COBS: A Compact Bit-Sliced Signature Index

Timo Bingmann[1(✉)], Phelim Bradley[2], Florian Gauger[1], and Zamin Iqbal[2]

[1] Institute of Theoretical Informatics,
Karlsruhe Institute of Technology, Karlsruhe, Germany
bingmann@kit.edu
[2] European Molecular Biology Laboratory,
European Bioinformatics Institute, Cambridge, UK

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. 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. in second place. However, unlike 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 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`. Experimental methods for "reading" DNA have been developing rapidly; there are various approaches, but all involve breaking the DNA and "reading" (typically called "sequencing") those fragments (these short strings are typically called "reads"). Read lengths started out moderately long (500–1000 characters) in the late 1990s, dropped down to 30 characters in 2008 with the advent of massively parallel technologies, and in the recent past, bleeding edge technologies have enabled reading of fragments as long as 1 million characters, albeit with a higher error rate.

The output of sequencing experiments are stored both in raw format (text files of the read strings) and "assembled format" – semi-heuristic best approximations to the underlying genome, also in text format, but of very variable quality,

N. R. Brisaboa and S. J. Puglisi (Eds.): SPIRE 2019, LNCS 11811, pp. 285–303, 2019.
https://doi.org/10.1007/978-3-030-32686-9_21

in particular when based on short read data. Unambiguous reconstruction of the original string from the substrings is mathematically impossible unless the fragments are longer than the longest repeated substring. Another complication is that a great deal of data is generated by sequencing unknown mixtures of different genomes (e.g. mixtures of bacteria from within the human gut, or samples from humans infected by three different types of malaria parasite), making it very hard to reconstruct the underlying genomes.

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 stored datasets, to find important genes or mutations, or combinations of mutations which are informative of function or ancestry. All of these search queries can be expressed in terms of exact or approximate matching of strings. 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 [6], whereas the small benchmark set of 100 000 microbial sequences used in our experiments already contain $2.2 \cdot 10^{10}$ distinct 31-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 [22,27]. 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 [36] 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 [41]. 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.

After reviewing related work in the following subsection, we present the new COBS index design in Sect. 2. In Sect. 3 we then report on our experimental evaluation of COBS and seven other k-mer indexing software packages.

1.1 Related Work

Considering q-grams or k-mers of a sequence are a staple in bioinformatics [8].

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 [32]. In an SBT the k-mers of each document are indexed into individual Bloom filters, 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 [32] a simple greedy clustering method is used, the bit union is stored in each inner node, and all nodes are RRR compressed [31] using SDSL [14]. The first improvement, the *Split Sequence Bloom Tree* (SSBT) [33], 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. This representation allows descendant nodes to omit storing the bits in the *similarity* filter explicitly, hence reducing space requirements while retaining the same information.

Simultaneously, Sun, Harris, Chikhi, and Medvedev proposed the *AllSome Sequence Bloom Tree* (AllSome-SBT) [34], 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. Again, this representation allows exclusion of bits already known from the parent node's filters, and thus reducing space and enabling better compression. Furthermore, the AllSome-SBT also improves on the clustering methods by employing an agglomerative hierarchical technique and by constructing batch Bloom filters for large query sets.

The currently smallest SBT variant is called *HowDe Sequence Bloom Tree* (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 the *how* vector signals if it is determined as zero or one. All determined bits can be omitted from any children. These two bit vectors are exactly the information needed to perform an efficient breadth-first search down the tree. Furthermore, the authors introduce a *culling* process to remove sparse inner nodes which don't reveal much information and thus speed up queries.

A completely different approach to indexing k-mers is taken by *Mantis* from Pandey et al. [28]. In Mantis, a counting quotient filter (CQF) [29] 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. Hence, bits set in the rows signal occurrence of any k-mer mapping to the color in the corresponding document list. Mantis then compresses the bit vectors in the color matrix using RRR or with a spanning tree based approach. The k-mer mapping is built from CQFs constructed by *Squeakr* [30], a k-mer counting tool. 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 [39] 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. Occurrence maps are then grouped depending on their density and encoding into disjoint buckets. To locate the correct occurrence map for a k-mer, a hierarchy of *Othello*s is built inside each bucket and over all bucket *Othello*s. An *Othello* [38] is a minimum perfect hash function mapping, which is fast and scalable but can introduce false positive results due to mapping of alien k-mers to random results.

BIGSI (BItsliced Genomic Signature Index) by Bradley et al. [5] 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* [20,21], 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* [35], which added a second data structure to encode paths in the graph present in the original reads. By contrast, *VARI* [26], *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* by Holley et al. [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 Subsect. 2.1, then turn to COBS' more compact bit-sliced representation in Subsect. 2.2, and discuss implementation details and algorithm engineering aspects in Subsect. 2.3.

2.1 Approximate Matching with Bloom Filters of Signatures

Given are an ordered set of *documents* $\mathcal{D} = [d_0, \ldots, d_{|\mathcal{D}|-1}]$, where each document d is composed of a set of *strings* $\{t_0, \ldots, t_{|d|-1}\}$. 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 $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, the documents are experiment samples, and the strings in each document can be reads or assembled genome sequences. When indexing web sites, the alphabet may be the ASCII characters or English words, the documents could be web pages, and the substrings may be words, sentences, or paragraphs.

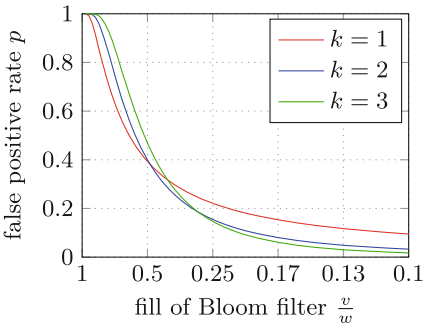

Fig. 1. Theoretical false positive rate of Bloom filters given fill and number of hash functions.

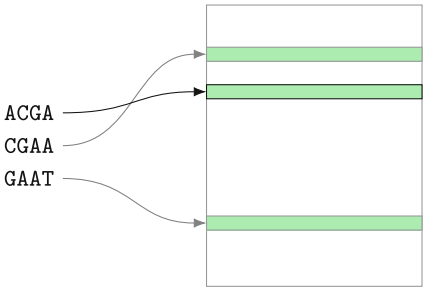

Fig. 2. Access pattern of the classical bit-sliced index.

To facilitate approximate pattern matching we consider *q-grams* of the strings [36], 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*.

A COBS index is composed of $|\mathcal{D}|$ Bloom filters [4], 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 $h_0, \ldots, h_{k-1}$ 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 $[f_0, \ldots, f_{|\mathcal{D}|-1}]$, 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 $w = -\frac{v \ln p}{(\ln 2)^2}$ [7,24].

To perform approximate matching for a pattern P, we follow previous work [36] 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 interested in the false positive rate of a *single* Bloom filter lookup. Instead we are concerned with the false positive 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 [32]). *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. Figure 1 shows the false positive rate $(1 - e^{-kv/w})^k$ of Bloom filters depending on its fill $\frac{v}{w}$ and the number of hash functions k. 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 Fig. 3). This is also called a "bit-sliced" layout [37] and was chosen for BIGSI and COBS to reduce the number of random accesses needed to evaluate a query. Each row of a term can be scanned sequentially, as shown in Fig. 2. This is particularly important if the index is read from external memory, where

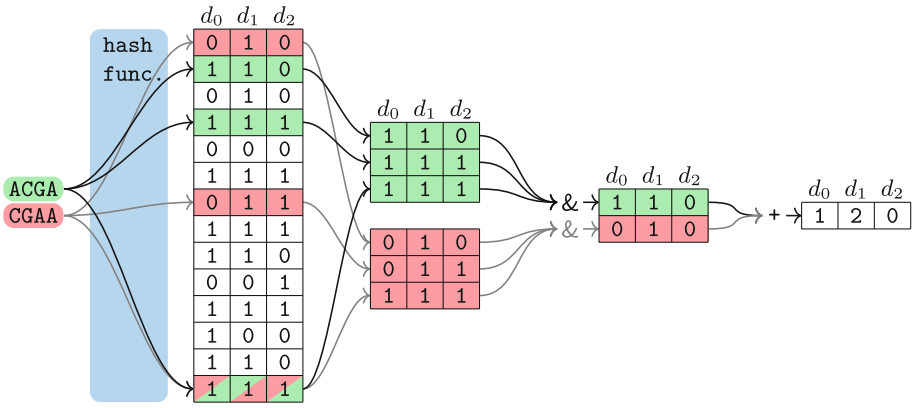

Fig. 3. Architecture of the bit-sliced signature index and query processing steps.

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 3 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 [40] 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 $\Theta(B)$ in external memory, thus keep the parameters constant for 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 4 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 $B = 8\,192$ 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.

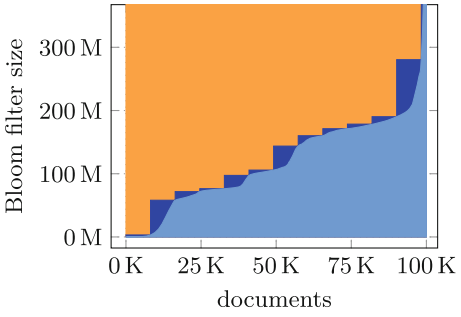

Fig. 4. Compact index composed of small sub-indexes containing B documents.

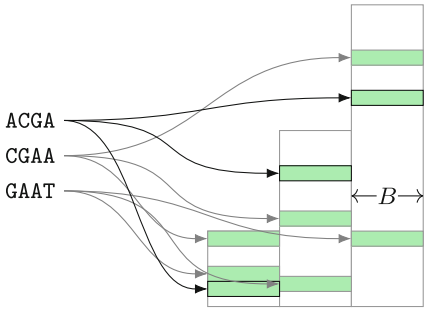

Fig. 5. Access pattern of our compact bit-sliced index.

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 [3] 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 Fig. 5, 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. We used xxHash [9] for hashing the q-gram strings. 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* (ClaBS) and *compact* (COBS). 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. We will refer to it as **ClaBS** in the experimental results.

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 B documents, as described in Subsect. 2.2. Each subindex is actually a ClaBS index. The subindices are simply concatenated into one large file.

While classic indexes with the same parameters can be concatenated straightforwardly, compact indexes are more difficult to merge. We may implement this in future versions of COBS by keeping some slack in the $\Theta(B)$ blocks and packing new documents into the best free block or by storing the subindices as separate files. This would allow incremental augmentation of COBS compact indices.

Parallelization. Due to the massive amount of data to process, we parallelized construction and query for shared-memory systems. ClaBS 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 in COBS can optionally be 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 simple memory transfers. 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 Sect. 3.

Single-Instruction Multiple-Data (SIMD). Besides the I/O bottleneck, extracting the bits from the index rows and adding them together required a considerable amount of running time in the query.

In the *ADD* step of the query process (Fig. 3), the rows are summed up to create the query result. In this illustration we hid the fact that the rows that are output from the *AND* step are bit-packed: each cell is represented by one bit. In the output of the *ADD* step, however, each document's score is represented by an integer specifying the number of matched query terms. This poses a problem

since the bits need to be unpacked before they can be processed. Ideally we would like to unpack and process multiple bits at once.

We use a straight-forward mapping to expand 8 bits output by the *AND* step to the $8 \cdot 16 = 128$ bits needed by the *ADD* step when using 16-bit score counters. This can be achieved by using one array lookup in a table of length 256 containing items of 128 bits. With these 128 bits, the final result can then be calculated by summing up the expanded values for each document using a single 128-bit SIMD instruction. The same approach can also be done with 32-bit score counters with 256-bit or two 128-bit instructions.

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.

Table 1. Software, references, git hashes, and commit dates used in experiments.

Software/Index	Git hash and commit date
SBT [32]	`977adfa` from March 1st 2019
SSBT (Split-SBT) [33]	`710c95f` from July 10th 2018
AllSome-SBT [34]	`4e1f2c5` from October 28th, 2018
HowDe-SBT [16]	`76e3c89` from March 1st, 2019
SeqOthello [39]	`68d47e0` from September 6th, 2018
Mantis [28]	`3853c82` from January 29th, 2019
BIGSI [5]	`2ab35e5` from May 15th, 2019 using BerkeleyDB 4.8.30
COBS and **ClaBS** [this]	`5328bd5` from May 24th, 2019

Software Packages. We acquired copies of the original source code of seven other index software packages via Github. The paper references, git hashes, and commit dates are listed in Table 1. More information about each package can be found in the related work Subsect. 1.1. We compiled all software from source and additionally used ntCard [25] (v1.1.0) as a preprocessing step for the SBTs, jellyfish [23] (v2.2.10) in other steps and as a library.

Data. Bradley et al. [5] 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 0000, 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 [5]. 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 3.4 M 31-mers on average with the minimum being zero and the largest containing 138 M 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.

For COBS' compact index $B = 1\,024$ documents were grouped into a sub-index in the largest instance with 100 000 documents.

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 we used 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 80 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 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 2 shows our results from all eight software packages for only 1 000 microbial DNA documents. The steps in the construction of each index are shown as separate rows if it was possible to measure these independently.

Table 2. Construction wall-clock time, CPU time, memory usage, and resulting index size for 1 000 microbial documents and all k-mer index software in our experiment

Phase	SBT	SSBT	AllSome-SBT	HowDe-SBT	Seq-Othello	Mantis	BIGSI	ClaBS	COBS
	Construction wall-clock time in seconds								
Count	2 018	1 974	1 954	1 959					
Bloom	114	117	140	144	295	232	1 881		
Build	3 097	21 378	1 401	68 034	2 225	987	2 574	99	43
Compress	1 768	5 187	80	3 802		45			
Total	6 996	28 657	3 576	73 939	2 520	1 264	4 455	99	43
	Construction CPU (User) time in seconds								
Count	4 574	4 511	4 475	4 488					
Bloom	11 133	10 967	10 234	10 278	28 123	19 162	169 345		
Build	855	5 178	449	66 872	2 198	943	1 767	1 604	1 430
Compress	1 569	4 832	1 663	2 857		3 423			
Total	18 131	25 489	16 821	84 495	30 320	23 527	171 113	1 604	1 430
	Construction maximum RSS memory usage in MiB								
Count	518	518	518	518					
Bloom	641	640	640	640	634	1 756	4 244		
Build	11 028	1 523	7 140	108 147	12 137	88 357	246 806	16 245	2 616
Compress	10 953	992	560	963		16 613			
Maximum	11 028	1 523	7 140	108 147	12 137	88 357	246 806	16 245	2 616
	Index size in MiB								
Size	19 844	3 254	21 335	1 911	4 410	16 486	27 794	16 236	3 022

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. Table 3 considers the time to run the query sets. We only show wall-clock for queries due to space; all query computations are performed with a single thread such that this is a fair comparison. Furthermore, for ClaBS and COBS the index is *completely loaded* into RAM such that the comparison with the others is fair. In future, it will become important to measure how many bytes were read from the disks per query, but in the current comparison we assume all index data is resident in RAM.

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

Table 3. Query wall-clock time for 1 000 microbial documents and all k-mer index software in our experiment

Phase	SBT	SSBT	AllSome-SBT	HowDe-SBT	Seq-Othello	Mantis	BIGSI	ClaBS	COBS
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	6 775	29 833	1 007	5 710	783	252	5 177	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

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 has a CPU/wall-clock speedup of 33, while BIGSI has 38, Mantis has 18, and SeqOthello 12. However, since COBS performs *the least amount* of computation and has among the highest speedups, the combination of these two factors really diminishes wall-clock construction time. Considering CPU user time, the index requiring most work for construction is BIGSI, probably due to the Python implementation. It however is parallelized, such that the wall-clock time is on par with the SBTs.

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.

The index sizes of all packages for the 1 000 microbial documents was smaller than the input in McCortex format (41 GiB input size). The software with the smallest index was the HowDe-SBT with only 1.9 GiB, followed by COBS with around 3.0 GiB and SSBT with 3.3 GiB.

In terms of query performance, the fastest index was COBS with 114 s to run all query sets three times, followed by ClaBS with 154 s. Mantis was a factor 2.2 slower, SeqOthello a factor 6.9 slower, and the fastest SBT version, AllSome-SBT, was a factor 8.8 slower.

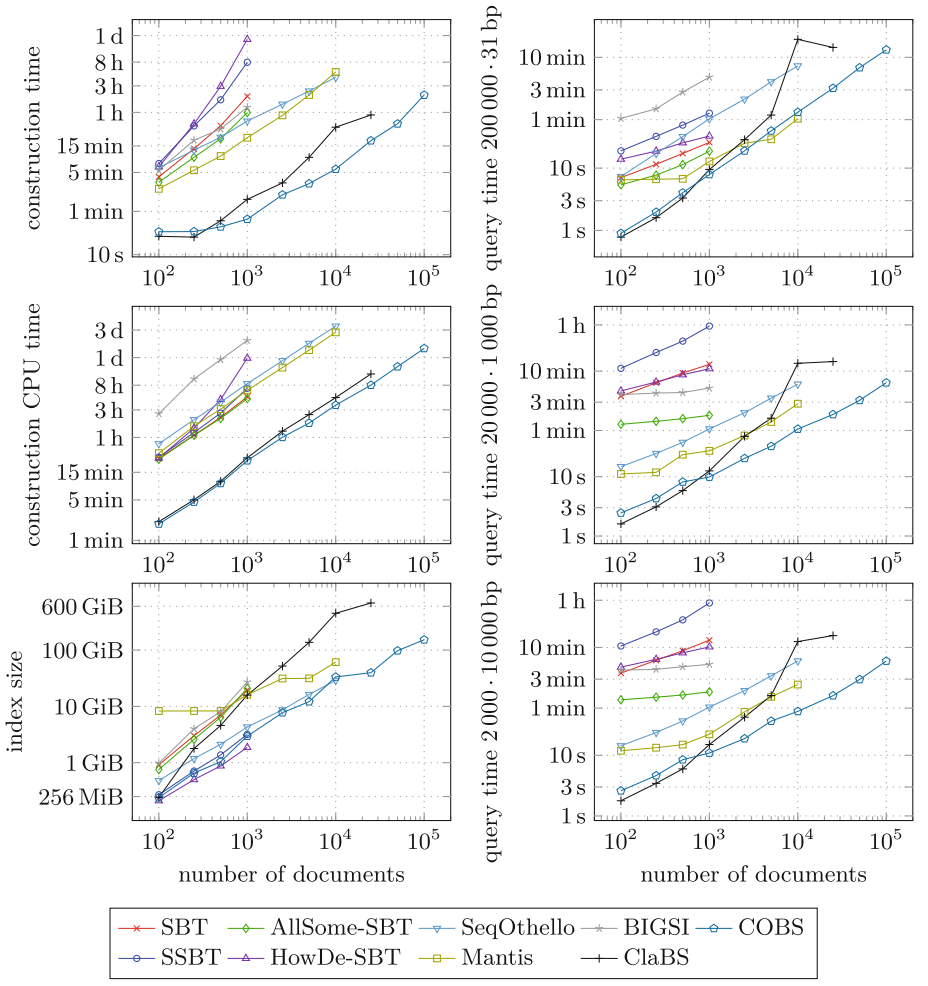

Fig. 6. Construction time, index size, query time for 200 000 · 31 bp, 20 000 · 1 000 bp, and for 2 000 · 10 000 bp round 2 after disk cache warm-up.

Using result checkers we verified that all software packages calculated correct results and counted the false positives contained in the returned list of the single k-mer query set ($\ell = 31$). The notable exception was SeqOthello, which produced false positives consistently for each multi-k-mer query and started returning false negative (missing) results when run on the 10 000 dataset. We could not investigate this issue further. The SBT variants and SeqOthello showed a very low false positive rate less than 0.5 %. Mantis produced zero false positives as expected. BIGSI and ClaBS are nearly identical in underlying data structure design, and deliver around 2.6 % false positives on single k-mer queries. COBS is designed to deliver about the prescribed error rate of 0.3, hence the 22.7 %

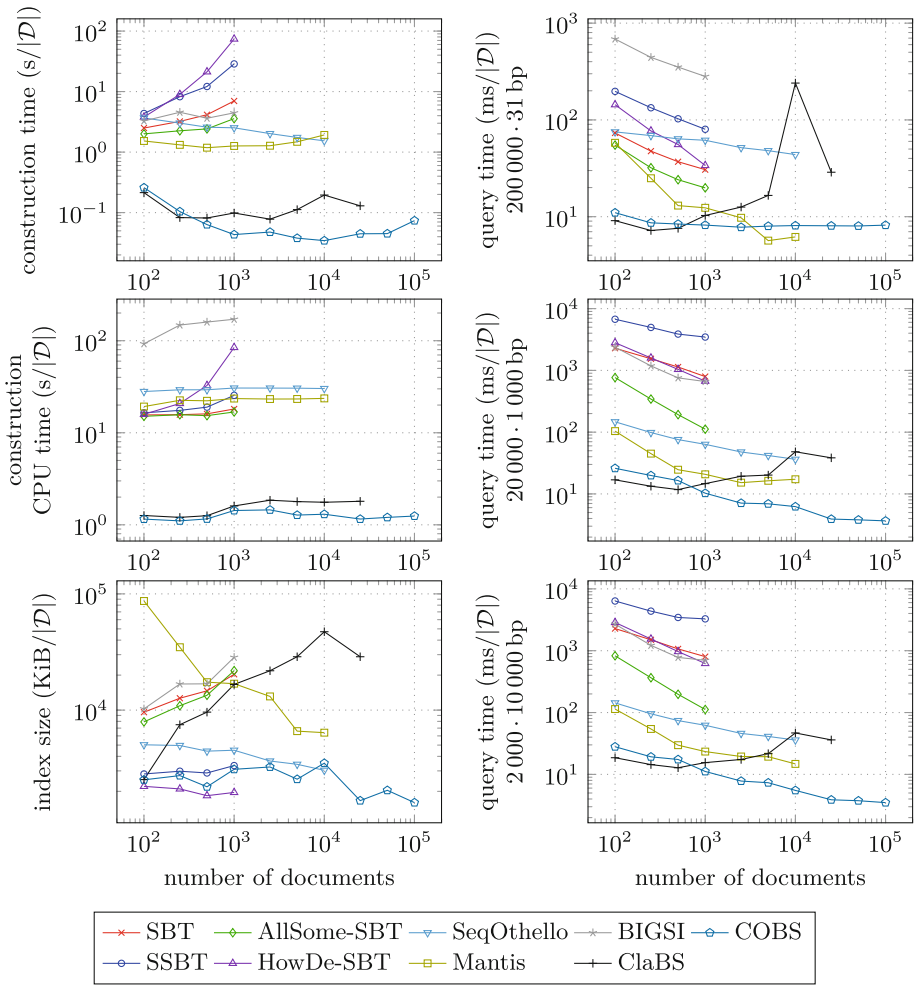

Fig. 7. Construction time, index size, query time for 200 000 · 31 bp, 20 000 · 1 000 bp, and for 2 000 · 10 000 bp in the first round divided by the number of documents $|\mathcal{D}|$.

false positives, which enables us to construct a more compact index. We also calculated the number of false positives in larger multi-k-mer query sets, and found all indexes except SeqOthello but including COBS to return *zero* false positives for all queries with $\ell \geq 100$ in the experiment.

Figure 6 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. Figure 7 shows 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 ClaBS's index size appears to increase with the number of documents (due to the maximum document size), the size per document of COBS 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 index scales to our full 100 000 microbial dataset.

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 for multi-q-gram queries 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. Our current COBS implementation also already supports 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. Almodaresi, F., Pandey, P., Patro, R.: Rainbowfish: a succinct colored de Bruijn graph representation. In: 17th International Workshop on Algorithms in Bioinformatics (WABI). LIPIcs, vol. 88, pp. 18:1–18:15. Schloss Dagstuhl, August 2017. preprint bioRxiv:138016
2. Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R.: A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics **34**(13), i169–i177 (2018)
3. Bingmann, T.: NVMe "disk" bandwidth and latency for batched block requests, March 2019. Online Article, http://panthema.net/2019/0322-nvme-batched-block-access-speed
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Bradley, P., den Bakker, H.C., Rocha, E.P.C., McVean, G., Iqbal, Z.: Ultrafast search of all deposited bacterial and viral genomic data. Nat. Biotechnol. **37**, 152–159 (2019)
6. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Comput. Networks ISDN Syst. **30**(1–7), 107–117 (1998)
7. Broder, A.Z., Mitzenmacher, M.: Network applications of Bloom filters: a survey. Internet Math. **1**(4), 485–509 (2003)
8. Chikhi, R., Holub, J., Medvedev, P.: Data structures to represent sets of k-long DNA sequences. Computing Research Repository (CoRR), arXiv:1903.12312:1–16, March 2019
9. Collet, Y.: xxHash: extremely fast non-cryptographic hash algorithm, 2014. Git repository. https://github.com/Cyan4973/xxHash. Accessed July 2019
10. Cook, C.E., Lopez, R., Stroe, O., Cochrane, G., Brooksbank, C., Birney, E., Apweiler, R.: The European Bioinformatics Institute in 2018: tools, infrastructure and training. Nucleic Acids Res. **47**(D1), D15–D22 (2019)
11. Crainiceanu, A., Lemire, D.: Bloofi: multidimensional bloom filters. Inf. Syst. **54**, 311–324 (2015)
12. Faloutsos, C., Christodoulakis, S.: Signature files: an access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst. (TOIS) **2**(4), 267–288 (1984)
13. Gauger, F.: Engineering a compact bit-sliced signature index for approximate search on genomic data. Master Thesis. Karlsruhe Institute of Technology, Germany, February 2018
14. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_28
15. Goodwin, B., et al.: BitFunnel: revisiting signatures for search. In: 40th ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 605–614. ACM, August 2017
16. Harris, R.S., Medvedev, P.: Improved representation of sequence Bloom trees. bioRxiv, pp. 501452, December 2018
17. Harrison, P.W., et al.: The european nucleotide archive in 2018. Nucleic Acids Res. **D47**(1), D84–D88 (2019)
18. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM Trans. Inf. Syst. (TOIS) **20**(2), 192–223 (2002)

19. Holley, G., Wittler, R., Stoye, J.: Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms Mol. Biol. **11**(1), 3 (2016)
20. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nat. Genet. **44**(2), 226 (2012)
21. Iqbal, Z., Turner, I., McVean, G.: High-throughput microbial population genomics using the cortex variation assembler. Bioinformatics **29**(2), 275–276 (2012)
22. Krugel, J.: Approximate Pattern Matching with Index Structures. Ph.D. thesis, Technische Universität München, Germany, February 2016
23. Marçais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics **27**(6), 764–770 (2011)
24. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
25. Mohamadi, H., Khan, H., Birol, I.: ntCard: a streaming algorithm for cardinality estimation in genomics data. Bioinformatics **33**(9), 1324–1330 (2017)
26. Muggli, M.D., et al.: Succinct colored de Bruijn graphs. Bioinformatics **33**(20), 3181–3187 (2017). preprint bioRxiv:040071
27. Navarro, G., Baeza-Yates, R.A., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Bull. Tech. Committee Data Eng. **24**(4), 19–27 (2001)
28. Pandey, P., Almodaresi, F., Bender, M.A., Ferdman, M., Johnson, R., Patro, R.: Mantis: a fast, small, and exact large-scale sequence-search index. Cell Systems, June 2018. preprint bioRxiv:217372
29. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: making every bit count. In: ACM International Conference on Management of Data, pp. 775–787. ACM (2017)
30. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: Squeakr: an exact and approximate k-mer counting system. Bioinformatics **34**(4), 568–575 (2018). preprint bioRxiv:122077
31. Raman, R., Raman, V., Srinivasa Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 233–242. SIAM, January 2002
32. Solomon, B., Kingsford, C.: Fast search of thousands of short-read sequencing experiments. Nat. Biotechnol. **34**(3), 300–312 (2016)
33. Solomon, B., Kingsford, C.: Improved search of large transcriptomic sequencing databases using split sequence Bloom trees. J. Comput. Biol. **25**(7), 755–765 (2018)
34. Sun, C., Harris, R.S., Chikhi, R., Medvedev, P.: AllSome sequence Bloom trees. J. Computat. Biol. **25**(5), 467–479 (2018)
35. Turner, I., Garimella, K.V., Iqbal, Z., McVean, G.: Integrating long-range connectivity information into de Bruijn graphs. Bioinformatics **34**(15), 2556–2565 (2018)
36. Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. Theoret. Comput. Sci. **92**(1), 191–211 (1992)
37. Wong, H.K.T., Liu, H.-F., Olken, F., Rotem, D., Wong, L.: Bit transposed files. In 11th International Conference on Very Large Data Bases (VLDB), pp. 448–457. VLDB Endowment, August 1985
38. Ye, Y., Belazzougui, D., Qian, C., Zhang, Q.: Memory-efficient and ultra-fast network lookup and forwarding using othello hashing. IEEE/ACM Trans. Networking **26**(3), 1151–1164 (2018)
39. Ye, Y., et al.: SeqOthello: querying RNA-seq experiments at scale. Genome Biol. **19**(1), 167 (2018). preprint bioRxiv:258772

40. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surveys (CSUR) **38**(2), 6 (2006)
41. Zobel, J., Moffat, A., Ramamohanarao, K.: Inverted files versus signature files for text indexing. ACM Trans. Database Syst. (TODS) **23**(4), 453–490 (1998)