbols represented by their left sub tree, while leaves contain pointers to the actual blocks. The computation done is exactly the same as for binary search, but the heap ordering makes the memory pattern an increasing stride in the same direction, allowing for more efficient predictive caching by modern microprocessors. For tree traversal to work properly, all but the final internal level in the tree need to be full (containing 2^ℓ elements where ℓ is the level) even if some sub trees are empty, this potentially doubles the memory footprint of the heap ordered binary search tree compared to a simple binary search.
- A **B$^+$-tree** [6] is a B-tree with pointers to blocks only stored in the leaves, while internal nodes only store the number of symbols represented by the sub trees. When $B = 2^k$ for some $k \in \mathbb{N}$, a fast templated branchless [27] binary search can be implemented for branch selection in the B-tree. While the internal nodes close to the root of the tree are likely to be present in cache, accessing nodes closer to the leaves are likely to trigger cache misses. In addition, the B-tree is comparatively space inefficient due to the need to store pointers internally.
- A **heap ordered B$^+$-tree** aims to be the best of both worlds, by storing the B$^+$-tree in contiguous memory without the need for child pointers. The children of node $A[i]$ will be at $A[Bi + 1..Bi + B]$. Internal branch selection can be done with templated binary searches and the memory access pattern follows a somewhat predictable increasing forward stride. While the need to keep internal levels of the tree full implies significant memory overhead, the low number of levels necessary to do block selection even for large data sets, keeps the memory overhead low in practice.

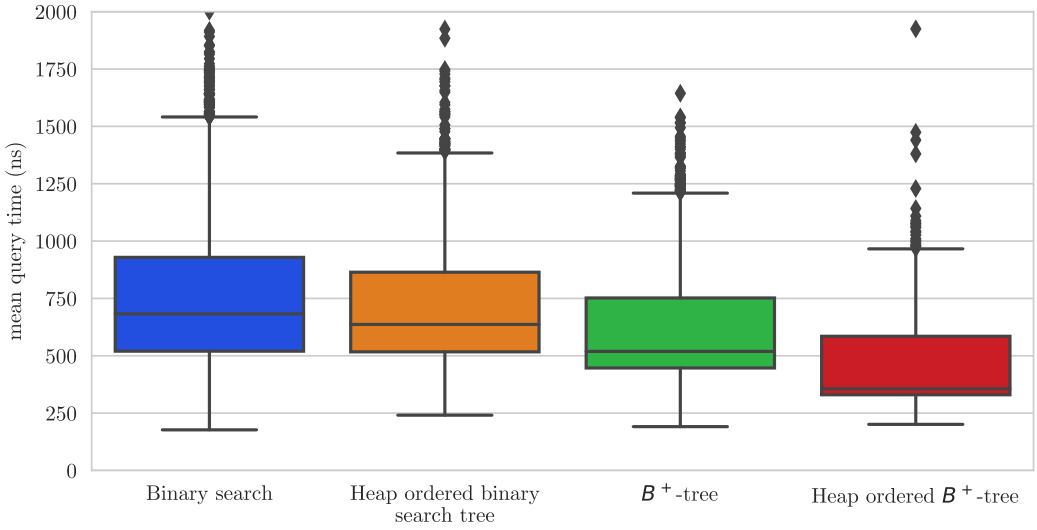

Figure 1 Predecessor search performance. Based on 10^5 $\mathsf{pred}(x)$ queries over 10^7 elements, with x chosen at random from $[0..m+10]$ where m is the largest element in the collection. Both B$^+$-tree and the heap ordered binary search tree are faster than a simple binary search, while the heap ordered B$^+$-tree is significantly faster still. The performance here and in Table 1 differ due to the overhead of outputting results of every query as opposed to just calculating summary statistics.

After benchmarking all of these potential approaches, we found that the heap ordered B$^+$-tree was the fastest and had reasonable memory overhead (Figure 1 and Table 1). The space complexity of our RB (for *run block*) implementation comprises of:

- $\mathcal{O}\left(r(\log\sigma + \log n)\right)$ bits for encoding runs,
- $\mathcal{O}\left(\frac{r}{b'}\sigma\log n\right)$ bits for encoding the block headers containing preceding character counts,
- $\mathcal{O}\left(\frac{r}{b'}w\right)$ bits for storing block pointers, and
- $\mathcal{O}\left(\frac{r}{b'}w\right)$ bits for storing the heap ordered B$^+$-tree,

making

$$\mathcal{O}\left(r(\log\sigma + \log n) + \frac{r}{b'}(\sigma\log n + w)\right)$$

bits an asymptotic upper bound for these RB structures.

Time complexity is $\mathcal{O}\left(\log\frac{r}{b'} + b'\right)$. We expect fewer than $\log r/b'$ memory accesses to uncached memory locations due to the apparent efficiency of pre-caching with increasing stride.

See Figure 2 for an example, illustrating the memory layout and query logic of our index structures.

Table 1 Performance of benchmarked predecessor search structures. 10^5 random predecessor queries on a set of ten million random (unique) elements. Heap ordered binary search trees and B$^+$-trees are faster than a traditional binary search, but have significant space overhead. Heap ordered B$^+$-tree is faster still with only minor overhead in space efficiency compared to binary search.

	binary search	heap ordered BST	B$^+$-tree	heap ordered B$^+$-tree
Mean query time	524.348ns	331.548ns	363.409ns	313ns
Space usage	76.294MB	128MB	157.511MB	78.3257MB

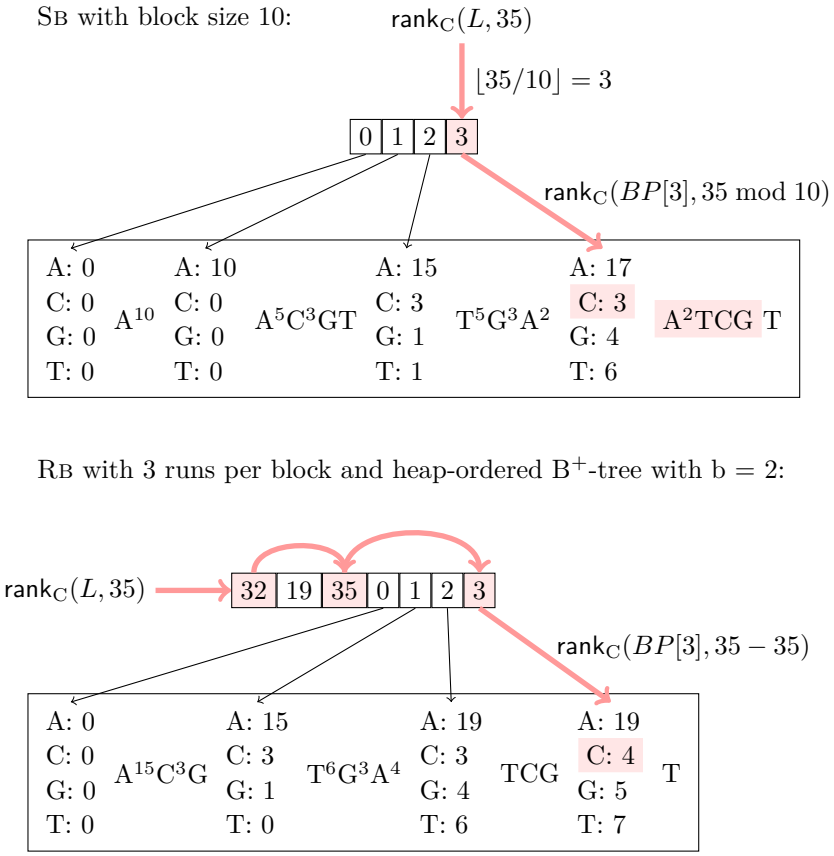

Figure 2 Illustrative example of our index structures. Indexes built on input sequence $L = A^{15}C^3GT^6G^3A^4TCGT$. Memory access pattern for rank$_C(L, 35)$ highlighted. For SB, five symbols, in four runs, containing one "C" character get read from block $BP[3]$, this one "C" count gets added to the number of preceding "C" characters in the block header (3), these get added together giving rank$_C(L, 35) = 4$. For RB the pred query tells us that $BP[3]$ is the target block, and that a total of 35 symbols precede the block, no scanning is needed and the result is the total count of "C" characters in the block header (4).

4 Performance

We implemented the designs described in Section 3 to verify their efficacy in practice, and to explore time-space trade-offs in performance by varying b' for our RB variant and varying b for SB variants. The source code is in C++, and it is available at `https://github.com/saskeli/block_RLBWT`.

4.1 Experimental Setup

Machine and Compilation. Tests were run on a machine with an AMD EPYC 7452 processor and 496GB of DDR4 RAM. The operating system was AlmaLinux 8.4 (with Linux 4.18.0-372.9.1.el8.x86_64 kernel). Code was compiled with GCC 12.2.0 and the performance-relevant flags: `-std=c++2a, -march=native, -Ofast` and `-DNDEBUG`. Experiments were repeated on a machine on the same HPC cluster with an Intel Xeon E7-8890 v4 processor (see Appendix B). We found results to be relatively stable across these two systems.

Indexes Measured. We use `sbt_rlbwt` (for *symbol blocks, two-byte*) to refer to the SB FM index variant from Section 3 that first divides the BWT into blocks contain equal numbers of symbols and then applies two byte run length encoding to each block; `sbv_rlbwt` (for *symbol blocks, variable width*) to refer to the SB index variant that divides the BWT into blocks containing equal numbers of symbols before using a variable width encoding to compress the blocks; and use `rb_rlbwt` for our RB index that divides the run-length encoded BWT into blocks of fixed runs.

We explored the space efficiency / query time trade-offs of varying block sizes for our indexes. We used block sizes of $2^8 \leq b \leq 2^{17}$ for SB variants, and $2^0 \leq b' \leq 2^9$ for RB, as these ranges cover what we consider practical values in most cases.

- `sbt_rlbwt`. Uses a default block size of $b = 2^{11}$, which yields reliably fast indexes and occasionally suffer in terms of compression performance.
- `sbv_rlbwt`. Uses a default block size of $b = 2^{14}$. This value enables good compression ratios most of the time while not completely sacrificing query performance.
- `rb_rlbwt`. With each block containing $b' = 32$ runs by default and using a heap ordered B^+-tree predecessor structure, with $B = 64$, for answering pred queries. We selected these parameters as good defaults for genomics data sets.

We further compared the performance of our variants using default parameters against other state-of-the-art FM-index implementations. These were:

- FBB. The fixed-block boosting with wavelet trees index of Gog et al. [13]. We obtained the code from the `SDSL-lite` library [12]. We realised that substituting the hybrid bit vector implementation used in the wavelet trees with other bit vector variants available in the `SDSL-lite` library led to interesting time / space trade-offs for the FBB implementations. We include four of these variants as `fbb_xx` in our results
  - `fbb_hyb` was built using the hybrid bit vector of Kärkkäinen et al. [17]. The `SDSL-lite` library implements the hybrid bit vector in the `hyb_vector` class.
  - `fbb_bv` was built using uncompressed bit vectors from the `SDSL-lite` library (class `bit_vector`) with separate rank support data structures.
  - `fbb_il` was built using the `SDSL-lite` implementation (`bit_vector_il`) of the bit vector from Gog et al. [14] that interleaves the partial rank queries with uncompressed bit vector blocks.
  - `fbb_rrr` uses the `rrr_vector` class of `SDSL-lite`, which implements the H_0-compressed bit vector representation of Raman et al. [26].
- RLBWT is the `rlmn` class of the `SDSL-lite` library that implements the run-length-encoded BWT representation with rank support of Mäkinen and Navarro [19].
- SRLBWT. The sparse RLBWT scheme of Belazzougui et al. [2], which is the component used for counting in the the r-index implementation [11]. We obtained its implementation from the `r-index`'s official repository[4].
- R-INDEX-F represents the scheme of Nishimoto and Tabei [25] as implemented by Brown et al. [5][5].

We remark that in preliminary experiments (not shown here) we also measured the performance of indexes based on balanced and Huffman-shaped wavelet trees applied to the entire BWT (with various internal bitvector representations), but found, as other authors

[4] `https://github.com/nicolaprezza/r-index`
[5] `https://github.com/drnatebrown/r-index-f`

Table 2 Details of the datasets used in performance benchmarks. The symbol σ denotes the alphabet size, n is the number of symbols in the dataset, and r is the number of runs in the BCR BWT of that dataset.

Dataset	σ	n	r	n/r
`hum50`	16	1.54×10^{11}	3.89×10^{9}	39.53
`ecoli3.6K`	16	1.90×10^{10}	1.57×10^{8}	120.55
`cov400K`	6	1.19×10^{10}	9.05×10^{6}	1317.79
`einstein`	140	4.68×10^{8}	2.90×10^{5}	1611.18
`worldleaders`	90	4.70×10^{7}	5.73×10^{5}	81.90
`coreutils`	235	2.05×10^{8}	4.68×10^{6}	43.82

have (e.g., [15, 13]), that these approaches were always inferior to the indexes listed above on our data sets. We therefore exclude these from further mention in the experiments for the sake of clarity.

Datasets. We used six repetitive collections for the experiments. See Table 2. They vary in length, alphabet size and level of repetitiveness to reflect different application scenarios. The dataset `hum50` consists of 50 different human assemblies. The dataset `ecoli3.6k` is the concatenation of 3600 E. coli genomes. The dataset `covid400k` contains 400,000 variants of the covid genome. `einstein` and `worldleaders` are each concatenations of different versions of a Wikipedia entry (for "Albert Einstein" and "World Leaders", respectively). `coreutils` contains different versions of the GNU coreutils' source code[6].

Benchmarks. For every dataset, we proceeded as follows. We built its BCR BWT string L [1] using the `grlBWT` tool [7]. Then, with L as input, we built an instance of each index listed above. We sampled sets of patterns of lengths 10, 30 and 50. For each set we sampled substrings of the desired length from ten thousand random positions in the input data set, such that the substrings contained only printable characters for the general data sets and no "N" symbols for genomic data sets. We measured the elapsed times for each index to count all patterns in each pattern set and then took the average, further dividing by pattern length to get a per symbol time for each pattern set.

4.2 Results

Time-space trade-offs. Increasing the block sizes for Sb variants had the expected effect of improving compression while sacrificing query performance, suggesting that when optimizing for query speed, block size should be reduced as much as possible within memory constraints. For Rb increasing block size improves compression to the detriment of query performance, but reducing the number of runs per block beyond a certain point decreases query performance. This performance decrease is expected if we observe that as the number of runs per block approaches one, the index becomes a heap ordered B^+-tree with single runs at the leaves, and the $\Theta(\log r/b)$ tree traversal becomes $\Theta(\log r)$. These performance trade-offs are shown in Figure 3.

[6] More details of the last three data sets can be found at corpus: `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

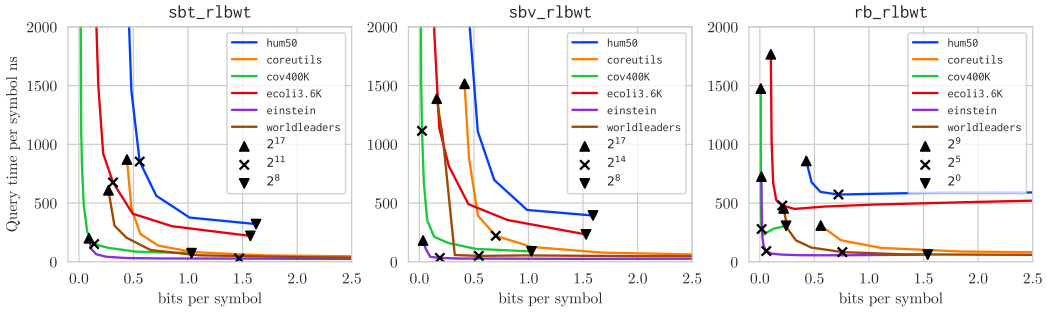

Figure 3 Performance trade-offs for varying block sizes, with maximum, default and minimum tested block sizes annotated as shapes. SB variants have a clear trade-off where increasing block size improves compression and slows down queries, while decreasing block size degrades compression performance and speeds up queries. The same is mostly true for RB as well but the increased overhead of the predecessor search when block sizes decrease limits how much speed up is possible. The sweet spot for minimizing both compressed size and query time differs between data sets. Default parameters for `sbt_rlbwt` and `rb_rlbwt` seem mostly reasonable but parameter tuning is recommended for best performance in specific applications.

Comparison against other FM-index implementations. Our index implementations are very competitive with other FM-index implementations, as can be observed in Figure 4. Our experiments show that as pattern length increases, the performance of our indexes improves in comparison to the competition (See Appendix A for figures with additional pattern lengths). We posit that this gain is due to the likelihood of both of the `rank` queries associated with a step of `extendLeft` targeting the same block increasing as the number matching suffixes becomes lower. This feature allows the second query to act on fully cached memory making the simple scanning approaches very fast in practice.

5 Concluding Remarks

We have described, engineered, and experimentally analysed two strikingly simple FM indexes, which often outperform more complex prior art. We close with three avenues for future work.

Firstly, our indexes were designed with genomics applications in mind and so, therefore, the approach we selected for encoding and decoding run heads is likely to be sub-optimal for data sets on larger alphabets. We believe that improving the run-head encoding with, e.g., Huffman codes, and allowing the character encoding, in some cases, to exceed eight bits, would improve compression with a negligible impact on query performance.

For larger alphabets, the space overhead incurred by the precomputed ranks stored at each block quickly becomes substantial. To mitigate this, it may be fruitful to treat rare symbols differently, possibly with an entirely different rank structure, avoiding the need to store a full σ precomputed ranks at each block.

Finally, as mentioned in Section 3, selecting encoding on a per-block basis was explored but discarded as the overhead for decoding was too high to be practical with out currently available block encodings. Block encodings designed specifically for fast decoding of blocks with specific attributes may prove fast enough in practice to be worth the overhead in encoding detection when querying.

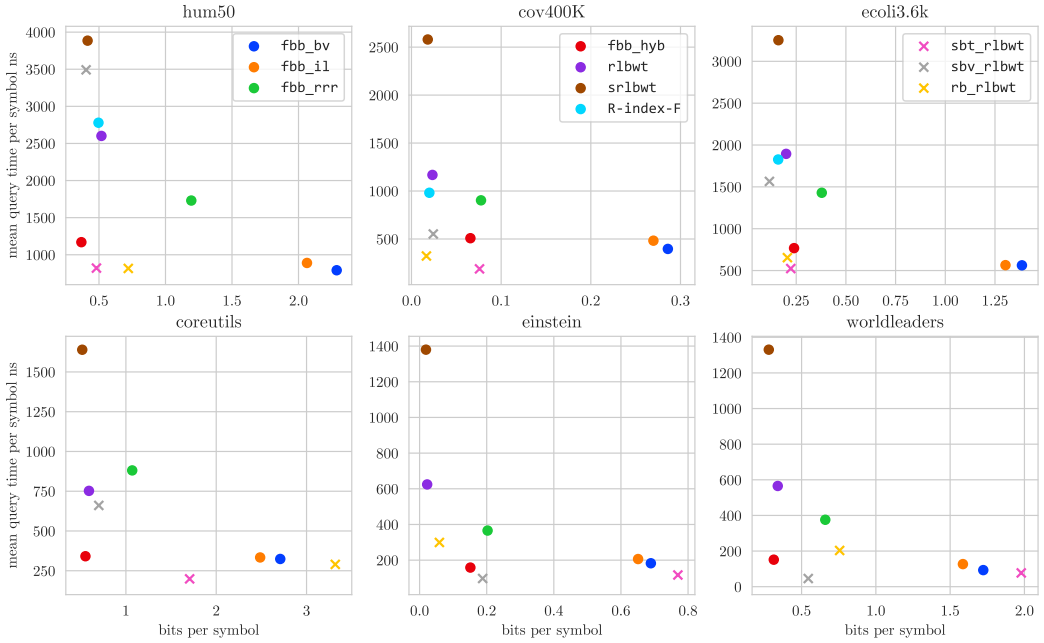

Figure 4 Mean match counting per symbol query times and bits per symbol for patterns of length 30. Our `sbt_rlbwt` and `rb_rlbwt` variants with default parameters are as fast or faster than the competition for genomics data, and compress significantly better than the closest competitors in query time. For the more general data sets we remain competitive but lose out in compression, possibly due to our implementation not doing any entropy-based compression, and thus encoding run heads inefficiently. However, the simple run head encoding does translate to good query performance.

References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

2 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Flexible indexing of repetitive collections. In *Proc. 13th Conference on Computability in Europe (CiE)*, pages 162–174, 2017.

3 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.

4 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4), 2015.

5 Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*, page article 16, 2022.

6 Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

7 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the BWT for repetitive text using string compression. In *Proc. 33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223, page article 29, 2022.

8 Dirk D. Dolle, Zhicheng Liu, Matthew Cotten, Jared T. Simpson, Zamin Iqbal, Richard Durbin, Shane A. McCarthy, and Thomas M. Keane. Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes. *Genome Research*, 27(2):300–309, 2017.

9 Jana Ebler, Peter Ebert, Wayne E. Clarke, Tobias Rausch, Peter A. Audano, Torsten Houwaart, Yafei Mao, Jan O. Korbel, Evan E. Eichler, Michael C. Zody, et al. Pangenome-based genome inference allows efficient and accurate genotyping across a wide spectrum of variant classes. *Nature Genetics*, 54(4):518–525, 2022.

10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

11 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):article 2, 2020.

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

13 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.

14 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

15 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Softw. Pract. Exp.*, 44(11):1287–1314, 2014.

16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. 14th Data Compression Conference (DCC)*, pages 302–311, 2014.

17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. 24th Data Compression Conference (DCC)*, pages 153–162, 2014.

18 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

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

20 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

21 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

22 Giovanni Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001.

23 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

24 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

25 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proc. 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, page article 101, 2021.

26 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

27 Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms (ESA)*, pages 784–796, 2004.

28 Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: astronomical or genomical? *PLoS Biology*, 13(7):e1002195, 2015.

29 Sebastiano Vigna. Quasi-succinct indices. In *Proc. Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 83–92. ACM, 2013.

A Additional match counting results on AMD EPYC 7452

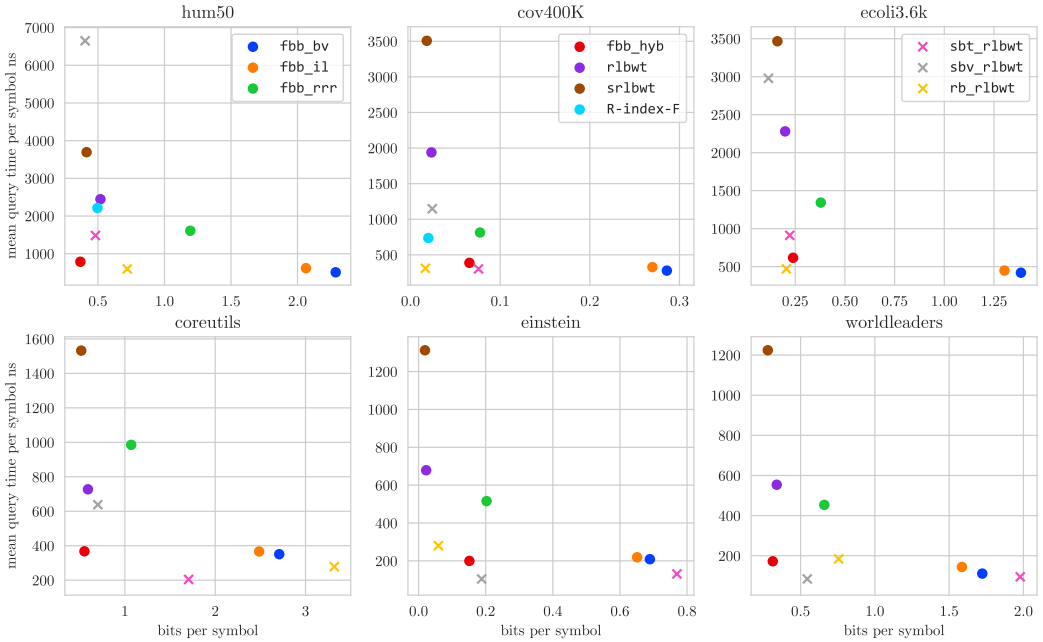

Figure 5 Mean match counting per symbol query times and bits per symbol for patterns of length 10.

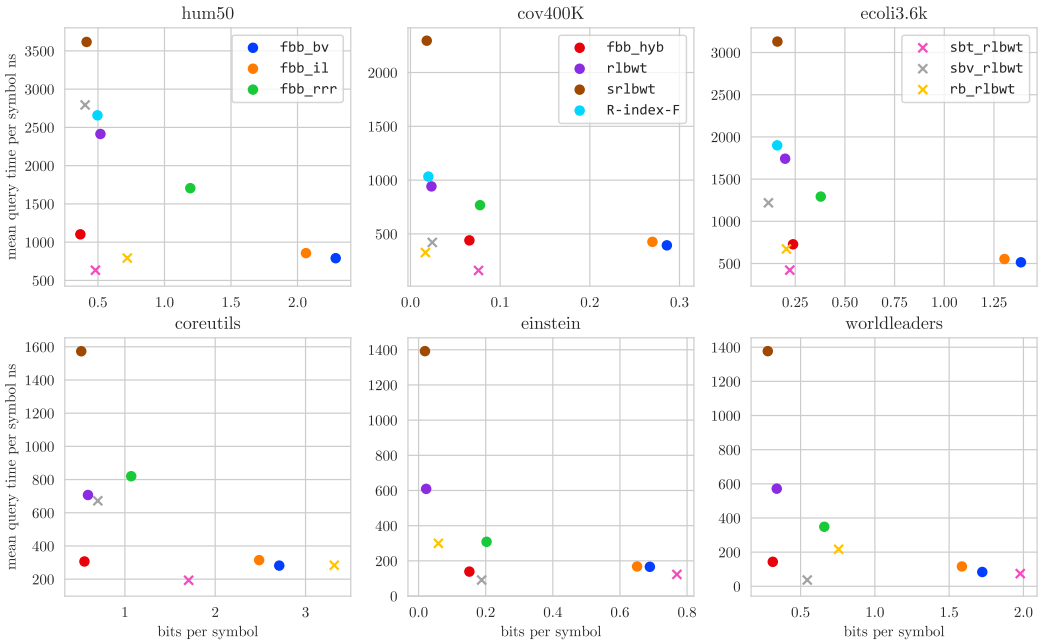

Figure 6 Mean match counting per symbol query times and bits per symbol for patterns of length 50.

B Intel Xeon E7-8890 v4 experiment results

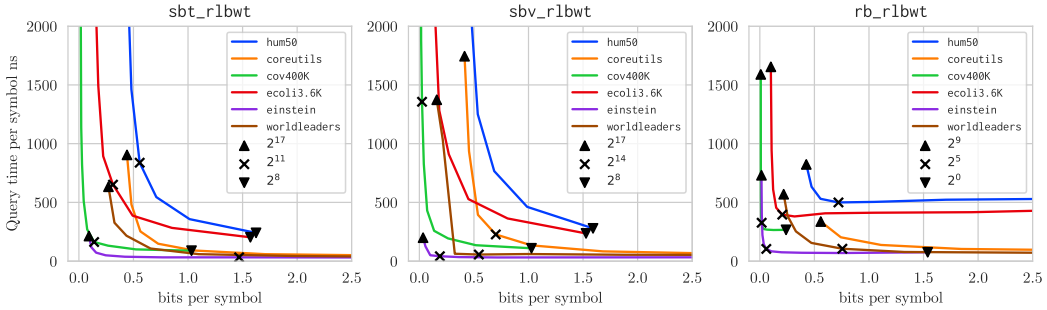

Figure 7 Performance trade-offs for varying block sizes. Run on Intel Xeon E7-8890 v4.

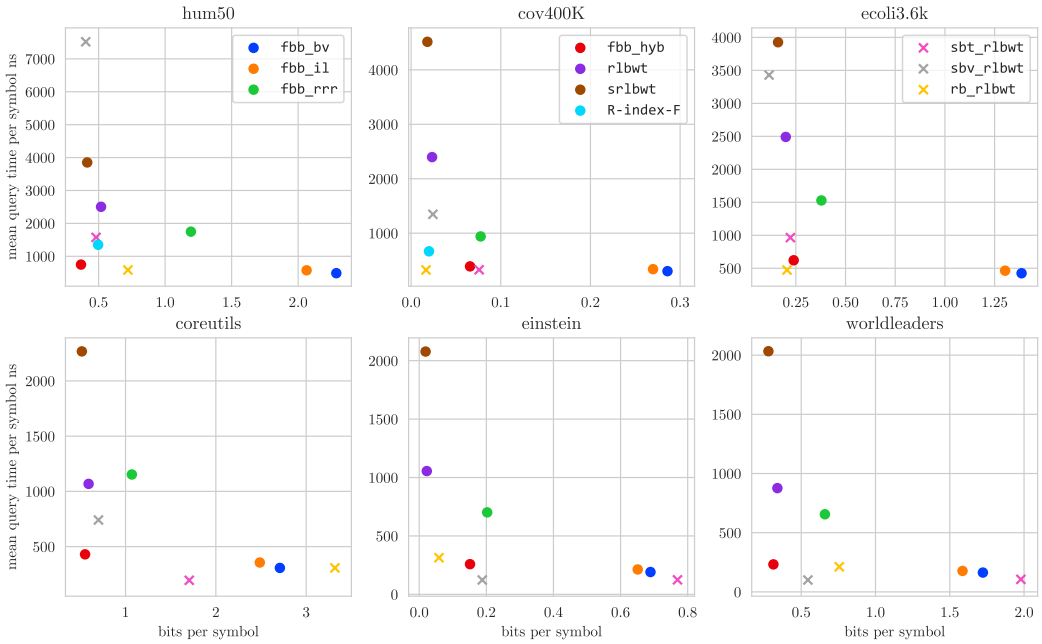

Figure 8 Mean match counting per symbol query times and bits per symbol for patterns of length 10. Run on Intel Xeon E7-8890 v4.

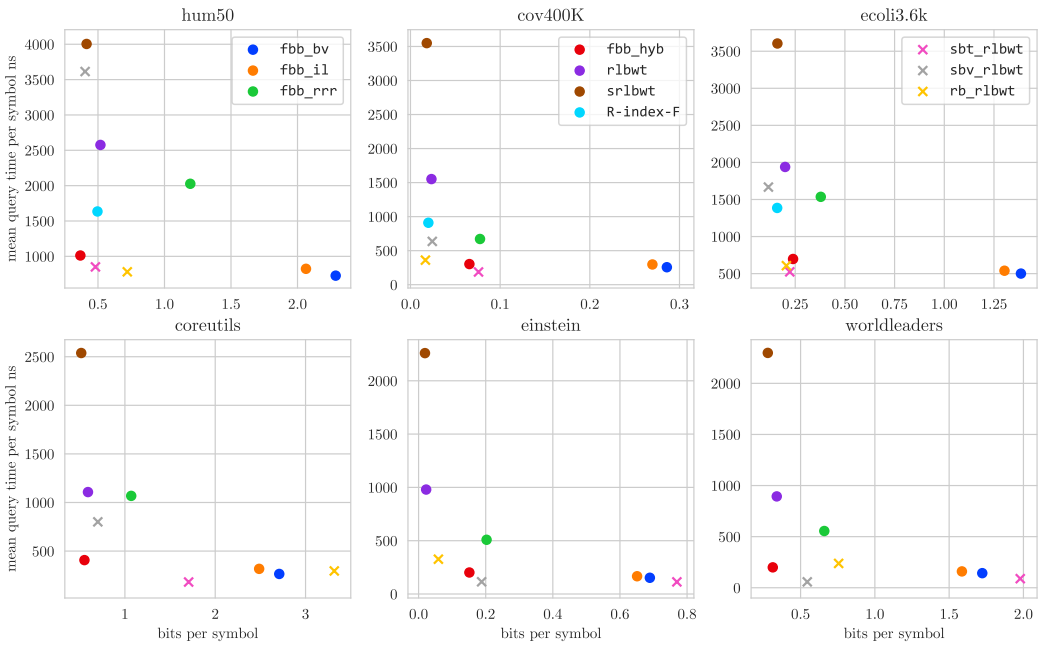

Figure 9 Mean match counting per symbol query times and bits per symbol for patterns of length 30. Run on Intel Xeon E7-8890 v4.

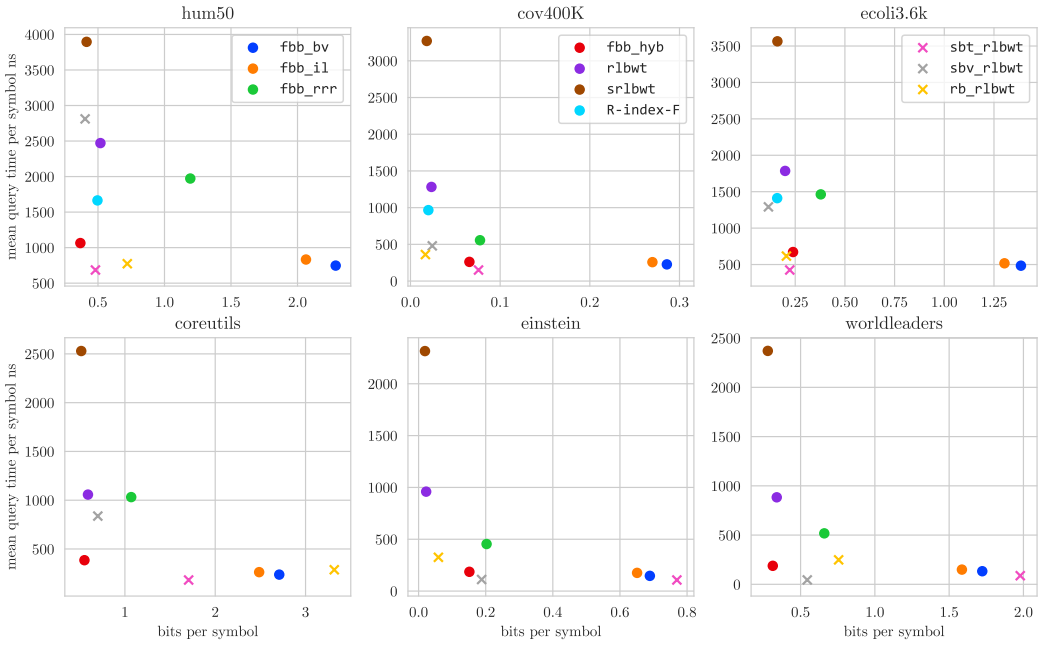

Figure 10 Mean match counting per symbol query times and bits per symbol for patterns of length 50. Run on Intel Xeon E7-8890 v4.