

COBS: a Compact Bit-Sliced Signature Index

Timo Bingmann¹, Phelim Bradley², Florian Gauger¹, and Zamin Iqbal²

¹ Institute of Theoretical Informatics,
Karlsruhe Institute of Technology, Germany

² European Molecular Biology Laboratory,
European Bioinformatics Institute,
Cambridge, United Kingdom

Abstract. We present COBS, a compact bit-sliced signature index, which is a cross-over between an inverted index and Bloom filters. Our target application is to index k -mers of DNA samples or q -grams from text documents and process approximate pattern matching queries on the corpus with a user-chosen coverage threshold. Query results may contain a number of false positives which decreases exponentially with the query length and the false positive rate of the index determined at construction time. We compare COBS to seven other index software packages on 100 000 microbial DNA samples. COBS' compact but simple data structure outperforms the other indexes in construction time and query performance with Mantis by Pandey et al. on second place. However, different from Mantis and other previous work, COBS does not need the complete index in RAM and is thus designed to scale to larger document sets.

1 Introduction

In this paper we present an approximate q -gram index named COBS [13], short for COmpact Bit-sliced Signature index, which is a cross-over between an inverted index and Bloom filters. The current application focus of COBS is to index DNA and protein k -mers from sequencing experiments, but the data structure can also be used for indexing q -grams from other domains such as English text.

In living cells, DNA exists as long contiguous molecules, typically textually encoded as strings of A, C, G, and T. The output of sequencing experiments are stored both in raw format (text files of the strings) and “assembled format” – semi-heuristic best approximations to the underlying genome, also in text format, but of very variable quality, in particular when based on short read data. As sequencing technology has advanced, it has also become much cheaper and more widespread, and its output has been stored in publicly available archives (e.g. the European Nucleotide Archive (ENA) and the Sequence Read Archive (SRA) which maintain mirrors of all the data. These archives now double in size every 18 months, and it is progressively more important to be able to search within the deposited datasets, to find important genes or mutations, or combinations of mutations which are informative of function or ancestry. In 2018, the ENA encompassed $1.5 \cdot 10^9$ microbial sequences and $8 \cdot 10^{15}$ base

pairs (i.e. characters) of read data [17], while the European Bioinformatics Institute reached 160 PB of storage capacity [10].

Despite the obvious similarities to standard document retrieval problems, the properties of DNA k -mer data are very different from traditional text corpora. Google’s index is reported to have in the order of 10^{13} documents containing 10^8 unique terms [7], whereas the small benchmark set of 100 000 microbial sequences used in our experiments already contain $2.2 \cdot 10^{10}$ distinct k -mers, of which $1.8 \cdot 10^{10}$ occur only once. The frequency of terms in a natural language is power-law distributed, with underlying terms generated over hundreds of years, resulting in just a few new terms per document. Microbial genomes however encode many billions of years of evolution; each new genome generates thousands of novel k -mers. There are also two other aspects whereby searching biological data differs from standard text retrieval. The first is that the index must support *approximate queries* allowing detection of closely related DNA to the query. Approximate pattern matching however is a notoriously difficult subject for text indices [21]. The second is that users often want all hits, not just the top few as is typical in web search.

For COBS we chose the robust q -gram indexing approach [33] and combined it with Bloom filters to reduce the term space size. This can be considered a variant of *signature files*, which have a long history in information retrieval [12] but were pushed to the sidelines for text search by inverted indexes [37]. Recently, they have been reconsidered as acceleration filters for large text search corpora [15] by engineering them to adapt to the collection’s characteristics. With COBS we venture to combine signature files with one-sided errors introduced by Bloom filters and inverted files to design an ultra fast and scalable q -gram index which supports approximate queries delivering a small reasonable number of expected false positives. Our contribution of making the signature files *compact* first enables the index to be applied to corpora with highly varying document sizes, such as microbial DNA samples.

1.1 Related Work

Considering q -grams or k -mers of a sequence are a staple in bioinformatics [9]. For example, BLAST [3] uses k -mers for seeding its search for maximal local alignments.

The earliest use of Bloom filters as an index for a collection of independent documents we could find is called *Bloofi* by Crainiceanu and Lemire [11]. They propose to use a Bloom filter for each document and to arrange them either in a B-tree or as a *Flat-Bloofi*. The latter is similar to BIGSI and COBS without compaction.

The currently most cited line of work on DNA k -mer indices for approximate search are the Sequence Bloom Trees (SBTs) first proposed by Solomon and Kingsford [29]. In an SBT the k -mers of each document are individually indexed into a Bloom filter, which are then arranged as the leaves of a binary tree. The inner nodes of the binary tree are union Bloom filters of their descendants. A query can then breadth-first traverse the tree pruning search paths which no longer sufficiently cover a given threshold Θ of the query k -mers.

In the original SBT [29] a simple greedy clustering method is used, the bit union is stored in each inner node, and all nodes are RRR compressed [28] using SDSL [14]. The first improvement, the Split Sequence Bloom Tree (SSBT) [30], splits the inner nodes into two Bloom filters: a *similarity* filter and a *remainder* filter, where the first contains all bits in both child filters and the second those set in either child minus the *similarity* filter. Simultaneously, Sun, Harris, Chikhi, and Medvedev proposed the AllSome Sequence Bloom Tree (AllSome-SBT) [31], which splits each inner node into an *all* and a *some* subfilter. The *all* filter contains bits in all leaves below the node, excluding those already set in the parent node, and the *some* filter all bits in some leaves but not all. The currently smallest SBT variant is called HowDe-SBT by Harris and Medvedev [16]. It decomposes the Bloom filters in each inner node into two bit vectors: the *det* vector signals if a particular bit is *determined* at this inner node, meaning that it is equal in all descendant leaves, and a *how* vector signaling if it is determined as zero or one. All determined bits can be omitted from any children.

A completely different approach to indexing k -mers is taken by *Mantis* from Pandey et al. [26]. In *Mantis*, a counting quotient filter (CQF) [27] is used to construct a mapping from k -mers to *color classes*, wherein k -mers with identical occurrence vectors for all documents are mapped to the same color class. Incidence of color classes to documents can then be represented as a matrix, in which columns are associated with documents and each row corresponds to a color class. *Mantis* differs from the other k -mer indexes referenced in this paper by being able to deliver *exact* approximate matching results without false positives.

SeqOthello [35] is another k -mer index software package. It contains an ensemble of encoding techniques for compressing the occurrence maps of k -mers in the document set. To locate the correct occurrence map a hierarchy of *Othellos* is built, which are minimum perfect hash function mappings and can introduce false positive results due to mapping of alien k -mers to random results.

BIGSI (BItSliced Genomic Signature Index) by Bradley et al. [6] is the direct ancestor of COBS and also a combination of Bloom filters and inverted indexes. *BIGSI* however is a prototype programmed in Python and uses a key-value database such as BerkeleyDB or RocksDB as storage back-end. It also does not contain the compaction feature introduced in COBS.

Related to k -mer indexing are colored de Bruijn graph representation data structures, which often contain an exact k -mer index but do not support approximate k -mer pattern searches. The original implementation, *Cortex*, stored k -mers in a hash table, along with booleans for the four possible forward and backward edges in a single byte. This was then followed by *McCortex* [20,32], which added a second data structure to encode paths in the graph present in the original reads. By contrast, *VARI* [25], *Rainbowfish* [1], and *pufferfish* [2] explore use of succinct data structures, the Burrows-Wheeler transform, and minimal perfect hash functions to save space and possibly even accelerate operations. The Bloom Filter Trie [19] is another colored de Bruijn graph representation based on the burst trie [18], wherein lookups for suffixes at compressed inner nodes are accelerated with Bloom filters.

2 A Compact Bit-Sliced Signature Index

In this section we present the index structure used in COBS. We first generally review Bloom filters as a q -gram index in subsection 2.1, then turn to COBS' more compact bit-sliced representation in subsection 2.2, and discuss implementation details and algorithm engineering aspects in subsection 2.3.

2.1 Approximate Matching with Bloom Filters of Signatures

Given are an ordered set of *documents* $\mathcal{D} = [d_0, \dots, d_{|\mathcal{D}|-1}]$, where each document d is composed of a set of *strings* $\{t_0, \dots, t_{|d|-1}\}$. The number of items in a set or array is denoted with $|\cdot|$. Each string t is a zero-based array of $|t|$ characters from a finite ordered *alphabet* Σ . In the context of indexing DNA, the alphabet is usually $\Sigma = \{\text{A, C, G, T}\}$, the documents are experiment samples, and the strings in each document can be reads or assembled genome sequences. When indexing web sites, the alphabet could be the ASCII characters or English words, the documents could be web pages, and the substrings may be words, sentences, or paragraphs.

To facilitate approximate pattern matching we consider q -grams of the strings [33], commonly called k -mers for DNA. For each string t with $|t| \geq q$ there are $|t| - q + 1$ consecutive substrings of length q . For a document d , we denote with $G_q(d)$ the union of all q -grams in the strings in d . Due to similarities with full-text indexing we also refer to the q -grams in a document as *terms*.

The COBS index is composed of $|\mathcal{D}|$ Bloom filters [5], each representing an approximate membership data structure with one-sided error. To construct a Bloom filter for a document d we assume k pairwise independent hash functions h_0, \dots, h_{k-1} with range $[0, w)$ and set the k bits $h_i(s)$ in an array f of w bits for each q -gram $s \in G_q(d)$. Testing for membership of a q -gram s is performed by checking if all k cells $h_i(s)$ are set, which can lead to false positives but never false negatives.

The entire document collection is thus represented by $|\mathcal{D}|$ bit arrays $[f_0, \dots, f_{|\mathcal{D}|-1}]$, each a Bloom filter with possibly different parameters. From previous work, the false positive rate p of a Bloom filter of size w with k hash functions and v inserted elements is known to be at most $(1 - (1 - \frac{1}{w})^{kv})^k \leq (1 - e^{-kv/w})^k$. Given a desired false positive rate p and number of elements v , one can calculate a partial derivative of the last bound to determine good approximate parameters $k = \frac{w}{v} \ln 2$ and $w = -\frac{v \ln p}{(\ln 2)^2}$ [8,23].

To perform approximate matching for a pattern P , we follow previous work [33] and determine the q -gram distance of P to all documents in the collection \mathcal{D} by testing each of the query's q -grams $G_q(P)$ on all documents. In COBS we present this positively as the q -gram *score* of the query for each document. The score is used to rank and return all documents containing at least a given percentage K of the $|G_q(P)|$ terms in the query.

As Solomon and Kingsford already noticed for SBTs, in the case of approximate pattern search on Bloom filters, we are not ultimately interested in the false positive rate of a single Bloom filter lookup. Instead we are concerned with the false positive

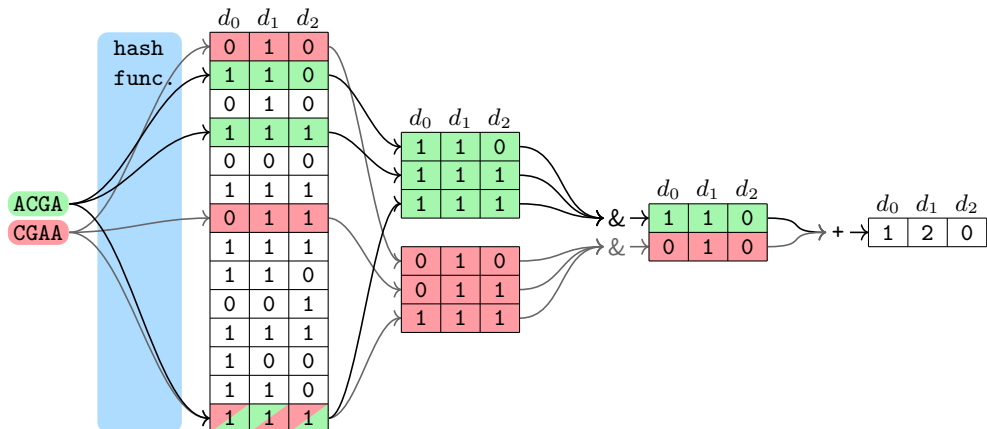


Fig. 1. Architecture of the bit-sliced signature index and query processing steps.

rate of a *query* P . More precisely, given $\ell = |G_q(P)|$ q -grams with the probability that more than $K\ell$ terms are false positives in the same filter.

Theorem 1 (False Positive Rate of a Query, Theorem 2 in [29]).

Let P be a query pattern containing $\ell = |G_q(P)|$ distinct terms. If we consider the terms as being independent, the probability that more than $\lfloor K\ell \rfloor$ false-positive terms occur in a filter f with false positive rate p is $1 - \sum_{i=0}^{\lfloor K\ell \rfloor} \binom{\ell}{i} p^i (1-p)^{\ell-i}$.

This theorem is derived by considering lookups of terms as independent Bernoulli trials and summing over the probability of zero to $\lfloor K\ell \rfloor$ false positives among the ℓ trials, which yields a binomial distribution. Given $K \geq p$, Solomon and Kingsford also apply a Chernoff bound and show that the false positive probability for a query to be detected in a document is $\leq \exp(-\ell(K-p)^2/(2(1-p)))$.

These repeated trials into the Bloom filter allow us to push the false positive rate p up higher than commonly used. Traditional uses of Bloom filters for approximate membership queries consider an error rate of 0.01 or less and multiple hash functions as desirable. Due to the inverse exponential relationship of a query’s false positive rate with its length, coupled with the fact that more hash functions cost more cache faults or I/Os, the minimum $k = 1$ and a high false positive rate around 0.3 are desirable for our q -gram index application.

For example, if we consider a query of length 100 containing $\ell = 70$ distinct 31-grams, a false positive rate of $p = 0.3$, and threshold $K = 0.5$, then Theorem 1 yields a false positive rate of about 0.000143. Which means there will be about 143 false positive results in one million documents on average.

2.2 Bit-Slicing and Compaction

Provided all Bloom filters are of the same size w , one can store them as a $w \times |\mathcal{D}|$ bit matrix such that a row contains all bit cells at one index in the $|\mathcal{D}|$ filters (see

left side of Figure 1). This is also called a “bit-sliced” layout [34] and was chosen for BIGSI and COBS to reduce the number of random accesses needed to evaluate a query. This is particularly important if the index is read from external memory, where scanning is much more efficient than random accesses. The approach however requires all Bloom filters to use the same hash functions and be the same size.

Figure 1 also illustrates how a query P is performed using the bit-sliced Bloom filter matrix. The q -grams of the query are hashed to determine the corresponding rows. These $k|G_q(P)|$ rows are then scanned and an *AND* join of k rows is performed to determine which q -grams occur in which document. This yields an indicator bit vector ordered by document number. All indicator vectors are then added together to calculate the score for each document. Only those documents reaching the query threshold $K|G_q(P)|$ are then reported as approximate matches. Due to the one-sided error of the Bloom filters, only *more* documents may be reported due to hash collisions; false negatives, i.e. missed hits, cannot occur.

One can also view the Bloom filter bit matrix as an inverted index: each row simply lists the document numbers containing the corresponding q -gram as indexes in a bit vector. Unlike a traditional inverted index however, *multiple* q -gram terms are superimposed in one row. This leads to false positive matches. In theory, one could apply all the methods developed by the information retrieval community [36] to these bit vectors or posting lists.

The current version of a bit-sliced index however relies on all documents and resulting Bloom filters having the same size. But larger documents result in denser bit vectors and smaller documents in sparser, as the number of bits set depends on the number of q -gram terms in the document. Depending on the dataset, this creates vastly different false positive rates in the bit matrix. Hence, we propose to *adapt* the size of each Bloom filter bit array to the document it indexes and aim to keep the false positive rate *constant*. We call this a *compact* bit-sliced signature index (the CO in COBS).

In theory one could adapt the Bloom filter size and hash function for each document. In practice we want to store bits of rows as blocks of size B in external memory, thus keep the parameters constant for $\Theta(B)$ consecutive documents. Furthermore, instead of calculating a new hash function for each filter, we propose to use only one function with a larger output range and then use a modulo operation to map it down to each individual filter’s size. Both practical optimizations only incur a small deviation from the optimal index size and false positive rates.

Figure 3 shows in light blue the desired Bloom filter size for the 100 000 microbial documents used in our experiments ordered by size and with false positive rate 0.3 and one hash function. The dark blue staircase function above the upward sloping curve shows batches of $B = 4\,096$ documents encoded with the maximum Bloom filter size of that block. The visible dark blue area is the minor overhead for encoding documents block-wise. If one uses only one Bloom filter size (the classic approach), then the index size would be the entire filled orange area, which extends upward to ensure the desired false positive rate for the largest document.

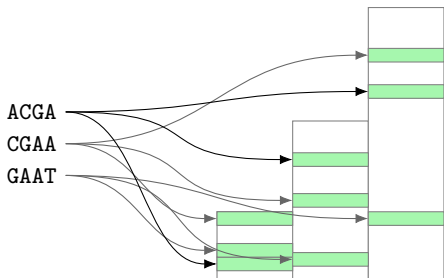


Fig. 2. Access pattern of our compact bit-sliced index.

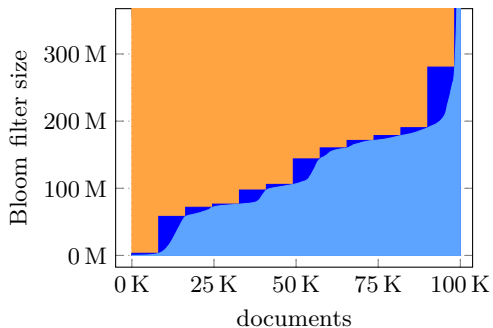


Fig. 3. Compact index composed of small sub-indexes containing $\Theta(B)$ documents.

Due to the variance in size of microbial and other real-world documents, the *compact* representation in COBS is essential. In designing COBS, we also considered that today’s SSD and NVMe storage technology now has orders of magnitude faster random access speeds [4] compared to rotational disks. Thus with these new storage devices, the batched random access for many smaller blocks of size B , as used in the compact layout and illustrated in Figure 2, first becomes viable.

2.3 Implementation and Engineering

We implemented COBS as a command line search engine tool using C++ and plan to provide a Python interface to the underlying algorithm library. The tool is open source and available from <https://panthema.net/cobs/>. It can read DNA FASTA files, multi-document protein FASTA files, McCortex, or text files as documents and extract q -grams from them. Depending on the format, the input data is broken into different q -gram sets: DNA reads are for example hashed independently, while English text is processed continuously. The q -grams or k -mers can optionally be canonicalized if their reverse complement are considered equivalent.

Classic and Compact. The COBS program can currently construct two index variants: *classic* and *compact*. In the classic index all documents are hashed using the same Bloom filter size, which depends on the desired false positive rate and the number of q -grams in the largest document. This is the non-compact version, which is similar to BIGSI, but was written for performance in C++ and with direct file accesses.

When constructing a compact index, the size of all documents are determined and the document set is reordered by size. Then a subindex is constructed for every $\Theta(B)$ documents, as described in subsection 2.2. Each subindex is actually a classic COBS index. The subindices are simply concatenated into one large file.

Parallelization. Due to the massive amount of data to process, we parallelized construction and query for shared-memory systems. Classic index construction we parallelized by building temporary indexes over batches of the documents and then

merging them into larger indexes. For compact index construction we parallelized construction of the subindices. Pattern search is parallelized by processing disjoint partitions of the document scores in parallel and then selecting the top scores sequentially using a partial sort operation.

Memory Mapped I/O. For querying an index, we map the file into virtual address space using `mmap`. The necessary rows of the inverted Bloom filter index are then read using a simple memory transfer. We experimented with directly issuing asynchronous I/O commands, but found only a negligible performance advantage that did not outweigh the higher code complexity.

Alternatively, COBS can also read the complete index into RAM and then run all queries. This was added to compare performance against other indexing software which only work in RAM, e.g. Mantis, in section 3.

3 Experimental Evaluation

In this section we present a comprehensive evaluation of eight software packages for indexing k -mers from read or assembled genomic sequence data.

Software Packages. We acquired a copy of the original **SBT** [29] (git hash `977adfa` from March 1st 2019), the Split-SBT (**SSBT**) [30] (git hash `710c95f` from July 10th 2018), the **AllSome-SBT** [31] (git hash `4e1f2c5` from October 28th, 2018), the **HowDe-SBT** [16] (git hash `76e3c89` from March 1st, 2019), **SeqOthello** [35] (git hash `68d47e0` from September 6th, 2018), Squeakr and **Mantis** [26] (git hash `3853c82` from January 29th, 2019), and **BIGSI** [6] (git hash `2ab35e5` from May 15th, 2019) using BerkeleyDB 4.8.30. More reference about each package can be found in the related work subsection 1.1. We compiled all from source and additionally used `ntCard` [24] (v1.1.0) as a preprocessing step for the SBTs, `jellyfish` [22] (v2.2.10) in other steps and as a library.

Data. Bradley et al. [6] previously indexed the complete global corpus of microbial DNA data, some 450,000 files. In doing this, they processed the raw data into k -mers. Since this contains low frequency errors from the sequencing instruments, they “de-noised” it using standard methods from `McCortex`, and stored the remaining k -mers in a binary format. We downloaded 100 000 of these files from http://ftp.ebi.ac.uk/pub/software/bigsi/nat_biotech_2018/ctx/. For microbial genomic read data k was chosen as 31, as this is large enough to (generally) guarantee uniqueness within a genome, without being so large as to frequently hit a sequencing error. For scaling experiments we selected random subsets containing 100, 250, 500, 1 000, 2 500, 5 000, 10 000, 25 000, and 50 000 documents from the 100 000 base set, each contained in the larger subsets. The 10 000 document subset is the same as used in one of the **BIGSI** experiments [6]. The average document size is 42.77 MiB stored in `McCortex` format, such that the entire 100 000 microbial dataset is 3.984 TiB in total. Each document contains 3.4M 31-mers on average with the minimum being zero and the largest containing 138M 31-mers. In total the 100 000 dataset contains 336 846 M 31-mers to index. While building the indexes using the various software all k -mers were included, without any occurrence threshold or cut-off.

Platform. We ran the experimental evaluation on a quad-socket Intel Xeon Gold 6138 2.0 GHz 4×20 -core machine with 768 GiB DDR4-2666 RAM and 4×2 TB NVMe Samsung 970 EVO SSD storage devices combined using RAID 0. The machine was running Ubuntu 18.04 with Linux kernel 4.15.0-48-generic and gcc 7.3.0. The combined SSDs reached 12.2 GiB/s sequential read, 2.3 GiB/s sequential write, 741 MiB/s random 4 KiB block read, and 1 188 MiB/s random 4 KiB block write speeds.

Queries. We designed four sets of batch queries to measure the performance of the indices, each set containing known true positives and true negatives in random order. In each batch all queries are of the same length $\ell \in \{31, 100, 1\,000, 10\,000\}$ base pairs (bp). To generate true positives, we first extracted all unitigs from the colored de Bruijn graph representation of each document using McCortex, and then randomly chose queries from all ℓ -grams in the unitigs. To generate true negatives, we generated random query strings of length ℓ , broke these down into k -mers, and checked that none of the k -mers were contained in any document. To balance the size, we selected 100 000 true positives and 100 000 true negatives for $\ell = 31$ and $\ell = 100$, for $\ell = 1\,000$ we selected 10 000 true positives and negatives each, and for $\ell = 10\,000$ we selected 1 000 each.

The queries are stored in FASTA format and annotated with their origin (random negative or the correct document id). After running the queries, we checked the results of each index software by comparing it against the true origin. Using the true negatives in the $\ell = k = 31$ set we can determine the false positive rate of each index.

Measurements. To evaluate the software we measured many different performance metrics while running construction and the batch queries. The machine was used exclusively when running the experiments. Using interfaces from the Linux kernel, we measured wall-clock time, CPU user time which captures time spent computing in any user thread, the maximum resident set size (RSS) in memory as returned by the `time` utility, the number of bytes read and written to the SSDs in each step, and the change in storage usage. We also recorded the resulting size of the index data files. We flushed the disk cache before each build phase or query batch. Each query batch was run three times: the first round started with a flushed (cold) cache, and the two subsequent rounds with a warm cache. The rounds are labeled r0, r1, and r2.

3.1 Results

In this section we present and discuss the results of our experiments with the eight index software packages. The machine we selected for the experiments is a large server-class platform with 160 cores and large amounts of RAM. While these properties are always good, we primarily chose it due to the 8 TB of fast SSD storage, which is many times faster than traditional rotational disks. For rapidly performing the experiments, this storage speed was crucial.

On the other hand the fast storage speed and massive multi-core processing power in our machine may highlight different aspects in the indexing software than previous comparisons. Most prominently, algorithms which previously only had to process data rates known from rotational disks (100s of MiB/s) may become a bottleneck when

Table 1. Construction wall-clock time, CPU time, memory usage, resulting index size, and query wall-clock time for 1000 microbial documents and all k -mer index software in our experiment

phase	SBT	SSBT	AllSome-SBT	HowDe-SBT	Seq-Othello	Mantis	BIGSI	COBS Classic	COBS Compact
Construction Wall-Clock Time in Seconds									
count	2018	1974	1954	1959					
bloom	114	117	140	144	295	232	1881		
build	3097	21378	1401	68034	2225	987	2574	99	43
compress	1768	5187	80	3802		45			
total	6996	28657	3576	73939	2520	1264	4455	99	43
Construction CPU (User) Time in Seconds									
count	4574	4511	4475	4488					
bloom	11133	10967	10234	10278	28123	19162	169345		
build	855	5178	449	66872	2198	943	1767	1604	1430
compress	1569	4832	1663	2857		3423			
total	18131	25489	16821	84495	30320	23527	171113	1604	1430
Construction Maximum RSS Memory Usage in MiB									
count	518	518	518	518					
bloom	641	640	640	640	634	1756	4244		
build	11028	1523	7140	108147	12137	88357	246806	16245	2616
compress	10953	992	560	963		16613			
maximum	11028	1523	7140	108147	12137	88357	246806	16245	2616
Index Size in MiB									
size	19844	3254	21335	1911	4410	16486	27794	16236	3022
ℓ	Query Wall-Clock Time in Seconds								
31 bp r0	31	80	20	34	62	12	281	10	8
31 bp r2	26	76	19	33	62	13	289	9	8
100 bp r0	663	3183	100	600	73	22	783	14	9
100 bp r2	649	3153	95	588	73	23	455	14	9
1000 bp r0	794	3466	112	670	63	21	660	15	10
1000 bp r2	781	3435	108	659	64	27	310	13	10
10000 bp r0	802	3273	112	622	62	23	699	16	11
10000 bp r2	790	3243	111	613	62	22	316	15	11
total r0-r2	6775	29833	1007	5710	783	252	5177	154	114
Document False Positive Rate for 31 bp Queries									
rate	0.004	0.004	0.004	0.004	0.001	0.000	0.027	0.024	0.227

dealing with SSD speeds (currently around 10 GiB/s). Furthermore, most of the index software packages had no built-in provisioning for utilizing multi-core parallelism. While we were able to accelerate embarrassingly parallel parts of the construction using bash (like creating Bloom filters for each file), in some software the main index build was still sequential. On the other hand, one can argue that index construction time is not as important as query performance, but it still limits scalability.

Table 1 shows our results from all eight software packages for 1000 microbial DNA documents. The steps in the construction of each index are shown as separate rows

if it was possible to measure these independently. We show both wall-clock time and CPU user time such that parallelized construction can highlight its speedup without obscuring the actual amount of computation. For queries we show only wall-clock due to space, but all query computations are performed with a single threaded such that this is a fair comparison. Furthermore, for COBS classic and COBS compact the index is *completely loaded* into RAM such that the comparison with the others is fair.

Considering construction wall-clock time, COBS compact is clearly the fastest index taking only 43 seconds on 1 000 documents. COBS classic is a factor 2.3 slower, Mantis a factor 30 slower, SeqOthello a 59 factor, and AllSome-SBT a factor 83 slower than COBS compact. The same is reflected in construction CPU time, with COBS compact being fastest and taking 1430 seconds. COBS classic is a factor 1.12 slower, AllSome-SBT a factor 11.8 slower, and Mantis a factor 16.5.

One can also see that we parallelized the Bloom filter construction (the “bloom” row) effectively for all indexes, while the build steps are usually only partially parallelized. COBS compact has a CPU/wall-clock speedup of 33, while BIGSI has 38, Mantis has 18, and SeqOthello 12. However, since COBS compact performs *the least amount* of computation and has among the highest speedups, the combination of these two factors really diminishes wall-clock construction time.

The amount of RAM required by the indexing software also limits their applicability, especially if the complete index itself needs to be constructed in RAM. BIGSI, HowDe-SBT, and Mantis have the highest main memory usage in the experiment. For BIGSI and Mantis memory was the limiting scalability factor, while for HowDe-SBT the construction time grew too long.

In terms of query performance, the fastest software was COBS Compact with 114 seconds to run all query sets three times, followed by COBS classic with 154 seconds. Mantis was 2.2 times slower, SeqOthello 6.9 times lower, and the fastest SBT version, AllSome-SBT, was 8.8 times slower.

Figure 4 shows scaling results for all software packages on increasing subsets of the indexed microbial document set. We skipped running the SBT variants for data sets larger than 10 000 because their construction time was growing super-linearly. SeqOthello and Mantis scaled much better in terms of construction time per document. These plots show that COBS scales well with an order of magnitude faster construction time per document than Mantis and SeqOthello, both in wall-clock and CPU time. While COBS classic’s index size appears to increase with the number of documents (due to the maximum document size), the size per document of COBS compact actually decreases because it can better pack documents into blocks.

As expected COBS’ query time for single k -mers increases linearly with the number of documents in the index, due to the scoring method without pruning. The query time of all other indexes also increases with the number of documents, but not quite linearly. The best index in terms of query time increase per document is the AllSome-SBT followed by HowDe-SBT, but only COBS compact index scales to the full 100 000 microbial dataset.

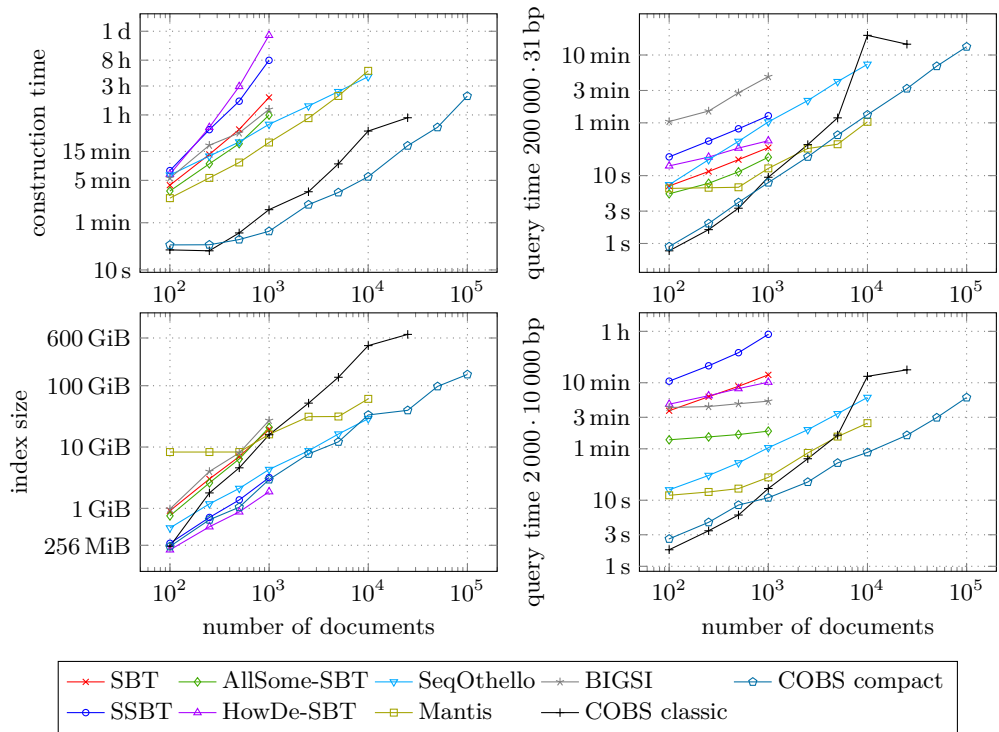


Fig. 4. Construction time, index size, query time for $200\,000 \cdot 31$ bp and for $2\,000 \cdot 10\,000$ bp round 2 after disk cache warm-up.

4 Conclusion and Future Work

With COBS we presented a signature index based on Bloom filters which enables approximate pattern matching on large q -gram datasets. It outperforms all other q -gram indexes in both construction and query time due to its simple data structure.

There are many avenues for future work on possible improvements to COBS' ideas. For example, dynamic operations on the index such as insertion, replacement, and removal of documents are very important for practical applications. We already provide a merge operation for classic indexes, but not for compact ones. Within time of an extended version of this paper we plan to support querying of multiple index files, such that a frontend may select different datasets or categories. Another important topic is better support for batch or bulk queries. And for further scalability it is important to explore distributed index construction and query processing.

Deriving from the simplicity of COBS are research avenues which could explore compression of rows in the Bloom filter matrix using techniques from information retrieval. And similar to Mantis' use of the CQF one could explore how to adapt other Bloom filter variants to the indexing problem with allowed false positives.

References

1. Fatemeh Almodaresi, Prashant Pandey, and Robert Patro. Rainbowfish: A succinct colored de Bruijn graph representation. In *17th International Workshop on Algorithms in Bioinformatics (WABI)*, volume 88 of *LIPICs*, pages 18:1–18:15. Schloss Dagstuhl, August 2017. preprint bioRxiv:138016.
2. Fatemeh Almodaresi, Hira Sarkar, Avi Srivastava, and Robert Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.
3. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.
4. Timo Bingmann. NVMe “disk” bandwidth and latency for batched block requests, March 2019. Online Article, <http://panthema.net/2019/0322-nvme-batched-block-access-speed>.
5. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
6. Phelim Bradley, Henk C. den Bakker, Eduardo P. C. Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*, 37:152–159, February 2019.
7. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.
8. Andrei Z. Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.
9. Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k -long DNA sequences. *Computing Research Repository (CoRR)*, arXiv:1903.12312:1–16, March 2019.
10. Charles E. Cook, Rodrigo Lopez, Oana Stroe, Guy Cochrane, Cath Brooksbank, Ewan Birney, and Rolf Apweiler. The European Bioinformatics Institute in 2018: tools, infrastructure and training. *Nucleic Acids Research*, 47(D1):D15–D22, January 2019.
11. Adina Crainiceanu and Daniel Lemire. Bloofi: Multidimensional Bloom filters. *Information Systems*, 54:311–324, December 2015.
12. Chris Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems (TOIS)*, 2(4):267–288, October 1984.
13. Florian Gauger. Engineering a compact bit-sliced signature index for approximate search on genomic data. Master Thesis. Karlsruhe Institute of Technology, Germany, February 2018.
14. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *LNCS*, pages 326–337. Springer, June 2014. preprint arXiv:1311.1249.
15. Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. BitFunnel: Revisiting signatures for search. In *40th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 605–614. ACM, August 2017.
16. Robert S. Harris and Paul Medvedev. Improved representation of sequence Bloom trees. *bioRxiv*, page 501452, December 2018.

17. Peter W. Harrison, Blaise T. F. Alako, Clara Amid, Ana Cerdeño-Tárraga, Iain Cleland, Sam Holt, Abdulrahman Hussein, Suran Jayathilaka, Simon Kay, Thomas M. Keane, Rasko Leinonen, Xin Liu, Josué Martínez-Villacorta, Annalisa Milano, Nima Pakseresht, Jeena Rajan, Kethi Reddy, Edward Richards, Marc Rossello, Nicole Silvester, Dmitriy Smirnov, Ana Luisa Toribio, Senthilnathan Vijayaraja, and Guy Cochrane. The European Nucleotide Archive in 2018. *Nucleic Acids Research*, 47(D1):D84–D88, January 2019.
18. Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, 20(2):192–223, April 2002.
19. Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11(1):3, April 2016.
20. Zamin Iqbal, Isaac Turner, and Gil McVean. High-throughput microbial population genomics using the cortex variation assembler. *Bioinformatics*, 29(2):275–276, November 2012.
21. Johannes Krugel. *Approximate Pattern Matching with Index Structures*. PhD thesis, Technische Universität München, Germany, February 2016.
22. Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k -mers. *Bioinformatics*, 27(6):764–770, 2011.
23. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
24. Hamid Mohamadi, Hamza Khan, and Inanç Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, 2017.
25. Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. preprint bioRxiv:040071.
26. Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell Systems*, June 2018. preprint bioRxiv:217372.
27. Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In *ACM International Conference on Management of Data*, pages 775–787. ACM, 2017.
28. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. In *13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242. SIAM, January 2002.
29. Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3):300–312, February 2016.
30. Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence Bloom trees. *Journal of Computational Biology*, 25(7):755–765, April 2018.
31. Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. AllSome sequence Bloom trees. *Journal of Computational Biology*, 25(5):467–479, May 2018. preprint bioRxiv:090464.
32. Isaac Turner, Kiran V. Garimella, Zamin Iqbal, and Gil McVean. Integrating long-range connectivity information into de Bruijn graphs. *Bioinformatics*, 34(15):2556–2565, 2018.

33. Esko Ukkonen. Approximate string-matching with q -grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, January 1992.
34. Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *11th International Conference on Very Large Data Bases (VLDB)*, pages 448–457. VLDB Endowment, August 1985.
35. Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. SeqOthello: querying RNA-seq experiments at scale. *Genome Biology*, 19(1):167, 2018. preprint bioRxiv:258772.
36. Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006.
37. Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490, 1998.