bafiq: BAM Flag Index Tool for Faster Repeated Queries of Mapped Reads

Jan Kotrs[1,2,*], Matej Lexa[1]

[1]*Masaryk University, Faculty of Informatics, Botanicka 68a, 60200 Brno, Czech Republic*
[2]*Health Monk s.r.o., Korunni 2569/108, 10100 Praha, Czech Republic*

Abstract
We present *bafiq*, a command-line rust tool for repeated filtering and counting of sequencing reads stored in the BAM file format. This software can be easily incorporated into NGS computational workflows, especially in situations where one needs to repeatedly process mapped reads in large files based on their FLAG values. Commands in *bafiq* follow query language familiar from *samtools* and similar software where, however, indexing is only based on the location of reads in a reference genome. Our tool allows significant computational time savings in situations where a single BAM file is queried repeatedly based on different FLAG combinations, leveraging a flag-based index. We made an effort to match or outperform the commonly used *samtools* in sequence extraction tasks as well as in sequence counting tasks.

Keywords
NGS workflow, BAM flags, read alignment filtering, BAM index

1. Introduction

Although long-read sequencing has been gaining traction steadily, short reads are still the most prevalent driver of today's NGS pipelines as a standard tool in biomedical research and clinical practice. Established and time-tested procedures and workflows now exist for every analysis imaginable. A wet-lab library preparation and sequencing phase is generally followed by computational analysis of the data. After pre-processing the raw sequencing reads, many such workflows initiate their computational phase by immediately comparing the reads to reference genomes, a step known as read mapping [1]. The standard data formats for storing the mapped (or aligned) reads are SAM/BAM [2], and the more recent CRAM [3] file formats.

The main goal of read mapping is to identify the regions of the reference genomes from which the analyzed sequencing reads may have originated. The process, however, is subject to possible errors that are then reflected in the contents of the alignment files (SAM/BAM/CRAM). Some reads may not be mapped to the reference at all, some may only map partially, some may map to multiple locations[4]. Depending on the sequencing technology, a small percentage of reads also contain errors [5], further complicating the picture. Computational tools designed to operate on the alignment files therefore not only prepare and convert the data for the next workflow processing step (e.g. tabulation of variants and their statistical analysis) but also tweak and filter their content so as to only work with an appropriate subset of sequencing reads in any given downstream analysis.

Tools that have become popular in this area are sometimes called BAM processors. They often convert and filter the BAM files before printing out selected subsets. In addition to the canonical software *samtools* based on the htslib library [2], equivalents have been made for other environments by wrapping these in Java [6], Python [7], or R [8]. Other BAM processors include Picard [9], *samblaster* [10], *biobambam* [11], and Scramble [12]. They often focus on extra flexibility, as is the case of *samql* ([13]), or speed which was the priority of *sambamba* [14]. The flexibility of *samql* comes in the form of

ITAT'25: Information Technologies – Applications and Theory, September 26–30, 2025, Telgárt, Slovakia
*Corresponding author.
honza@kotrs.cz (J. Kotrs); lexa@fi.muni.cz (M. Lexa)
0000-0002-4213-5259 (M. Lexa)

an SQL-like language for querying BAM files. The speed improvements are mostly based on parallel execution and code optimizations.

A well-known speed-centered addition to BAM/CRAM files is an index based on their coordinate-based sorting. This functionality made its way into samtools and many other BAM processors as well. The index divides the mapped and sorted reads in a BAM file into bins, which are then referenced in the index for fast data retrieval from a given range. Genome browsers such as IGV [15] rely on this mechanism to modify visualized browser content quickly even when working on very large BAM files.

In our work with BAM files we often needed to process them in various ways that, unlike the example above, depended on other components of the read data, not the coordinates. BAM format allows for reads to be marked using FLAGs and TAGs. For example, we may want to eliminate read pairs that do not map concordantly, or reads where their mate in the pair did not map. Or, we may want to only work with reads mapping with a certain minimum mapping quality (MAPQ value). While existing tools offer arguments to select specific flags or tags, these operations do not take advantage of indexing, as if the queries and filtering operations were not supposed to be done more than once within a given workflow. We noticed that this is not always the case. One may want to count certain types of reads in the alignment files, repeat the counting procedure for different settings of a variable, then filter the file for only a subset of reads to create a ”cleaned” version for further work. Also in certain educational settings, an instructor may require students to query the same BAM file as part of a structured exercise.

To support higher speeds in these scenarios without compromising on the real-world data volume, we propose to create a new type of index file that can be stored along with the original alignment file, similarly to the coordinate-based (.bai) index that can be made using *samtools*. The *bafiq* tool described here is a command-line program written in rust that takes a BAM file as an input along with a number of filtering arguments. It then either counts or extracts the reads in SAM format that pass the query criteria. When queried, *bafiq* first looks for an existing flag index (.bfi) of the input file. If it does not exist, it is created upon first query. Any subsequent calls to *bafiq* on the same input BAM file will take advantage of the index and return results faster. Counts are returned immediately as they are part of the index, sequence retrieval based on the query follows standard I/O limitations with the added benefit of the index allowing selective compressed blocks skipping.

2. BAM Flag Index (.bfi)

2.1. Read counts

The SAM format [2] defines a set of 16 binary FLAG values forming the total space of all FLAG combinations stored as 16-bit integers, or 2^{16} = 65,536. However not all flags are used, bits 12-15 are reserved, therefore the more realistic boundary is much smaller: 2^{12} = 4,096. Furthermore many of the flag combinations do not make biological sense. For example a single read cannot be *first in pair* and *second in pair* at the same time. Thus in a typical human NGS BAM file we observe up to 100 of flag combinations actually used out of which just 4 represent over 90% (Table 1).

Leveraging this observation makes pre-computing of all the flag combinations and the number of the corresponding reads upfront and embedding that into the *.bfi* index feasible. This way all *count* queries are resolved immediately as *bafiq* retrieves that count directly from the index. That also means the very first *bafiq* count query time is almost exclusively spent by building the index.

2.2. Read sequences

Learning the sequence number of given flag combinations in a BAM file is useful in itself however the downstream analysis often also requires the sequence extraction. For that reason the *.bfi* index stores the gzip file byte offsets of the blocks which contain at least one sequence of the given flag combination. This allows for faster extraction of the actual sequences due to skipping of entire blocks during the retrieval process without processing the whole file start to finish.

Flags	Flag names	Count	Percentage
0x053	paired, mapped, read rev. strand, first in pair	16,607,130	24.04%
0x0a3	paired, mapped, mate rev. strand, second in pair	16,607,130	24.04%
0x063	paired, mapped, mate rev. strand, first in pair	16,502,892	23.88%
0x093	paired, mapped, read rev. strand, second in pair	16,502,892	23.88%
0x4a3	paired, mapped, mate rev. strand, second in pair, PCR/optical dup.	358,112	0.52%
0x453	paired, mapped, read rev. strand, first in pair, PCR/optical dup.	358,112	0.52%
0x493	paired, mapped, read rev. strand, second in pair, PCR/optical dup.	316,126	0.46%
0x463	paired, mapped, mate rev. strand, first in pair, PCR/optical dup.	316,126	0.46%
0x051	paired, read rev. strand, first in pair	258,982	0.37%
0x0a1	paired, mate rev. strand, second in pair	257,880	0.37%

Table 1
Top 10 most frequent flag combinations of chr1 sequence in a BAM file produced by NGS. The source chr1 BAM file was obtained from the The 1000 Genomes Project's high coverage WGS effort[16]. Statistics were obtained using *bafiq's* index viewer (Section 6.1)

2.3. Index Compression

The BAM file is compressed using gzip which stores sequences in blocks. For reference a BAM file containing human chromosome 1 sequences obtained from a WGS (whole-genome sequencing) experiment of 30x read depth (common scenario) contains close to 70M sequences. A typical BAM block contains 180 sequences on average (see Section 6.1). That is roughly 380,000 blocks potentially (worst-case scenario) distributed among the flag combinations. The lists of blocks containing sequences of given flag combinations are referred to as *bin*s. As each block can host sequences with different flag combinations it can be part of multiple bins. As a result the total number of block offsets stored across all bins can grow quickly. Therefore we have briefly explored possible means of compression of the index itself to keep the size reasonable ideally without compromising *bafiq's* speed.

2.3.1. Bin Sparsity

As established earlier the list of bins is quite sparse (100 out of 4,096). Stripping empty bins is thus the first obvious step. That in itself however does not provide large enough space-saving benefit so we explored further.

2.3.2. Delta Encoding

Each bin contains an ordered sequence of byte offsets which are growing in nature. Thus storing the offsets as deltas from the first saves a lot of space and is trivial to decode at runtime.

2.3.3. Compression Gains

For the referenced example of chromosome 1 the *.bfi* index compression gains amount to ~30% using just the 2 methods:

```
Original size:       6,496,336 bytes (6.2 MB)
Compressed size:     4,471,326 bytes (4.3 MB)
Compression ratio:   1.45x
Space saved:         31.2%
```

If we break it down by technique the ratio is as expected:

```
Bin Sparsity:     129,280 bytes saved ( 6.0%)
Delta encoding: 2,025,458 bytes saved (94.0%)
```

2.3.4. Dictionary Compression

We have also explored the dictionary compression which would replace repeated offset subsequences in the index with a literal to be expanded upon decompression. However the initial implementation did not scale well ($\mathcal{O}(n^2)$ in complexity) and was not the primary focus so we have decided not to include it. It might be a subject of future optimizations should the index size becomes an issue.

3. Index Building Strategies

Since our main goal was to enable speedy exploratory analysis using flags we have explored and compared multiple strategies (Table 2).[1] of reading the input BAM file and building the index.

The index build process boils down to (1) read, (2) decompress, (3) index and optionally (4) compress. Each of the steps can be solved sequentially, in parallelized fashion or some combination of the two with various benefits and tradeoffs. Considering our goals we have decided to implement following 3 strategies: *Channel Producer-Consumer (CPC)* (utilizing *crossbeam-channels*[17]), *Work Stealing (WS)* (utilizing *rayon*[18]) and *Constant Memory (CM)*. The first two (*CPC and WS*) leveraging memory mapping and different types of parallelization orchestration with potentially large memory requirements in favor of speed. The third one is more memory conservative keeping an all-times low profile of mere 100MB. It does not memory-map the whole input file rather streams it by chunks and drains the processed ones from the memory as fast as possible.

The *Work Stealing* strategy leverages *ryon*[18] crate implementation of a parallel computation scheduling concept dating back to 1999 from the Cilk runtime[19]. The core idea is as worker threads becoming available to take new tasks they ”steal” tasks from queue of other busy worker threads. This requires less synchronization then a central channel contention used by the *Channel Producer-Consumer* strategy.

3.1. Discovery Phase

The first computational challenge is the fast *gzip* block discovery in the BAM file with respect to the underlying OS I/O limitations. Two approaches were explored: (1) *Discover-All-First* and (2) *Streaming*. The *Discover-All-First* approach uses single thread to read the whole file (mapped into memory) organized into *gzip* blocks for downstream processing. While the *Streaming* approach is feeding the downstream *gzip* block processors continuously as they are being discovered.

By nature of the memory mapping (leveraging *memmap2*[20] crate) *bafiq* eventually loads the whole input file into memory. As that could cause pressure in low-memory environments (although not completely, the memory recovers as needed over time) the *Constant Memory* strategy streams the input from the disk and uses capped buffer so that the memory usage stays linear in respect to the input size.

3.2. Decompression Phase

The next phase takes in discovered *gzip* blocks and decompresses them in memory for the downstream BAM records processing. Here the challenge is how to efficiently utilize existing decompression libraries (as the decompression itself is outside of the scope of this work).

We are leveraging *libdeflater*[21] crate for fast decompression with all 3 strategies using thread local buffer to process.

3.3. Flags Extraction Phase

In this phase the just decompressed block becomes addressable for the contents. With the simple operation of reading just the flag bit we opted-out of using the *htslib-rust*[22] crate to avoid unnecessary

[1]As the naïve single-core non-parallel strategy scanning the BAM file start to finish was under-performing even on the smallest test file (1.3GB, over 1m 30s) we have decided not to include it in further performance exploration.

Table 2
Comparison of Index Building Strategies

Dimension	Channel Producer-Consumer	Work Stealing	Constant Memory
Perf. @ 10t, 1.3GB BAM	1.998 s	1.056 s	4.561s
Memory Usage	1.3 GB	1.3 GB	100 MB
Coordination	Crossbeam channels	None	Sequential chunks
Processing Model	Producer-consumer	Batch work-stealing	Micro-batch streaming
File Access	Memory-mapped	Memory-mapped	Streaming file I/O
Block Discovery	Streaming	Discover-all-first	Streaming
Merge Strategy	Parallel tree	Parallel tree	Immediate persistence

abstraction (however we're leveraging it for writing SAM format output later in the sequence extraction task).

3.4. Index Assembly Phase

Both *WS* and *CPC* strategies build up local indices within scoped threads and merge at the end of decompression and extraction phases. The *CM* uses the same merge-tree algorithm although merges to the main index after every micro-batch is processed to minimize memory impact.

4. Querying

To define flag combination for querying the BAM sequences we opted for *samtools* specification using *-f* (required flags) and *-F* (forbidden flags) to keep familiarity.

As the SAM[2] specification defines following flags (bits) can be combined to query for sequences:

Bit	Hex	Read Flag
1	0x1	Paired read
2	0x2	Properly paired
4	0x4	Read unmapped
8	0x8	Mate unmapped
16	0x10	Read is reverse complemented
32	0x20	Mate is reverse complemented
64	0x40	First read in pair
128	0x80	Second read in pair
256	0x100	Secondary alignment
512	0x200	Fails quality checks
1024	0x400	PCR or optical duplicate
2048	0x800	Supplementary alignment

5. Benchmarking

We have measured the CPU and memory utilization as well as overall time to finish a task of all 3 selected *bafiq* strategies alongside samtools as a reference. As *bafiq's* advantage comes primarily after the index is built we also added our version of quick counting and filtering (*bafiq fast-count*) more resembling to what *samtools view -c* does to demonstrate the underlying performance.

Apart from the actual index building we bench-marked the sequence retrievals using the *.bfi* index.

Two short-read BAM files originating from a typical NGS WGS 30x pipeline were selected for benchmarking - chr22 (1.3GB) and chr1 (8.2GB) aligned to *hg38* reference. As many of *bafiq* features build on parallelized processing all benchmarking runs had explicit *–threads* setting aligned with samtools for fair comparison.

All benchmark runs were performed either to produce or consume uncompressed indices as the final index size was not an issue (~0.5% of the input file).

5.1. Bench 1: first query (*bafiq index*)

As seen in the index building benchmarks (Table 3) the best performing strategy was *Work Stealing* with its index building time close to *samtools view -c* across the different number of threads available, even slightly faster when both constrained to 2 threads. As the subsequent query for sequence count is fetched pre-computed from the index and resolve under 20ms we consider the index building a time-equivalent task to *bafiq index + bafiq query*.

We can clearly see that the cost of channel management overhead and work organization required by *Channel Producer-Consumer* strategy is very costly especially with limited threads but becomes less significant as the number of available threads grows.

Due to the memory-mapping feature both *CPC* and *WS* strategies can exhaust memory up to the original BAM file size which is not pracical for full-genome alignments. Because of that we have included more memory-conservative strategy (*Constant Memory (CM)*) which, although not as performant as *WS*, keeps a constant memory footprint (~100 MB) independent of the input file size.

5.2. Bench 2: sequence extraction (*bafiq view*)

The sequence retrieval (Table 4) of *bafiq view* leverages stored gzip block byte offsets (indexed for each of the flag combination) so that reading from the original BAM file does not depend on a pre-scanning step anymore and can limit its read to focused blocks. As each block stores 180 sequences on average traversing it is relatively cheap operation.

Where *.bfi* index performs quite well are use-cases of retrieving rare flag combinations relative to the bulk of records (chr22 0x4 ~3,922). However its utility as a performance helper decreases with the abundance of matching reads (as seen in chr22 0x2 and 0x10 flags).

Despite that observed performance decrease in most measured scenarios the index-based retrieval was still faster than without it.

6. Supplementary Tools

6.1. Index Viewer (*bafiq-viewer*)

As the *.bfi* index file is stored in a binary form to save space we have developed a tool to quickly glance over the contents including basic index statistics. For example a *.bfi* index of chromosome 1 can look as follows:

```
Total records:     69095506
Total bins:        56
Non-empty bins:    56
Total blocks:      379169
Reads per block:   min=35, max=310, avg=182.2
```

7. Discussion

We have shown that an exploratory analysis of aligned reads based on FLAG combinations can vastly benefit from a dedicated index to obtain sequence counts (from *s* to *ms*) and for sequence retrieval

Table 3
Bench 1: Index Building Task, chr22 (1.3GB)

Strat.	T	Time (s)	Pk RAM	Pk/Avg CPU
CM	2	10.778	0GB	163% / 148%
CPC	2	34.456	1.3GB	242% / 200%
WS	2	3.113	1.3GB	196% / 188%
BFC	2	2.696	1.2GB	250% / 210%
S	2	6.536	0GB	143% / 136%
CM	4	7.356	0GB	250% / 225%
CPC	4	9.538	1.3GB	442% / 397%
WS	4	1.950	1.3GB	379% / 321%
BFC	4	1.495	1.2GB	466% / 423%
S	4	1.985	0GB	395% / 391%
CM	6	6.670	0GB	314% / 288%
CPC	6	4.600	1.3GB	677% / 610%
WS	6	1.634	1.3GB	555% / 439%
BFC	6	1.056	1.2GB	683% / 533%
S	6	1.340	0GB	650% / 624%
CM	10	7.226	0GB	369% / 346%
CPC	10	2.122	1.3GB	1060% / 943%
WS	10	1.195	1.3GB	865% / 630%
BFC	10	0.611	1.2GB	1127% / 1074%
S	10	0.927	0GB	1003% / 968%
CM	24	7.276	0GB	458% / 430%
CPC	24	0.970	1.3GB	2424% / 1741%
WS	24	0.854	1.3GB	1750% / 1463%
BFC	24	0.413	1.2GB	2469% / 2469%
S	24	0.912	0GB	1059% / 1026%

Table 4
Bench 2: Sequence Extraction Task

File	Flag	Tool	T	Time (s)	Records
chr22	0x4	SV	2	5.787	3,922
		BV	2	0.307	3,922
		SV	24	1.206	3,922
		BV	24	0.171	3,922
chr22	0x10	SV	2	7.763	5,364,679
		BV	2	7.954	5,364,679
		SV	24	2.139	5,364,679
		BV	24	3.931	5,364,679
chr22	0x2	SV	2	10.046	10,544,216
		BV	2	10.912	10,544,216
		SV	24	6.094	10,544,216
		BV	24	8.909	10,544,216
chr1	0x4	SV	2	33.986	13,028
		BV	2	1.187	13,028
		SV	24	5.353	13,028
		BV	24	0.466	13,028
chr1	0x10	SV	2	319.660	34,551,825
		BV	2	161.876	34,551,825
		SV	24	191.471	34,551,825
		BV	24	164.540	34,551,825
chr1	0x2	SV	2	358.796	67,657,186
		BV	2	232.801	67,657,186
		SV	24	351.908	67,657,186
		BV	24	287.492	67,657,186

Table 3: CM = *constant-memory*, CPC = *channel-producer-consumer*, WS = *work-stealing*, BFC = *bafiq-fast-count**, S = *samtools*, T = *number of threads*, Pk/Avg = *peak/average*; **Table 4**: SV = *samtools view*, BV = *bafiq view*

scenario where the flag combination matches smaller subset. However in some cases where the flag combination matches large proportion of sequences of the BAM file the benefits of stored offsets in current implementation diminish.

Nevertheless the code base and insights from individual presented strategies can serve as a demonstrator of utility of building narrowly focused indices. The present FLAG index scope could be also extended to include other fields from the header (such as TAG or MAPQ).

8. Declaration on Generative AI

During the preparation of this work, the authors used *Claude Sonnet 4* in order to: Assist in implementing parts of the rust code base. After using the tool, the authors reviewed and edited the content as needed and take full responsibility for the code content.

The authors have not employed any Generative AI tools for the publication content.

References

[1] S. Schbath, V. Martin, M. Zytnicki, J. Fayolle, V. Loux, J.-F. Gibrat, Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis, Journal of Computational Biology 19 (2012) 796–813. doi:10.1089/cmb.2012.0022.

[2] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, 1000 Genome Project Data Processing Subgroup, The sequence alignment/map format and SAMtools, Bioinformatics 25 (2009) 2078–2079. doi:10.1093/bioinformatics/btp352.

[3] G. Cochrane, B. Alako, C. Amid, L. Bower, A. Cerdeño-Tárraga, I. Cleland, et al., Facing growth in the European Nucleotide Archive, Nucleic Acids Research 41 (2012) D30–D35. doi:10.1093/nar/gks1175.

[4] M. Cechova, Probably correct: Rescuing repeats with short and long reads, Genes 12 (2020) 48. doi:10.3390/genes12010048.

[5] F. Pfeiffer, C. Gröber, M. Blank, K. Händler, M. Beyer, J. L. Schultze, G. Mayer, Systematic evaluation of error rates and causes in short samples in next-generation sequencing., Sci Rep 8 (2018). URL: https://doi.org/10.1038/s41598-018-29325-6.

[6] HTSJDK, https://github.com/samtools/htsjdk, 2025.

[7] A. Heger, J. Marshall, K. Jacobs, et al., Pysam, https://github.com/pysam-developers/pysam, 2025.

[8] M. Morgan, H. Pagès, V. Obenchain, N. Hayden, Rsamtools: binary alignment (BAM), FASTA, variant call (BCF), and tabix file import. R package version 2.24.0, 2025. doi:10.18129/B9.bioc.Rsamtools.

[9] Picard Toolkit, https://broadinstitute.github.io/picard/, 2019.

[10] G. G. Faust, I. M. Hall, SAMBLASTER: fast duplicate marking and structural variant read extraction, Bioinformatics 30 (2014) 2503–2505. doi:10.1093/bioinformatics/btu314.

[11] G. G. Faust, I. M. Hall, Biobambam: tools for read pair collation based algorithms on bam files, Source Code Biol. Med. 9 (2014) 13. doi:10.1093/bioinformatics/btu314.

[12] B. J. K., The Scramble conversion tool., Bioinformatics 30 (2014) 2818–2819. doi:10.1093/bioinformatics/btu390.

[13] C. T. Lee, M. Maragkakis, SamQL: a structured query language and filtering tool for the SAM/BAM file format., BMC Bioinformatics 22 (2021) 474. doi:10.1186/s12859-021-04390-3.

[14] A. Tarasov, A. J. Vilella, E. Cuppen, I. J. Nijman, P. Prins, Sambamba: fast processing of NGS alignment formats, Bioinformatics 31 (2015) 2032–2034. doi:10.1093/bioinformatics/btv098.

[15] H. Thorvaldsdóttir, J. T. Robinson, Mesirov, J. P., Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration, Briefings in Bioinformatics 14 (2012) 178–192. doi:10.1093/bib/bbs017.

[16] M. Byrska-Bishop, U. S. Evani, X. Zhao, A. O. Basile, H. J. Abel, A. A. Regier, A. Corvelo, W. E. Clarke, R. Musunuri, K. Nagulapalli, S. Fairley, A. Runnels, L. Winterkorn, E. Lowy, H. G. S. V. Consortium, P. Flicek, S. Germer, H. Brand, I. M. Hall, M. E. Talkowski, G. Narzisi, M. C. Zody, High-coverage whole-genome sequencing of the expanded 1000 genomes project cohort including 602 trios, Cell 185 (2022) 3426–3440.e19. doi:10.1016/j.cell.2022.08.004.

[17] A. d'Antras, J. Stone, A. Crichton, F. S. Klock II, S. Kazlauskas, *crossbeam: Tools for concurrent programming in Rust*, https://github.com/crossbeam-rs/crossbeam, 2025. Accessed: 2025-08-20.

[18] N. Matsakis, J. Stone, *rayon: Data parallelism in Rust*, https://crates.io/crates/rayon, 2025. Accessed: 2025-08-20.

[19] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, Journal of the ACM 46 (1999) 720–748.

[20] R. Levien, contributors, *memmap2: Memory-mapped file IO for Rust*, https://crates.io/crates/memmap2, 2025. Accessed: 2025-08-20.

[21] A. Kewley, *libdeflater: Fast DEFLATE/zlib/gzip compression and decompression*, https://github.com/libdeflater/libdeflater, 2025. Accessed: 2025-08-20.

[22] J. Köster, C. Schröder, P. Marks, D. Lähnemann, M. Holtgrewe, J. Gehring, *rust-htslib: Rust bindings to HTSlib*, https://github.com/rust-bio/rust-htslib, 2025. Accessed: 2025-08-20.

A. Online Resources

The source code for the *bafiq* tool is available via GitHub (https://github.com/honzakotrs/bafiq).