QuadRank: Engineering a High Throughput Rank

Ragnar Groot Koerkamp
Karlsruhe Institute of Technology, Germany

Abstract

Motivation. Given a text, a query $\mathsf{rank}(q, c)$ counts the number of occurrences of character c among the first q characters of the text. Space-efficient methods to answer these rank queries form an important building block in many succinct data structures. For example, the FM-index [13] is a widely used data structure that uses rank queries to locate all occurrences of a pattern in a text.

In bioinformatics applications, the goal is usually to process large inputs as fast as possible. Thus, data structures should have high *throughput* when used with *many threads*.

Contributions. We first survey existing results on rank data structures. For the $\sigma = 2$ binary alphabet, we then develop BiRank, which has 3.28% space overhead. BiRank merges the central ideas of two recent papers: (1) we interleave (inline) offsets in each cache line of the underlying bit vector [31], reducing cache misses, and (2) these offsets are to the *middle* of each block so that only half of each needs popcounting [18]. In QuadRank (14.4% overhead), we extend these techniques to the $\sigma = 4$ (DNA) alphabet.

Both data structures typically require only a single cache miss per query, making them highly suitable for high-throughput and memory-bound settings. To enable efficient batch-processing, we support *prefetching* the cache lines required to answer upcoming queries.

Results. BiRank and QuadRank are around 1.5× and 2× faster than similar-overhead methods that do not use interleaving. Prefetching gives an additional 2× speedup, at which point the dual-channel DDR4 RAM bandwidth becomes a hard limit on the total throughput. With prefetching, both methods outperform all other methods apart from SPIDER [31] by 2×.

When using QuadRank with prefetching in a toy count-only FM-index, QuadFm, this results in a smaller size and up to 4× speedup over Genedex, a state-of-the-art batching FM-index implementation.

Conclusion. Optimizing data structures for high throughput, by minimizing cache misses and branch-misses and adding support for prefetching, can result in significant speedups when benchmarks are adjusted accordingly.

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

Keywords and phrases Rank, Succinct Data Structures, Cache Performance, Prefetching

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.20

Supplementary Material *Software*: `https://github.com/RagnarGrootKoerkamp/quadrank` [20] archived at `swh:1:dir:83a2ce8d57ab4f246e470f1226e19ecafb51fffe`

Acknowledgements I thank Heng Li for motivating me to start this project in an attempt to speed up BWA-MEM. I also thank those who were involved in discussions surrounding this project or gave feedback on the text: Rick Beeloo, Piotr Beling, Felix Leander Droop, Simon Gene Gottlieb, Florian Kurpicz, Rob Patro, Giulio Ermanno Pibiri, and Peter Sanders.

1 Introduction

Given a fixed text $T = t_0 \dots t_{n-1}$ of length n over an alphabet Σ of size σ, a query $\mathsf{rank}(q, c)$ counts the number of occurrences of symbol $c \in \Sigma$ in the first q ($0 \le q \le n$) characters of the text[1]:

[1] Like Rank9 [43] and most (but not all) other implementations, we follow Dijkstra's advice [10] and start numbering at zero.

$$\mathsf{rank}(q, c) := \sum_{i \in \{0,\ldots,q-1\}} [t_i = c].$$

In most literature, the binary alphabet of size $\sigma = 2$ is used, in which case the text is simply a string of n bits. In this case, we also write $\mathsf{rank}(q) := \mathsf{rank}(q, 1)$ to count the number of 1 bits.

Of interest are space-efficient data structures that can answer these queries quickly. Indeed, there exist *succinct* data structures [22] that use $n + o(n)$ bits of space to answer queries on a binary text in $O(1)$ time in the RAM-model with word-size $w = \Theta(\lg n)$. When the bitvector itself is stored explicitly, a tight lower bound on the space usage is $n + \Omega(n \log\log n / \log n)$ bits [33, 16].

A fast and widely used implementation is Rank9 [43], which has a fixed 25% space overhead. Many subsequent works have reduced the space overhead to as little as 1.6%, as detailed in Section 2. In practice, most implementations have fixed overhead, making them *compact* ($n + O(n)$ bits) but not *succinct*.

FM-index. A primary application of Rank queries is in the *FM-index* [13], a succinct data structure that can efficiently locate all occurrences of a pattern in a text and is used in tools such as BWA-MEM [32] and Bowtie [30, 29]. Whereas most of the literature on rank structures assumes a binary alphabet ($\sigma = 2$), in this case the DNA alphabet has size $\sigma = 4$. Indeed, BWA-MEM implements its own rank structure over a 2-bit alphabet[2], and this paper started as an attempt to speed this up.

Wavelet tree. For alphabets of arbitrary size, *wavelet trees* [21] or the *wavelet matrix* [8] can be used for succinct rank queries. They both need $\lg_2 \sigma$ queries to a binary rank structure. Recently, quad wavelet trees [6] have been introduced, following earlier theoretical [14] and practical [3] results on multi-ary wavelet trees. Quad wavelet trees use rank over a *quad vector* as a building block, and thus need only $\log_4 \sigma$ rank queries, leading to 2× to 3× speedups over binary wavelet trees.

Multithreading and batching. In the past, increasing CPU frequencies led to faster code, but nowadays, improvements are mostly in increasing parallelism. Furthermore, total compute of a CPU increases faster than the available memory bandwidth, resulting in the need for *communication-avoiding algorithms* that minimize their memory bandwidth [11]. Indeed, in bioinformatics applications, one often has many independent queries (DNA sequences) that need to be processed (searched in an FM-index) as fast as possible. In particular, this allows using all cores/threads of the CPU as well as processing queries in *batches* inside each thread, to hide the memory latency.

Current benchmarks usually measure the throughput of answering rank queries in a for loop on a single thread, but this does not take into account the possibility for batching, nor does it capture the effects of running many threads in parallel. As we will see, in a high-throughput setting, many existing methods become bottlenecked by the total memory bandwidth of the CPU. We specifically design our data structures to use the memory bandwidth maximally efficient.

[2] `https://github.com/lh3/bwa/blob/master/bwt.c`

Contributions. We develop two data structures, BiRank and QuadRank, that support high-throughput rank queries over texts over alphabets of size 2 and 4.

Both of them integrate a number of existing techniques (see next section), and are *not* designed to support select queries, since these are not needed for the motivating FM-index application, thus allowing for more optimizations. Specifically, BiRank has 3.28% overhead and integrates (1) inlining of counts into the bitvector [31], which reduces cache misses, (2) *pairing* with mask-lookup [18], halving the number of popcounts, and (3) an additional third layer [47] that is modified to be only half its usual size.

QuadRank extends the ideas of BiRank, but has roughly 4× larger space overhead (14.4%) since it stores metadata for each symbol. It combines the cache locality of the implementation in BWA-MEM [32] with the low overhead of quad vectors [5] and a transposed bit layout for faster queries [1, 18]. QuadRank is optimized for returning ranks for all 4 symbols at once by using AVX2 instructions, which is useful for approximate pattern matching in an FM-index.

Both data structures need only a single cache line from RAM to answer queries as long as the input is less than roughly 16 GiB. The main novelty is in combining all these existing ideas and applying them to the size-4 alphabet, while also adding support for *prefetching* of cache lines to enable much more efficient batch-processing of queries. As a side-effect, we also added prefetching to some other libraries, yielding up to 2× speedups.

Results. For both data structures, we implement a number of variants that have different space-time trade-offs. When used in a for loop, BiRank is up to 1.5× faster than the next-fastest rust implementation of equal size, with the speedup being larger when using many threads. Prefetching memory improves the throughput of many libraries by around 1.5×, and improves BiRank by 2×. In this setting, all methods are bottlenecked by the memory throughput, and BiRank is 2× faster than all others because it only needs to read 1 instead of 2 cache lines from RAM. Similarly, QuadRank is at least 1.5× faster than the next-fastest Rust library, QWT [6], and 2× faster after adding prefetch instructions, again being bottlenecked by the RAM throughput.

Inspired by genedex [12], we develop QuadFm, a toy-implementation of a count-only FM-index that uses batching, prefetching, and multithreading. At 14.4% overhead (2.29 bits/bp), our implementation is over 1.5× faster using QuadRank compared to using QWT's quad vector, and at 100% space overhead, QuadFm is 4× faster than genedex, a state-of-the-art FM-index implementation.

2 Background

We briefly go over some previous papers containing rank structures for either $\sigma = 2$ or $\sigma = 4$ in chronological order and list their main technical contributions. Both [47] and [26] contain a nice overview as well. Note that many of these papers develop a rank structure in the context of the larger *rank and select* problem, where there are different design trade-offs. Additionally, work on *compressed* bitvectors is omitted here.

Most data structures are schematically depicted in Figure 1.

Terminology. For later reference, we summarize our terminology. The raw data is split into *superblocks* (the second level, L2) that are further split into *blocks* (L1). *Block* is used for both the raw bits themselves, as well as the cache line containing them. For each superblock, an L2 *offset* is stored, representing the number of 1-bits before the superblock. For each block, an L1 *delta* is stored, typically representing the number of 1-bits preceding it inside

the superblock. Conceptually, these levels form a *summary tree*, named as such by Kurpicz et al. [27]. We follow their notation, and number the levels of the tree *bottom-up*, breaking tradition with e.g. the presentation in [47] and [26]. The root has all superblocks (L2) as children, each superblock has its contained blocks (L1) as children, and each block has the contained bits as children.

The *overhead* of a data structure is the increase in space consumption relative to the size of the input data. We use bp (base pair) as the unit for 2-bit encoded DNA characters, and occasionally use the Rust syntax `u64` for a 64-bit variable and `u64x4` for a 256-bit SIMD register containing 4 64-bit words. A *symbol* is an element of the alphabet Σ, whereas a *character* is an element of a string.

Classic succinct approach. As a baseline, Jacobson [22] stores the bitvector, and additionally two levels of blocks alongside this. *Blocks* consist of $\lfloor \log(n)/2 \rfloor$ bits, and $\lfloor \log n \rfloor$ blocks form a *superblock*. Level L2 of the tree then contains a $\lceil \log n \rceil$ bit *offset* for each superblock, counting the number of set bits preceding it. Level L1 stores for each block a $\lceil \log \log n \rceil$ bit *delta* counting the number of 1-bits preceding it inside its superblock. The number of 1-bits in a (prefix of) a block is obtained via a lookup in a precomputed table of size $2^{(\log n)/2} = \sqrt{n}$.

A practical approach. González et al. [17] observe that the classic method above has 66.85% overhead in practice for $n = 2^{30}$. They replace the large lookup table by a smaller table of per-byte popcounts. (Meanwhile, CPUs natively support 64-bit `popcount` instructions.) They use 256-bit superblocks with a 32-bit offset, containing 8 32-bit blocks, each with their own 8-bit delta. Alternatively, they introduce a *single*-level tree storing a 32-bit L2 offset after every e.g. $4 \cdot 32$ bits and omitting L1. This requires popcounting more words, but has the benefit of improved cache locality compared to a two-level tree.

Rank9: interleaving levels. *Rank9* [43] has 25% overhead and *interleaves* the L2 and L1 levels of the classic tree. Each block is 64 bits, and 8 blocks form a 512-bit superblock, exactly matching a cache line. For each superblock, the interleaved tree stores a 64-bit integer with the offset of the superblock, and 7 9-bit deltas (for all but the first block) in an additional 64-bit word. This needs two cache misses per query (for the L2 array and bits), and is very fast in practice. Specifically it only needs to popcount a single 64-bit word, which is done using *broadword programming* (also known as SWAR, SIMD Within A Register).

Poppy: reducing space. *Poppy* [47] is optimized for space and has only 3.125% overhead. First, it makes the observation that performance is largely determined by the number of cache misses. Thus, it uses larger blocks of 512 bits. It then re-uses Rank9's interleaved index with two modifications. Each superblocks contains 4 blocks, and for each superblock it stores a 32-bit offset (L2) followed by 3 10-bit popcounts of the first 3 blocks. Queries then require a prefix-sum over these counts. To handle 64-bit outputs, it stores an additional layer (L3) of the tree, with a full 64 bit offset after every 2^{32} input bits.

BWA-MEM: DNA alphabet. BWA-MEM [32] implements a 100% overhead rank data structure on $\sigma = 4$ DNA. It interleaves L2 offsets with the data, and requires only a single cache miss per query. In each cache line, it stores 4 64-bit offsets (one for each DNA character), followed by 256 bits encoding 128 bp.

SDSL. The succinct data structure library (SDSL) [15] implements Rank9 and introduces `rank_support_v5`, which has 6.25% overhead. It uses superblocks of 2048 bits. For each, it interleaves a 64-bit offset (L2) and 5 11-bit deltas (packed into 64 bits) to all but the first of 6 blocks covering $6 \cdot 64$ bit. `rank_support_il` interleaves 64-bit offsets with 512-bit blocks.

EPR-dictionaries: arbitrary σ. EPR-dictionaries [38] work for arbitrary alphabet. For $\sigma = 4$, they use 64-bit (32 bp) blocks and have 42% overhead, and interleave an independent 2-level rank structure for each character. Compared to earlier work, space is saved by storing a packed representation of the text instead of σ (1-hot) encoded bitvectors that each indicate which text positions contain each symbol $c \in \Sigma$.

B-trees. Pibiri and Kanda [37] diverge from the classic approach and introduce a rank and select structure based on highly tuned B-trees that have 3.6% overhead. Each rank query traverses roughly $\log_{16} n$ levels of the tree, with the middle levels packing 16 32-bit values in a cache line. Due to efficient caching of the top levels of the tree, performance is similar to poppy, although not as fast as rank9.

AWFM: transposed layout and batching/prefetching. The AWFM-index and its Rust implementation AWRY [1] builds an FM-index on a size $\sigma = 6$ alphabet of 4 DNA characters as well as a sentinel and ambiguity symbol. It uses blocks of 256 3-bit characters, preceded by 5 64-bit offsets that are padded to 512 bits. Each block is encoded using a similar *strided* or *transposed layout*: instead of concatenating the 3 bits of each character, it stores 3 256-bit vectors containing bit 0, bit 1, and bit 2 of each character. This allows for more efficient popcounting. The FM-index processes queries in batches of size 4, and prefetches memory needed for the next rank operation as soon as possible.

Pasta: larger L1 values and faster queries. *PastaFlat* [26, 25] has the same 3.125% space overhead as Poppy, but improves query time by 8% by avoiding Poppy's need to take a prefix sum over L1 counts. Pasta doubles the metadata for each superblock to 128 bits, covering 8 512-bit blocks of 4096 bits in total. It stores a 44-bit offset (L2) followed by 7 12-bit deltas (L1) from the start of the superblock to each block. A second structure, *PastaWide* (3.198% overhead) uses 16-bit values for L1, which allows faster select queries using SIMD instructions. Here, each superblock covers 128 blocks and stores a 64-bit L2 value, this time *not* interleaved with the L1 values, and the L3 level is dropped.

Quad vectors: extending PastaFlat to $\sigma = 4$. *Quad wavelet trees* internally use *quad vectors* [5, 6], which have a layout very similar to PastaFlat. Super blocks cover eight 512 bp blocks and stores 128 bits of data for each of the 4 symbols. This takes 4× more space per character, but since the text doubles in space as well, the overhead only doubles to 6.25%. Alternatively, 256 bp (512 bit) blocks can be used to reduce the number of cache misses, using 12.5% overhead.

SPIDER: interleaving bits for minimal cache misses. *SPIDER* [31] has only 3.3% overhead and reduces the number of cache misses from 2 to (nearly) 1 by interleaving L1 with the bitvector itself (like BWA-MEM), instead of interleaving L1 with L2: each cache line stores a 16-bit L1 delta, and 496 input bits. L2 superblocks store a 64-bit offset for each 128 blocks, taking only 0.1% extra space and thus likely fitting in a cache.

Pairing: halving the overhead. *Pairing* (pfBV) [18] is an idea that halves the memory overhead again, to 1.6%. Compared to PastaWide, instead of storing 16-bit (L1) deltas to the start of *each* 512-bit block, here we store 16-bit deltas to the middle of each *pair* of 512-bit blocks. Then, the second block can add a prefix-popcount to this as usual, while the first block can *subtract* a *suffix*-popcount instead. Similarly, the 64-bit L2 offset is to the *middle* of a twice-as-large superblock. This is similar to the *alternate counters* idea for the FM-index [7], where, for alphabet size 4, each block stores half the offsets. A small complication with this design is that conditionally popcounting a prefix *or* suffix of bits is slightly slower. Instead, Gottlieb and Reinert [18] introduce a lookup table that stores a precomputed mask for each position. Lastly, for $\sigma = 4$, this paper uses the transposed layout of AWFM, but calls it *scattered* instead.

2.1 Further implementations

In anticipation of the evaluations, we list some specific Rust implementations.

QWT. `qwt` (github:rossanoventurini/qwt) implements `RSQVector256` and `RSQVector512` corresponding to the Quad Vectors in the paper [6] with 12.5% and 6.25% overhead. It further contains `RSWide`, which implements the PastaFlat structure [26] (omitting the L3 layer), and `RSNarrow`, which exactly implements Rank9.

Sux. `sux` (github:vigna/sux-rs) [44] contains an implementation of Rank9, as well as five versions of *RankSmall*. These are all variants on Rank9, but use Poppy's 64-bit L3 to allow for 32-bit L2 values. They vary in the number of `u32` used to store the L1 values and the width of the L1 values. A special case is `RankSmall3` (3.125% overhead), which stores 3 11-bit values in a single 32-bit word by using 0-extension for the implicit high 0-bit of the first value.

Bitm. `bitm` (github:beling/bsuccinct-rs) is part of bsuccinct [2]. Its `RankSimple` (6.25% overhead) stores a 32-bit L2 offset for every 512 bit block. `RankSelect101111` (read: 10-11-11) has 3.125% overhead and is the same as `RankSmall3` of sux.

Genedex. `genedex` (github:feldroop/genedex) [12] implements variants of the data structures of [18]. It is designed for $\sigma > 2$, but also supports $\sigma = 2$. `Flat512` stores the text using 4 indicator bitvectors and uses 4 interleaved copies of SPIDER, one for each symbol. `Flat64` is the same but with 64-character blocks. `Condendensed512` implements the flattened bit vectors (fBV) of [18], with blocks representing 512 transposed characters, and using σ interleaved copies of PastaWide.

Further Rust implementations. We did not include the following libraries in the evaluations because they are not (close to) Pareto optimal. `Bio` [28] has a `RankSelect` structure that stores a 64-bit offset after every few 32-bit words, but is not very optimized. `RsDict` [23] implements a compact encoding [36], making it relatively slow. `Sucds` [24] implements Rank9, which is already covered. `Succinct` [42] provides both Rank9 and JacobsonRank, which is both slower and larger. `Vers_vecs` [9] implements PastaWide, but with superblocks spanning 2^{13} rather than 2^{16} bits.

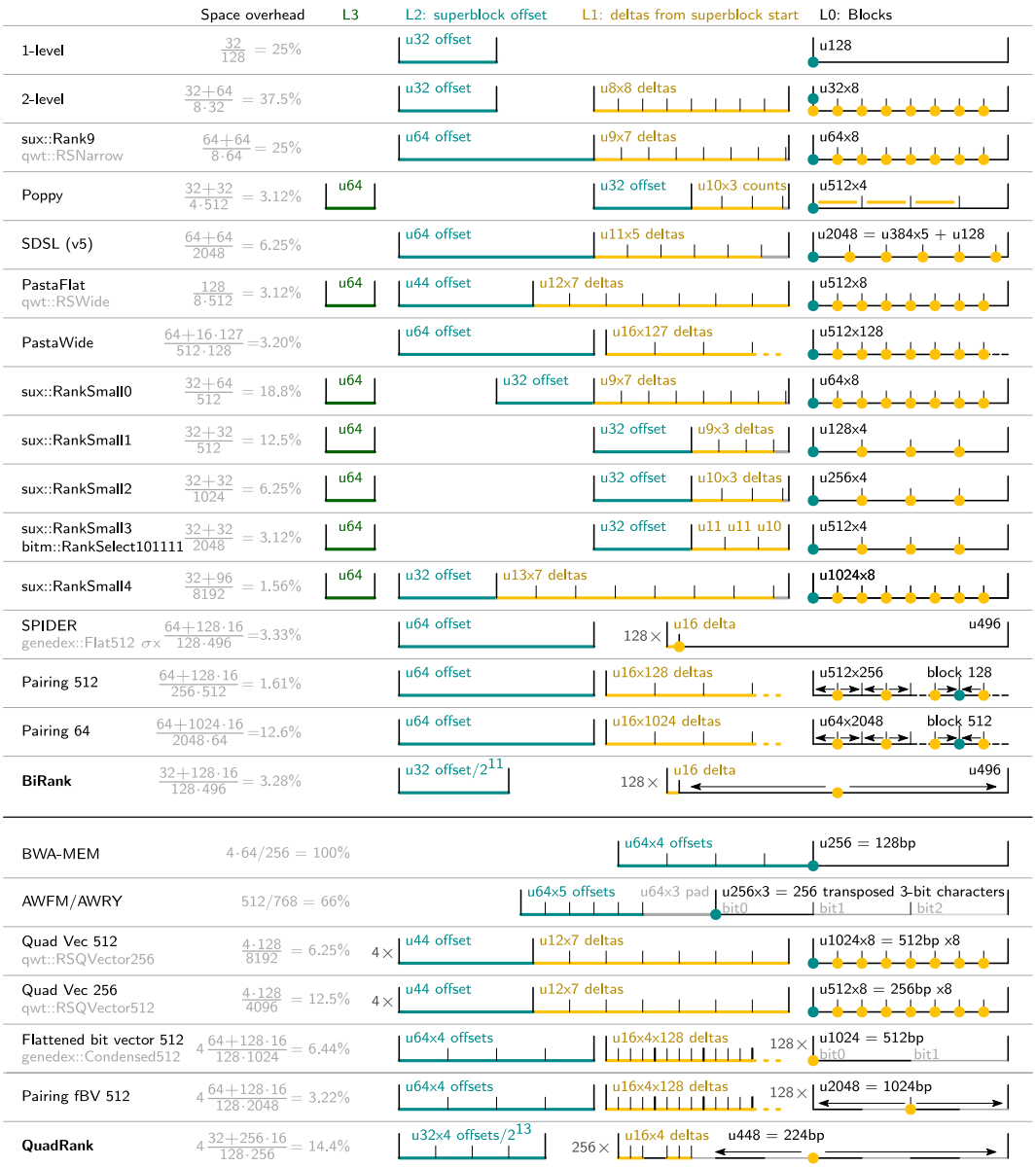

Figure 1 Schematic overview of rank data structures. The top and bottom half are for $\sigma = 2$ and $\sigma = 4$ respectively. Each line shows a data structure (and notable (re)implementations) with its overhead and the layout of a single *superblock* (not to scale). Each structure stores up to 3 vectors containing (interleaved) superblocks offsets, block deltas, and raw bits. On the right (black) are the *blocks* containing (bitpacked) data. Each superblock contains a single L2 *offset* (teal) that is either absolute, or sometimes relative to a 64-bit L3 value (green). They usually count the number of 1-bits/characters before the start of the superblock as indicated by the teal dot, or to the middle of the superblock for pairing variants. L1 *deltas* (yellow) count from the start/middle of the superblock to the start of each block (yellow dots). Only for poppy they count individual blocks (yellow lines). For pairing, pairing fBV, BiRank, and QuadRank, L1 deltas are to the middle of each (pair of) block(s). AWFM, (pairing) fBV, and QuadRank store the text *transposed*, alternating words of low and high bits.

3 BiRank

Let $T = t_0 \dots t_{n-1}$ be a text of n binary characters $\{0, 1\}$. For a query q ($0 \le q \le n$), $\mathsf{rank}(q) = \sum_{i \in \{0,\dots,q-1\}} [t_i = 1] = \sum_{i \in [q]} t_i$ counts the number of 1-bits in the first q characters of the text.

BiRank is a data structure that answers $\mathsf{rank}(q)$ queries in constant time using 3.28% space overhead. It can be constructed in parallel on multiple threads from a slice of already-packed data and provides `rank` and `prefetch` functions. The description below refers to the lines of the simplified Rust code for querying QuadRank in Appendix A.

Single cache-miss queries. We aim to minimize the number of cache misses on large (many GB) inputs, to enable efficient usage in high-throughput settings where the memory bandwidth is the bottleneck. A single cache miss is inevitable, and so we must avoid any further cache misses. This means that any additional data should fit in L3 cache, which is not the case for non-interleaved layouts. For example, pairing has 1.6% overhead, which would only support 1 GiB of input with a 16 MiB L3 cache.

Interleaved L1. Thus, like SPIDER [31], BiRank inlines a 16-bit L1 delta b_j into each block/cache line j (lines 7-8 of Listing 1), so that each of $\lceil n/B \rceil$ blocks covers $B := 512 - 16 = 496$ bits. The $\lceil n/S \rceil$ superblocks cover $S := 128 \cdot B$ bits each, and a second much smaller array stores a 32-bit L2 offset s_i for each superblock i.

Shifted 32-bit L2 offset. Poppy [47] uses an additional 64-bit third level L3, so that 32-bit L2 values are sufficient. Even though this L3 layer is already very small, we remove it completely. Rather than directly encoding $\mathsf{rank}(i \cdot S)$, the rank at the start of a superblock, we store

$$s_i := \lfloor \mathsf{rank}(i \cdot S)/2^{11} \rfloor$$

in the 32-bit L2 value. The remainder $\mathsf{rank}(i \cdot S) \bmod 2^{11}$ will be added to the 16-bit b_j delta for each block in the superblock. This configuration supports inputs up to 2^{43} bits, or 1 TiB, since $n < 2^{43}$ implies $\mathsf{rank}(i \cdot S)/2^{11} < 2^{32}$.

Size of a superblock. Each superblock must contain at most 2^{16} bits, so that the 16-bit b_j can represent their deltas. Thus, we could fit $\lfloor 2^{16}/B \rfloor = 132$ blocks inside each superblock, but we round this down for computational efficiency: $S := 128 \cdot B$.

We also implemented a variant of the pairing of superblocks technique of [18], which doubles the superblock size S and halves the cache usage, see Appendix B. We decided not to use it though: while the reduced cache usage could be beneficial, in practice, the gains are inconsistent and small at best.

Overhead. The overhead of the block deltas is $16/B = 3.226\%$, whereas the superblocks have an overhead of $32/S = 0.05\%$. Thus, the total overhead is 3.28%, and the superblock array fits in a 16 MiB L3 cache for inputs up to 32 GiB.

L1-delta to the middle. To reduce the amount of work needed for popcounting, we apply a variant of the pairing technique [18]: the 16-bit L1 value is not the delta from the start of the superblock to the *start* of the current block ($\mathsf{rank}(j \cdot B)$), but instead to the *middle* of the current block ($\mathsf{rank}(j \cdot B + 240)$, after taking into account a 16-bit padding, line 9):

$$b_j := \mathsf{rank}(j \cdot B + 240) - 2^{11} \cdot s_{\lfloor j/128 \rfloor}.$$

By construction, these values are indeed bounded by $b_j \le S + (2^{11} - 1) < 2^{16}$ (line 8).

Queries. A query for position q first determines the superblock $i_q = \lfloor q/S \rfloor$ and block $j_q = \lfloor q/B \rfloor$. Then, we compute the rank of the middle of the block as $\mathsf{rank}(j \cdot B + 240) = 2^{11} \cdot s_{i_q} + b_{j_q}$. We then make a case distinction on whether q lies left or right of the middle of its block to determine the final value

$$\mathsf{rank}(q) = 2^{11} \cdot s_{i_q} + b_{j_q} + \begin{cases} \sum_{k \in \{j_q B + 240, \dots, q-1\}} t_i & \text{if } q \geq j_q \cdot B + 240, \\ -\sum_{k \in \{q, \dots, j_q B + 240 - 1\}} t_i & \text{if } q < j_q \cdot B + 240. \end{cases} \tag{1}$$

In code, we popcount up to 256 bits (lines 15 and 18): either a suffix of the first half, or a prefix of the second half. The conditional negation (line 20) is optimized to a branchless `cmov` instruction.

Masking. Instead of a for loop over the 64-bit words in the block and bit-shifting, we prefer a branchless technique that always covers the full 256 bits. Uncounted bits are masked out (line 18) via a 256-bit mask that is precomputed (line 11) for each $0 \leq (q \bmod B) < 512$, again following [18]. These are simply stored as a 16 KiB array `[u256; 512]`, which fits in a typical 32 KiB L1 cache.

Prefetching. In order to facilitate efficient batching algorithms (see Appendix D), we provide a `prefetch(q)` function that starts loading the two cache lines containing $s_{\lfloor q/S \rfloor}$ and $b_{\lfloor q/B \rfloor}$ that are needed for $\mathsf{rank}(q)$. For simplicity and reliability, we prefetch into all levels of the cache hierarchy.

Parallel construction. The parallel construction algorithm used for both BiRank and QuadRank builds on the `rayon` crate. First, we count the number of 1-bits in each superblock, which is trivial to parallelize. Then, we take a prefix sum over these counts in a (non-parallel) linear pass, so that we know the number of 1-bits preceding each superblock. This then allows us to fully construct all superblocks (and their contained blocks) in parallel.

3.1 Variants

We consider a few larger but faster variants of BiRank. The ones marked with a * are the best for each overhead and chosen for the evaluations.

1. **BiRank16*** (3.28% overhead) is the original as described above and inlines a 16-bit value in each cache line.
2. **BiRank32** (6.67%) is identical but stores a 32-bit value instead, doubling the overhead. This allows for a much ($\approx 2^{16}\times$) smaller L2 array.
3. **BiRank16x2*** (6.72%) stores *two* 16-bit deltas, to 1/4th and 3/4th into the block. Then, only a quarter of the cache line (2 64-bit words) has to be popcounted.
4. **BiRank23_9** (6.67%) takes a middle ground: it stores a 23-bit L1 delta to 1/4th of the block, and a 9-bit "L0.5" delta (≤ 256) from there to 3/4th.
5. **BiRank64** (14.3%) directly stores a 64-bit value instead, completely removing the need for a separate L2 level.
6. **BiRank32x2*** (14.3%) doubles the overhead again and stores two 32-bit L1 values, shrinking the L2 array.
7. **BiRank64x2*** (33.3%) *again* doubles the overhead, and completely removes the L2 level.
8. **BiRankR9** (33.3%) is an inline version of Rank9: it inlines a 64-bit L2 offset, followed by a 64-bit word containing 6 9-bit deltas to the start of each remaining 64-bit word.

4 QuadRank

QuadRank is the extension of BiRank to the 2-bit DNA alphabet. It can be constructed in parallel on multiple threads from bitpacked data. Rank queries can be either for a specific symbol ($\mathsf{rank}(q, c)$), or for all 4 symbols at once ($\mathsf{rank}_4(q)$). We do not provide a dedicated function to count a range, as is commonly used for the FM-index, because the associated branch-misses would hurt performance, and cache lines are automatically reused anyway.

As with BiRank, QuadRank is optimized for having as few cache misses as possible. In particular, the data-layout is nearly the same, but with the L2 and L1 data replicated for each symbol: each cache line contains 4 16-bit deltas $b_{j,c}$ and $B_4 = (512 - 4 \cdot 16)/2 = 224$ characters. Superblocks cover 256 blocks ($S_4 := 256 \cdot B < 2^{16}$), and for each we store 4 shifted 32-bit offsets $s_{i,c} := \lfloor \mathsf{rank}(i \cdot S_4, c)/2^{13} \rfloor$. We now divide by 2^{13}, since $S + (2^{13} - 1) < 2^{16}$, which allows inputs up to 2^{45} characters or 8 TiB. The overhead over the b_i deltas is $4 \cdot 16/(2B) = 14.29\%$ and the overhead of the offsets is $(4 \cdot 32)/(2S) = 0.11\%$, for 14.40% overhead in total. A 16 MiB L3 cache can support over 14 GiB (61 Gbp) of input.

To compute the rank of all 4 symbols at once, relatively more time is spent on popcounting than in the binary case. Thus, we detail our optimizations to compute all 4 ranks efficiently.

Transposed layout. Compared to the layout for binary input, the main difference is that we now store the input data in *transposed* (or *strided*) layout [1, 18] (as opposed to *packed*). Ignoring the inlined $b_{j,c}$ deltas for the moment, the 256 characters in a block are split into 4 groups of 64. Each group of 64 characters is encoded as two 64-bit values, one consisting of the *negation* of all low bits, and one of the *negation* all high bits. The 4 16-bit deltas replace the 64 bits corresponding to the 32 first characters (Listing 1, line 8), as shown in the bottom row of Figure 1. The positions matching a symbol are then found by and-ing the two values together (line 18), after possibly negating one or both (lines 13-17). This layout makes more efficient use of popcount instructions, since each counted word now contains up to 64 1 bits, compared to 32 with the packed layout.

rank1. Computing the rank for a single character is similar to before (1): we retrieve the superblock offset s_{i_q}, multiply it by 2^{13}, and then add the $b_{j_q,c}$ for the current block and character. Lastly, we add or remove the count for up to 128 characters in either the first or second half of the cache line, processed in two chunks of 64 characters.

4-way popcount. To return the rank of all 4 symbols, we essentially do the above method 4 times in parallel in `u64x4` 256-bit AVX2 SIMD registers. In particular, we use a single SIMD lane for each symbol (Listing 1). To popcount the number of 1-bits in each lane, we use Mula's algorithm [34, 35]. Essentially, this splits each byte into two 4-bit nibbles and for each does a `_mm256_shuffle_epi8` instruction to do a 16-way lookup returning the precomputed number of 1 bits in each nibble. It then adds these two values, resulting in per-byte popcounts, and finally uses the `_mm256_sad_epu8` instruction to take a horizontal sum of the 8 bytes in each 64-bit lane. We convert the counts to `u32x4` and then conditionally negate them using `_mm_sign_epi32(counts, u32x4::splat(pos-96))`, which multiplies each lane by the sign of (q mod B) − 96 (i.e., -1, 0, or 1).

4.1 Variants

Again, we consider a number of slightly faster variants that use larger inline values. Since returning all 4 counts takes more compute, we specifically focus on methods that reduce the amount of characters to be counted from 128 to 64. There is more variation here than in

the binary case: we can use packed (P) or transposed layout (T), and we can avoid using the pairing technique (bidirectional (B) vs forward (F)) to save the small CPU overhead for negating values. This is just a small selection of possibilities, and not all implementations were equally optimized. Those marked with a * are the fastest for each overhead and have been chosen for the evaluations.

- **QuadRank16*** (TB, 14.40% overhead) is as described above and inlines 4 16-bit values containing the rank to the middle of each block.
- **QuadRank32** (TB, 33%) instead uses 4 32-bit values, making the L2 array much smaller.
- **QuadRank24_8*** (TB, 33%) leaves space for 3 groups of 64 characters and splits this into 3 sub-blocks, storing an L1 delta to the end of the first and third group. This way, only a 64-character popcount remains.
- **QuadRank7_18_7** (PB, 33%) uses a normal packed layout. It stores an 18-bit L1 to the middle of 6 32-character blocks, and two 7-bit "L0.5" deltas to 1/6th and 5/6th.
- **QuadRank64*** (TB, 100%) stores 4 64-bit values, as does BWA-MEM, removing the L2 array. This only leaves space for 128 characters, so each half is now only 64 of them.
- **QuadRank32_8x4** (PF, 100%) uses packed layout. It stores a 32-bit L1 delta to the start of the block, and 4 8-bit "L0.5" deltas to each 32-character sub-block.
- **QuadRank32x2** (PF, 100%) stores 2 32-bit L1 deltas to the start and halfway point, and does a forward scan.

5 Results

Both our implementation of BiRank and QuadRank and the evaluations can be found at github:ragnargrootkoerkamp/quadrank. All experiments are run on an AVX2 Intel Core i7-10750H Skylake CPU with 6 cores and hyper-threading enabled. The frequency is pinned to 3.0GHz. Cache sizes are 32 KiB L1 and 256 KiB L2 per core, and 12 MiB shared L3 cache. Main memory is 64 GiB as dual-channel 32 GiB 3200MHz DDR4 sticks, with the memory controller running at 2933 MHz.

Benchmarks on a 92-core AMD Zen 4 EPYC with 12 DDR5 memory channels can be found in the appendix and give mostly similar results (Appendix C.3). The appendix also contains additional plots analysing the CPU time for very small inputs (Appendix C.2), as well as statistics on the number of measured last-level cache misses per query (Appendix C.1).

We only compare Rust implementations, since our aim is to provide a ready-to-use Rust library as well. Furthermore, cross-language function calls would likely prevent the compiler from optimizing all code equally, and re-implementing comparable benchmarks in C++ and getting all libraries to work was deemed infeasible.

5.1 BiRank

We compare BiRank and its variants against the Rust crates mentioned in Section 2.1. In order to make the evaluations with prefetching fair, we have created PRs adding support for this to each of them.[3]

[3] `github:vigna/sux-rs/pull/98`, `github:rossanoventurini/qwt/pull/6`, `github:feldroop/genedex/pull/4`, `github:beling/bsuccinct-rs/pull/14`.

Benchmark setup. For each run, we first build each data structure on a random 4 GiB input (in parallel, if possible) and generate 10 million random query positions. Then we run three types of benchmarks. In the first, we measure the average **latency** of sequential queries, by making each query dependent on the result of the previous one. In the second, we measure the *inverse throughput* (i.e., amortized time per query, which we will just call throughput) when processing random queries in a **for loop**: `for i in 0..Q { BiRank::rank(queries[i]) }`. We stress here that CPUs can use pipelining and out-of-order execution to execute multiple (up to at least 4) iterations of the loop in parallel. Thus, we add a third mode where we explicitly process many items at once and we add **prefetching**, where we prefetch the required cache lines 32 iterations ahead: `for i in 0..Q { BiRank::prefetch(queries[i+32]); BiRank::rank(queries[i]); }`[4] We then repeat these three benchmarks when running in parallel on 1, 6, and 12 threads, where each thread has its own independent set of 10 M queries. Each reported measurement is the median of 3 runs.

Excluded libraries. SPIDER (github:williams-cs/spider) [31] was not yet implemented in Rust, so we made a variant of BiRank that approximately uses SPIDER's linear-scan for popcounting inside a block. Unfortunately, we were unable to compare the performance against the original C implementation. Pairing (github:seqan/pfBitvectors) [18] also has only been implemented in C++. Nevertheless, genedex was reported to be faster (personal communication). Lastly, we exclude the dynamic B-tree (github:jermp/mutable_rank_select) of [37], but consider a Rust reimplementation of this work a promising direction for future work on select specifically.

Lower bounds. The results are in Figure 2. There are different lower-bounds on the throughput: the measured latency of the RAM is around 80 ns/read, which gives a lower-bound of $80/t$ ns/query for t threads (dotted red lines). When using only a single thread, we are further bound by its maximum random access throughput of around 7.5 ns per cache line (dashed red lines). In all other cases, we are limited by the 2.5 ns/cache line throughput of the RAM (solid red lines).

Analysis. The first observation is that the latency of all methods is similar, as this is always limited by the memory latency. Furthermore, processing queries in a loop is around 4× faster with 1 thread, and up to 8× with added prefetching. Using 12 threads halves the gap, but nevertheless, processing multiple independent queries in each thread should be preferred to exploit instruction-level parallelism.

Looking at the middle column, we see that BiRank is just slightly better than other methods when using a single thread. This grows to 1.4× speedup when using many threads, where it likely benefits from the reduced memory bandwidth. We see that BiRank16 (the default) is smaller but slightly slower than the larger BiRank variants. BiRank16x2 has double the overhead and is slightly faster, while the variants with larger overhead (that shrink/remove the array of superblock offsets) only provide very minimal gains.

After adding prefetching (right column), we see that all methods improve compared to the shaded data points without prefetching, most somewhere around 1.5×. Whereas non-interleaving methods are limited to around 16 ns/query, BiRank is able to reach the hard limit of 8 ns/query that each thread needs per cache line. Here, the larger variants benefit from requiring a bit less compute compared to the smaller variants. When using

[4] In practice, we must also prefetch the upcoming query values themselves. We keep the memory system so busy that it does not have time to do this by itself, leading to cache misses on the query values themselves if we do not prefetch them.

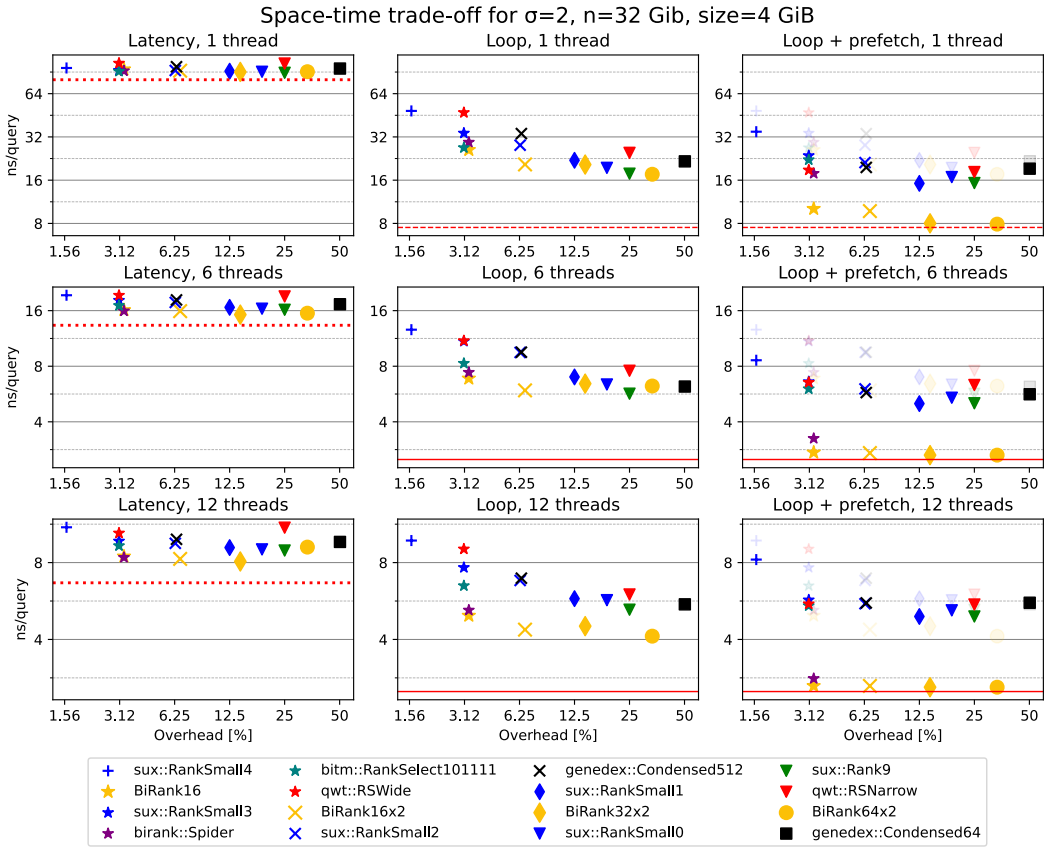

Figure 2 Log-log space-time trade-offs for rank structures on binary input of total size 4 GB. The top/middle/bottom row show results for 1/6/12 threads on a CPU with 6 cores. The left/middle/right column show results for the latency, the throughput of a for loop, and the throughput of a for loop with prefetching. Red lines indicate: (left) the roughly 80 ns RAM latency divided by the number of threads, (top mid/right) the 7.5 ns/read maximum random-access RAM throughput of 1 thread, and (rest) the 2.5 ns/cache line total random-access RAM throughput. In the right column, the transparent markers repeat the for-loop throughput. The legend is sorted by increasing overhead.

prefetching from multiple threads, the situation is the same: BiRank can fully exhaust the RAM random-access throughput, even in its smallest configuration, whereas other methods are 2× as slow. We also see that prefetching speeds up BiRank around 2× compared to just a plain for loop. A special case here is SPIDER, which is also interleaved. With a single thread, the branch-misses of popcounting in a for loop hurt its performance, but it becomes as fast as BiRank and memory bound when multithreading.

5.2 QuadRank

We now run the same set of experiments to compare QuadRank against QWT and genedex on size-4 alphabets. On additional feature is that we compare both $\mathsf{rank}(q, c)$ and $\mathsf{rank}_4(q)$. For the other libraries, rank_4 is implemented naively by simple calling rank four times, whereas QuadRank is primarily optimized for this case.

Overall, the situation here is similar to the binary case. In most settings (single or multithreaded, in a for-loop or with prefetching), the default 14.4% overhead version of QuadRank is around 1.4× faster than the 12.5% overhead version of QWT, and 2× faster

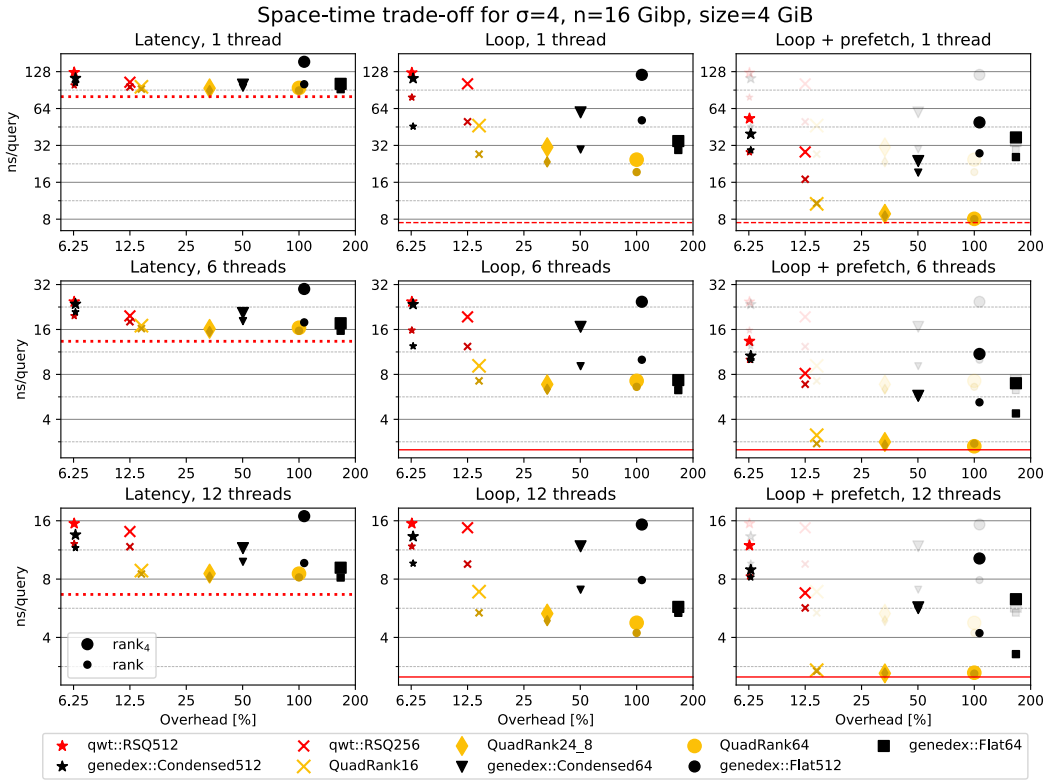

Figure 3 Space-time trade-off of rank structures on size 4 alphabet on 4 GiB input. Compared to Figure 2, here we benchmark both $\mathsf{rank}(q, c)$ (small markers), and rank_4 (large markers).

for rank_4. Also at large overhead, QuadRank is faster than all genedex variants. In high-throughput settings, QuadRank can again saturate the memory bandwidth and answer one query per cache line, being 2× faster than all other methods below 100% overhead. With prefetching, QuadRank shows little to no overhead for computing rank_4: Even with the additional SIMD operations to compute all ranks, it is still memory bound.

6 Conclusion

We surveyed a large number of existing rank structures and, inspired by them, developed BiRank and QuadRank. Their main novelty is in bringing together many independent parts, and applying them to size-4 alphabets. We benchmarked them in a high-throughput setting, with many threads and batching of multiple queries inside each thread.

For binary input, the previous best data structure is SPIDER [31]. BiRank is usually slightly faster than our reimplementation of SPIDER. When multithreading, both are around 1.5× faster than other methods. Additional prefetching improves all methods, and doubles the throughput of BiRank, making it 2× faster than all other methods with prefetching. BiRank benefits from single cache-miss queries, compared to two for most other methods, and thus makes optimal use of the limited memory bandwidth. For QuadRank, the improvement over existing methods is already 1.5× without multithreading, and again 2× with prefetching.

Using batching and prefetching with our memory-bandwidth-frugal methods allows up to 3× higher throughput than sequential processing, and this increases to 6× speedup on a server with many more cores but higher memory latency.

These results also translate to up to 4× speedups over the state-of-the-art when used in an FM-index, and we hope that multithreading, batching, and prefetching become standard in both applications and benchmarks.

In general, we observe that designing data structures for high-throughput, by minimizing cache misses and branch-misses and adding support for prefetching, can give big gains when benchmarks are adjusted accordingly.

Future work. Future work remains in generalizing and optimizing the library for other platforms than AVX2. As the current code was optimized for Skylake (2015), it is likely that more modern platforms (Golden Cove, 2021 or zen 5, 2024) admit different trade-offs. For AVX512, there are dedicated popcount instructions that could be used, while ARM NEON only supports 128-bit instructions and will need further work.

Additional features could be in-place parallel construction and a $\mathsf{prefix\_rank}(q, c)$ operation that counts the number of occurrences of characters *at most* c to support the bidirectional FM-index. Lastly, our FM-index could be extended with support for locate queries.

References

1 Tim Anderson and Travis J. Wheeler. An optimized fm-index library for nucleotide and amino acid search. *Algorithms for Molecular Biology*, 16(1), December 2021. doi:10.1186/s13015-021-00204-6.

2 Piotr Beling. Bsuccinct: Rust libraries and programs focused on succinct data structures. *SoftwareX*, 26:101681, May 2024. doi:10.1016/j.softx.2024.101681.

3 Alex Bowe. Multiary Wavelet Trees in Practice. Master's thesis, School of Computer Science and Information Technology RMIT University, Melbourne, Australia, 2010. URL: https://raw.githubusercontent.com/alexbowe/wavelet-paper/thesis/thesis.pdf.

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

5 Matteo Ceregini, Florian Kurpicz, and Rossano Venturini. Faster wavelet trees with quad vectors. *arXiv*, 2023. doi:10.48550/arXiv.2302.09239.

6 Matteo Ceregini, Florian Kurpicz, and Rossano Venturini. Faster wavelet tree queries. In *2024 Data Compression Conference (DCC)*. IEEE, March 2024. doi:10.1109/dcc58796.2024.00030.

7 Alejandro Chacon, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Boosting the fm-index on the gpu: Effective techniques to mitigate random memory access. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(5):1048–1059, September 2015. doi:10.1109/tcbb.2014.2377716.

8 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, January 2015. doi:10.1016/j.is.2014.06.002.

9 Cydhra. Vers - very efficient rank and select (vers_vecs). github.com/Cydhra/vers, 2023.

10 Edsger W. Dijkstra. Why numbering should start at zero. https://www.cs.utexas.edu/~EWD/transcriptions/EWD08xx/EWD831.html, 1982.

11 Jack J. Dongarra. The evolution of mathematical software. *Communications of the ACM*, 65(12):66–72, November 2022. doi:10.1145/3554977.

12 Felix Leander Droop. Genedex: a small and fast FM-index for Rust. https://github.com/feldroop/genedex, 2025.

13 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, SFCS-00, pages 390–398. IEEE Comput. Soc, 2000. doi:10.1109/sfcs.2000.892127.

14 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, May 2007. doi:10.1145/1240233.1240243.

15 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. *From Theory to Practice: Plug and Play with Succinct Data Structures*, pages 326–337. Springer International Publishing, 2014. doi:10.1007/978-3-319-07959-2_28.

16 Alexander Golynski. Optimal lower bounds for rank and select indexes. In *Automata, Languages and Programming*, pages 370–381. Springer Berlin Heidelberg, 2006. doi:10.1007/11786986_33.

17 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings of WEA 2005*, 2005. WEA.

18 Simon Gene Gottlieb and Knut Reinert. Engineering rank queries on bit vectors and strings. *Algorithms for Molecular Biology*, 20(1), December 2025. doi:10.1186/s13015-025-00291-9.

19 Simon Gene Gottlieb and Knut Reinert. Search schemes for approximate string matching. *NAR Genomics and Bioinformatics*, 7(1), January 2025. doi:10.1093/nargab/lqaf025.

20 Ragnar Groot Koerkamp. RagnarGrootKoerkamp/quadrank. Software, swhId: swh:1:dir:83a2ce8d57ab4f246e470f1226e19ecafb51fffe (visited on 2026-04-28). URL: https://github.com/RagnarGrootKoerkamp/quadrank, doi:10.4230/artifacts.25696.

21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

22 Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988.

23 Sujay Jayaker. RsDict: Fast rank/select over bitmaps. https://github.com/sujayakar/rsdict, 2020.

24 Shunsuke Kanda. Succinct data structures in rust (sucds). https://github.com/kampersanda/sucds, 2021.

25 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. *arXiv*, 2022. doi:10.48550/arXiv.2206.01149.

26 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE 2022*, pages 257–272. Springer International Publishing, 2022. doi:10.1007/978-3-031-20643-6_19.

27 Florian Kurpicz, Niccolò Rigi-Luperti, and Peter Sanders. Theory meets practice for bit vectors supporting rank and select. *arXiv*, 2025. doi:10.48550/arXiv.2509.17819.

28 Johannes Köster. Rust-bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, October 2015. doi:10.1093/bioinformatics/btv573.

29 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, March 2012. doi:10.1038/nmeth.1923.

30 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3), March 2009. doi:10.1186/gb-2009-10-3-r25.

31 Matthew D. Laws, Jocelyn Bliven, Kit Conklin, Elyes Laalai, Samuel McCauley, and Zach S. Sturdevant. Spider: Improved succinct rank and select performance. In *SEA 2024*, volume 301, pages 21:1–21:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SEA.2024.21.

32 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv*, 2013. doi:10.48550/arXiv.1303.3997.

33 Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *SODA '05*, pages 11–12. Society for Industrial and Applied Mathematics, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070435.

34 Wojciech Muła. SSSE3: fast popcount. `http://0x80.pl/notesen/2008-05-24-sse-popcount.html`, May 2008.

35 Wojciech Muła, Nathan Kurz, and Daniel Lemire. Faster population counts using avx2 instructions. *The Computer Journal*, 61(1):111–120, May 2017. `doi:10.1093/comjnl/bxx046`.

36 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *SEA 2012*, pages 295–306. Springer Berlin Heidelberg, 2012. `doi:10.1007/978-3-642-30850-5_26`.

37 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Information Systems*, 99:101756, July 2021. `doi:10.1016/j.is.2021.101756`.

38 Christopher Pockrandt, Marcel Ehrhardt, and Knut Reinert. EPR-Dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional FM indices. In *RECOMB 2017*, pages 190–206. Springer International Publishing, 2017. `doi:10.1007/978-3-319-56970-3_12`.

39 Luca Renders, Lore Depuydt, and Jan Fostier. Approximate pattern matching using search schemes and in-text verification. In *Bioinformatics and Biomedical Engineering*, pages 419–435. Springer International Publishing, 2022. `doi:10.1007/978-3-031-07802-6_36`.

40 Luca Renders, Lore Depuydt, Travis Gagie, and Jan Fostier. Columba: fast approximate pattern matching with optimized search schemes. *Bioinformatics*, 41(12), December 2025. `doi:10.1093/bioinformatics/btaf652`.

41 Arang Rhie, Sergey Nurk, Monika Cechova, Savannah J. Hoyt, Dylan J. Taylor, Nicolas Altemose, Paul W. Hook, Sergey Koren, Mikko Rautiainen, Ivan A. Alexandrov, Jamie Allen, Mobin Asri, Andrey V. Bzikadze, Nae-Chyun Chen, Chen-Shan Chin, Mark Diekhans, Paul Flicek, Giulio Formenti, Arkarachai Fungtammasan, Carlos Garcia Giron, Erik Garrison, Ariel Gershman, Jennifer L. Gerton, Patrick G. S. Grady, Andrea Guarracino, Leanne Haggerty, Reza Halabian, Nancy F. Hansen, Robert Harris, Gabrielle A. Hartley, William T. Harvey, Marina Haukness, Jakob Heinz, Thibaut Hourlier, Robert M. Hubley, Sarah E. Hunt, Stephen Hwang, Miten Jain, Rupesh K. Kesharwani, Alexandra P. Lewis, Heng Li, Glennis A. Logsdon, Julian K. Lucas, Wojciech Makalowski, Christopher Markovic, Fergal J. Martin, Ann M. Mc Cartney, Rajiv C. McCoy, Jennifer McDaniel, Brandy M. McNulty, Paul Medvedev, Alla Mikheenko, Katherine M. Munson, Terence D. Murphy, Hugh E. Olsen, Nathan D. Olson, Luis F. Paulin, David Porubsky, Tamara Potapova, Fedor Ryabov, Steven L. Salzberg, Michael E. G. Sauria, Fritz J. Sedlazeck, Kishwar Shafin, Valery A. Shepelev, Alaina Shumate, Jessica M. Storer, Likhitha Surapaneni, Angela M. Taravella Oill, Françoise Thibaud-Nissen, Winston Timp, Marta Tomaszkiewicz, Mitchell R. Vollger, Brian P. Walenz, Allison C. Watwood, Matthias H. Weissensteiner, Aaron M. Wenger, Melissa A. Wilson, Samantha Zarate, Yiming Zhu, Justin M. Zook, Evan E. Eichler, Rachel J. O'Neill, Michael C. Schatz, Karen H. Miga, Kateryna D. Makova, and Adam M. Phillippy. The complete sequence of a human y chromosome. *Nature*, 621(7978):344–354, August 2023. `doi:10.1038/s41586-023-06457-y`.

42 Jesse A. Tov. Succinct data structures for rust (succinct). `https://github.com/tov/succinct-rs`, 2016.

43 Sebastiano Vigna. Broadword implementation of rank/select queries. In Catherine C. McGeoch, editor, *WEA 2008*, pages 154–168, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. `doi:10.1007/978-3-540-68552-4_12`.

44 Sebastiano Vigna and Tommaso Fontana. Sux. `https://github.com/vigna/sux-rs`, 2024.

45 Mohsen Zakeri, Nathaniel K. Brown, Omar Y. Ahmed, Travis Gagie, and Ben Langmead. Movi: A fast and cache-efficient full-text pangenome index. *iScience*, 27(12):111464, December 2024. `doi:10.1016/j.isci.2024.111464`.

46 Mohsen Zakeri, Nathaniel K. Brown, Travis Gagie, and Ben Langmead. Movi 2: Fast and space-efficient queries on pangenomes. *bioRxiv*, October 2025. `doi:10.1101/2025.10.16.682873`.

47 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA 2013*, pages 151–163. Springer Berlin Heidelberg, 2013. `doi:10.1007/978-3-642-38527-8_15`.

A Code snippets

```rust
type Block = [[u64; 4]; 2];         // 2 halves of 4 64-bit values.
// Little-endian mask values:
// pos   0: 111...111, pos   1: 011...111, ..., pos 127: 000...001
// pos 128: 000...000, pos 129: 100...000, ..., pos 255: 111...110
static MASKS: [[u64; 2]; 256] = ...;
fn rank1(block: &Block, mut q: u64, c: u8) -> u64 {
  let block_u16: &[u16; 32] = transmute(&block); // Cast block to u16's.
  let delta = block_u16[c + (c&2)] as u64;  // Jump over transposed data.
  q += 32;                          // Adjust for skipping 32 characters.
  let half = block[q/128];          // Read the low or high half of bits.
  let masks: [u64; 2] = MASKS[q];   // Read bitmask for the relevant bits.
  let mut popcount = 0;
  let cl = -(c as u64 & 1);         // Cast 0 or 1 to 0 or u64::MAX.
  let ch = -(c as u64 >> 1);
  for i in 0..2 {
    let l = half[2*i  ] ^ cl;
    let h = half[2*i+1] ^ ch;       // l&h indicates occurrences of c.
    popcount += (l & h & masks[i]).count_ones();
  }                                 // The if is optimized into a cmov.
  if q < 128 { delta - popcount } else { delta + popcount }
}
```

Listing 1 Simplified code for computing the character count of a prefix of a block in QuadRank16.

B Pairing superblocks

Here we give a variant on the idea of pairing superblocks [18]. Just like we store block offsets b_j to the middle of a block and then branch on adding or removing to/from that, we can also let the superblock offsets s_i be to the middle of a double-sized superblock. Let $S' = 2S = 256 \cdot B$. We now store $\lfloor n/S' \rfloor$ 32-bit superblock offsets $s'_i := \lfloor \mathsf{rank}(i \cdot S' + S'/2)/2^{11} \rfloor$. Given a block j in superblock $i_j := \lfloor j/256 \rfloor$, its delta is incremented by S' when it is in the lower half of the superblock:

$$b'_j := \mathsf{rank}(j \cdot B + 240) - 2^{11} \cdot s'_{i_j} + \begin{cases} 0 & \text{if } j \geq 256 \cdot i_j + S'/2 \\ (S'/2 - 240) & \text{if } j < 256 \cdot i_j + S'/2 \end{cases}.$$

For blocks in the upper half of a superblock this is $< S'/2$ as before. In the left half, the uncorrected value is the negation of the number of 1 bits between the block middle $j \cdot B + 240$ and superblock middle $i_j \cdot S' + S'/2$, which is between 0 and $S'/2 - 240$. After the correction, we obtain a non-negative value $< S'/2$ again. Queries are modified accordingly to remove this extra term again:

$$\mathsf{rank}(q) = 2^{11} \cdot s'_{i_q} + b'_{j_q} - \begin{cases} 0 & \text{if } q \geq S' \cdot i_q + S'/2 \\ S'/2 & \text{if } q < S' \cdot i_q + S'/2 \end{cases} + \begin{cases} \sum_{k \in \{j_q B + 240, \ldots, q-1\}} t_i & \text{if } q \geq j_q \cdot B + 240 \\ -\sum_{k \in \{q, \ldots, j_q B + 240 - 1\}} t_i & \text{if } q < j_q \cdot B + 240 \end{cases}.$$

C Additional results

C.1 Cache misses per query

Table 1 and Table 2 contain experimental measurements of the number of last-level cache misses each method has on an input of size 4GiB, that is, the average number of cache lines fetched from RAM per query. As expected, BiRank and Quadrank have 1 cache miss per query, while nearly all other methods require at least 2 cache misses.

Table 1 Average number of last-level cache misses per query on an input of 4 GiB, $\sigma = 2$.

Ranker	Space overhead (%)	Cache misses per rank
sux::Rank9	25.0	2.05
sux::RankSmall0	18.8	2.17
sux::RankSmall1	12.5	2.03
sux::RankSmall2	6.3	2.25
sux::RankSmall3	3.1	2.11
sux::RankSmall4	1.6	2.31
qwt::RSNarrow	25.0	2.05
qwt::RSWide	3.1	1.99
genedex::Condensed64	50.2	3.73
genedex::Condensed512	6.4	3.27
bitm::RankSelect101111	3.1	2.25
birank::Spider	3.3	**1.02**
BiRank64x2	33.3	**1.01**
BiRank32x2	14.3	**1.01**
BiRank16x2	6.7	**1.02**
BiRank16	3.3	**1.02**

Table 2 Average number of last-level cache misses per query on an input of 4 GiB, $\sigma = 4$.

Ranker	Space overhead (%)	Cache misses per rank	per rank_4
genedex::Flat64	166.9	**1.38**	2.02
genedex::Flat512	106.6	**1.35**	2.32
genedex::Condensed64	50.2	2.46	2.58
genedex::Condensed512	6.4	3.34	3.41
qwt::RSQ256	12.5	2.01	2.00
qwt::RSQ512	6.3	2.22	2.35
QuadRank64	100.0	**1.01**	**1.01**
QuadRank24_8	33.3	**1.01**	**1.00**
QuadRank16	14.4	**1.05**	**1.05**

C.2 Throughput for small inputs

In Figure 4 and Figure 5 we benchmark on small 128 KiB inputs that fit comfortably in the L2 cache. This way, experiments will be mostly CPU-bound, and we get an idea of the maximum performance of each method and their relative computational cost.

As expected, we see a space-time tradeoff, with methods that need more space typically being faster. Rank9 (25% overhead) is the fastest, while RankSmall0 is small but slow. The BiRank variants are all roughly equally as fast, with the BiRank16 variant (3.28% overhead) being slightly slower, but still faster than other methods. The server (see next section) is up to 2× faster, and surprisingly, even the smallest BiRank variant is very fast.

For QuadRank, we see that the laptop is faster with rank than rank_4 queries, while, again surprisingly, the server is faster for rank_4 queries.

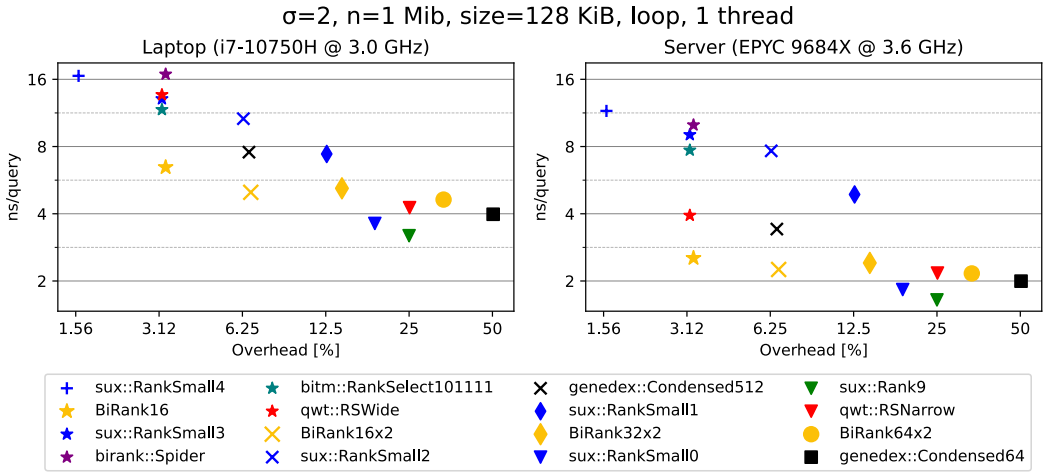

Figure 4 Space-time trade-off plot of the inverse throughput of rank queries in a for loop on a small 128 KiB binary input that fits in L2 cache.

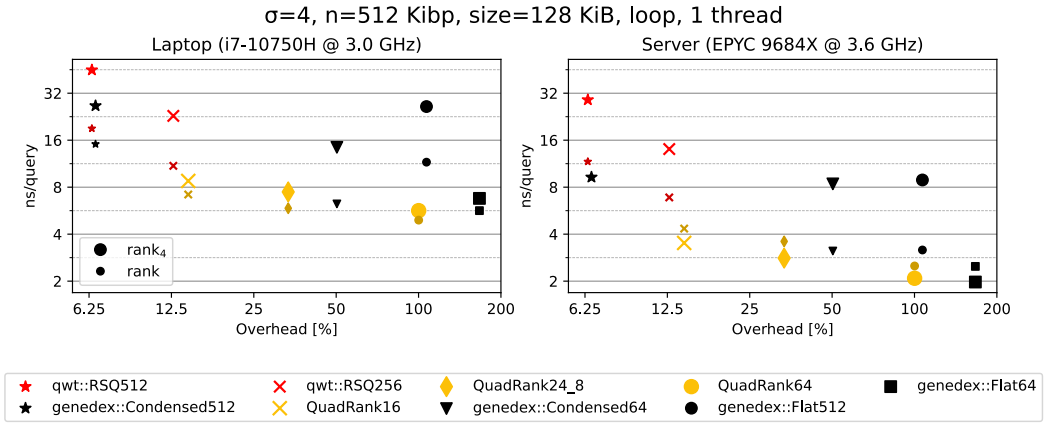

Figure 5 Space-time trade-off plot of the inverse throughput of rank queries in a for loop on a small 128 KiB input over alphabet size 4 that fits in L2 cache. Small markers indicate the time for a rank query that counts only one symbol, while large markers always return all four ranks for rank_4.

C.3 AMD EPYC evals

We replicate the experiments on a large AMD Zen 4 EPYC 9684X server with 96 cores, 192 threads, 96 MiB of L3 cache for each 8 cores, and 12-channel DDR5 RAM. It has a base clock frequency of 2.55 GHz, but during our experiments, it ran consistently at 3.7 GHz with 1 thread, at 3.4 GHz when using all 192 threads, and at 3.0 GHz when using 192 threads with batch processing. We run experiments using 1, 48, and 96 threads. Results for 192 threads are nearly identical to using 96 threads.

Figure 6 and Figure 7 show the results, which follow a similar trend as the earlier results: BiRank and QuadRank are consistently faster, with the improvement of BiRank becoming more pronounced as we add more threads. With prefetching, BiRank saturates the memory bandwidth when using 92 threads, and is then 2× faster than nearly all other methods. One notable difference with the laptop benchmarks is that here, SPIDER is as fast as BiRank when using prefetching, suggesting that on zen 4, the associated branch misses do not cost nearly as much. A further difference is that benefit of independent queries over sequential queries is up to 4× here, compared to at most 2× on the laptop. Likely, this is due to the server having a 60% higher RAM latency (130 ns vs 80 ns), as well as it having a larger reorder window that can process more loop iterations in parallel. At the same time, this reduces the impact of prefetching from 2× to 1.5×.

For $\sigma = 4$, we see that larger data structures are often *slower*. Likely, this is because the 1.1GiB L3 cache can hold a larger fraction of the data when the overhead is small.

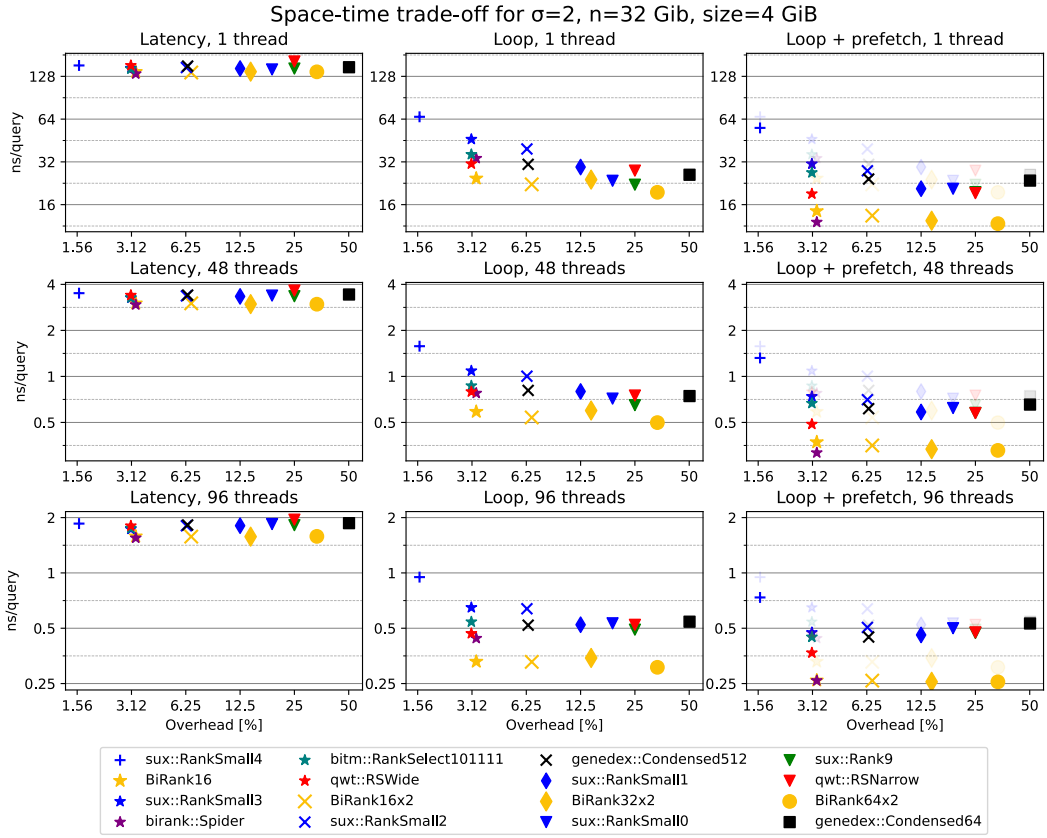

Figure 6 Scaling with input size for size 2 alphabet.

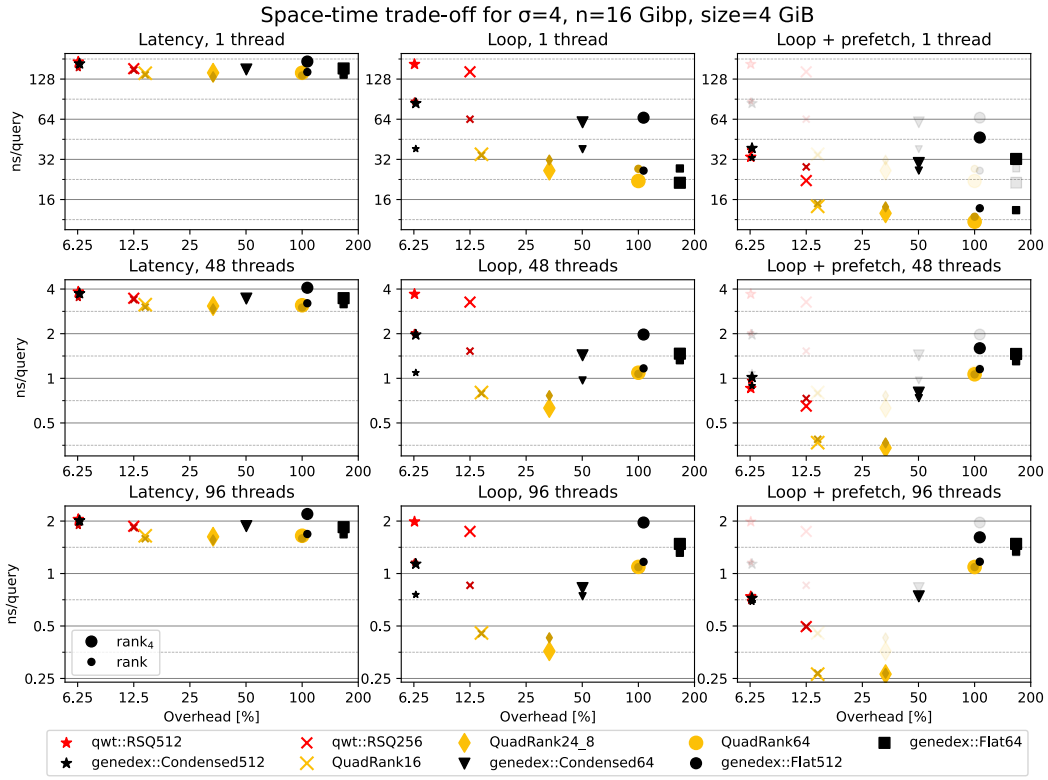

Figure 7 Scaling with input size for size 4 alphabet.

D QuadFm: A Batching FM-index

To showcase an application of our high-throughput data structure, we develop a toy implementation QuadFm of an FM-index for the $\sigma = 4$ DNA alphabet. Inspired by Movi [45, 46] and genedex [12] we process queries in batches and prefetch memory for upcoming rank queries. For simplicity, our implementation only counts the number of matches and does not support locating them. It only supports exact forward searching, and does not implement bidirectional search or search schemes [39, 40, 19], nor in-text verification. We use a prefix lookup table for the first 8 characters, and handle a single sentinel character (`$`) by storing its position in the Burrows-Wheeler Transform (BWT) [32, 7, 4].

The main function `query_batch` takes a batch of 32 queries and returns for each the BWT-interval where each query matches. Similar to genedex, during the processing, we keep an array of the indices of *active* queries whose interval is not empty yet. As long as there are active queries, we loop over those queries twice. First, we detect queries that were completed and *swap-pop* them from the active list, and then prefetch the memory needed for the rank queries. In a second loop, we perform the rank queries and LF-mapping for each active query. We do *not* optimize pairs of rank queries for small ranges, to avoid branch-misses.

D.1 Results

We test our FM-index implementation by building it for a 3.1 Gbp human genome [41]. We simulated 500 000 150 bp reads and applied 1% uniform random substitution errors to them. We then count the number of occurrences of each read in both forward and reverse-complement direction.

We compare the size and query throughput of QuadFm against AWRY [1] and Genedex [12], which are both configured to also use an 8-character prefix lookup table and a large suffix-array sampling factor. In particular, Genedex already supports query batching. We instantiate QuadRank with each of the rank structures for $\sigma = 4$. As before, we benchmark on 1, 6, and 12 threads on the laptop CPU, and test in three modes: *sequential*, where queries are done 1-by-1, *batch*, which does 32 queries in parallel, and *batch+prefetch*, which additionally prefetches cache lines for the next iteration over all queries.

In Figure 8, we see that genedex (blue) is faster than AWRY (purple). Genedex is consistently around 1 bit/bp larger than using the same rank structures in QuadFm (black), since it uses a size-5 alphabet to handle the sentinel, but otherwise they are comparable in performance. Using QuadRank makes QuadFm both smaller and up to 40% faster in multithreaded settings. Prefetching consistently doubles the throughput, and with 12 threads, QuadFm with QuadRank16 (2.29 bits/bp) is over 4× faster than Genedex' smallest variant (3.2 bits/bp) and 1.65× faster than its fastest variant (6.75 bits/bp).

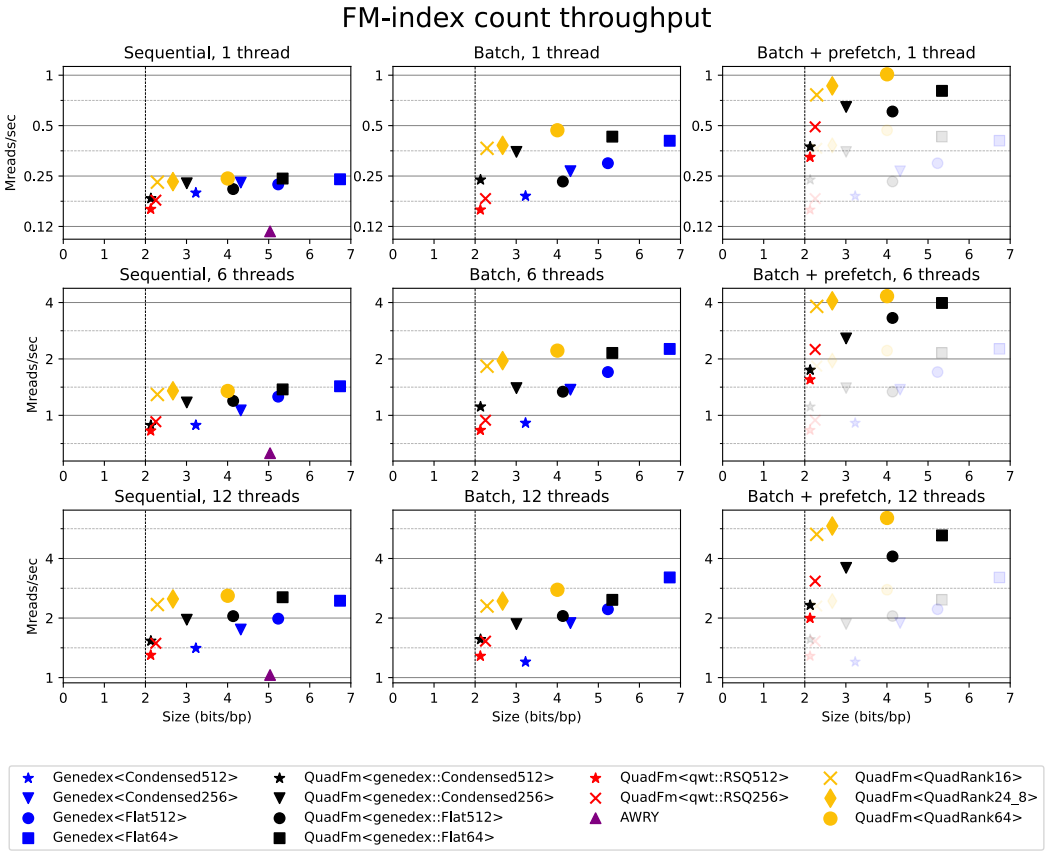

Figure 8 Size and throughput of counting exact matches of 150 simulated queries with a 1% error rate in an FM-index on a human genome. Vertical grey lines indicate the 2 bits/bp lower bound.